Design and Optimization of Decoders for Low-Density Parity Check Codes Synthesized from OpenCL Specifications

by

Andrew James Maier

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Department of Electrical and Computer Engineering
University of Alberta

Abstract

Open Computing Language (OpenCL) is a high-level language that allows developers to produce portable software for heterogeneous parallel computing platforms. OpenCL is available for a variety of hardware platforms, with compiler support being recently expanded to include Field-Programmable Gate Arrays (FPGAs). This work investigates flexible OpenCL designs for the iterative min-sum decoding algorithm for both regular and irregular Low-Density Parity Check (LDPC) codes over a range of codeword lengths.

The computationally demanding LDPC decoding algorithm offers several forms of parallelism that could be exploited by the Altera-Offline-Compiler (AOC version 15.1) for OpenCL. By starting with the recommended design approaches and optimizing based on experimentation, the highest throughput regular LDPC decoder produced a maximum corrected codeword throughput of 68.22 Mbps for 32 decoding iterations at the compiler-selected FPGA clock frequency of 163.88 MHz for a length-2048 (3,6)-regular code. Designs for three of the DOCSIS 3.1 [7] standard irregular codewords were investigated and implemented using the AOC. The designs prove that OpenCL on FPGAs can produce high-throughput results for industry-sized irregular LDPC codes with significantly shorter design time compared to other custom hardware and software applications.

“The most exciting phrase to hear in science, the one that heralds the most discoveries, is not ‘Eureka!’ (I found it!) but ‘That’s funny...’”
Isaac Asimov

Dedicated to my beloved family and friends...

Acknowledgements

Firstly, I would like to thank my supervisor, Dr. Bruce Cockburn, for his support, dedication, and advice throughout both my undergraduate degree and my M.Sc. program. Dr. Cockburn, I'm extremely grateful for your guidance with problem solving, technical writing, and conference presentations. I can't thank you enough for your dedication to my M.Sc. program and my research.

Throughout my degree, I was fortunate to have a variety of experiences and meet some incredible people that have truly impacted my life. I would like to thank David Sloan, not only for your help throughout the years, but for becoming a great friend. We've experienced so much in the past few years from competing on BattleBots and engineering random embedded systems to our LAN party Factorio days. Thank you, Saeed Ansari, for being such an amazing friend. Even though we only met in the last few months of my program, thank you for all of your help in my studies and for all of our adventures.

I'd like to thank my lovely girlfriend, Danielle Schultze, for her support over the last year and a bit of my degree. You pushed and motivated me to be my best each and every day.

I would also like to thank Khalid Al-Almoudi, Waleed El-Halwagy, and everyone else in the VLSI lab for creating such a welcoming environment. And, of course, I have to thank Mehdi Shaghaghi, Parisa Zadeh, and Andrew Hakman for all of our talks and your advice.

A huge thank you to all of my family and friends. After a lot of hard work in the lab, sometimes the best remedy is just spending time and hanging out with friends.

Most of all I need to thank my loving parents, my sister, and my grandparents for their support not just in my University years but throughout my entire life. Thank you for everything you've done for me. I truly could not have done it without you guys.

Finally, thank you to the CMC Microsystems for providing the training, licenses, and technical support for my research and to NSERC.

Contents

List of Figures

List of Tables

List of Acronyms

AOC	Altera-Offline-Compiler
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BPSK	Binary Phase-Shift Keying
CN	Check Node
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
DOCSIS	Data Over Cable Service Interface Specification
ECC	Error Correcting Codes
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDL	Hardware Description Language
LDPC	Low-Density Parity Check
LLR	Log-Likelihood Ratio

LTE	Long-Term Evolution
MIMD	Multiple-Instruction, Multiple-Data
MISD	Multiple-Instruction, Single-Data
MPI	Message Passing Interface
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
PCIe	Peripheral Component Interconnect Express
QC-LDPC	Quasi-Cyclic Low-Density Parity Check
RAM	Random-Access Memory
SDRAM	Synchronous Dynamic Random-Access Memory
SIMD	Single-Instruction, Multiple-Data
SISD	Single-Instruction, Single-Data
VN	Variable Node
XOR	Exclusive-OR

Chapter 1

Introduction

Error Correcting Codes (ECCs) have become essential in digital communication systems [28]. Many systems now include forward error correction based on low-density parity check (LDPC) block codes and turbo codes [32] that approach the Shannon capacity limit [30]. LDPC codes are available in both block-based and convolutional forms. This thesis considers only the block-based LDPC codes. Many recent communication standards require LDPC codes including the 4G LTE wireless standard [2], the 10GBASE-T Ethernet standard [36], the DOCSIS 3.1 cable modem standard [7], and the satellite communication standard DVB-S2 [6]. The large number of computations and the minimum allowed throughputs for the LDPC decoding algorithms, such as the 1 Gbps required for the DOCSIS 3.1 upstream codes [7], introduce many implementation challenges. Fortunately, the decoding algorithm presents several forms of parallelism that are potentially exploitable using a suitable hardware platform and programming environment.

With the adoption of LDPC codes in industry, a number of high-throughput hardware implementations have been described in the literature. Blanksby and Howland [16] designed a fully parallel 1Gbps decoder on one custom integrated circuit; however, their architecture is only practical for shorter codewords due to the large required hardware area for the interleaver network. Many other hardware-aware LDPC decoders have been proposed to handle longer codes, such as those used in DVB-S2 [34]. Researchers, such as Liao et al. [23], designed code-aware decoders to reduce the hardware cost without significantly compromising error correction performance for parallel and serial decoder designs. Although these designs produce a high data throughput, they are limited in that they cannot be easily reconfigured to handle different codeword sizes. It is now common for standards, such as DOCSIS 3.1 [7], to use multiple codeword lengths. A bit-serial decoder design was introduced by Tyler Brandon et al. [17] that is suitable for longer codes without assuming constraints on the structure of the parity check matrix. Researchers have also investigated stochastic decoder designs in an attempt to reduce the hardware cost of decoding long codewords [33].

Software decoders, along with their hardware counterparts, have been investigated that provide the flexibility to change codes. Implementations have been profiled using a variety of heterogeneous computational platforms including massively parallel Graphics Processing Units (GPUs) and multi-core Central Processing Units (CPUs). The massively parallel architecture of GPUs make them an attractive accelerator platform for problems involving largely single-instruction multiple-data (SIMD) vector types. To harness the power of these powerful computational platforms, high-level parallel languages, such as Compute Unified Device Architecture (CUDA) [1] and Open Computing Language (OpenCL) [4], were developed to allow the programmer to produce portable software that is optimized by the compiler for any compatible device. Andrade et al. investigated a flexible OpenCL decoder design for the Quasi-cyclic LDPC codes used in the WiMAX (802.16) standard [15].

1.1 Motivation

LDPC decoders have been programmed using OpenCL for implementations on both GPUs [14] and Field-Programmable Gate Arrays (FPGAs) [27]. LDPC decoders that use Khronos' OpenCL [4] to compile designs to a GPU were able to achieve throughputs similar to custom hardware implementations [19]. The primary benefit to using these high-level languages to specify decoder implementations for synthesis is the significant reduction in design time and effort when compared to a custom hardware Application-Specific Integrated Circuit (ASIC) design [18] [20]. The synthesized high-level designs can then be re-used for multiple possible target platforms.

Using the Altera-Offline-Compiler (AOC), this thesis investigates the viability of the high-level synthesis OpenCL language on reconfigurable architecture FPGAs against custom hardware and software approaches, using the LDPC decoding algorithm as a benchmark problem.

1.2 Thesis Overview

This thesis has four key chapters. Chapter 2 introduces the background knowledge required to implement an LDPC decoder with OpenCL. Decoder designs for a set of random regular LDPC codes (sometimes also called symmetric LDPC codes) are implemented and evaluated in Chapter 3. Chapter 4 uses the knowledge gained from the regular designs to implement the more challenging DOCSIS 3.1 industrial irregular (sometimes also called asymmetric) codes [7]. Finally, we will evaluate the portability aspect of OpenCL source code by compiling the source code onto OpenCL GPUs in Chapter 5. Chapter 6 then summarizes the contributions of the thesis and proposes directions for future work.

Chapter 2

Background

2.1 Heterogeneous Parallel Architecture

All of the described architectures were compiled and executed onto the custom-built heterogeneous computing platform called *Hextet*. As shown in Figure 2.1, Hextet's CPU is a multi-threaded Intel hexa-core i7 64-bit microprocessor. This CPU is capable of executing 12 simultaneous threads of software instructions. The system has the Nvidia GeForce GTX690 (x2 chips) graphics processor with 4GB of available RAM [12]. The FPGA used to evaluate the architectures using the Altera-Offline-Compiler is the Nallatech P385N-A7 board that contains the Altera Stratix V FPGA and 8GB of off-chip SDRAM. Note that the connection between the FPGA and the CPU is done via an 8-lane PCI express (PCIe) generation 2 bus whereas the connection between the GPU and the CPU is done via a 16-lane PCIe generation 3 bus.

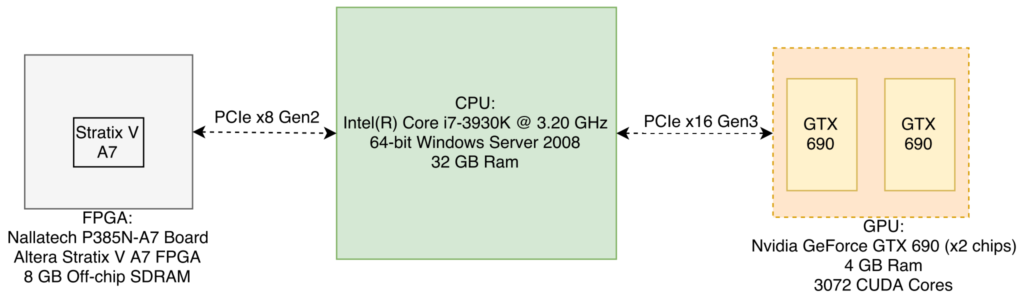


Figure 2.1: Hextet System Configuration

2.2 Parallel Programming Environments

With the introduction of more heterogeneous architectures, many different parallel programming environments have been created [25]. These programming environments allow the developer to program at a higher level, leaving the responsibility of communication and low-level device operation to compiled code. OpenCL [4], initially released on December 9, 2008 by the Khronos Group, is an example of one of these parallel programming environments. Initially supported by specific CPU manufacturers, OpenCL expanded to be compatible with most commerical GPUs and, most recently, FPGAs from the two largest vendors, Xilinx and Altera.

Some other relevant parallel programming environments target some of the same execution devices as OpenCL. Nvidia's CUDA was developed to execute on the company's own GPUs and hardware accelerators [1]. CUDA is similar in programming style to OpenCL in that it involves a host CPU and enabled device. ArrayFire, formerly AccelerEyes, produced an add-on to the Matlab programming language called Jacket [5]. Jacket was introduced to enable Matlab source code to define parallel data structures and data operations on Nvidia GPUs for computational acceleration. More recently, MathWorks included the Parallel Computing Toolbox in Matlab [31].

Other heterogeneous parallel programming environments include the OpenMP and MPI standards. OpenMP is an application programming interface available for most compilers and that provides a compiler-directed interface for launching multi-threaded applications [8]. Message Passing Interface (MPI) is a library specification developed by the MPI forum that provides an interface for communicating with nodes in a heterogeneous platform [11]. OpenMP and MPI are commonly used in conjunction to produce multi-threaded applications which are launched on many devices in a system.

2.3 OpenCL Basics

The OpenCL standard was developed to facilitate the design of portable software-specified systems for heterogeneous parallel computing platforms [4]. Supported platforms currently include multi-core central processing units (CPUs), massively parallel graphics processing units (GPUs), and field-programmable gate arrays (FPGAs). Ideally, system designs specified in OpenCL can be simply recompiled to immediately and efficiently access the available parallel resources of the given parallel architecture.

Figure 2.2 illustrates a general OpenCL architecture. The host device, a CPU, is connected to an OpenCL capable device through the OpenCL device's memory. In contemporary computers the connection is usually made over a PCIe bus. Work is sent by the host to the OpenCL device in the form of instructions and data transfers. Large data input sets are usually transferred into the Global Memory of the OpenCL device using direct memory access (DMA). Host software then initiates execution of software in the OpenCL device, as shown by the high-level flowchart in Figure 2.3. The OpenCL software is partitioned into host code and OpenCL kernel code. The host code, which is written in conventional C/C++ using the OpenCL application programming interface (API), is responsible for setting up and controlling the computations in the OpenCL device. Kernel source code, which is written in OpenCL, is executed on the OpenCL device(s). The typical OpenCL program consists of a host program and OpenCL kernel code for execution on an OpenCL-enabled device. For most devices, the OpenCL kernel code is compiled and built at runtime by the host. An executing instance of a kernel module is called a "work-item". Multiple work-items can be instantiated from the same kernel source code. The number N of work-item instances and the (up to three) resizeable dimensions of the matrix of work-items are declared when the kernel is defined in the host program. In Figure 2.2, the N work-items can be organized in up to three dimensions to suit the data that is to be processed and thus simplify the program design. Work-items are in turn organized into "work-groups", which can also be

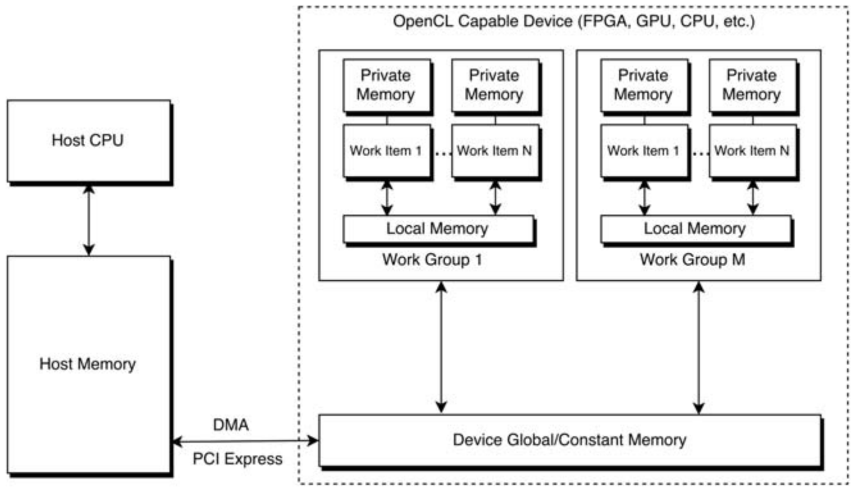


Figure 2.2: OpenCL Architecture

indexed in up to three dimensions to suit the natural organization of the data.

Five basic kinds of data storage can be accessed by an OpenCL program, as illustrated in Figure 2.2: host memory, device global memory, device constant memory, local memory, and private memory. "Host memory" usually corresponds to the RAM of the CPU host. "Global memory" is the relatively large RAM that is directly accessible by the OpenCL device. On the Nallatech 385n FPGA board [12], the global memory is implemented with a high-capacity 25 GB/s synchronous DRAM (SDRAM) that is directly connected to the Altera Stratix V GX A7 FPGA [13]. Global memory can be accessed by all work-items during execution. However, the achieved data bandwidth to global memory in an SDRAM depends on the success with which the significant random-access latency to SDRAM pages can be hidden by containing related global data within SDRAM pages and by minimizing the number of times new pages that need to be opened. "Constant memory", inspired by the constant memory that is provided in GPUs, is another form of global memory that is restricted to read-only access after the OpenCL system has been initialized and device execution has begun. This memory is cached on the OpenCL device to provide quick and repeated access. "Local memory" is intended to model memory that is smaller in capacity than global memory, but that provides faster access. Local memory can be used to cache data that originates in global mem-

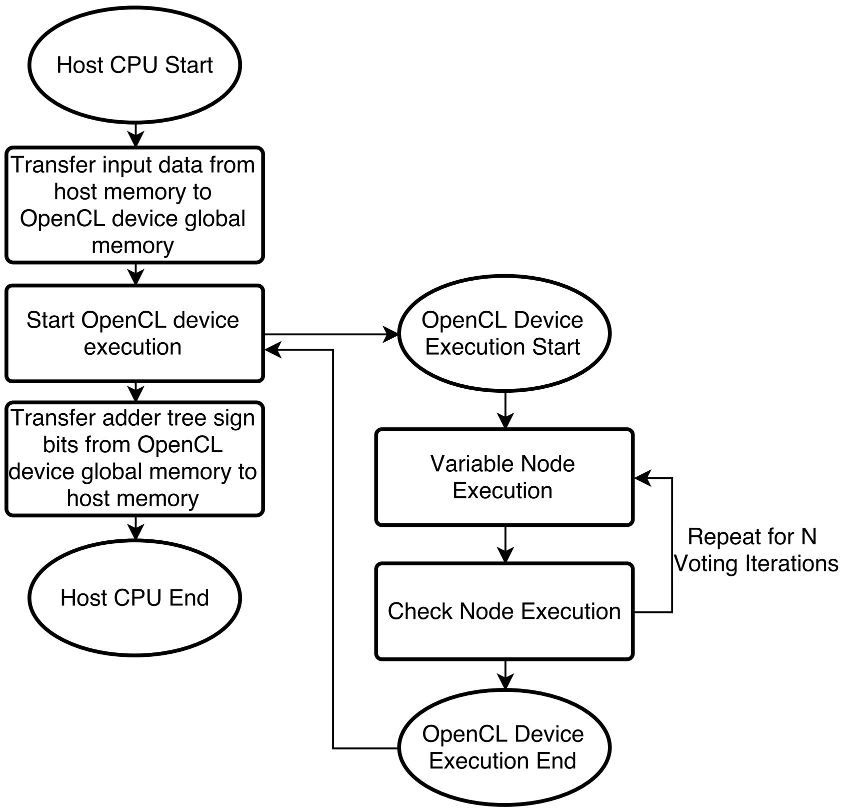

Host CPU Start
Transfer input data from host memory to OpenCL device global memory
Start OpenCL device execution
Transfer adder tree sign bits from OpenCL device global memory to host memory
Host CPU End
OpenCL Device Execution Start
Variable Node Execution
Check Node Execution
Repeat for N Voting Iterations
OpenCL Device Execution End

Figure 2.3: Execution Flow in the Host CPU Controlling Execution Flow in the OpenCL Device

OpenCL Memory Type	FPGA Location
Private Memory	On-Chip Registers
Local Memory	On-Chip Memory Blocks
Global Memory	Off-Chip SDRAM
Constant Memory	Cached On-Chip Memory Blocks

Table 2.1: OpenCL Memory Types on FPGAs

ory, thus effectively speeding up access to that data. In an FPGA-based device, local memory would be implemented on the FPGA chip using block RAMs and/or registers. FPGA synthesis tools can be used to optimize performance by jointly selecting the logic and on-chip memory configurations. Local memory is associated with work-groups. Each work group has exclusive access to its own local memory. In GPU-based devices, "program memory" is implemented in registers first and then in allocated regions of the global memory if needed for each work-item. In FPGA-based devices, however, extra required private memory can be synthesized in the FPGA fabric memory blocks in the same way as local memory, and so both kinds of memory might be zero-latency, depending on the OpenCL compiler.

2.4 OpenCL for FPGAs

Due to the irregular structure and large data bandwidth requirements of the LDPC interleaver network, efficient memory management using the most appropriate OpenCL memory types is essential for optimized decoder performance. In order to easily accommodate longer LDPC codes, the decoder must be designed to simplify scalability with respect to codeword length. Before the decoder can begin, we need to transfer all of the data efficiently from the host to the FPGA device.

As Section 2.3 introduced, there are four main memory types in OpenCL: private memory, local memory, global memory, and constant memory. The placement of the memory types is decided by the device compiler and can be made specific for each OpenCL-enabled device. For the FPGA, memory is located as per Table 2.1.

Just prior to the start of device execution, the host code reads in the pre-compiled kernel code and configures the FPGA hardware appropriately. The host must then transfer the converted channel measurements, the interleaver connection information, and device configuration information down to the device via the PCIe connection using a DMA transfer. The LDPC min-sum decoding algorithm is a suitable problem for implementation onto an FPGA from an OpenCL program. The algorithm contains many forms of exploitable parallelism, but there remains the non-trivial challenge of implementing the interleaver network. The decoder problem allows us to evaluate the performance of the compiler by investigating many different possible architectures.

This thesis investigates using the Altera-Offline-Compiler (AOC) version 15.1 to generate and execute OpenCL-based designs on the Nallatech 385n-A7 FPGA board. Most common OpenCL programs compile the kernel source code at runtime of the host using the *clBuildProgramWithSource* API function. Since compilations for the FPGA take significantly longer compared to other OpenCL-enabled devices, kernels must be compiled with the AOC prior to the execution of the host. The OpenCL program is then built using the *clBuildProgramWithBinrary* API function at runtime. With the long final compilation times, Altera introduced an emulator tool to help with the functional debugging of kernels before fully compiling them. Figure 2.4 shows the typical compilations for an OpenCL design with the AOC. To debug the functionality of an OpenCL kernel for an FPGA, the Altera emulator is generally used first. The compilation flow for emulation, shown on the left of Figure 2.4, only takes a couple of minutes and is activated using the '-march=emulator' compiler flag. There are some setup steps required to compile and execute the emulator of which batch files were developed. The batch files used for this process are attached in Appendices V and W.

After a successful emulation, the design is then typically fully compiled for the FPGA with the Altera profiling system active (using the '–profile' compiler flag). This step allows the developer to view the profile information about the kernel(s) with the Altera profiling GUI. The GUI shows information related to pipeline stall, used memory bandwidth, and execution times for every kernel. Finally, once the developer is satisfied with the emulator and profiler results, the final design can be

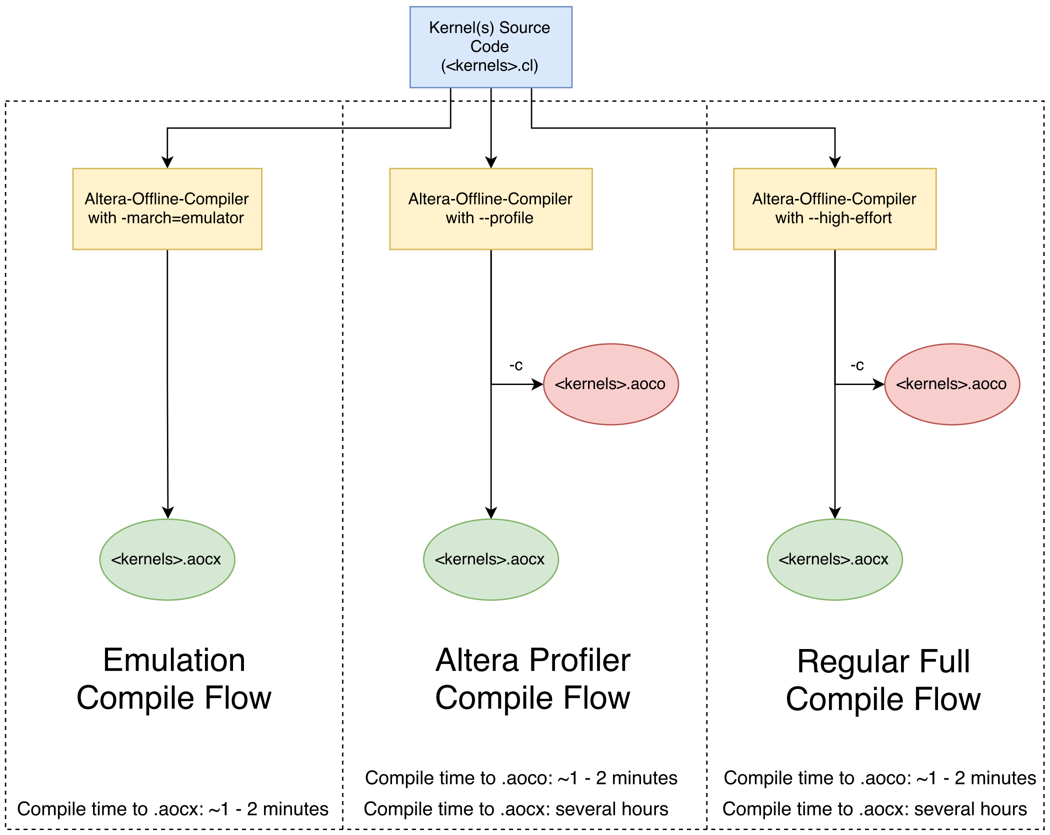

Kernel(s) Source Code (<kernels>.cl)
Altera-Offline-Compiler with -march=emulator
Altera-Offline-Compiler with --profile
Altera-Offline-Compiler with --high-effort
-c
<kernels>.aoco
-c
<kernels>.aoco
<kernels>.aocx
<kernels>.aocx
<kernels>.aocx
Emulation Compile Flow
Altera Profiler Compile Flow
Regular Full Compile Flow
Compile time to .aocx: ~1 - 2 minutes
Compile time to .aoco: ~1 - 2 minutes
Compile time to .aocx: several hours
Compile time to .aoco: ~1 - 2 minutes
Compile time to .aocx: several hours

Figure 2.4: Compile Flow Diagram for Designing with the Altera-Offline-Compiler

compiled using the AOC. By using the built-in OpenCL profiling for the command queues, the execution time on the device(s) can still be measured and reported accurately. Note that for the profiling and final compilation flows, we can choose to stop the emulation after the step (.aoco generation) has completed. In this first stage of compilation, the OpenCL source code is compiled into System Verilog files that are then used by Quartus in the second stage to complete the hardware generation.

2.5 Programming the Host Code

All OpenCL programs require one host and at least one device. The host can be any CPU capable of communicating with an OpenCL device. We use Hextet's multi-core CPU i7 as the host for all OpenCL programs. The host source code must complete a series of "setup" steps in order to launch an OpenCL kernel, as outlined in the specification [4]. These setup steps include creating and identifying the OpenCL device(s), creating the context and program, creating the kernels, and launching the application. In addition to the setup, the host is responsible for the memory management between the device (i.e., transferring data to and from the OpenCL device) both before and after the execution. However, in order to debug kernels with the Altera-Offline-Compiler, we can use the built-in emulator tool. This tool requires some setup on the CPU to compile and run programs. The setup batch source code for the emulator are included in Appendix V. The setup batch source code for executing an emulation are included in Appendix W.

2.6 Low-Density Parity-Check Codes and Decoding Algorithms

The standard LDPC decoding algorithm, also called the belief propagation algorithm, iteratively transforms a codeword vector containing digitized and nonlinearly binned input bit measurements into a binary output codeword that satisfies all of the parity constraints specified by the code [32]. Let $\vec{r}$ be a binary codeword of length $n_{VN} \geq 2$. An (n_{VN},n_{CN}) LDPC code has an associated $n_{CN} \times n_{VN}$ binary parity check matrix H such that for all valid codewords $\vec{r}$, $H\vec{r}^T = 0$ assuming modulo-2 arithmetic. Given a possibly corrupted codeword, an optimal LDPC decoder is one that finds the valid codeword $\vec{r}$ that is the most likely to be the original codeword. The optimal maximum-likelihood decoding algorithm, which exhaustively considers every possible codeword candidate, is not practical because of the huge space of possible codewords. Instead, an efficient belief propagation decoding algorithm is used [32]. The belief propagation decoding algorithm for LDPC block codes are best described as node operations performed on belief messages (i.e., signed values) exchanged between the two types of nodes of a bipartite graph, called the Tanner graph of the LDPC code (e.g. Figure 2.5). The two types of nodes in a Tanner graph are called Variable Nodes (VNs), denoted by '=' symbols in Figure 2.5, and Check Nodes (CNs), denoted by '+' symbols in Figure 2.5. The signed channel measurements are stored in separate VNs (one measured bit per VN). The edges of the Tanner graph form the regular interleaver network, which is determined by the

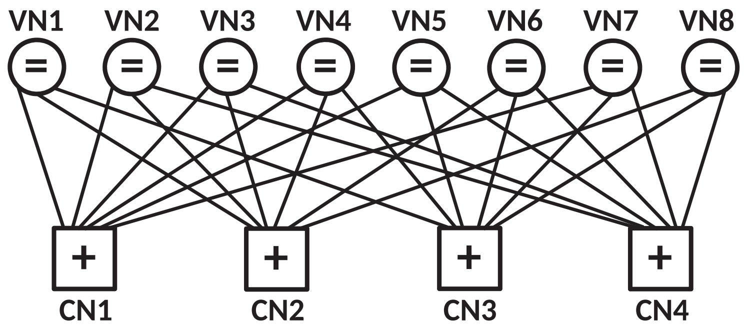


Figure 2.5: Tanner Graph for a length-8 (w_{VN},w_{CN}) = (3,6)-regular LDPC Code

parity check matrix H. For the example Tanner graph in Figure 2.5, the associated parity check matrix H is shown in Equation 2.1. Matrix H is designed to maximize the coding gain of the corresponding LDPC code [32]. Each check node verifies the parity of a parity constraint involving the signs of the values sent to it from the connected VNs for all valid codewords x, $Hx^T = 0$.

$$H = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \tag{2.1}$$

In the first phase of one decoding iteration, messages are computed independently in each VN and then sent, via the interleaver network, to the connected CNs. In Figure 2.5 there are $n_{VN} = 8$ VNs and $n_{CN} = 4$ CNs. This code is called (3,6)-regular because each VN is connected to 3 CNs (i.e., VN degree $w_{VN} = 3$), and each CN is connected to 6 VNs (i.e., CN degree $w_{CN} = 6$). We denote the number of the connections from each VN by w_{VN} and the number of connections from each CN by w_{CN}. In the second phase of each decoding iteration, the CNs independently process their incoming messages (that is, verifying their parity check constraint), and send back adjusted messages to their connected VNs. The messages returned by the CNs to the VNs contain correction factors to the channel measurements that are intended to make the parity constraints more likely to hold true over the codeword. Each VN sums up the one stored nonlinearly binned channel measurement with the $w_{VN} = 3$ CN messages to determine the current consensus belief of the one corresponding bit value in the corrected codeword. Several alternative VN and CN calculations have been proposed [29]. Following common practice, we used the simplified “min-sum” belief propagation algorithm (reviewed later) instead of the theoretically more effective but more computationally complex “sum-product” algorithm [32].

Chapter 3

Regular LDPC Decoder Implementations

3.1 Host Code

As explained in Section 2.3, every OpenCL program requires a host CPU program in order to launch kernels and read/write results. In our benchmark problem, the host is responsible for setting up the OpenCL kernel(s) and preparing the LDPC input data. In our simulation experiments, following standard practice in such simulations, the host program uses a known valid codeword (all-zero) for each LDPC code, and then adds Gaussian noise based on an input Signal-to-Noise-Ratio (SNR) value. Although the codeword is known to the host, it is not known to the decoder. As such, the decoder processes the information in exactly the same manner as if the codeword was not known to the host. All of the information required for the decoder is also generated by the host and passed down to the kernel on the device. This includes the interleaver connections, the number of variable/check nodes, and their number of inputs. By reading in the LDPC code (given in the standard A-List format [24]) file, the host program determines the interleaver network and passes the information down to the OpenCL kernel(s). The host source code for the regular decoder is attached in Appendix A.

A common representation of LDPC codes is in the A-List format [24]. This format describes all aspects of the code starting from the number of nodes (both variable and check nodes) as well as outlining the connections between them. The host code for all the decoders takes in a LDPC code specification using this format and creates a parity H matrix based off of the information. The type of LDPC codes used in this chapter are called block codes (or block-LDPCs). These codes have a regular structure in which entire blocks of the code are decoded in comparison to convolutional LDPC codes which are a continuous stream of data. Since we are only really concerned with the throughput performance of the decoders and not the error-correcting performance of each LDPC code itself, the exact structure of the code (i.e., the connections between nodes) does not matter as long as it satisfies the (3,6)-regular property. Because of this, to test the designs in this chapter we use a series of randomly generated (random connections between VNs and CNs subject to the available degrees of each node) LDPC codes varying in length from 16 VNs to 2048 VNs. Although the connections between each VN and CN are randomly generated, each VN is connected to 3 CNs while each CN is connected to 6 VNs. This way a larger variety of codes can be tested to determine throughput patterns. In order to generate these random LDPC codes, a C# program was developed that takes in the required code length and generates an A-List output file for the randomly generated LDPC code. The source code for this program is attached in Appendix B.

3.2 LDPC Decoding Algorithm Optimizations

Given the OpenCL language and the LDPC min-sum decoding algorithm, we will start by generating a design that should produce reasonably efficient results on the FPGA. The typical OpenCL model contains a SIMD-structured program with a single sequence of instructions acting on multiple sets of data that reside in the device global memory. This software architecture performs especially well on GPU architectures due to the SIMD hardware architectures of graphics processors. A similar approach can be taken to the problem of producing an efficient and optimized LDPC decoder. Although the irregular structure of the interleaver network does not fit the simple SIMD architecture, the decoding operations for the VNs and CNs can still be written in SIMD operations. We know that GPUs perform well with a SIMD structure, but we need to know what is better for the FPGA.

3.2.1 SIMD Vectorization

In a fully SIMD program, the most efficient code ensures that all operations are ‘vector friendly’. Most normal operations such as addition, subtraction, etc, can be altered such that they process multiple data simultaneously. However, if-then-else chains create performance bottlenecks by introducing different branches for the threads to execute. This ultimately results in a larger amount of hardware generated for all instruction paths. Synchronization hardware must then be used at the end of the if-else chain. If the paths are unbalanced, the resulting penalty can be significant.

To avoid this situation we can execute all possible instruction paths for all of the threads of the SIMD program and then multiplex in the correct data in the final calculation. This ensures thread synchronization and functional correctness. An example of this approach is shown in Algorithm 1. The vectored == operation in line 8 compares the two vector inputs $someValue$ and $zeroVector$ for equality. In this case, the resulting *conditional* vector is $(1, 0, 1, 0)$. A masking equation is then used on the conditional array to store the correct value (either 1 if the value in someValue was 1 or -1 if the value in someValue was 0) in the $result$ vector. These bitwise operations are easy to implement in hardware and allow for each index in the vector to have their own result. In this case the result in line 9 is $(-1, 1, -1, 1)$

Algorithm 1 Vectored SIMD If-Else

```
1: procedure Vectored SIMD If-Else
2:     cl_int4 $someValue = (cl\_int4)(0, 1, 0, 1)$
3:     cl_int4 $result$
4:     cl_int4 $zeroVector = (cl\_int4)(0, 0, 0, 0)$
5:     cl_int4 $oneVector = (cl\_int4)(1, 1, 1, 1)$
6:     cl_int4 $negOneVector = (cl\_int4)(-1, -1, -1, -1)$
7:
8:     cl_int4 $conditional = (someValue == zeroVector)$
9:     $result = (conditional \& oneVector) | (\sim conditional \& negOneVector)$
10: end procedure
```

An experiment was constructed in order to determine the most appropriate approach for implementing this level of parallelism on the FPGA platform. The experiment tested branching/unpacking a vector in order to execute multiple if-else

Table 3.1: Conditional Array Values And Resulting Vector

1	1	1	1	1	0
1	1	1	1	1	0
1	1	1	1	1	0
...	...	...	...	...	...

$\longrightarrow$

1	1	1	1	1	-1
1	1	1	1	1	-1
1	1	1	1	1	-1
...	...	...	...	...	...

chains or using the approach of executing all possible instruction paths. The basic outline of the program is a simple iterative walk-through of a two-dimensional array with the result from each component determined by the number present within the first conditional array. The conditional array (the array walked-through) carries an initial value of either 1 or 0. A value of 1 forces the code to use the first block of the if-else and the 0 forces it to use the latter. The values initialized into the conditional array are structured to create a pattern similar to that experienced during the distribution of the minimum and second minimum of a check node (descibed later).

As shown in Table 3.1, the pattern within the conditional array will cause the first path of the if-else to be executed 5 out of the 6 iterations. Two versions of the code were compiled: the vectorized version and the branching version. The branching version simply takes apart the vector, branches for each entry, and stores the results. The source code for the branching version is attached in Appendix C. The vectorized version, as illustrated in Algorithm 1, uses a masking equation to properly store the answer for each portion of the vector. The source code for the vectorized version is attached in Appendix D. During compilation, an iteration loop (of 1,000,000 times) was introduced to force the data through the critical section repeatedly. These iterations should ultimately place the results of the performance to be based on either the branching or vectoring instead of any overhead introduced by the experiment. The results shown in Table 3.2 show that both kernels had similar execution times with the branch-less vectored code winning by 14 ms (percentage difference of around 0.8

The same experiment was then done on the NVIDIA GTX 690 GPU in Hextet with the same OpenCL source code. The GPU architecture is heavily optimized to

Table 3.2: Branching vs. Vectored Performance Comparison

Kernel	Execution Time	Clock Frequency
Branching	1.747 seconds	293.9 MHz
Vectored	1.733 seconds	296.2 MHz

execute SIMD efficiently and with high performance. As such, we should expect the vectored version on the GPU to greatly outperform the branching. The results from the GPU Execution, shown in Table 3.3, display a significant increase in execution time for the branching code by a factor of $17.9925(branching)/0.00236(vectored) = 8996.25$ times. Table 3.4 shows that the vectored code executed 8996.25 times faster than the branching version on the GPU and that the vectored code executed 1.01 times faster than the branching code on the FPGA. By comparing these results with those from the FPGA execution in Table 3.2, the GPU significantly outperformed the FPGA with the vectorized code by executing $1.733/0.00236 = 866.5$ times faster; however, the FPGA greatly outperformed the GPU for the branching case by executing $17.9925/1.747 = 10.3$ times faster. These results prove that although the GPU outperforms the FPGA for SIMD programs, the FPGA will is better suited to programs with a more MIMD structure. They also show, however, that even the algorithms intended for FPGA targets should strive for a SIMD structure whenever possible.

Device	Kernel	Execution Time (s)
GPU	Branching	17.9925
GPU	Vectored	0.00236

Table 3.3: NVIDIA GPU Vector and Branch Performance

Device	Branching / Vectored Ratio
FPGA	1.01
GPU	8996.25

Table 3.4: Branching/Vectored Execution Time Ratio for the FPGA and GPU

3.2.2 Nodes

We will now consider suitable algorithms for each of the VNs and CNs in OpenCL.

Variable Nodes

For each VN output, the node needs to send back to the connected CN a recomputed sum of the incoming messages from those CNs plus the channel measurement. However, to make the calculations more SIMD and to have more efficient hardware, the VN computes the overall total and then produces the outgoing message for each CN by subtracting the message received from the respective CN. The pseudo-code shown in Algorithm 2 shows this approach. Each VN simply adds all of its inputs (the incoming messages plus the initial channel measurement) giving an overall total. The corresponding input message is then subtracted from this total to get each output message.

This algorithm is illustrated in Figure 3.1 for a VN with three inputs and three outputs. The VN datapath performs the "sum" part of the min-sum algorithm. In particular, it includes a two's complement adder tree (to sum up the one initial channel measurement encoded previously on the host as a finite-precision signed Log-Likelihood Ratio (LLR) and the several signed incoming messages) followed by an array of parallel (SIMD) subtractors. These subtractors (as mentioned above) subtract the value of each incoming message from a CN to obtain the corresponding outgoing message to that same CN. The incoming messages in Figure 3.1 are denoted by **In1**, **In2**, and **In3**. The outgoing messages **Out1**, **Out2**, and **Out3** are obtained by subtracting **In1**, **In2**, and **In3**, respectively, from the overall sum, i.e., the output of the adder tree. The subtraction is done following standard practice to avoid resending information back to the same connected check node and thus to avoid unwanted interactions in the message information computed in the VNs and CNs in successive decoding iterations. The output messages from the VNs are converted to sign-and-magnitude format to simplify the implementation of the CNs. For example, if a specific VN has an initial channel measurement of $ch = 2$ and input messages $(In_1, In_2, In_3) = (6, -4, 3)$ from the connected CNs, the overall total will be first calculated as $total = ch + In_1 + In_2 + In_3 = 7$. The individual inputs are then subtracted from the overall total as $Out_i = total - In_i$ producing $(Out_1, Out_2, Out_3) = (1, 11, 4)$ in this example.

Algorithm 2 Variable Node Min-Sum Algorithm

```
1: procedure VARIABLE NODE EXECUTION
2:     $total = 0$
3:     for each input $i \in N$,where $N = \#of inputs$ do
4:         $total = add\_sat(total, i)$
5:     end for
6:     for each input $i \in N$ do
7:         $current = total - valueOf(input\ i)$
8:         Check for overflow and saturate against $MAX\_NEG\_LLR$
9:         Find the location for the outgoing message $input\_index$
10:        Convert $current$ to sign & magnitude
11:        Write the value of $current$ to its check node at $input\_index$ index
12:    end for
13: end procedure
```

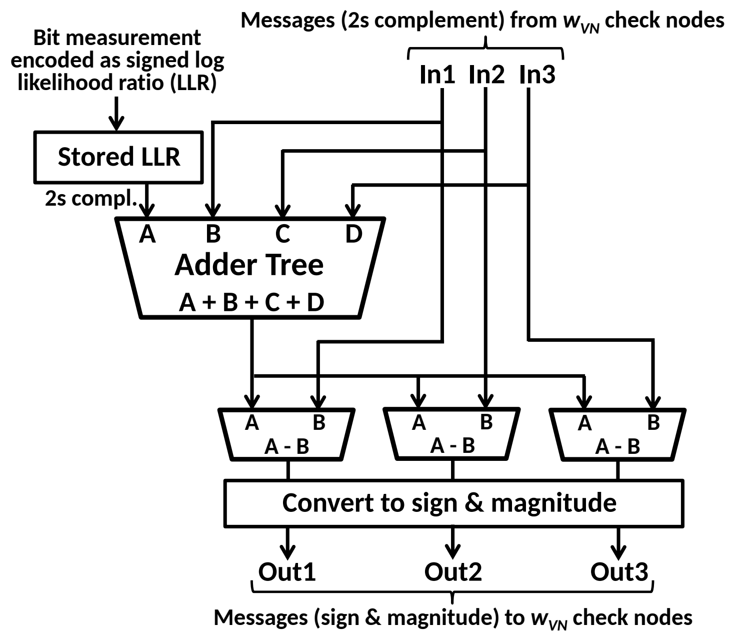

Messages (2s complement) from w_{VN} check nodes
Bit measurement encoded as signed log likelihood ratio (LLR)
In1 In2 In3
Stored LLR
2s compl.
A B C D
Adder Tree
A + B + C + D
A B
A - B
A B
A - B
A B
A - B
Convert to sign & magnitude
Out1 Out2 Out3
Messages (sign & magnitude) to w_{VN} check nodes

Figure 3.1: Variable Node Architecture for a (w_{VN},w_{CN}) = (3,6)-regular LDPC code

Check Nodes

The check node computation completes the "min" portion of the min-sum decoding algorithm. It is more complex to implement than the variable node computation as it involves finding both the first and second minimums magnitudes of the input values. The input message magnitude that is found to be the first minimum will be given the second minimum magnitude message as its output. All other inputs will be given the first minimum magnitude as their corresponding output message. The check node also attempts to correct the parity of the check constraints that involve the message from the connected VNs. Essentially the check nodes compute correction factors that are sent back through the interleaver to the variable nodes. The architecture illustrated in Figure 3.2 and the pseudo-code in Algorithm 3 define the CN computation. For a (3,6)-regular LDPC code (as shown in Figure 3.2), each CN has $w_{CN} = 6$ incoming messages. The incoming sign bits and magnitudes are processed separately in parallel. All of the sign bits are *XOR*ed together in a parity tree to give an overall CN parity, which is 0 if the corresponding CN constraint is satisfied and is 1 otherwise. The sign of each outgoing message is formed by taking the XOR of the overall parity with the sign of the corresponding incoming message. This has the effect of flipping all of the message signs if the CN parity check was not satisfied; otherwise, the message signs of the outgoing messages are the same as those of the incoming messages. The CN processes the magnitudes of the incoming messages by determining the first and second minimum magnitudes. For example, if the incoming messages have magnitudes [4 5 1 10 2 6], the third magnitude (i.e., 1) will be found to be the first minimum, and the fifth magnitude (i.e., 2) will be found to be the second minimum. The magnitude of each outgoing message is the minimum of the magnitudes of all incoming messages, neglecting the magnitude of the corresponding one incoming message. Thus for the example incoming message magnitudes, the corresponding outgoing messages will have magnitudes [1 1 2 1 1 1]. As a final processing step in the CN, the outgoing signed messages are converted into two's complement form to simplify the implementation of the VNs.

Algorithm 3 Check Node Min-Sum Algorithm

```
1:  procedure CHECK NODE EXECUTION
2:      Find the absolute value first minimum from the $N$ inputs
3:      Find the absolute value second minimum from the $N$ inputs
4:      Calculate the parity across all incoming messages
5:      Find the location for the outgoing message $input\_index$
6:      for each input $i \in N$ do
7:          $current\_parity = parity \oplus$(sign bit of input i)
8:          if magnitude of $input\ i = first\_minimum$ then
9:              $output = current\_parity | second\_minimum$
10:         else
11:             $output = current\_parity | first\_minimum$
12:         end if
13:         Convert output to 2's complement representation
14:         Write output to the variable node at $input\_index$ index
15:     end for
16: end procedure
```

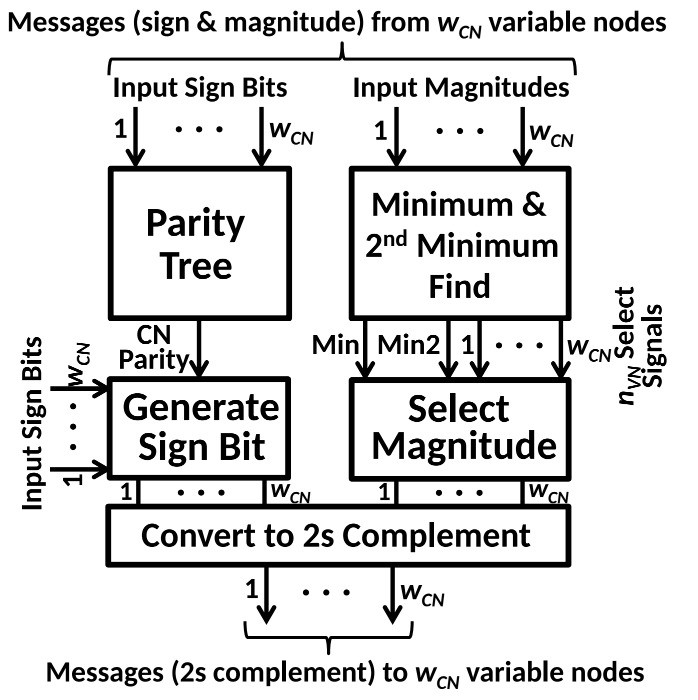


Figure 3.2: Check Node Architecture for a $(w_{VN},w_{CN}) = (3,6)$-regular LDPC code

3.2.3 Number Representation

Log Likelihood Ratio (LLR) Representation

For the incoming initial channel measurements it is natural and convenient to use a floating-point representation in simulation; however, as introduced in the algorithms for each node and following common practice, we use a fixed-precision binary representation of the log likelihood ratio (LLR), where $LLR(x) = \ln \frac{P(x=1)}{P(x=0)}$, for the "belief" messages in our decoder. For example, in a simple binary representation approach, the channel contains floating-point values from -1 to +1 (where -1 represents binary 1 and +1 represents binary 0). The -1 to +1 representation is known as Binary Phase-Shift Keying (BPSK) [28]. As the FPGA has a limited number of floating-point arithmetic units, integer operations are preferred as they take up less space and perform efficiently . In order to accurately represent a floating point number, we use the log likelihood ratio format with saturated linear binning. Although the fixed-precision binary LLRs are less accurate than using floating point throughout, the errors introduced by the LLR operation will provide similar decoder performance as long as the LLRs are sufficiently wide. The standard practice is to use a bit width of at least 6 for the LLRs; however, some reported designs use a minimum of 4 bits [26]. That is, the errors introduced will be corrected the LDPC decoding algorithm itself. The saturated linear binning distributes the bins equally from $-MaxValue$ to $+MaxValue$. Any values that fall outside of that range are saturated to $+/-MaxValue$. This format stores the value in a signed fixed-point integer format allowing for more efficient arithmetic to be synthesized.

Depending on the number of bits available for storing LLRs, we quantize the floating-point values into a fixed number of bins. Through experimentation with a variety of bit widths, an appropriate number of bits for each LLR was determined to be around 6-7 bits to ensure satisfactory decoder performance. To simplify the programming, we used an LLR bit width of 8 bits as they can be stored in exactly 1 byte of data (the *cl_char* OpenCL datatype width). As this is a signed integer representation, 1 bit is used for the sign bit and the remaining 7 bits are exponentially binned amongst the -1 to +1 floating point values. This means that there are $2^8 = 256$ bins centered around 0. Because we require the use of both positive and negative numbers, our available bins with 8 bits are from -127 to +127. Based on the BPSK configuration with floating point values, the respective bin cutoff points are given in Table 3.5. The values in Table 3.5 are calculated by the formula $MaxValue = (1/BINS\_PER\_ONE\_STEP)$ and $BinCutoff = i/MaxValue$, where $0 < i < (MAX\_LLR + 1)$, $MAX\_LLR = 2^{(N\_BITS)} - 1 = 127$. Only the magnitude is binned in the LLR conversion process: the sign value remains unaffected.

Sign & Magnitude vs 2's Complement

The operations within the variable node involve simple additions and subtractions of both positive and negative numbers. Thus the simplest way to store the data for this node is in 2's complement format. This, however, is not the case for the check node algorithm. The check node algorithm operates separately on the magnitudes and sign bits for each message. Thus, the most efficient fit for the check node is to store the data in the sign & magnitude representation. This allows the algorithm to use masking to obtain the sign or magnitude of each message.

Since both the variable node and check node implementations are simpler and more efficient with two different number representations, we need a simple way to convert between the message formats. The simplest way is to use the algorithm described in Algorithm 4. Note that in this example we are dealing with 16-bit numbers; however, the masks can be simply adjusted to accommodate any length of vector. Also note that for both representations, positive numbers have the same form and only negative numbers require conversion. When we need to convert a negative number we simply flip each bit in the number (as shown by the exclusive-or in line 5) and then add '1'. This algorithm works for converting numbers from 2's complement to sign & magnitude and vice versa.

1	0.244094	32	7.811024	63	15.377954	94	22.944883
2	0.488189	33	8.055119	64	15.622047	95	23.188976
3	0.732283	34	8.299212	65	15.866142	96	23.433071
4	0.976378	35	8.543307	66	16.110237	97	23.677166
5	1.220472	36	8.787402	67	16.354332	98	23.921261
6	1.464567	37	9.031496	68	16.598425	99	24.165356
7	1.708661	38	9.275591	69	16.842520	100	24.409449
8	1.952756	39	9.519685	70	17.086615	101	24.653543
9	2.196851	40	9.763780	71	17.330709	102	24.897638
10	2.440945	41	10.007874	72	17.574804	103	25.141733
11	2.685040	42	10.251968	73	17.818897	104	25.385828
12	2.929134	43	10.496063	74	18.062992	105	25.629921
13	3.173229	44	10.740158	75	18.307087	106	25.874016
14	3.417323	45	10.984252	76	18.551182	107	26.118111
15	3.661417	46	11.228347	77	18.795277	108	26.362206
16	3.905512	47	11.472442	78	19.039370	109	26.606300
17	4.149606	48	11.716536	79	19.283464	110	26.850395
18	4.393701	49	11.960630	80	19.527559	111	27.094488
19	4.637795	50	12.204724	81	19.771654	112	27.338583
20	4.881890	51	12.448819	82	20.015749	113	27.582678
21	5.125984	52	12.692914	83	20.259844	114	27.826773
22	5.370079	53	12.937008	84	20.503937	115	28.070868
23	5.614173	54	13.181103	85	20.748032	116	28.314960
24	5.858268	55	13.425198	86	20.992126	117	28.559055
25	6.102362	56	13.669291	87	21.236221	118	28.803150
26	6.346457	57	13.913386	88	21.480316	119	29.047245
27	6.590551	58	14.157480	89	21.724409	120	29.291340
28	6.834646	59	14.401575	90	21.968504	121	29.535433
29	7.078740	60	14.645670	91	22.212599	122	29.779528
30	7.322835	61	14.889764	92	22.456694	123	30.023623
31	7.566929	62	15.133859	93	22.700788	124	30.267717
32	7.811024	63	15.377954	94	22.944883	125	30.511812
33	8.055119	64	15.622047	95	23.188976	126	30.755907
34	8.299212	65	15.866142	96	23.433071	127	31.000000

Table 3.5: Saturated LLR Bin Cutoff Points

Algorithm 4 Conversion Between 2's Complement And Sign & Magnitude

```
1: procedure Conversion($value$)
2:     if $value > 0$ then
3:         return $value$
4:     else
5:         $value = value \oplus (0x8FFF)$ // assume 16-bit value
6:         $value = value + 1$
7:         return $value$
8:     end if
9: end procedure
```

Vectorization

The regular LDPC decoder problem allows us to exploit many different forms of parallelism, as shown in Table 3.6. Because of the irregularity of the interleaver, the problem is a combination of SIMD and MIMD parallelism. Although the interleaver network has an irregular structure, the LDPC decoding algorithm contains much SIMD structure within the VN and CN portions independently, as explained in Section 3.2. All of the VN operations are relatively simple with the same sequence of additions and subtractions executed multiple times but with different data. We can exploit these attributes by vectorizing our design to effectively multiply the throughput of the implementation. There are two different ways we can achieve vectorization: through the vectored OpenCL data types and through the *num_simd_work_items* AOC kernel attribute. Both of these ways increase the width of the data path allowing more throughput. We will discuss the latter strategy later on in Section 3.5. Using the vectored OpenCL data types, we are able to increase the number of codewords that we can decode simultaneously at the expense of extra required hardware on the FPGA and increased latency for memory transfers.

As previously discussed in Section 3.2.3, we use 8-bit (char type) integers to represent the LLR values of the min-sum algorithm. We can simply redefine these to use the OpenCL vectored char type cl_char$n$, where $n = 1$, 2, 4, 8, or 16 represents the vector width. For example, a variable of type cl_char16 contains 16 individual char values.

SIMD vector data types are essential components when programming for massively parallel GPU platforms due to their compatibility with SIMD hardware architecture. For example, such data types are used frequently in image processing systems. Many SIMD programming techniques have been developed to optimize parallel instructions operating on vector data types. By vectorizing our design to decode n codewords simultaneously, we need to alter our execution path to handle the data. In order to create a SIMD-friendly execution path, we need to vectorize all operations and remove all conditional branching statements. Conditional branches are undesirable because they require the generation of all possible instruction paths. The FPGA can minimize the performance loss associated with conditional branching, however, at the cost of extra hardware. To avoid these situations, all possible instruction paths are executed for all of the vector indexes and the correct results for each index are multiplexed out following the final calculation. This ensure that all components of the vector execute the same operations simultaneously while ensuring that each component calculation only selects and stores their correct result. This ensures the hardware is efficiently used as all operations are simply compiled with a larger data width on the FPGA (e.g., an 8-bit adder is transformed to a 32-bit adder when using cl_char4 instead of cl_char).

Vectorizing the VN operation is simple because of the supported vector types in OpenCL; however, more extensive changes had to be made to the CN design for the minimum calculations. In order to vectorize the first and second minimum calculations, a two-stage tournament structure was implemented. All CN input magnitudes enter the first tournament tree to determine the first minimum magnitude. Adjacent magnitudes are compared using a bitwise comparison vector operation. The lesser magnitude at each comparison continues forward to the next round, while the larger magnitude is sent to the second minimum tournament tree. This process continues until the first and second minimum magnitudes over all inputs are determined.

Parallelism	**Evaluated Kernels**	**Description**
Loop Pipelining	Design A[1]only	Launch successive iterations to pipeline loop hardware running the VNs before the CNs for each iteration.
Vectorization	Designs A,B,C	Vectorize all operations and widen the LLR vectors to decode multiple codewords simultaneously.
Hardware Pipelining	Designs B[2],C[3]	Launch multiple work-items onto a single hardware pipeline.
Pipeline Replication	Design C only	Replicate the hardware pipeline to create multiple execution lanes to effectively increase the degree of vectorization.

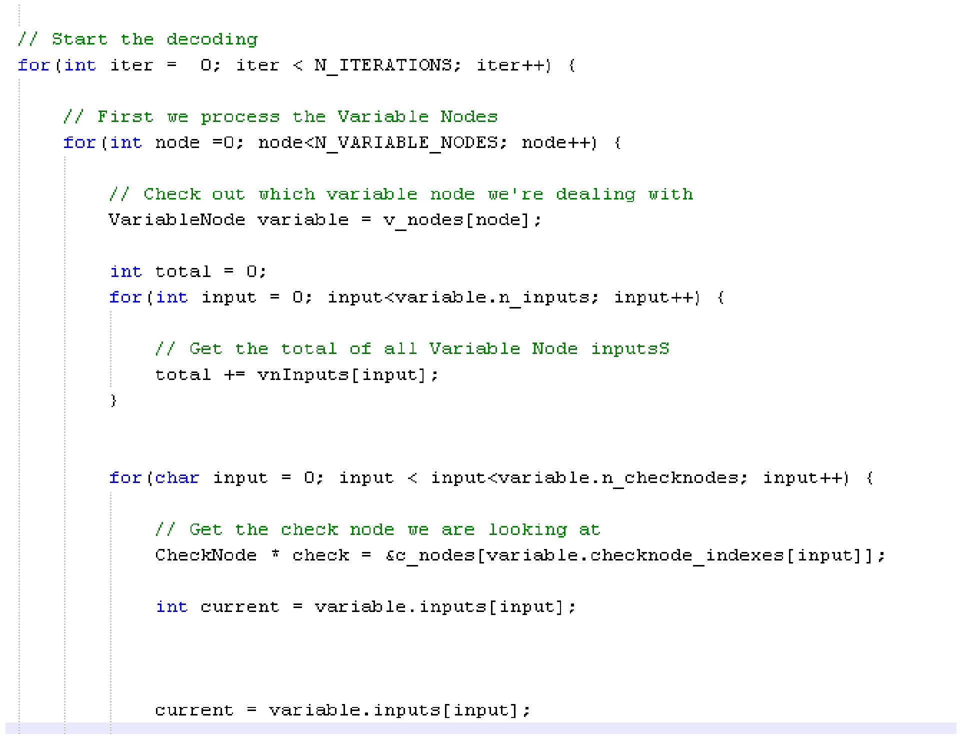
Table 3.6: Types of Parallelism

3.3 Single Work-Item, Single Kernel (Design A)

In [9] Altera recommends forming an initial design as a single work-item, single kernel OpenCL program. The advantage of this approach is that the kernel code is essentially regular C code. Most of the same source code that was functionally tested in a C program can be re-used when migrating the software to OpenCL. Figures 3.3 and 3.4 are samples of source code taken from our original C simulation program and the kernel for Design A in OpenCl, respectively. As the Figures show, the source codes have many similarities. Both source codes iterate with *for* loops for the number of iterations and the nodes within. The variable node algorithm then begins by calculating the total of all of its inputs for both source code samples.

By not partitioning the work into a variable number of work-items and work-groups, the programmer doesn't need to be concerned with fitting the memory model of the program across multiple work-items. In this software architecture, the compiler is given more flexibility to perform hardware optimization as it only needs to generate an efficient pipeline for the execution of a single work-item. Performance

[1]Design A: Single Work-Item, Single Kernel
[2]Design B: Original Local Memory
[3]Design C: Replicated Local Memory



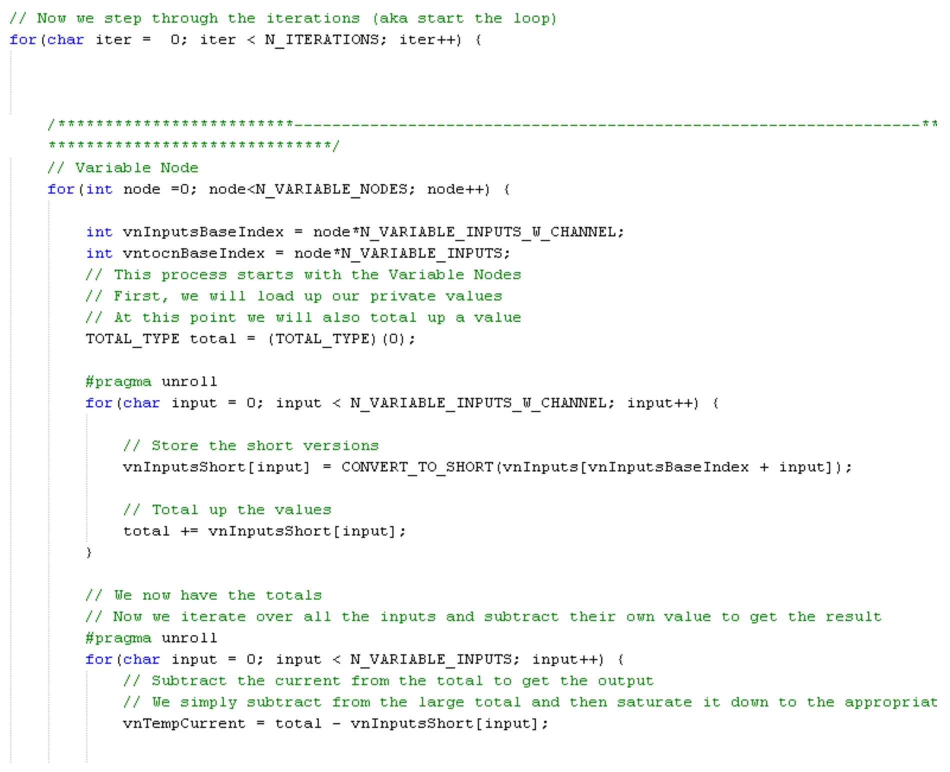
```
// Start the decoding
for(int iter =  0; iter < N_ITERATIONS; iter++) {

    // First we process the Variable Nodes
    for(int node =0; node<N_VARIABLE_NODES; node++) {

        // Check out which variable node we're dealing with
        VariableNode variable = v_nodes[node];

        int total = 0;
        for(int input = 0; input<variable.n_inputs; input++) {

            // Get the total of all Variable Node inputsS
            total += vnInputs[input];
        }


        for(char input = 0; input < input<variable.n_checknodes; input++) {

            // Get the check node we are looking at
            CheckNode * check = &c_nodes[variable.checknode_indexes[input]];

            int current = variable.inputs[input];



            current = variable.inputs[input];
```

Figure 3.3: Sample of LDPC C Simulation Source Code

is then gained through loop pipelining that is automatically generated and inserted into the design by the compiler, as shown in Table 3.6. In our decoder problem, each pass through the pipeline corresponds to one decoding iteration for one codeword.

Another advantage of initially implementing a single work-item, single kernel design is the optimization report generated for such a design by the Altera Offline Compiler [9]. This report gives the developer pipeline optimization results generated by the compiler. The compiler details whether or not it was able to infer pipeline execution for every loop in the implementation. Within these loops, the compiler then details the stalls and critical paths by revealing when each successive loop iteration is launched. Pipeline stalls can arise from data dependencies, memory dependencies, and even loop-iteration dependencies. This information can be used to develop improved designs that have fewer pipeline stalls.

As a starting point, we designed a single work-item, single-kernel LDPC decoder implementation with the AOC for OpenCL, which we call decoder Design A. The resulting kernel was indeed similar to the C simulation code using the algorithms



```
// Now we step through the iterations (aka start the loop)
for(char iter =  0; iter < N_ITERATIONS; iter++) {



    /*************************-----------------------------------------------------------------**
    *****************************/
    // Variable Node
    for(int node =0; node<N_VARIABLE_NODES; node++) {

        int vnInputsBaseIndex = node*N_VARIABLE_INPUTS_W_CHANNEL;
        int vntocnBaseIndex = node*N_VARIABLE_INPUTS;
        // This process starts with the Variable Nodes
        // First, we will load up our private values
        // At this point we will also total up a value
        TOTAL_TYPE total = (TOTAL_TYPE)(0);

        #pragma unroll
        for(char input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {

            // Store the short versions
            vnInputsShort[input] = CONVERT_TO_SHORT(vnInputs[vnInputsBaseIndex + input]);

            // Total up the values
            total += vnInputsShort[input];
        }

        // We now have the totals
        // Now we iterate over all the inputs and subtract their own value to get the result
        #pragma unroll
        for(char input = 0; input < N_VARIABLE_INPUTS; input++) {
            // Subtract the current from the total to get the output
            // We simply subtract from the large total and then saturate it down to the appropriat
            vnTempCurrent = total - vnInputsShort[input];
```

Figure 3.4: Sample of Design A in OpenCL

discussed in Section 3.2. The host begins the decoding process for each codeword by sending the appropriate codeword data and buffer pointers to the kernel via input parameters. By using the OpenCL memory buffers, the initial channel measurements for the codeword are read into the kernel for all VNs via the on-board SDRAM. Along with the pointers into the SDRAM, the host also passes a pointer for the resulting data, the interleaver connection information, and relevant constant values used in the computation (i.e., maximum values and bit masks). All other constant information, such as the LLR width and codeword size, are predefined in the kernel using C compiler directives. As a result, the compiler is given information pertaining to the size of pointers and data widths at compile time, allowing it more flexibility to make optimizations.

As shown in Figure 3.5, after loading the initial channel measurements (e.g., the noisy codeword bits) into registers in the VNs, the single work-item begins execution by looping over the n_{VN} VNs in the code using the VN algorithm discussed in Section 2.6. The same single work-item then loops over all of the n_{CN} CNs as per the second phase of the decoding iteration. This two-phase sequence is then repeated depending on the number of iterations (e.g., 32) used by decoder. Finally, the decoded codeword is transferred to the host from the VNs.

Since the flow of each VN is completely unrolled for one iteration and mapped to a single pipeline, the compiler is able to launch successive VN calculations in pipelined fashion in a shared hardware datapath in this implementation. At the end of each VN computation, the output data needs to be sent to the appropriate CNs according to the interleaver network specified in the LDPC code. Since these connections are constant for each codeword over all the iterations, the connections need only be loaded once by the host into global or constant memory.

The device source code (kernel source code) for this design is attached in Appendix F.

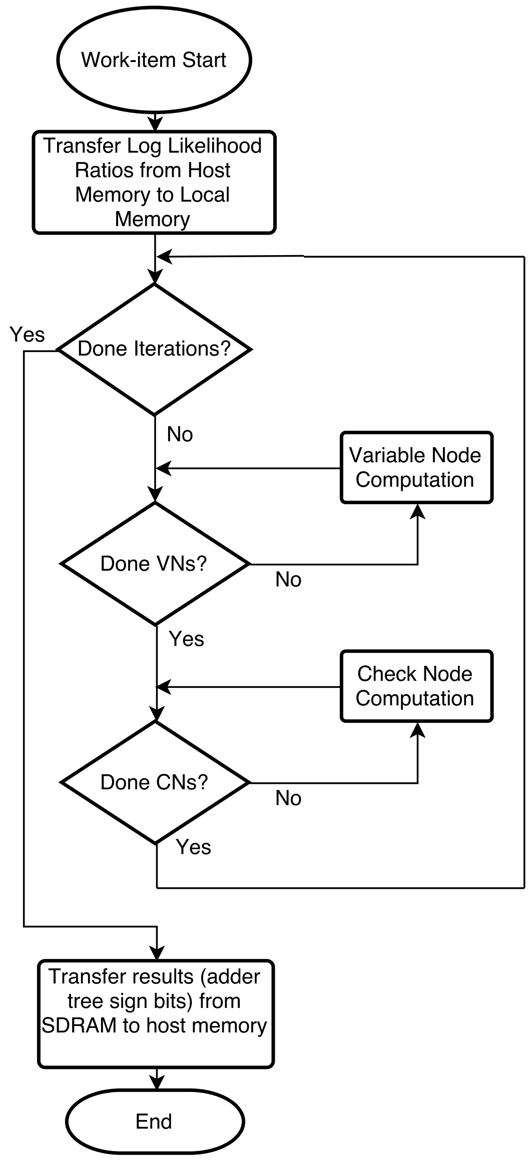
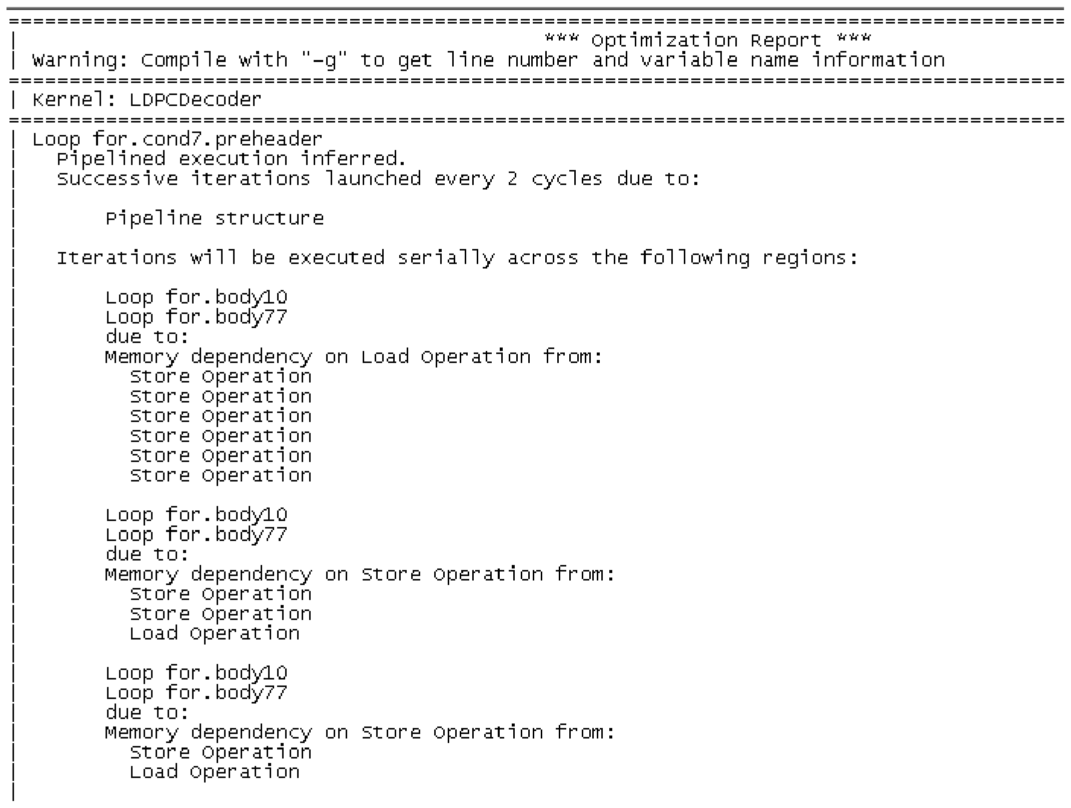

Work-item Start
Transfer Log Likelihood Ratios from Host Memory to Local Memory
Done Iterations?
Yes
No
Variable Node Computation
Done VNs?
No
Yes
Check Node Computation
Done CNs?
No
Yes
Transfer results (adder tree sign bits) from SDRAM to host memory
End

Figure 3.5: Flow Diagram of the Single Work-Item, Single Kernel Design A

3.3.1 Memory Model

As mentioned in Section 2.4, memory management in OpenCL is one of the more important design decisions when maximizing kernel efficiency and performance. In order to achieve the most efficient decoder design for the single work-item, single kernel architecture (Design A), we need to organize the interleaver data appropriately. Since all of the data initially arrives from the host into the global memory (SDRAM), our first implementation simply used buffers within the global memory to provide storage for each VN and CN. Each VN and each CN is allocated an array within the global memory to store their respective inputs. To implement the interleaver, we pass a lookup table from the host to the kernel that contains an index for the connection from each node. For example, each VN indexes into a VN-to-CN interleaver connection array. Every entry in this connection array is an index into the CN inputs storage array. This way, every input for each VN is able to look up their index into the CN inputs storage array. Since all of the data is pre-calculated, the index locations simply need to be looked up. If the code needs to be changed, only the look-up tables and loop counters need to be changed.

As we know from Section 2.4, there are four different memory types available to the programmer in the OpenCL device. Although global memory (implemented on the external SDRAM) is much larger, its random access latency is relatively long compared to the other memory types. With the input arrays located in the global memory, the optimization report from the compiler details that successive loop iterations for both the VN and CN are not able to launch at every clock cycle due to the memory dependencies from the increased latency of the global memory loads and stores. Figure 3.6 shows a screenshot image of the Altera optimization report [9] which reveals that successive loop iterations are launched every two cycles. As shown in the report (also attached in Appendix E), the iterations are slowed down by the memory dependencies from accessing the external SDRAM global memory. Therefore, to increase the throughput of the design, we decided to implement the interleaver network in local memory. Implementing the interleaver in local memory



```
================================================================================
|                                         *** Optimization Report ***
| Warning: Compile with "-g" to get line number and variable name information
================================================================================
| Kernel: LDPCDecoder
================================================================================
| Loop for.cond7.preheader
|   Pipelined execution inferred.
|   Successive iterations launched every 2 cycles due to:
|
|       Pipeline structure
|
|   Iterations will be executed serially across the following regions:
|
|       Loop for.body10
|       Loop for.body77
|       due to:
|       Memory dependency on Load Operation from:
|         Store Operation
|         Store Operation
|         Store Operation
|         Store Operation
|         Store Operation
|         Store Operation
|
|       Loop for.body10
|       Loop for.body77
|       due to:
|       Memory dependency on Store Operation from:
|         Store Operation
|         Store Operation
|         Load Operation
|
|       Loop for.body10
|       Loop for.body77
|       due to:
|       Memory dependency on Store Operation from:
|         Store Operation
|         Load Operation
|
```

Figure 3.6: Altera Optimization Report for Design A

involves moving the VN and CN input storage arrays into local memory. However according to the Altera-Offline-Compiler for OpenCL, the host is only able to access the global memory. Because of this restriction, the channel measurements must be first read into local memory from the global memory before beginning the decoding iterations. This is shown as the first step in Figure 3.5.

Since the interleaver network remains constant throughout execution, we can consider using the *constant* memory type of OpenCL. Constant memory behaves similarly to global memory on the FPGA in that it can be written into the SDRAM by the host. However, since the compiler knows that the data will never change, it can cache the constant memory values into on-chip FPGA memory upon the first load operation into the kernel. The constant memory cache size is adjusted by a compiler option. Constant memory can have either a function scope, in which the data is only accessible within that kernel, or file scope, where the constant memory is shared across all work-items of every kernel. This memory type is attractive for implementing the interleaver network as the data is frequently read and constant. In this design, we can store the two lookup arrays for the VN-to-CN and CN-to-VN connections. All other required constant data (such as bit masks and maximum values) can also be stored using the constant data type to achieve shorter read latency. It is important to note that the cache size must be adjusted for each required codeword because if it is compiled to be too small, a relatively large number of stalls could result as the data would be constantly overwritten. To avoid any unnecessary cache miss delays with the constant memory, we need to ensure that the size of the constant memory is appropriate for each codeword. Table 3.7 shows the calculations of the required constant memory cache sizes in bytes (B). Both connection arrays for the nodes take up the same constant memory size with a regular code. As shown in Table 3.7, the array sizes for the variable node connections are calculated by $(NUMBER\_OF\_VARIABLE\_NODES \times NUMBER\_OF\_VN\_INPUTS) \times sizeof(short)$. The connection array uses the OpenCL data type *short* (which is 2 bytes in size) as this is the smallest data size allowable to store the indices. The minimum size allocatable is 16 KB (16384 bytes) for the constant memory cache. The "Miscellaneous Usage" column is made up of the constant mask vectors used in the computation and the "Amount Given" column is the compiled constant cache size. All of the tested codes use the default minimum size of 16 KB where the last two decoder sizes (1536 and 2048) require more constant memory. Since the constant memory cache size is required to be a power of 2, the last two decoder sizes are compiled with a cache size of 32 KB (32786 bytes).

LDPC Code Length	VN Lookup Usage (Bytes)	CN Lookup Usage (Bytes)	Miscellaneous Usage (Bytes)	Total Usage (Bytes)	Amount Given (Bytes)
16	$(16\times3)\times 2 = 96$	$(8\times6)\times 2 = 96$	144	336	16384
32	$(32\times3)\times 2 = 192$	$(16\times6)\times 2 = 192$	144	528	16384
...	...	...	...	...	...
1024	6144	6144	144	12432	16384
1536	9216	9216	144	18576	**32768**
2048	12288	12288	144	24720	**32768**

Table 3.7: Required Constant Memory Cache Sizes in Bytes for the Regular Codes

3.3.2 Results

As shown in Figure 3.7, the throughput for Design A has a maximum of 0.023 Mbps for a length-32 (3,6)-regular LDPC code. Figure 3.8 shows the compiler-selected clock frequencies for Design A. The decreasing clock frequencies reveal that the hardware pipeline becomes saturated around the length-32 (3,6)-regular decoder design. The normalized throughput of throughput / clock frequency is shown in Figure 3.9. Although the figure shows a linear increase from the length-64 code, the ratio appears to remain fairly constant throughout each decoder length as the increase is only by 0.00007. The reason for the non-monotonicity of Figures 3.7 and 3.9 is hidden by the compiler itself, however it was found that different random structures of the regular LDPC codes produced variations in the throughputs. One possible reason for these variations is due to the memory contention of the random interleaver connections. Because the variations only exist with the shorter LDPC codes, they are likely due to the overhead of the OpenCL execution. Note that the decoders of length-512 and higher were unable to compile for Design A on the FPGA. The limiting factor was the available FPGA logic element resources on the chip. Design A for the length-256 decoder used 98% (230025/234720) of the available logic elements whereas the length-512 decoder estimated a usage of 161% (**375552**/234720) and therefore did not successfully complete placement on

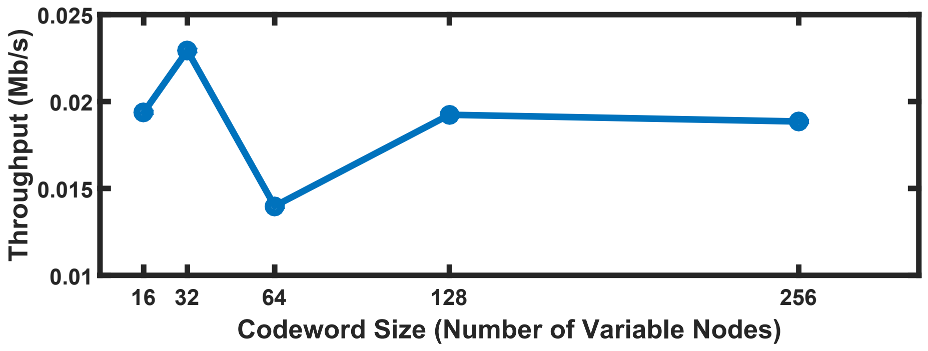


Figure 3.7: Total Coded Throughput vs. Codeword Size for Design A Using Compiler-Selected Clock Frequencies

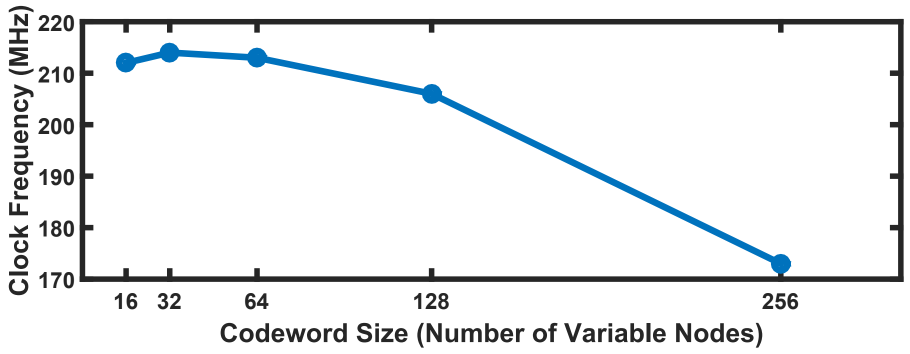


Figure 3.8: Compiler-Selected Clock Frequencies vs. Codeword Size for Design A

the FPGA.

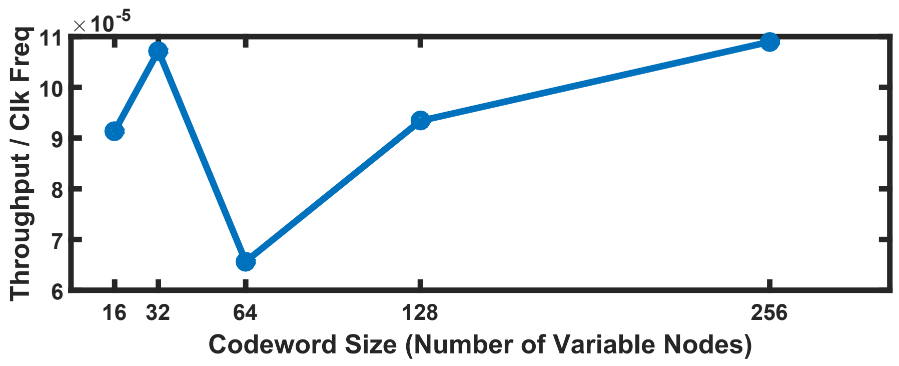


Figure 3.9: Coded Throughput / Clock Frequency vs. Codeword Size for Design A

3.3.3 Benefits

The exercise of implementing and evaluating the Single Work-Item, Single Kernel Design A, was beneficial in many ways. A fully functioning decoder was realized and the synthesized design had already achieved fairly impressive performance. However, the main benefit from this design is the information given in the optimization report from the Altera compiler. By locating the pipeline stalls for each loop iteration, we were able to improve the performance. The design was fairly simple with only minor changes from the version to make data representations to fit the OpenCL format. Because of this, the design time was relatively short.

3.3.4 Limitations

This type of design is a recommended starting point for OpenCL algorithm designs as the AOC generated optimization report shows vital data related to the potential performance. Although a few alterations have been made to fit the OpenCL memory model, the code still resembles a sequential C program and therefore should require less development time. A more efficient OpenCL design can be implemented onto an FPGA in a matter of days, which is attractive when compared to the much longer design time (and development risk) required for a custom ASIC implementation. However, upon implementing this design with a length-256 (3,6)-regular code, the achieved throughput was only 0.019 Mbps using 32 decoding iterations. It is important to note that this throughput is calculated using the total bits of the decoder (information bits + parity check bits). By comparison, a custom bit-serial ASIC design for a length-256 (3,6)-regular code by Brandon et al. has an information bit throughput of 250 Mbps at 250 MHz using 32 decoding iterations [17].

With a few more optimizations, we can significantly improve the throughput while still requiring only a fraction of the custom ASIC design time. Because of the irregular memory access patterns of the LDPC decoder algorithm required by the interleaver network, it's difficult to implement a completely stall-free loop pipeline. In order to better control the pipeline, we can use the OpenCL NDRange kernel design to efficiently launch multiple work-items simultaneously. With this flexibility, we can create and launch multiple execution pipelines and greatly increase the throughput.

3.4 An NDRange Decoder Implementation (Design B)

The first implementation using a single work-item, single kernel design provided us with detailed pipeline information that was used to optimize the node operations. For example, the latency of implementing the interleaver within global memory introduces pipeline stalls, and we found that the number of stalls can be significantly reduced through the use of local memory. To overcome some of the other limitations introduced by the single work-item Design A, we expanded the kernel to have a multiple work-item, single kernel architecture. This allowed us to efficiently execute multiple work-items simultaneously through the single shared hardware pipeline, as discussed in Table 3.6. Thus through a more careful design of the decoder, we effectively reduced the performance loss due to the pipeline stalls that affected the single work-item architecture.

Dividing the kernel into multiple work-items (NDRange Kernel) involves appropriately partitioning the memory. Although the NDRange kernel allows us to create many work-groups with multiple work-items in each, it is important to note that local memory is only shared within work-groups which execute the same kernel source code as illustrated in Figure 2.2. In order to hold the interleaver within the local memory of the OpenCL memory model, we, therefore, still require the VNs and CNs to remain within a single kernel. The VN and CN input arrays will remain within the local memory of the kernel and are declared to be the same size as with Design A. Depending on the number of work-items launched, we need to split up the execution of all VN and CN operations. A simple solution would be to launch the kernel with n_{VN} work-items such that each work-item executes one VN calculation. However, due to the (3,6)-regular structure of the code, there are twice as many VNs as CNs. This would mean that half of the work-items would be inactive for the CN calculation. A more efficient solution is to launch the kernel with n_{CN} work-items where each work-item performs two VN calculations and one CN calculation. This effectively widens the pipeline for the VN portion to accommodate two independent VN calculations.

While executing multiple work-items, we need to take precautions to ensure that all VN calculations complete before the CNs are able to read their inputs; otherwise this would compromise the correct operation of the decoder. In order to do this, we need to introduce work-item synchronization code (i.e., the OpenCL*barrier*). The work-items need to be synchronized after the VN and CN calculations.

The OpenCL kernel source code for this design is attached in Appendix G.

3.4.1 Memory Model

As mentioned above, partitioning the kernel into multiple work-items complicates the memory model compared to the single work-item, single kernel approach. In a word, the data needs to be properly parallelized and divided amongst the work-items. This requires careful programming of the kernel code to ensure that each work-item operates on the correct data. Through the use of the optimization report, we found that we could decrease the memory access latencies by implementing the interleaver network within the local memory rather than the global memory. As we know from Section 2.3, local memory only exists within each work-group of the execution. Thus in order to communicate via local memory, the design must be within one single kernel.

3.4.2 Results

As shown in Figure 3.10, with all of the above optimizations, Design B is able to achieve a maximum throughput of 51.98 Mbps when decoding the length-2048 (3,6)-regular LDPC code, whereas Design A was only able to achieve 0.023 Mbps for a length-32 (3,6)-regular LDPC code. During high-level synthesis, the AOC selects the optimal clock frequency for each kernel. This frequency is determined based on a number of factors such as pipeline depth and expected memory efficiency [9]. Due to the high-level nature of OpenCL, the programmer does not have direct control over many compiler-controlled design details, such as the pipeline depth. As shown in Figure 3.11, the irregularities in the throughput curve correlate closely with variations in the compiler-selected clock frequencies. The lower achieved throughputs (such as those experienced at codeword sizes 768 to 1536) appear to have lowered generated clock frequencies, thus contributing to longer execution times.

Comparing this design (B) to the single work-item, single kernel version (Design A), we were only able to compile a decoder for the shorter codewords due to the increased complexity and size of the required synthesized hardware. As a direct comparison, the single work-item, single kernel solution generated a throughput of 0.023 Mbps whereas the above multiple work-item design produced 20.4 Mbps while using the length-64 (3,6)-regular LDPC code, effectively generating 25 times the throughput.

The performance gain appears to level off above the length-1024 (3,6)-regular code size limiting the throughput to just above 50 Mbps. Because the only variation between the compiled kernels for each code size is the number of launched work-items, it appears that for codes larger than (n_{VN},n_CN)=(1024,512), the one shared hardware pipeline reaches saturation, as illustrated by Figure 3.12. That figure shows the ratio throughput/f_{CLK}. In order to allow even greater throughput, we need to look at replicating the design to more fully use the available hardware resources in the FPGA to increase the number of simultaneously active work-items and to also overcome the bottleneck of a single saturated pipeline.

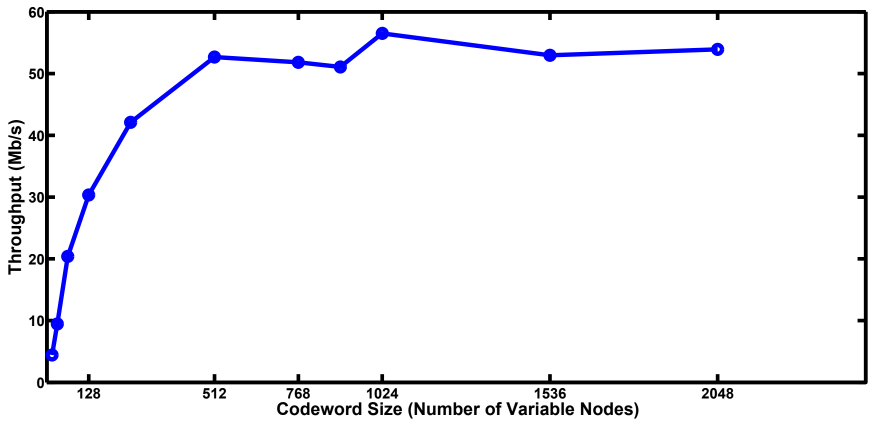


Figure 3.10: Total Coded Throughput vs. Codeword Size for the Local Memory Design B Using Compiler-Selected Clock Frequency

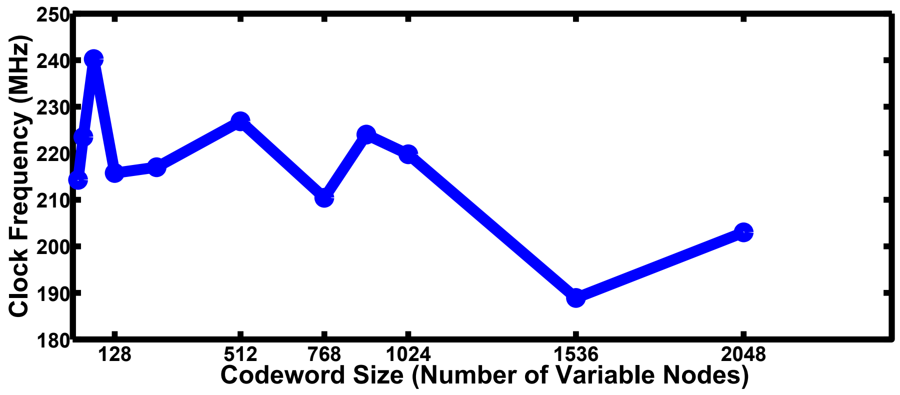


Figure 3.11: Compiler-Selected Clock Frequencies vs. Codeword Size for the Local Memory Design B

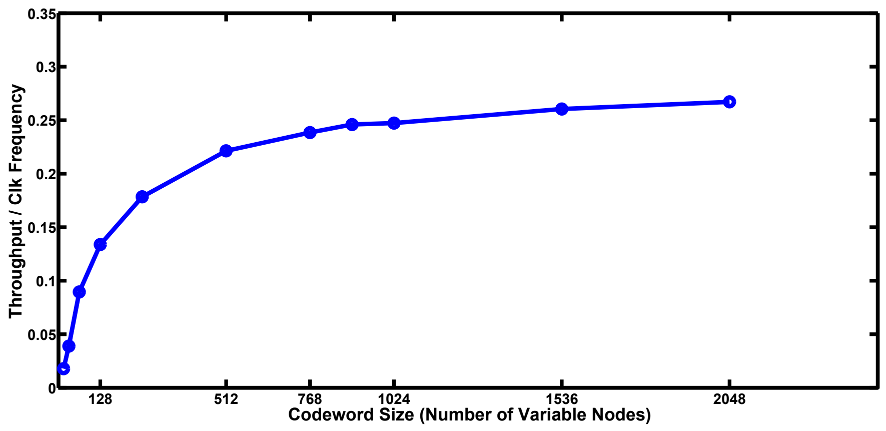


Figure 3.12: Coded Throughput / Clock Frequency vs. Codeword Size for the Local Memory Design B

3.5 Replication of the NDRange Decoder Hardware (Design C)

As introduced in Section 3.2.3, we are able to use kernel attributes, such as *num_simd_work_-items*, to increase the width of the datapath compiled from vectorized code. We can also adjust the *num_compute_units* attribute to replicate the hardware design over the available FPGA fabric. As recommended by Altera in [9], *num_simd_work_items* should be used primarily because the hardware cost per performance increase achieved is relatively low. Instead of copying the entire hardware pipeline, this attribute simply vectorizes the data path and generates the hardware appropriately. For example, with *num_simd_work_items* = 2 the compiler generates a single hardware pipeline that is twice the width thus allowing twice the number of work-items as discussed in Table 3.6. In this manner, the same number of work-items is launched; however, the execution paths are divided along the synthesized vectored execution lanes. The parallel lanes then execute simultaneously and in synchronism. Thus, the throughput can be increased by processing more nodes of the same codeword simultaneously. Figure 3.13 illustrates the effects of the parallelism attributes *num_simd_work_items* and *num_compute_units* on the hardware pipeline. The *num_simd_work_items* attribute increases the effective width of the hardware

LDPC Code Length	*num_simd_work_items* used
16	2
32	2
...	...
2048	2

Table 3.8: Replication of the NDRange Decoder Design C *num_simd_work_items*

pipeline by vectorizing the operations. Note that this is independent of the OpenCL vector types explored in Section 3.2.3. That is, the OpenCL vector types (i.e., *cl_char16*) can be used in any of the configurations for Figure 3.13. When using the *cl_char16* type, each pipeline width is of *cl_char16*. For middle example case in Figure 3.13 where $num\_simd\_work\_items = 4$ and $num\_compute\_units = 1$ each of the SIMD pipelines will have the width to be able to process a *cl_char16* type effectively giving the overall pipeline a width of $size(cl\_char16) \times 4$. Increasing the *num_compute_units* attribute replicates the entire hardware pipeline to another location in the FPGA. Each work-group in OpenCL will be scheduled to run on an available compute unit in the FPGA. For example, if there is only one launched work-group, only one compute unit will be used.

Since the decoder design involves multiple work-items in a single work-group, it is not worthwhile to increase the number of compute units as the OpenCL scheduler assigns a work-group to a single compute unit. By manually testing the estimated hardware usage of each compiled kernel, we could empirically determine optimal values for the SIMD attribute. It's important to note here that the compile times are significantly higher with the higher utilization percentages of the FPGA. Due to hardware resource limitations, we were only able to introduce a SIMD vector width of two for all of the codeword sizes, as shown in Table 3.8. This limitation could be overcome by using larger FPGA fabrics.

The OpenCL kernel source code for this design is attached in Appendix H

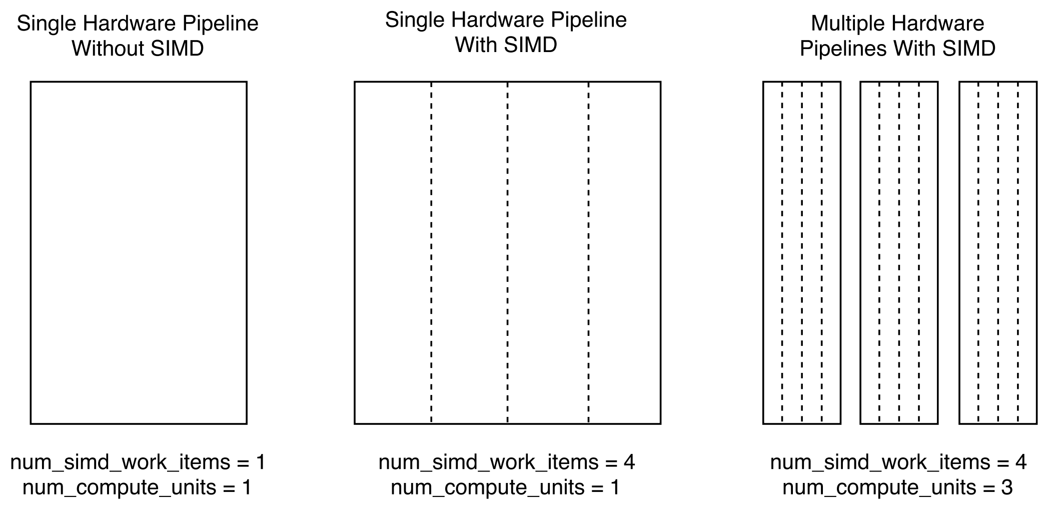


Figure 3.13: OpenCL Parallelism Kernel Attributes

3.5.1 Results

After replicating the NDRange local memory kernel using *num_simd_work_items*, we were able to achieve an average coded throughput of 64.93 Mbps, as shown in Figure 3.14. While the plot results were obtained by averaging 3 runs for each codeword size, the maximum achieved throughput for the length-2048 codeword was actually 68.22 Mbps. The results confirm that upper performance limit for the non-replicated implementation (Design B) is due to the saturation of the pipeline as the throughput continues to be pushed with larger codes. This is emphasized by the comparison shown in Figure 3.15 as the throughputs are similar until the codeword size reaches $n_{VN} = 768$. For longer codes, the replicated kernel continues to increase in throughput. At the maximum calculated throughput, the replicated version produced 1.204 times the throughput of the original local memory implementation (compared at 53.92 Mbps).

As with the original results, the throughput nonmonotinicities (although smaller in magnitude) experienced at codeword size 1024 correlate to drops in the compiler-selected clock frequency, as shown in Figure 3.16. The throughput / clock frequency ratio for Design C is shown in Figure 3.17. The ratio continues to increase linearly after a codeword size of 1024. Comparing this ratio with Design B in Figure 3.18,

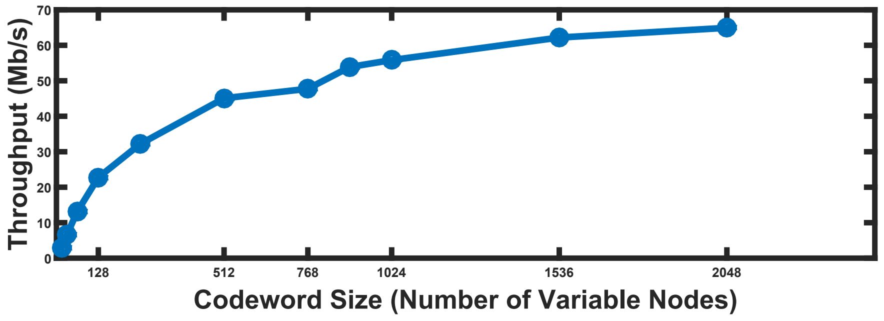


Figure 3.14: Total Coded Throughput Results vs. Codeword Size for Design C

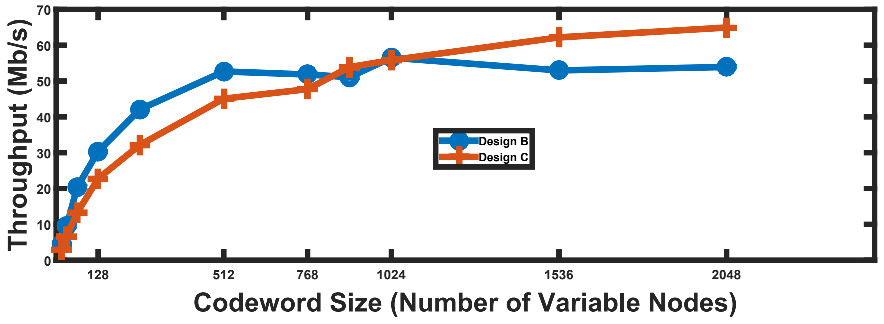


Figure 3.15: Total Coded Throughput vs. Codeword Size for Design B (Local Memory) and Design C (Local Memory Replication)

we can see that while the increase in the ratio for Design B slows down markedly with the increasing codeword size, the ratio for Design C continues linearly with a positive slope.

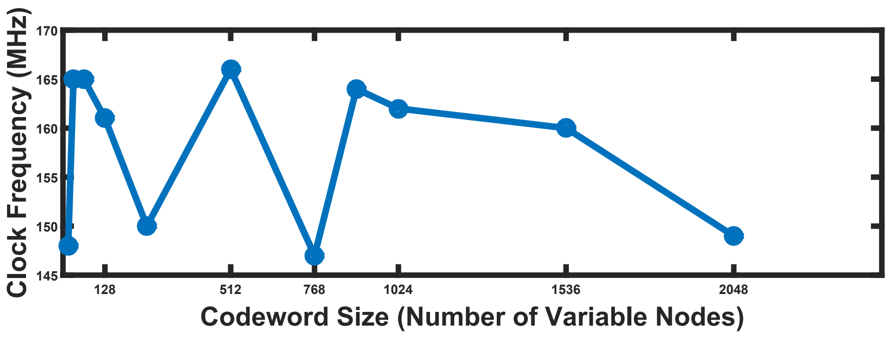


Figure 3.16: Compiler-Selected Clock Frequencies for Design C

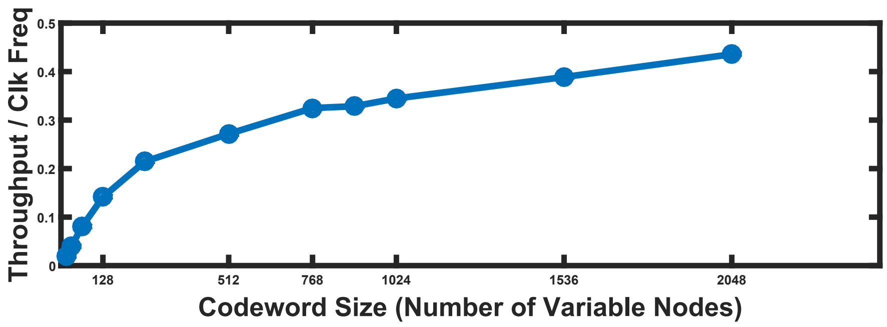


Figure 3.17: Coded Throughput / Clock Frequency vs. Codeword Size for the Local Memory Design C

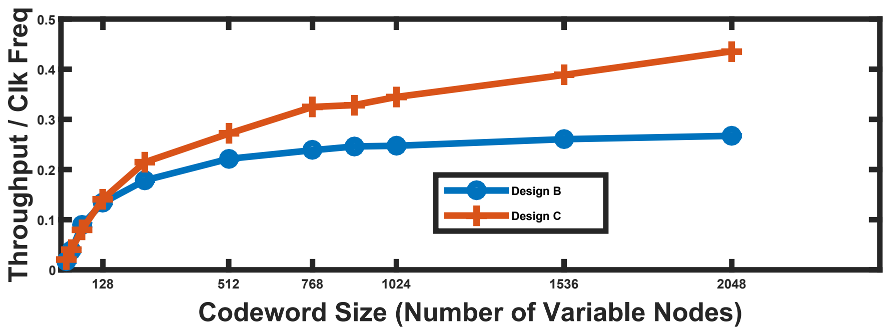


Figure 3.18: Coded Throughput / Clock Frequency vs. Codeword Size for Designs B and C

3.6 Design Comparison

The regular LDPC decoder provided a challenging and instructive benchmark case for the OpenCL compiler as it involved many different forms of potentially exploitable parallelism. While the compiler was able to efficiently exploit the automatic pipeline parallelism in Design A, further changes were made to investigate how best to exploit the three other types of designer inserted parallelism shown in Table 3.6. By following the recommendations of the Altera design tool, the optimization guide [9] and through experimentation, we were able to achieve a high-throughput decoder. By combining hardware pipelining, vectorization, and replication in the final local memory Design C, we were able to achieve a maximum throughput of 68.22 Mbps for a randomly designed length-2048 (3,6)-regular LDPC code.

Another LDPC OpenCL-based decoder reported by Falcao et al. had a throughput of around 7 Mbps (with 30 decoding iterations) for a length-1024 (3,6)-regular code [18] whereas our final Design C achieved a throughput of 55.8 Mbps (with 32 decoding iterations) for the same code size, as shown in Figure 3.14.

For a length-1024 (3,6)-regular code, Falcao et al. reported a throughput of around 7 Mbps (with 30 decoding iterations) for an OpenCL-based FPGA-targetted decoder [18] whereas our final Design C achieved a throughput of 55.8 Mbps (with 32 decoding iterations) for the same code size, as shown in Figure 3.14. Falcao used the Xilinx Virtex-6 LX760 FPGA (which is similar in size to the FPGA used in this thesis research) with the SOpenCL for FPGA compiler [18]. It would be possible for our experimental version to achieve a higher throughput for the length-1024 (3,6)-regular code by simply compiling for the longer (2048,1024) code and decoding twice the number of codewords in the same amount of hardware, however the interleaver would need to provide the connections for two half-length codewords. This would be straightforward to develop as the interleaver connections are stored in a look-up table.

Pratas et al. reported a multiple kernel throughput implementation of 17 Mbps for a length-64800 DVB-S2 standard LDPC code with 10 decoding iterations [27].

Although our implementations investigated shorter codes, our Design C was able to produce increasing throughput with LDPC code size, with a majority of the results surpassing 17 Mbps. The designs investigated by Andrade et al. [15] reported comparable throughputs of just over 100 Mbps. However, because their design targeted irregular codes, the results are not directly comparable with due to the significantly different interleaver networks and node designs.

Although many parameters needed empirical tuning and some software alterations were required to give the compiler the necessary flexibility to produce optimized hardware, we were able to produce a relatively efficient and flexible decoder design for regular LDPC codes within a fairly short development time. In order to optimize for medium length (i.e., 512 or 1024) regular codewords, our design was implemented as a single kernel, single work-group design. However, this is not especially scalable when introducing the irregular LDPC codes in Chapter 4 as the performance bottleneck becomes the available local memory. A balance must be struck between programming the decoder design to allow a greater variety of codewords lengths and optimizing the decoder design for a few of those lengths.

Generating the single work-item, single kernel (A) design allowed us to locate and reduce the pipeline bottlenecks through the use of the Altera Optimization Report [9]. Although this approach proved the simplest and resembled sequential C code design, more efficient optimizations could be made with an NDRange kernel (B). By expanding past the initial implementation of the single work-item, single-kernel decoder (A) design, we were able to achieve throughputs comparable to other optimized FPGA-based LDPC decoders using OpenCL described in the literature. The OpenCL compiler allowed us to entirely avoid designs expressed in hardware description languages such as Verilog or VHDL.

Although the OpenCL-based decoder design produces respectable decoding throughput results, there are many limitations with this design approach. The local memory for each FPGA device is significantly smaller than the available global memory. Because of the irregularity of our interleaver network, we were not able to partition the decoder into separate work-groups and therefore the decoder design eventually will be limited by the local memory capacity for longer codewords. With efficient memory usage, we were able to compile a kernel for a random length-2048 (3,6)-regular LDPC code; however, this design is not sufficient for the industrial-sized irregular codes such as those present in the DOCSIS 3.1 standard [7].

In order to handle larger regular or irregular LDPC codes, different OpenCL software architectures must be researched.

Chapter 4

Irregular LDPC Decoder Implementations

Irregular LDPC codes, such as those present in the DOCSIS 3.1 standard [7], are often used in industrial applications instead of regular LDPC codes. The DOCSIS 3.1 upstream standard uses Quasi-Cyclic Low-Density Parity Check (QC-LDPC) codes rather than traditional block LDPC codes. Quasi-Cyclic LDPC codes are block LDPC codes that have a defined structure for the connections between the nodes allowing for easier decoder implementation. Regular LDPC codes are those codes where the variable nodes all have the same degree and the check nodes all have the same degree. This is not true for irregular codes, which have a variable number of connections to the VNs and to the CNs. Because of this difference, designing an OpenCL-based decoder for irregular codes has some unique challenges.

4.1 Design Challenges

Because of the irregular number of connections between nodes for each irregular LDPC code, we are more limited in our design of the OpenCL kernel. In the regular NDRange decoder, discussed in Section 3.4, we were able to use the OpenCL device local memory as the interleaver between the VNs and CNs. We were also able to optimize the independent minimum and second minimum calculation tournaments in the CNs by assuming the same input fan-in for each check node.

4.1.1 Multiple Kernel Design

All of the designs discussed earlier in Chapter 3 exploit the ability to use local memory as the interleaver and to use a single NDRange kernel design. The previously discussed designs exploited the static ratio between the number of VNs and CNs in (3,6)-regular LDPC codes by having each work-item of the single kernel design handle two variable node calculations and one check node calculation. Irregular codes do not have the same number of connections for each VN and CN, nor is the ratio between the number of nodes static between different irregular codes (sometimes called asymmetric codes).

We will look into designing decoders for three of the irregular LDPC codes used in the DOCSIS 3.1 standard (for upstream data transmit): a short code (length-1120 (1-5,15-16)-irregular), a medium code (length-5940 (1-5,26-27)-irregular), and a long code (length-16200 (1-5,33-34)-irregular). In the notation for the irregular codes the node connection ratio is shown as the available range of fan-outs for each node type. For example, the length-1120 (1-5,15-16)-irregular code has a range of fan-outs from the VNs of 1-5 and the range of fan-outs from the CNs is 15-16. Table 4.1 specifies these three codes in more detail. All of the connection information for the DOCSIS 3.1 codes started as a circular LDPC description file whereas the regular decoders used the A-List format. More information about the circular LDPC description file type is found in [7], however in order to use the LDPC code with our host code we need to convert these files to A-List format. A program was developed that takes in the circular matrix file and outputs the generated A-List file for the irregular code. The source code for this program is attached in Appendix I As shown in column 4 of the table, the ratio of the number of variable nodes to the number of check nodes is not consistent throughout the different LDPC codes. Furthermore, the maximum check node fanout (the number of connections to a check node) increases with the length of the code due to the decreasing ratio of parity check constraints to code bits. The longer the code in the DOCSIS 3.1 standard, the larger the ratio of information bits compared to the number of parity bits. Because the ratio and fanout is different

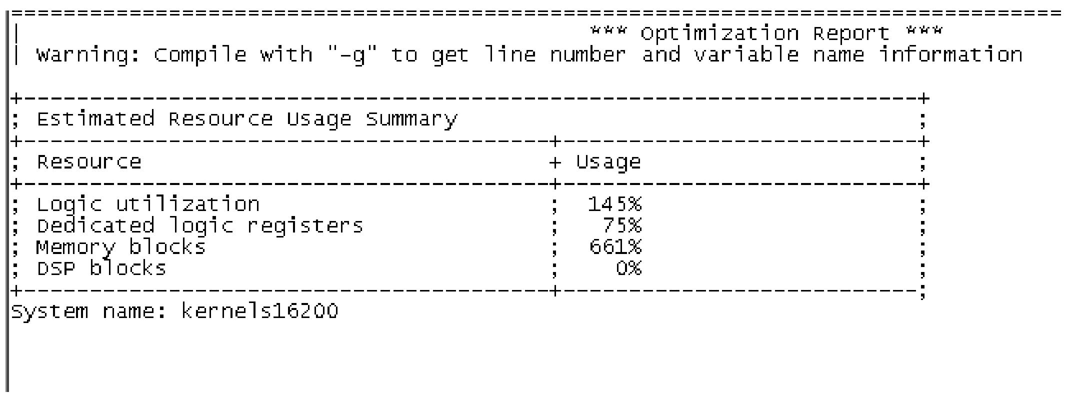
Code	Number of VNs	Number of CNs	Ratio of VN/CN	Maximum VN Fanout	Maximum CN Fanout
Short Code	1120	280	4	5	16
Medium Code	5940	900	6.6	5	27
Long Code	16200	1800	9	5	34

Table 4.1: DOCSIS 3.1 Code Definitions [7]

for the three codes, we are not able to create a single kernel design for all three of the codes. It would be possible to design an efficient decoder using local memory as the interleaver for each code individually, however it would require a much higher ratio of VN to CN execution. For example, for the shortest code of length 1120 we would need each work-item to execute one variable node and $1120/280 = 4$ check nodes. This approach, however, would exceed the amount of available logic on the FPGA and is not scalable to the longer irregular codes. Because this approach is not scalable, we therefore investigated designs that use multiple kernels.

4.1.2 Memory Usage

Because of physical memory limitations, there is a limited amount of memory for each OpenCL memory type. Resource allocation and usage for OpenCL on FPGAs is different compared to when compiling for other more static device types. Local memory, for instance, in OpenCL on FPGAs is synthesized in the on-chip FPGA device memory whereas for most other OpenCL devices this memory has a static limit for each device (i.e., 32 KB) as this local memory amount is available for each work group. Because the compiler controls the location and width of all hardware pipelines, it is able to reassign the memory blocks on the device and allow for dynamic local memory allocation. This allows designs some flexibility in demanding more local memory, however the local memory resource requirements for the large fanout and ratios of each DOCSIS 3.1 irregular code are still too large to efficiently fit the interleaver solely onto the FPGA. Although the idea of using the local memory for the irregular decoders as in Chapter 3, Figure 4.1 shows a screenshot of the estimated resource usage for the longest irregular code of length 16200. This



```
================================================================================
|                                                  *** Optimization Report ***
| Warning: Compile with "-g" to get line number and variable name information

+--------------------------------------------------------------------+
; Estimated Resource Usage Summary                                   ;
+----------------------------------------+---------------------------+
; Resource                               + Usage                     ;
+----------------------------------------+---------------------------+
; Logic utilization                      ;  145%                     ;
; Dedicated logic registers              ;   75%                     ;
; Memory blocks                          ;  661%                     ;
; DSP blocks                             ;    0%                     ;
+----------------------------------------+---------------------------;
System name: kernels16200
```

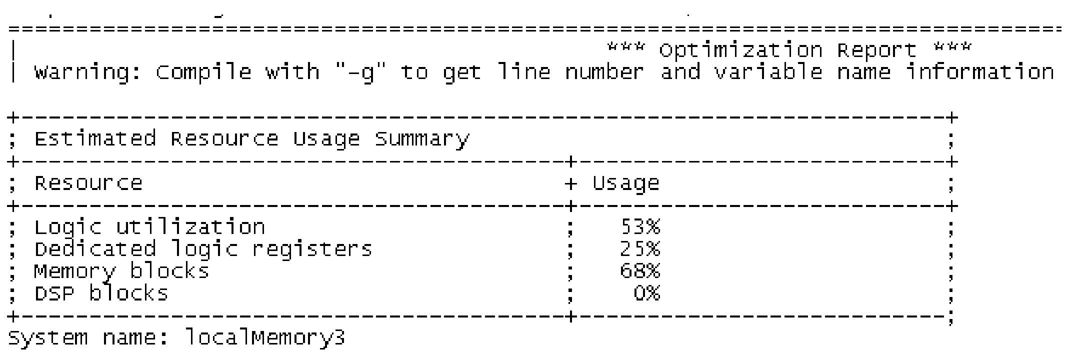
Figure 4.1: Irregular Decoder Local Memory Estimated Resource Usage

Resource	Estimated Usage	Actual Usage
Logic Utilization	53%	49%
Dedicated Logic Registers	25%	24.4%
Memory Blocks	68%	61%
DSP Blocks	0%	0%

Table 4.2: Design C Actual Resource Usage vs. Estimated Resource Usage

estimated resource usage report is generated before the hardware synthesis before the design is passed to Quartus for place and route. The estimations shown in Figure 4.1 reveal a significant amount of memory block usage as well as logic utilization. Although the hardware synthesis tools regularly optimize the compilation using less resources than estimated, the estimations are usually quite accurate. As a direct comparison with a compiled regular decoder, Figure 4.2 shows a screenshot of the estimated resource usage for the length-2048 (3,6)-regular decoder. The actual (Quartus reported) FPGA resource usage for the length-2048 (3,6)-regular decoder is shown in Table 4.2. The actual resource usage was only a difference of a few percentages compared to the estimations for the regular decoder and since the decoding algorithm itself is the same for the irregular decoders, we should see a similar difference in resource usage. Not surprisingly, the compilation of the local memory interleaver for the length-16200 (1-5,33-34)-irregular decoder failed due to lack of resources on the FPGA after the compiler estimated 661% memory usage.

As discussed in Section 2.4, our FPGA device uses the on-board (but off-chip) SDRAM to implement the OpenCL global memory. The size of the global memory



```
===========================================================================
|                                          *** Optimization Report ***
| Warning: Compile with "-g" to get line number and variable name information

+--------------------------------------------------------------------+
; Estimated Resource Usage Summary                                   ;
+----------------------------------------+---------------------------+
; Resource                               + Usage                     ;
+----------------------------------------+---------------------------+
; Logic utilization                      ;   53%                     ;
; Dedicated logic registers              ;   25%                     ;
; Memory blocks                          ;   68%                     ;
; DSP blocks                             ;    0%                     ;
+----------------------------------------+---------------------------;
System name: localMemory3
```

Figure 4.2: Design C Estimated Resource Usage

is significantly larger compared to the other available OpenCL memory types and this makes it an attractive alternative for storing the large amount of data required by the DOCSIS 3.1 irregular codes.

In Section 3.3.1, the constant memory cache size usage was investigated for the regular decoder. Although the interleaver connections are inherently different for the DOCSIS 3.1 irregular codes, the philosophy of using the constant memory cache for the interleaver lookup tables remains most efficient approach for this chapter as well. In this design, we will also store the two lookup arrays for the VN-to-CN and CN-to-VN connections. However, due to the larger number of connections for each individual node, we need to keep a closer eye on the compiled cache sizes. As initially introduced in Sections 2.3 and 2.4, constant memory initially exists on the external SDRAM (the same location as the global memory) until it is eventually cached on the FPGA. Although keeping the cache size constant does not change the functional correctness of the decoder, a cache that is too small would introduce pipeline stalls from the constant replacing of data into the cache. By ensuring that the constant memory cache size is sufficiently large to hold all of the constant memory variables, we can avoid any potentially performance-killing situations. Table 4.3 calculates the total constant memory usage for the three irregular codes. The array sizes for the variable node connections are calculated by $(NUMBER\_OF\_VARIABLE\_NODES \times NUMBER\_OF\_VN\_INPUTS) \times sizeof(int)$. Due the large sizes of the irregular code connection arrays, we are forced to use the *int* OpenCL type to store the array indices. Comparing these results with those calculated for the regular decoders in

LDPC Code Length	VN Lookup Usage (Bytes)	CN Lookup Usage (Bytes)	Miscellaneous Usage (Bytes)	Total Usage (Bytes)	Amount Given (KB)
1120	$(1120 \times 5) \times 4 = 22400$	$(280 \times 16) \times 4 = 17920$	$144 \times 2 = 288$	40608	64
5940	$(5940 \times 5) \times 4 = 118800$	$(900 \times 27) \times 4 = 97200$	288	216288	256
16200	$(16200 \times 5) \times 4 = 324000$	$(1800 \times 34) \times 4 = 244800$	288	569088	1024

Table 4.3: Required Constant Memory Cache Sizes for the Irregular Codes

Table 3.7, the irregular decoders require significantly more constant memory. As with the regular decoders, the miscellaneous usage is made up of the constant mask vectors and is exactly doubled when compared to the regular decoders. The amount given, as in Table 3.7, is the compiled amount of constant memory cache for that LDPC code length. The compiled constant memory cache size must be rounded to the nearest power of two (i.e., 16 KB, 64 KB, etc.).

4.1.3 Check Node Minimum-Find Tournaments

As introduced in Section 2.6, the min-sum decoding algorithm requires that each check node find the first and second minimum magnitudes of the incoming messages. A two-stage tournament structure was then implemented in Section 3.2 for the regular decoder case. The regular decoder tournament for (3,6)-regular codes assumed a fixed number of 6 inputs to each CN allowing for the tournament structure to remain constant over all LDPC codes. The same structure cannot be used in the irregular case as the number of inputs is no longer fixed to 6 but variable depending on the irregular code. In order to handle the irregularity of the number of check node inputs, a program was made to produce the source code for the tournament structure. This minimum tournament generation program assumes the maximum number of inputs to be the maximum fanout for each of the DOCSIS 3.1 codes described in Table 4.1. The number of rounds required for both the first and second minimum magnitude tournaments depends on the maximum fan-in input for the specified LDPC code. For example, for the length-1120 (1-5,15-16)-irregular code the first minimum magnitude tournament requires $log_2(16) = 4$ rounds. The source code for the this program is attached in Appendix U.

4.2 Global Memory Interleaver with Channel Synchronization

With all of these design challenges in mind, the seemingly simplest approach is to create a two-kernel decoder using the device global memory for the interleaver storage. As briefly introduced in Section 3.3.1, using the global memory as the interleaver will inherently introduce more latency when compared with the local memory implementation. The main compensating advantage to this approach is that the two kernels can work on two independent buffers thus doubling the number of codewords decoded simultaneously. Without the size restriction of local memory, we are able to store many more codewords in the interleaver.

4.2.1 Channel Synchronization

With two independently and simultaneously executing kernels on the FPGA, the most efficient tool to use for synchronization is the Altera OpenCL extension channel. These channels allow direct kernel-to-kernel communication within the OpenCL device, without requiring any communication via the host or external global memory. The communication channel is implemented as a First-In-First-Out (FIFO) buffer on the FPGA.

Since the variable nodes and check nodes must pass data between each other, we need to synchronize the two sets of node calculations at the beginning of each iteration. Table 4.4 shows how the variable nodes and check nodes will operate on two different codeword sets (data sets) using a “ping-pong” approach for a number N of iterations. Using this schedule, both kernels will be active simultaneously on different data sets (‘a’ and ‘b’) allowing for roughly twice the throughput. For example we see from Table 4.4 that for $N = 8$ decoding iterations the variable node kernel will need to repeat for $2N + 1 = 17$ iterations while the check node will only

VN Iteration #	0	1	2	3	4	5	...	2N - 2	2N - 1	2N	2N + 1
VN Data Set	a	b	a	b	a	b	...	a	b	**a**	**b**
CN Data Set	-	a	b	a	b	a	...	b	a	b	-
CN Iteration #	-	0	1	2	3	4	...	2N - 3	2N - 2	2N - 1	-

Table 4.4: Synchronization for N Iterations

need $2N - 1 = 15$ iterations. This is due to the first and last iterations of the variable node being used to fill and empty the global memory buffer pipeline. The bolded iterations in Table 4.4 only exist to transfer the data back to the host for each variable node. For an industrial application, these last two iterations would be used to load in and compute the next two codeword data sets.

4.3 Single Work-Item (Design D)

Much like with the regular decoder, designing first with a single work-item approach is the most simple. By following this approach, as recommended by Altera [10], we are able to simplify the program design such that it resembles a simple sequential C version. The Altera OpenCL optimization guide is also available to provide design guidance for these single work-item approaches [9].

Figure 4.3 illustrates the hardware layout for the multiple kernel approach. Both kernels operate independently and simultaneously on their own respective hardware pipelines in the FPGA. The kernels perform their iterations as outlined in Table 4.4. The use of the two buffers ‘X’ and ‘Y’ in the external SDRAM allows the kernels to operate simultaneously on different data sets of codewords.

4.3.1 Optimization Report

The original approach for Design D only used the global memory for the interleaver and operations. The compiler-produced optimization report for single work-item designs as well as the kernel profiling information [9] showed a relatively large number of pipeline stalls due to global memory reads in both kernels. From this information, and the advice presented in the Altera Optimization Guide [9], the source code was changed to pre-load all of the data for each node type into the on-chip

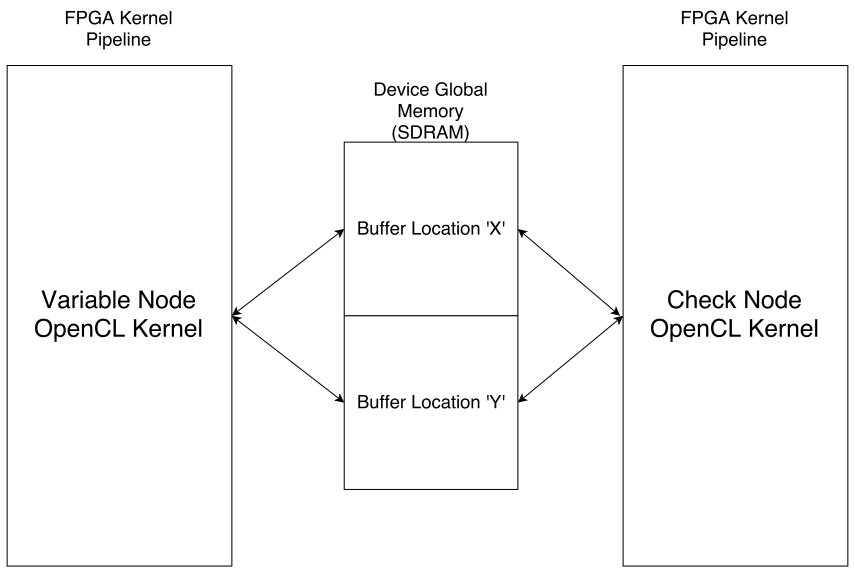


Figure 4.3: Multiple Kernel Pipeline Block Diagram

FPGA memory. A combination of local memory and private memory (implemented with device registers) was used to temporarily hold the data during the computation. This change significantly decreased the global memory read stall percentage for both pipelines.

4.3.2 Results

Design D uses design insights developed earlier for the regular decoder design such as vectorization using the *cl_char*, the memory model usage, and the optimizations made in Section 3.2. The source code for Design D is attached in Appendices M, N, and O. Figure 4.4 shows the results for Design D with the three irregular LDPC codes detailed in Table 4.1. The effective throughput of the design appears to decrease linearly with the increase in irregular code size with the largest throughput experienced with the lowest length-1120 (1-5,15-16)-irregular code at 5.4 Mbps. The clock frequencies for the three designs are shown in Figure 4.5. The ratio of coded throughput/clock frequency is shown in Figure 4.6. This relatively low initial throughput result is analogous to that of the Single Work-Item, Single Kernel (Design A) for the regular LDPC codes. The reducing throughput shows that the pipeline hardware is saturated and producing stalls. To reduce these pipeline stalls,

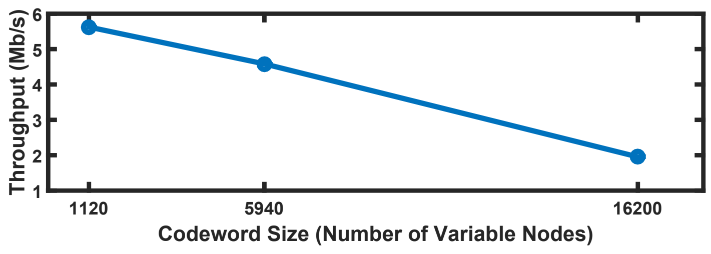


Figure 4.4: Single Work-Item (Design D) Results

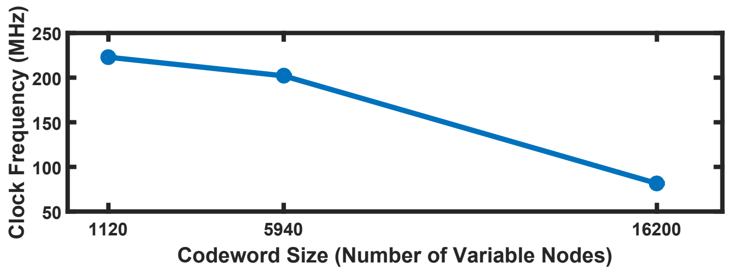


Figure 4.5: Single Work-Item (Design D) Clock Frequencies

the design needs to widen and introduce the SIMD hardware pipeline approach or generate more compute units to handle the increasing volume of computation. However, much like with the regular decoder case, the first step is to create an NDRange kernel design with multiple work-items for each kernel.

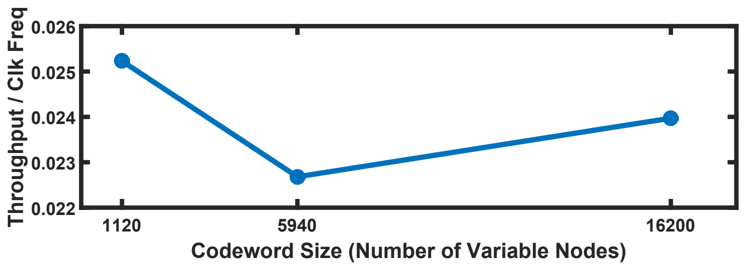


Figure 4.6: Coded Throughput / Clock Frequency for Design D

4.4 An NDRange Decoder Implementation (Design E)

An NDRange kernel design allows the developer the flexibility to launch multiple work-items on SIMD hardware pipelines or even on separate compute units, depending on how the work-items are organized within each work-group. As with the regular decoder, modifying the single work-item design for NDRange execution requires a change to the memory organization specifically with the local/private memory OpenCL types. The hardware structure shown in Figure 4.3 will remain constant for this NDRange design as well. However, the scheduler will be responsible to efficiently launching work-items to fully utilize the hardware. With reference to Table 3.6, this design (Design E) will be implementing both vectorization and hardware pipelining.

4.4.1 Work-Item Synchronization

Section 4.2.1 introduced the need to synchronize both kernels (VN and CN) with each other in order to control the order of execution for each dataset using the Altera OpenCL channel extension type. This remains true for the NDRange Design E, however work-items internally for each kernel must also synchronize to prevent them from producing incorrect LDPC decoder results. In order to implement this synchronization, we will use the OpenCL *void barrier(cl_mem_fence_flags flags)* function. The *barrier* function forces all work-items in a work-group to stall and wait until all work-items have executed the barrier [4]. Including the input parameter *CL_GLOBAL_MEM_FENCE* ensures that all global memory reads/writes before the barrier function will commit to memory prior to any reads/writes after the barrier. Since the barriers can only synchronize work-items within one specific work-group, we need to design the NDRange irregular decoder within one single work-group.

4.4.2 Emulation Results

After designing and debugging the Design E irregular decoder using the Altera OpenCL Emulator, as outlined in [10], the final execution on the FPGA produced an unexpected deadlock. This deadlock was found to be the result of the kernel synchronization method and kernel communication. With the initial single work-item approach, the pipeline contained synchronization via the Altera OpenCL channels with the two simultaneously executing kernels. Because of the irregular ratio of VN/CN for each irregular code, it was decided that the best approach is to have the first work-item of each kernel be responsible for handling the communication via the Altera OpenCL channels. By using barrier synchronization after each channel read, the decoder would produce the correct results. However, the OpenCL specification does not require compilers/devices to enforce the simultaneous operation of multiple kernels. It is unclear whether this compiler limitation will be changed in future updates, however only theoretical results are obtainable for this design and this time.

A variety of methods were investigated for debugging this problem. Initially, the functionality and correctness of the LDPC decoder was verified using the Altera emulator environment. The decoder appeared to producing the correct results, however as expected, the emulator was executing the kernels serially. Debug designs that use multiple print statements (global memory writes) were then programmed, compiled, and executed on the FPGA. These designs revealed that kernels were indeed executing simultaneously (or intertwined) and the point of deadlock was consistently after the attempted synchronization using the Altera OpenCL channels. Although the execution problems are not entirely known at this time, future compiler iterations will support a more developed communication method between kernels.

Although the Altera emulator only promises functionally-correct results and makes intelligent assumptions for the execution flow, we can obtain throughput results for proof of completion. These results are listed in Table 4.5. Please note that the emulation for the larger irregular code length-16200 (1-5,33-34)-irregular was not able run to completion on the Altera emulator due to the program maximum work-item limitations. Note that we cannot accurately estimate the throughput of the resulting device program as the emulator results vary significantly depending on other factors such as load on CPU, order of execution, etc.

Code Length	Throughput (Mbps)
1120	0.8594
5940	0.2918
16200	-

Table 4.5: NDRange Design E Emulation Results

4.4.3 NDRange Pipeline Replication

As shown from the emulation results, simply extending Design D to work with multiple work-items in the irregular case does not solve our issue of pipeline saturation because the throughput continues to decrease with the increased code length. The next step would be to increase the available hardware for execution, similar to what was done in Section 3.5. Increasing the width of the pipeline using the *num_simd_work_items* should increase the throughput for the larger codes. Unfortunately executing this attribute in emulation will not does not change the throughput as the program is executed serially on the CPU.

There are, however, other software architectures which will correctly execute on the FPGA device that we can evaluate. Since the issue is due to the synchronization method between the kernels for Design E the logical solution would be to move the synchronization responsibilities off of the device and onto the host.

4.5 Host Synchronization Control (Design F)

Moving the responsibilities of synchronization to the host allows for simplified device hardware and source code. For most OpenCL devices, specifically GPUs, this type of programming structure is not only normal, but usually preferred as it removes the inter-kernel communication and allows the device to simply execute on the provided data. In a heterogeneous platform, this would allow the host to better distribute the computations with the host acting as the central co-ordinating node.

Moving the synchronization to the host also requires that we let the host control the iterations of the LDPC decoding process. Figure 4.7 illustrates the execution of this design. The host program is responsible for starting the execution of both the

VN kernel and the CN kernel at the beginning of each iteration. At the end of each iteration, the host enforces the synchronization by waiting for both kernels to fully finish before continuing. With this configuration, the device is still able to operate on two separate data sets **a** and **b** using the same buffer locations **X** and **Y** shown in Figure 4.3.

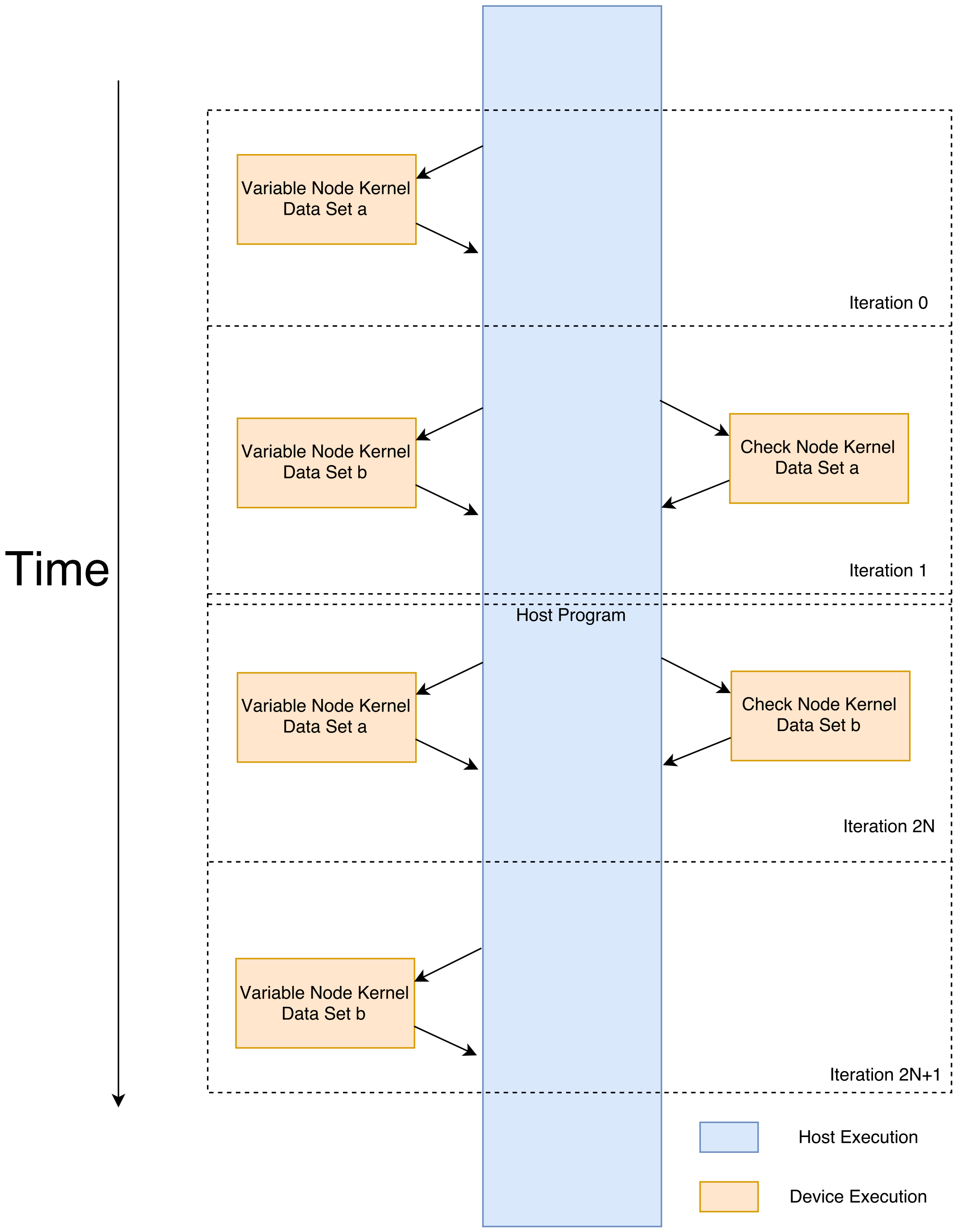

Time
Variable Node Kernel Data Set a
Iteration 0
Variable Node Kernel Data Set b
Check Node Kernel Data Set a
Iteration 1
Host Program
Variable Node Kernel Data Set a
Check Node Kernel Data Set b
Iteration 2N
Variable Node Kernel Data Set b
Iteration 2N+1
Host Execution
Device Execution

Figure 4.7: Host Synchronization Execution Flow Diagram

4.5.1 Pipeline Replication

As we learned from Section 3.5 and Design C, increasing the *num_simd_work_items* variable whenever possible can remove pipeline saturation and boost performance depending on the available logic on the FPGA. Although we weren't able to test the previous NDRange implementation to determine the exact location of the pipeline saturation, using all available hardware will ensure the best possible performance. Table 4.6 shows that a *num_simd_work_items* = 2 was only able to fit on the compilation of the length-1120 decoder whereas the other two were not able to fit a replicated pipeline.

4.5.2 Results

With the complexity of the synchronization transferred from the OpenCL source code to the host, all three LDPC code sizes were able to compile and execute on the FPGA test bed. Figure 4.8 shows the total coded information throughput achieved for the Host Synchronization (Design F) setup for the three irregular codeword sizes introduced in Table 4.1. With this NDRange kernel setup, we can see that the throughput increases with the larger codewords (opposite of that shown in Figure 4.4 for Design D). Figure 4.9 shows the compiler-selected clock frequencies for each of the kernels and Figure 4.10 plots the ratio of Throughput / Clock Frequency. The decrease in clock frequency in the largest codeword (length-16200) is likely due to the saturation of the hardware pipeline. However, as Figure 4.10 shows, the design appears to still be scalable as the throughput / clock frequency ratio continuously increases with the larger codewords. This same situation was originally encountered for Design C as discussed in Section 3.5. The solution in that design was to increase the available hardware pipeline for the execution by replication (using

Code Length	*num_simd_work_items* used
1120	2
5940	1
16200	1

Table 4.6: Host Synchronization Design F *num_simd_work_items*

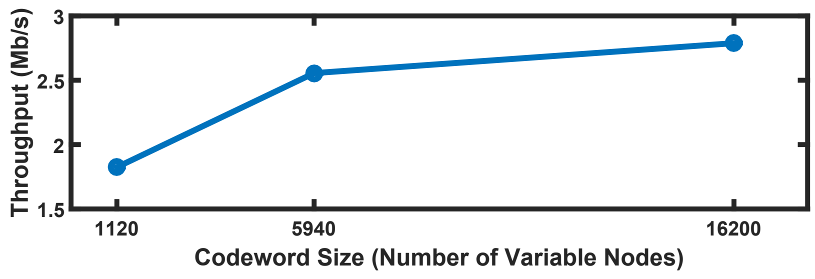


Figure 4.8: Host Synchronization (Design F) Throughput Results

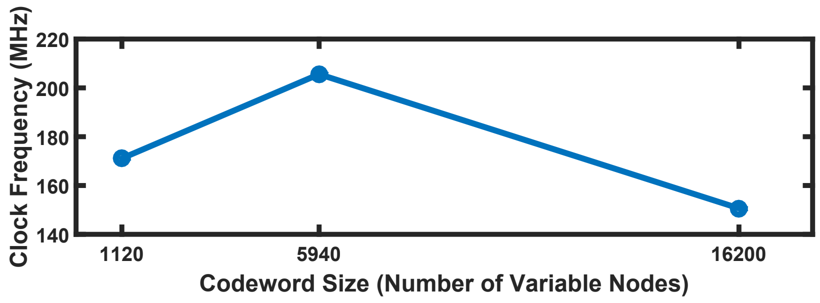


Figure 4.9: Host Synchronization (Design F) Clock Frequencies

the *num_simd_work_items* attribute) as shown in Table 4.6, however we are limited by the number of resources on the FPGA in this case.

The achieved throughput values, however, are not as high as Design D when the execution remained solely on the FPGA. Communicating via the host device increases the decoding execution time and lowers the throughput.

The source code for this design (using the length-1120 (1-5,15-16)-irregular code as an example) is attached in Appendix S.

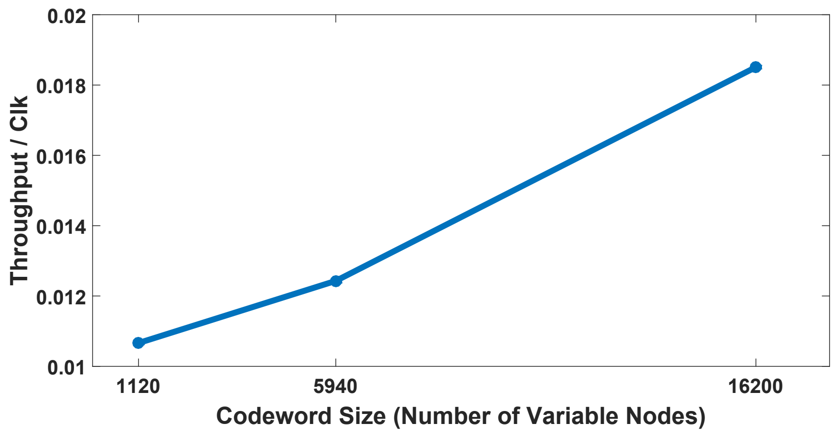


Figure 4.10: Host Synchronization (Design F) Throughput / Clock Frequency Ratio

4.5.3 Limitations

Although this structure allows the device code to be simplified and promotes heterogeneous programming, the latency associated with the constant communication with the host proved to be a bottleneck. The Altera OpenCL profiling graphical user interface (GUI) tool allows profiling-enabled compiled kernels to be investigated. Figure 4.11 shows the profiler results for the length-1120 (1-5,15-16)-irregular LDPC code. For each kernel, the blue stripes within the timeline show the active execution points and their duration. The execution points in between each active portion reveal that not only does the host synchronization add a significant amount of time, but that the kernels were also not executing simultaneously. Even without executing simultaneously, the figure shows that the majority of the execution time was taken up by the combination of the host communication and the host execution time.

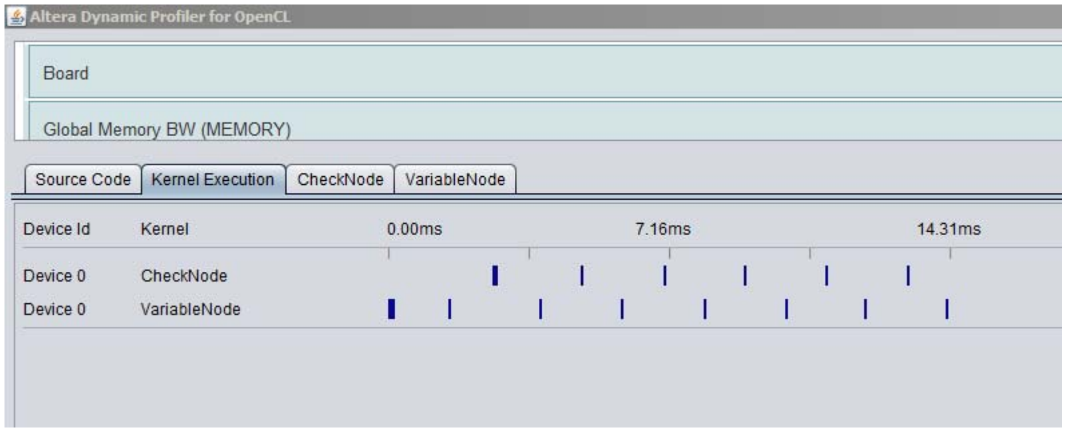


Figure 4.11: Host Synchronization Visual Profiler

4.6 Design Comparison

Comparing these designs with the results achieved for the regular decoders, it is clear to see that the decoder for irregular LDPC codes provides a significant challenge for an OpenCL-based design. The irregularity of the node fan-ins as well as the node ratios removes assumptions that could be used for optimization. For three of the main codes of the DOCSIS 3.1 [7] standard, the different designs discussed in this chapter produced varying results as summarized in Table 4.7. The highest performing design was the host controlled synchronization (Design F) at 2.79 Mbps for the length-16200 codeword. Note that all of the results for Design E (as noted by '*') are from emulation only. All of the host code is attached in Appendices J, K, and L.

Code Length	Design D Throughput (Mbps)	Design E Throughput (Mbps)	Design F Throughput (Mbps)
1120	5.62	0.86*	1.82
5940	4.58	0.29*	2.56
16200	1.95	-*	2.79

Table 4.7: Irregular Results for Designs D, E, and F

Chapter 5

Cross-Platform Portability

Chapter 2 introduced OpenCL as a high-level programming environment that claims to provide portability across heterogeneous parallel hardware platforms consisting of CPUs, GPUs, FPGA fabrics, and other hardware devices [4]. Although most vendors have their own dedicated OpenCL compilers and device drivers, OpenCL source code remains the same regardless of the target device (minus vendor-specific enhancement attributes). Although our LDPC decoder implementations were developed for the Stratix V GX A7 FPGA, the same kernel designs can be directly recompiled for any OpenCL-enabled device. To confirm this property we investigated running the designs discussed in the previous chapters on the NVIDIA GTX 690 3072 CUDA-core GPU in Hextet using the Intel SDK compiler for OpenCL version 6.1 2016 R2 [21]. Note that although we are using the Intel SDK compiler for OpenCL to compile kernels for an NVIDIA GTX 690 GPU, the device driver (and thus the interface) is provided by NVIDIA. Although using the Intel compiler doesn't allow us to use NVIDIA-specific language enhancements, the OpenCL code executed will be functionally equivalent.

Although the source code may be portable, the differences in the devices are still important constraints, especially in the maximum available memory capacities. When compiling for an FPGA, local and private memory (as discussed in Section 2.4) are implemented in on-chip block RAM. Because of this, the amounts of local and private memory are decided at compile-time allowing the compiler to configure as much memory as required subject to the available memory in the device. For fixed architecture devices, such as the GPU and the CPU, memory synthesis is not possible. Consequently in the LDPC decoder implementations, large amounts of local memory are requested for the longer codewords as we must store not only the interleaver network but also the connection information for each node. Because of these restrictions, it is not possible to port the same OpenCL code from the FPGA to the GPU for certain code scenarios.

The GPU is actually made up of two GeForce GTX 690 Kepler GPU chips (each with 1536 CUDA cores). The results from a device information query application are attached in Appendix X. The "OpenCL Device Query" program is a freely available sample program downloaded from NVIDIA directly that displays the properties of all present OpenCL devices [3]. Because the memory cannot be synthesized on the GPU hardware like in the FPGA using OpenCL, we must operate within the maximum available memory capacities for each type. Table 5.1 shows the available memory sizes for the four OpenCL memory types. Private memory is implemented in registers in the GPU cores (much like in the FPGA). The local memory is implemented on the GPU chip except it exists in the shared memory between GPU cores on the chip. Much like on the FPGA, the global memory is located off-chip in the GPU DRAM with 2 GB allocated for each of the two GPU chips. The constant memory is cached in the GPU core. Along with the memory capacity limitations, NVIDIA also restricts the allowable dimensions and number of work-items/work-groups. As introduced in Section 2.3, work-items can be organized into up to three dimensions in which each dimension is a work-group. The NVIDIA GTX 690 limits the maximum work-group size to [1024,1024,64] for dimensions [x,y,z], respectively. For example, the maximum number of work-items within a single work-group for a 1-dimensional organization is 1024. Additionally, the maximum number of work-groups is also 1024.

Memory Type	Size	Physical Location
Private Memory	64 KB	On-Chip (Registers)
Local Memory	48 KB	On-Chip (Shared Memory)
Global Memory	4 GB (2 GB/chip)	Off-Chip (GDDR5 RAM)
Constant Memory	64 KB	On-Chip (Cached)

Table 5.1: NVIDIA GTX 690 GPU Device Information

5.1 Regular LDPC Decoder Designs

5.1.1 Design A

The strategy behind Design A was to take the simulation C code and transform it into an OpenCL application using the single work-item, single-kernel, approach recommended by Altera [10]. Although the design only resulted in a throughput of 0.023 Mbps for a length-32 (3,6)-regular code, the process revealed the best path for evolving a design with higher throughput. This approach does not appear to be as successful when compiled directy onto the NVIDIA GTX 690 GPU. Figure 5.1 shows that the throughput of Design A on the GPU flattens at around 0.000350 Mbps whereas the FPGA version in Section 3.3 obtained a throughput of 0.023 Mbps for a length-32 (3,6)-regular code. That is to say that executing the decoder on the FPGA outperformed the NVIDIA GPU by a factor of over 180 times. Figure 5.3 compares the throughput from executing Design A on the FPGA versus on the GPU. The throughput of the FPGA outperformed the GPU for all of the executed codes. Note that the decoders of length-512 and higher were unable to compile for Design A on the FPGA as mentioned in Section 3.3. The limiting factor was the available FPGA logic resources on the chip. The main likely reason for the performance gap from the FPGA to the GPU is the lack of loop pipelining on the GPU. Figure 5.2 reveals a linear increase in execution time. Because of the simplicity of generating Design A from the C simulation code, the FPGA is very useful here as a rapid prototyping device that outperforms powerful SIMD GPUs.

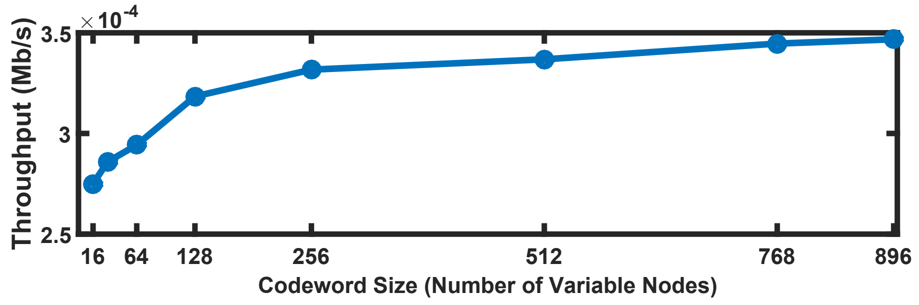


Figure 5.1: Regular Decoder Design A Throughput on the NVIDIA GPU

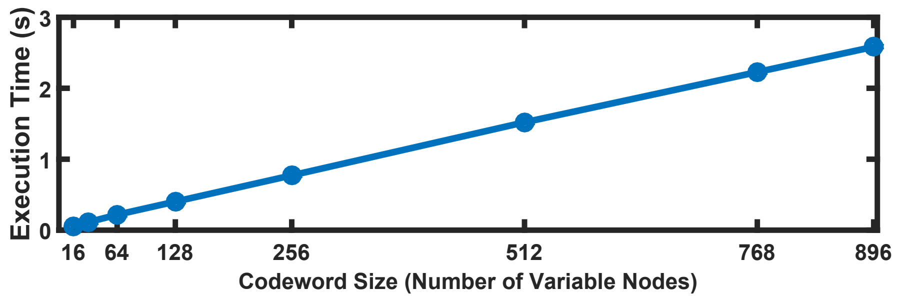


Figure 5.2: Regular Decoder Design A Execution Time on the NVIDIA GPU

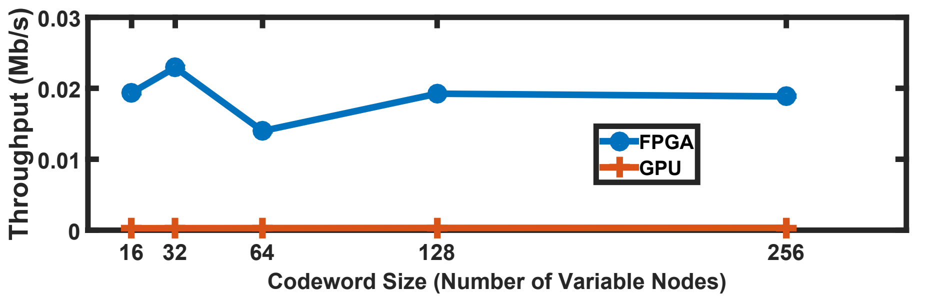


Figure 5.3: Design A Throughput FPGA vs. GPU Execution

5.1.2 Design B

The source code for Design B (given in Section 3.4 and Appendix G) was compiled and executed directly on the GPU. Figure 5.4 shows the throughput achieved from executing the kernel on the NVIDIA GPU. The decoder designs for LDPC codewords of length-512 and longer were not able to execute on the GPU. This is directly due to the required amount of local memory for the longer LDPC codewords as the GPU has a local memory size maximum of 48 KB. The maximum achieved throughput of 0.337 Mbps for a length-256 (3,6)-regular LDPC is predictably significantly lower than the 42.1 Mbps of the FPGA execution because most of the optimizations to the design were made with the FPGA target in mind. However, comparing the throughput shown in Figure 5.4 with the throughput plot for Design A in Figure 5.1, expanding the kernel design to an NDRange multiple work-item architecture now allows the GPU throughput to increase linearly. The execution time for Design B is no longer linearly increasing with the codewords thus producing higher throughput and validating that the limitation of throughput on Design A on the GPU is due to a lack of loop pipelining.

Figure 5.5 compares the throughput achieved from executing Design B on the GPU and on the FPGA. As the scaling of the plot shows, all of the throughput results from the GPU are lower than that from the FPGA by a factor of over 100 times.

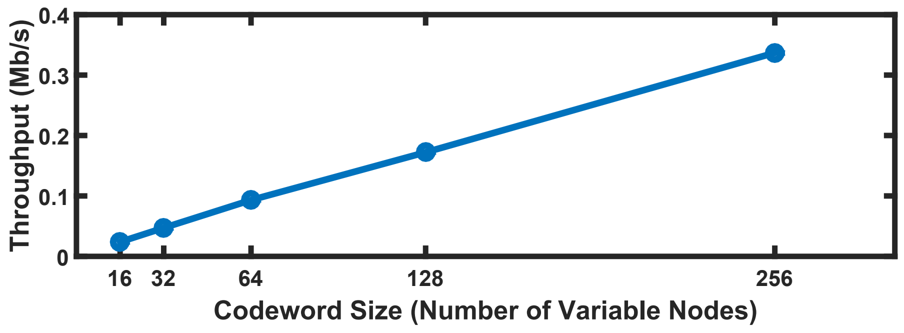


Figure 5.4: Regular Decoder Design B Throughput on the NVIDIA GPU

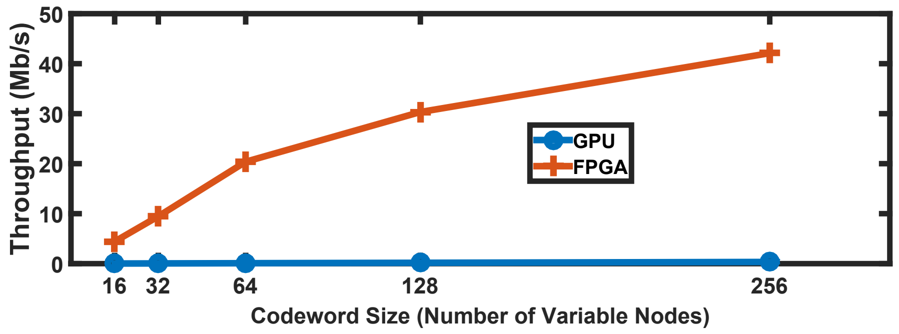


Figure 5.5: Regular Decoder Design B on the NVIDIA GPU vs. the FPGA

5.1.3 Design C

The regular decoder Design C (source code available in Appendix H) implementation makes use of the Altera OpenCL attribute *num_simd_work_items* as introduced in Section 3.5. Using this attribute when compiling the kernel for FPGAs results in the 'widening' of the hardware pipeline to accommodate more work-items that execute simultaneously. As discussed in Section 3.5, another attribute, *num_compute_units*, allows the developer to compile multiple copies of the entire hardware pipeline that execute independently of each other (illustrated in Figure 3.13).

While these attributes can greatly increase kernel performance, unfortunately, they are unique to the Altera-Offline-Compiler (AOC) for FPGAs and are not part of the OpenCL specification. Because of this, it is not possible to compile Design C using the Intel SDK for OpenCL to execute on the GPU.

5.2 Irregular LDPC Decoder Designs

5.2.1 Design D

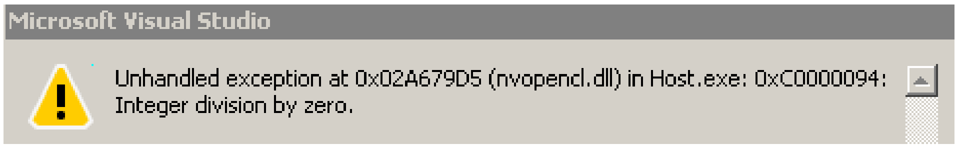
Design D (attached in Appendices M, N, and O) started once again with a single work-item approach to simplify the device code (like in Design A) since the irregular decoder designs introduced a significant new and difficult challenge due to the larger size requirements of the LDPC interleaver networks. Design D introduced the Altera OpenCL 'channel' attribute which allows kernels to communicate directly within the FPGA chip using a First-In First-Out (FIFO) buffer. However, because the channel attribute is only available with the Altera-Offline-Compiler for FPGAs, the direct source code cannot be compiled for the GPU. In the 2.0 specification, OpenCL introduced the concept of the *Pipe* which effectively provides the same functionality as the Altera channels. However, unfortunately we are once again limited as the NVIDIA GTX 690 driver only supports OpenCL 1.2.

In order to run this design on the GPU, we must slightly alter the OpenCL source code to use a different method of synchronization between the kernels. Since the OpenCL pipe is not yet supported, the next simplest method for synchronization without involving the host is to use the device global memory. By allocating a buffer in the global memory and synchronizing the kernels based its value, we can indeed compile the code and execute it on the NVIDIA GPU. This source code is attached in Appendix T for this slightly altered version.

This slightly altered source code was taken directly from the FPGA code for Design D, recompiled with the Intel SDK, and executed on the NVIDIA GPU. The result was that the program failed to terminate. Research into this 'hanging' program behaviour revealed that simultaneous kernel execution is only supported for NVIDIA devices using the OpenCL 2.0 standard [4]. As we remember from the GPU information program (results in Appendix X), the NVIDIA GTX 690 GPU driver operates using the OpenCL 1.2 specification and thus simultaneous kernel execution is not available. Since Design D uses kernel synchronization that requires both kernels to execute simultaneously (or intermittently at the very least), executing Design D on the GPU will not complete.

5.2.2 Design E

In Section 5.2.1 we encountered the problem that the current NVIDIA GTX 690 OpenCL driver is operating using the OpenCL 1.2 specification and thus does not support concurrent kernel execution. Design E has a very similar architecture as Design D in that it requires both kernels (VN and CN) to execute simultaneously or intermittently in order to properly synchronize at the beginning of each decoding iteration. Because of this requirement, it is again not possible to execute Design E


Microsoft Visual Studio

```
Unhandled exception at 0x02A679D5 (nvopencl.dll) in Host.exe: 0xC0000094:
Integer division by zero.
```

Figure 5.6: Design F GPU Execution Error

on the GPU to completion.

5.2.3 Design F

Fortunately, for the investigation on OpenCL portability described in this chapter, the method of synchronization was also a problem for Design E on the FPGA in Section 4.4. With the complexity of synchronization moved to the control of the host, we are able to compile and execute the design on the GPU without any source code alterations with the only difference in the organization of work-items within a work-group. While the FPGA version launched all of the work-items in a single work-group to make use of the synthesized local memory, the maximum number of work-items in a work-group the GPU is 1024.

Unfortunately, due to reasons hidden by the GPU driver, the execution on the GPU fails just prior to launching the kernels onto the device. The error code, attached as Figure 5.6, appears to be an issue within the GPU driver itself. Upon further research, the issue was empirically determined to be due to the private memory requirements. The kernels internally require more than the available 64 KB in registers for private memory. Without a significant change to the memory structure, the design is unfortunately unable to execute on the GPU.

Chapter 6

Contributions and Future Work

The objective of this thesis was to evaluate the ability of the Altera-Offline-Compiler for OpenCL [10] to produce efficient flexible high-throughput LDPC decoders that make effective use of the available types of parallelism. The LDPC decoder algorithm produced a challenging and instructive benchmark problem with several different potentially exploitable forms of parallelism for both the regular and irregular LDPC codes. Clarifying the various design trade-offs was a specific objective of this project. Compared with other custom hardware and software techniques, the high-level synthesis tool OpenCL allowed reasonable performance to be achieved with a significantly shorter design time and much lower development risk. While the compiler was able to efficiently produce much of the hardware pipeline parallelism (such as those in Designs A and D), further design changes were required to increase performance past the recommended practices in the Altera Optimization Guide [9].

6.1 Conclusions

This thesis introduced three separate designs (Designs A, B, and C) for a regular LDPC decoder in Chapter 3. By investigating the three other types of parallelism shown in Table 3.6, and by experimenting past the recommended practices in the Altera Optimization Guide [9], an efficient high-thoughput decoder resulted. By combining hardware pipelining, vectorization, and replication in the final regular LDPC design (C), we were able to achieve a maximum throughput of 68.22 Mbps for a random length-2048 (3,6)-regular LDPC code. For a length-1024 (3,6)-regular code, Falcao et al. reported a throughput of around 7 Mbps (with 30 decoding iterations) for an OpenCL-based FPGA-targetted decoder [18] whereas our final Design C achieved a throughput of 54.8 Mbps (with 32 decoding iterations) for the same code size, as shown in Figure 3.14. Falcao used the Xilinx Virtex-6 LX760 FPGA (which is similar in size to the Altera Stratix V A7) with the SOpenCL for FPGA compiler.

Comparing our designs with fully parallel ASIC solutions, Blanksby et al. reported a throughput of 1 Gbps for a length-1024 (3,6)-regular LDPC code [16] whereas our equivalent decoder achieved only 54.1 Mbps. The HDL implementation by Karkooti et al. where their length-1535 (3,6)-regular VHDL LDPC decoder achieved 127 Mbps [22] and our equivalent decoder achieved just over 62 Mbps. Another HDL design by Zhang et al. for a length-9216 (3,6)-regular LDPC decoder design produced 54 Mbps [35]. Although our decoder was not able to synthesize for any LDPC codes longer than length-2048, our throughput at the length-2048 achieved 68.2 Mbps. Even though our decoder throughput is not generally comparable to the ASIC and FPGA HDL implementations, the design time for our OpenCL implementation is significantly shorter. Although no direct design time comparison were made in this thesis, Pratas et al. surveyed design times for a series of techniques [27]. Table 6.1 estimates the achieved throughput / capital cost ratio for three types of LDPC implementations: OpenCL, FPGA HDL, and ASIC. Although the throughput for the FPGA HDL and OpenCL implementations is much lower than that of the ASIC, the hardware costs and design times are significantly smaller. Note that the throughput values in this table for the OpenCL, HDL, and ASIC implementations were taken from the decoder sizes shown in Table 6.2.

Chapter 4 extended the regular LDPC decoder in Design C into three more designs (Designs D, E, and F) that decoded irregular codes, which are often preferred over regular codes in industrial applications. The chapter evaluated designs for three of the longer LDPC codes of the DOCSIS 3.1 [7] standard. By taking a similar design approach from that of the regular LDPC decoder chapter, the first design used the

Implementation	Hardware Cost	Design Time	Throughput (Mbps)	Throughput / Cost
OpenCL	$\sim \$5,000$	Days to Weeks (Shortest)	68.2	0.0136 Mbps/\$1
FPGA HDL	$\sim \$5,000$	Weeks to Months (Medium)	127	0.0254 Mbps/\$1
ASIC (first prototype)	$\sim \$10M+$	Months to Years (Longest)	13210	0.00132 Mbps/\$1
ASIC (production)	$\sim \$10$	Months to Years (Longest)	13210	1321 Mbps/\$1

Table 6.1: Throughput Cost Analysis

Implementation	LDPC Code	Source Citation
OpenCL	Length-2048 (3,6)-regular	This Thesis
FPGA HDL	Length-1536 (3,6)-regular	Karkooti et al. [22]
ASIC	Length-1024 (3,6)-regular	Onizawa et al. [26]

Table 6.2: LDPC Codes Used in Throughput Cost Analysis

recommended single work-item format. However, the complexity of the irregular codes was such that the synthesized designs were not able to fit into a similar design and a new communication approach was investigated. The single work-item design (Design D) used the global memory (external SDRAM) as the interleaver between the VNs and CNs. In order to allow more codewords to be simultaneously processed, a multiple kernel (one for the VNs and one for the CNs) was developed such that the kernels used two data buffers in the global memory to operate simultaneously. Communication between the two kernels was doing using the built-in Altera channel OpenCL extension. This design produced a maximum total codeword throughput of 5.4 Mbps with the length-1120 (1-5,15-16)-irregular LDPC code. Following the same design decisions as in Chapter 3, the single work-item approach was expanded to an NDRange approach. Unfortunately due to an unknown deadlock issue with the communication and synchronization (discussed in Section 4.4), the decoder was unable to successfully terminate on the FPGA. In an attempt to move the complexity of the communication from the OpenCL kernel to the host, Design F was developed. This design required each iteration to be started by the OpenCL host ensuring that both kernels fully complete before continuing which got rid of the need for the Altera OpenCL channels. This final design produced a maximum codeword throughput of 2.79 Mbps for a length-16200 (1-5,33-34)-irregular LDPC decoder. It was discovered in the Altera profiling GUI that the host communication formed the majority of the execution time. The DOCSIS 3.1 standard requires that the Quasi-Cyclic LDPC (QC-LDPC) upstream codes, detailed in Table 4.1, must have a throughput of at least 1 Gbps [7]. Although the largest produced throughput was only 2.79 Mbps for a length-16200 (1-5,33-34)-irregular LDPC code which does not meet the DOCSIS 3.1 requirement, the design proves that high-level synthesis tools for OpenCL can be used for the industrial LDPC codes.

Chapter 5 briefly investigated the portability aspect of OpenCL code by providing the source code to the Intel SDK compiler for execution on Hextet's NVIDIA GTX 690 GPU. The execution of Design A on Hextet's NVIDIA GTX 690 GPU produced a maximum throughput of 0.000350 Mbps whereas the same design on the FPGA obtained a throughput of 0.023 Mbps: the difference of a factor of 180 times. This difference is likely due to the fact that the FPGA compiler pipelines the loop iterations for a single work-item design whereas the same is not true for the GPU compiler. The throughput of Design B for the GPU was a maximum of 0.337 Mbps for the length-256 (3,6)-regular LDPC decoder whereas the same design on the FPGA achieved 42.1 Mbps of codeword throughput. Since the pipeline replication present in Design C on the FPGA was controlled by the Altera OpenCL extension attribute *num_simd_work_items*, there is no difference between Design B and Design C when execute on the GPU. Design D for the irregular DOCSIS 3.1 QC-LDPC codes made use of the Altera OpenCL extension channels which are not available on the GPU compiler. In the 2.0 OpenCL specification, the concept of the *Pipe* was introduced which provides the effectively the same functionality as the Altera channels, however the NVIDIA GTX 690 is limited to support OpenCl 1.2. Due to the lack of requirement in the OpenCL 1.2 specification for kernels to be executed simultaneously in hardware, it was determined that designs E and F were also unable to execute on the GPU. The design predictably produced significantly lower throughput compared to the FPGA version as many optimizations had been made for the target FPGA device. The notion of portability allows developers to create heterogeneous platforms which harness multiple OpenCL devices for even more complex computations. With the updating of drivers and introduction of more OpenCL features, portability is being more and more widely supported.

With the introduction of larger and more powerful OpenCL-enabled FPGAs and compiler upgrades in the future, all the results reported above should only increase. This makes the source code even more valuable as it can be recompiled and used taking advantage of newly available hardware.

6.2 Contributions of this Thesis

Both Chapters 3 and 4 provided insight into software architecture designs of LDPC decoders on FPGAs using OpenCL. For both the regular and irregular first designs, the approach of developing a single work-item design proved extremely helpful. This step gave valuable information with the Altera Optimization Report and was the closest to the serial C simulation source code. In the LDPC case, both of the first designs showed that using the NDRange type kernels with multiple work-items would produce higher throughputs. Using the available Altera attribute *num_simd_work_items* in combination with the OpenCL vector data types produced efficient hardware pipelines for the NDRange kernels.

As a starting point for FPGA-based OpenCL designs, it was determined that the best designs kept the data and execution paths in the FPGA chip as much as possible. I recommend pre-loading all of the data onto the FPGA chip and using a combination of local and private memory whenever possible. The use of constant memory can also be highly beneficial so long as the compiled constant memory cache size is sufficient to hold all of the data and avoid cache misses.

Section 3.2.1 discussed an experiment where branching source code was evaluated against vectorized source code. The vectorized source code used a masking equation and vector operations to obtain the same results as branching for each individual vector element. As expected, the vectorized approach had the shortest execution time on both the FPGA and the GPU. The branching source code, however, was a close second on the FPGA executing only 1% slower than the vectorized source code. This result was not the same for the GPU as the branching execution took over 7600 times longer (17.9925 second vs 0.00236 seconds). The similar execution time for the branching code on the FPGA reveals a strength for MIMD operations. This is because the FPGA can simply generate hardware pipelines for all execution paths whereas the GPU must attempt to predict the execution path.

6.3 Future Work

Future work for these designs will include generating a pipeline structure to stream codewords into and out of the OpenCL device either via the host or a direct I/O connection. The resulting vectorization of the decoder should be used as a "ferris wheel" type pipeline in that codewords could be sent back to the host at every decoding iteration. In this design, at every decoding iteration the next codeword location in line would be removed and re-loaded with the next codeword. Once the "ferris wheel" is completely full each decoding iteration would involve the removal of decoded codeword and the entrance of a new one. This would allow for the continuous flow of codewords and thereby increase the throughput by reducing or completely hiding the overhead introduced by the setup of the decoder implementation.

For the DOCSIS 3.1 QC-LDPC codes, a flexible decoder design could be investigated which is able to handle all three of the mentioned codes (in Table 4.1). The design would consist of the compiled hardware pipeline for the longest of the three codes (length-16200). When necessary, the new interleaver connection lookup tables could be sent to the FPGA so that it can handle one of the two other LDPC code sizes (length-5940 or length-1120). Since the hardware pipeline doesn't require reconfiguring between the three QC-LDPC code sizes, the time required to change between codewords would be quite short as the host only needs to transfer the new lookup tables to the device.

The work with the OpenCL framework in this thesis leads to an emphasis on recommending careful planning and adaptation of the memory model for OpenCL and how it maps to each individual OpenCL device. We found that the Altera AOC optimization report provides a great resource for determining performance bottlenecks including the efficiency of memory transfers. Further use of the newest OpenCL features, such as pipes and Altera channels, should be investigated. With the release of larger FPGAs, the source code could first be directly taken, recompiled, and executed on the device. Since there are more resources available, this would allow the decoder to be compiled for longer LDPC codes and would result in more throughput achieved. A larger FPGA would allow the interleaver networks for multiple LDPC codes to be stored simultaneously avoiding the need to re-download them from the host.

As OpenCL-to-FPGA compiler technology improves, the performance potential of FPGA hardware will be better unlocked to complement the parallelism of CPUs and GPUs.

Bibliography

[1] "NVIDIA CUDA reference manual," NVIDIA Corporation, Tech. Rep., 2007.

[2] "Long-Term Evolution (LTE)," 3rd Generation Partnership Project (3GPP), 3GPP Mobile Communications Standard Release 8, Dec. 2008.

[3] "OpenCL Programming Guide for the CUDA Architecture v2.3," NVIDIA Corporation, Tech. Rep., 2008.

[4] "The OpenCL specification," Khronos OpenCL Working Group, Tech. Rep., Dec. 2008.

[5] "Jacket - GPU Engine for MATLAB Getting Started Gude," AccelerEyes, 75 5th St NW STE 204, Atlanta, GA, 30308, USA, Tech. Rep., April 2010.

[6] "DVB fact sheet: 2nd generation satellite DVB-S2 (EN 302 307)," DVB Project, Geneva, Switzerland, Standard, Aug. 2012.

[7] "Data-over-cable service interface specifications DOCSIS(R) 3.1: Physical layer specification CM-SP-PHYv3.1-I01-131029," Cable Television Laboratories, Louisville, CO, USA, Standard, Oct. 2013.

[8] "OpenMP Application Program Interface Version 4.0," OpenMP ARB, Tech. Rep., July 2013.

[9] "Altera SDK for OpenCL best practices guide," Altera, Programming Reference, May 2015.

[10] "Altera SDK for OpenCL getting started guide," Altera, Standard Release, November 2015.

[11] "MPI: A Message-Passing Interface Standard Version 3.1," Message Passing Interface Forum, Tech. Rep., June 2015.

[12] "Nallatech OpenCL FPGA accelerator cards," Nallatech, Tech. Rep., Oct. 2015.

[13] "Stratix V device overview," Altera, Tech. Rep., Oct. 2015.

[14] K. Abburi, "A scalable LDPC decoder on GPU," in *VLSI Design (VLSI Design), 2011 24th International Conference on*, Jan 2011, pp. 183–188.

[15] J. Andrade, G. Falcao, and V. Silva, "Flexible design of wide-pipeline-based WiMAX QC-LDPC decoder architectures on FPGAs using high-level synthesis," *Electronics Letters*, vol. 50, no. 11, pp. 839–840, May 2014.

[16] A. J. Blanksby and C. J. Howland, "A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder," *Solid-State Circuits, IEEE Journal of*, vol. 37, no. 3, pp. 404–412, 2002.

[17] T. Brandon, R. Hang, G. Block, V. C. Gaudet, B. Cockburn, S. Howard, C. Giasson, K. Boyle, P. Goud, S. S. Zeinoddin *et al.*, "A scalable LDPC decoder ASIC architecture with bit-serial message exchange," *Integration, the VLSI Journal*, vol. 41, no. 3, pp. 385–398, 2008.

[18] G. Falcao, M. Owaida, D. Novo, M. Purnaprajna, N. Bellas, C. Antonopoulos, G. Karakonstantis, A. Burg, and P. Ienne, "Shortening design time through multiplatform simulations with a portable OpenCL golden-model: The LDPC decoder case," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, April 2012, pp. 224–231.

[19] G. Falcão, L. Sousa, and V. Silva, "Massive parallel LDPC decoding on GPU," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, 2008, pp. 83–90.

[20] K. Hill, S. Craciun, A. George, and H. Lam, "Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA," in *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, July 2015, pp. 189–193.

[21] E. L. (Intel), "Developer guide for intel sdk for opencl," Tech. Rep., 2016.

[22] M. Karkooti and J. R. Cavallaro, "Semi-parallel reconfigurable architectures for real-time LDPC decoding," in *Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on Information Technology*, vol. 1. IEEE, 2004, pp. 579–585.

[23] E. Liao, E. Yeo, and B. Nikolic, "Low-density parity-check code constructions for hardware implementation," in *Communications, 2004 IEEE International Conference on*, vol. 5, June 2004, pp. 2573–2577, Vol. 5.

[24] D. MacKay and E.-C. Codes, "David mackays gallager code resources," *Dostupnỳ z URL:¡ http://www. inference. phy. cam. ac. uk/mackay/CodesFiles. html*, 2009.

[25] Mittal, Sparsh and Vetter, Jeffrey S, "A survey of CPU-GPU heterogeneous computing techniques," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, p. 69, 2015.

[26] N. Onizawa, T. Hanyu, and V. C. Gaudet, "Design of high-throughput fully parallel ldpc decoders based on wire partitioning," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 18, no. 3, pp. 482–489, 2010.

[27] F. Pratas, J. Andrade, G. Falcao, V. Silva, and L. Sousa, "Open the gates: Using high-level synthesis towards programmable LDPC decoders on FPGAs," in *Global Conference on Signal and Information Processing (GlobalSIP), 2013 IEEE*, Dec 2013, pp. 1274–1277.

[28] J. Proakis and M. Salehi, *Digital Communications*, 5th ed. McGraw-Hill Education, 2007.

[29] C. B. Schlegel and L. C. Perez, *Trellis and Turbo Coding: Iterative and Graph-Based Error Control Coding.* John Wiley & Sons, 2015.

[30] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, pp. 379–423, 623–656, July 1948.

[31] Sharma, Gaurav and Martin, Jos, "MATLAB®: a language for parallel computing," *International Journal of Parallel Programming*, vol. 37, no. 1, pp. 3–36, 2009.

[32] L. Shu and D. J. Costello, *Error Control Coding*, 2nd ed. Pearson Education India, 2004.

[33] S. Tehrani, S. Mannor, and W. Gross, “Fully parallel stochastic LDPC decoders,” *Signal Processing, IEEE Transactions on*, vol. 56, no. 11, pp. 5692–5703, Nov 2008.

[34] P. Urard, E. Yeo, L. Paumier, P. Georgelin, T. Michel, V. Lebars, E. Lantreibecq, and B. Gupta, “A 135Mb/s DVB-S2 compliant codec based on 64800b LDPC and BCH codes,” in *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, Feb 2005, pp. 446–609, Vol. 1.

[35] T. Zhang and K. K. Parhi, “A 54 mbps (3, 6)-regular FPGA LDPC decoder,” in *2002.(SIPS'02). IEEE Workshop on Signal Processing Systems.* IEEE, 2002, pp. 127–132.

[36] Z. Zhang, V. Anantharam, M. Wainwright, and B. Nikolic, “An efficient 10GBASE-T Ethernet LDPC decoder design with low error floors,” *Solid-State Circuits, IEEE Journal of*, vol. 45, no. 4, pp. 843–855, April 2010.

Appendix A

Regular Decoder Host Source Code (C)

```
// This is the start of the Channel LDPC implementation
#include <iostream>
#include <fstream>
#include <ctime>
#include <math.h>
#include <malloc.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "E:\Andrew\Projects\common\ax_common.h"




#define ALTERA

#define MANUAL_VECTOR_WIDTH 16

int N_VARIABLE_NODES = 2048;
int N_CHECK_NODES =(N_VARIABLE_NODES/2);
#define MAX_VARIABLE_EDGES 3
#define MAX_CHECK_EDGES 6

#define PROJECT_TITLE "Single Kernel Vector LDPC"
#define PROJECT_DESCRIPTION "This host code supports the single kernel implementation (
    ↪ single kernel multiple work−item) of the LDPC decoder using the char4 data type.
    ↪ Within the char4 data type, each of the chars will contain a different codeword to
    ↪  decode"


// These options apply to the code being used as well as the number of iterations to do
int LDPC_SIZE = N_VARIABLE_NODES; // Note this is overwritten when a value is passed in
    ↪ via command line

#define FILENAME "matrix_"
#define OUTPUT_FILE "results_"
#define N_ITERATIONS 32

#define N_VARIABLE_INPUTS MAX_VARIABLE_EDGES
#define N_CHECK_INPUTS MAX_CHECK_EDGES


#define N_VARIABLE_INPUTS_W_CHANNEL (N_VARIABLE_INPUTS+1)


#define MAX_LINE 1024

static char * INPUT_FILENAME = (char*) malloc(sizeof(char)*MAX_LINE);
static char * OUTPUT_FILENAME = (char*) malloc(sizeof(char)*MAX_LINE);

#if MANUAL_VECTOR_WIDTH==2
#define LLR_TYPE cl_char2
#define TOTAL_TYPE cl_short2
#define CONVERT_TO_SHORT(a) convert_short2(a)
#define CONVERT_TO_CHAR(a) convert_char2(a)
#elif MANUAL_VECTOR_WIDTH==4
#define LLR_TYPE cl_char4
#define TOTAL_TYPE cl_short4
#define CONVERT_TO_SHORT(a) convert_short4(a)
#define CONVERT_TO_CHAR(a) convert_char4(a)
#elif MANUAL_VECTOR_WIDTH==8
#define LLR_TYPE cl_char8
```

```
#define TOTAL_TYPE cl_short8
#define CONVERT_TO_SHORT(a) convert_short8(a)
#define CONVERT_TO_CHAR(a) convert_char8(a)
#elif MANUAL_VECTOR_WIDTH==16
#define LLR_TYPE cl_char16
#define TOTAL_TYPE cl_short16
#define CONVERT_TO_SHORT(a) convert_short16(a)
#define CONVERT_TO_CHAR(a) convert_char16(a)
#else
#define LLR_TYPE cl_char
#define TOTAL_TYPE cl_short
#define CONVERT_TO_SHORT(a) convert_short(a)
#define CONVERT_TO_CHAR(a) convert_char(a)
#endif


// Integer options
#define LLR_BASE_TYPE char
#define N_BITS_LLR (sizeof(LLR_BASE_TYPE)*8) // 8 bits per byte
#define MAX_LLR ((1 << (N_BITS_LLR-1))-1)
#define SIGN_BIT (MAX_LLR+1)
#define MAX_NEG_LLR (0-MAX_LLR)
typedef LLR_TYPE LLR;

LLR vector_max_llr;
LLR vector_min_neg_llr;
LLR vector_n_bits_llr;
LLR vector_sign_bit;
LLR vector_too_neg_llr;
LLR allones;
LLR ones;
LLR zeros;
TOTAL_TYPE total_vector_max_llr;
TOTAL_TYPE total_vector_min_neg_llr;


#define BINS_PER_ONE_STEP 4
#define SATURATION_POINT int(MAX_LLR/BINS_PER_ONE_STEP)



//#define PRINT_DEBUG
//#define QUICK_DEBUG

#ifdef PRINT_DEBUG
#undef QUICK_DEBUG
#endif

#define DEBUG_FILENAME "output_debug_integer.csv"
#define CHECKNODE_DEBUG_FILENAME "checknode_debug.csv"
#define VARIABLENODE_DEBUG_FILENAME "variablenode_debug.csv"




#pragma region OpenCL Specific Declarations
static char * KERNEL_FILENAME  = (char*) malloc(sizeof(char)*MAX_LINE);
static char const* COMPILER_OPTIONS = "-w";


#pragma endregion

#define DEBUG 1
#define AOCL_ALIGNMENT 64





#pragma region Error Checking Macros
#define SYSTEM_ERROR -1
#define CheckPointer(pointer) CheckPointerFunc(pointer,__FUNCTION__, __LINE__)
#define ExitOnErrorWMsg(msg) ExitOnErrorFunc(msg,__FUNCTION__,__LINE__)
#define ExitOnError() ExitOnErrorWMsg("Error! Please refer to")
#pragma endregion

#pragma region Structure Defintions
// This structure holds all data required for the variable nodes.
// Note that the last input will be the channel input for each variable node
// the n_inputs will always be n_checknodes+1
typedef struct VariableNodes
{
  LLR inputs[MAX_VARIABLE_EDGES+1];
  cl_short checknode_indexes[MAX_VARIABLE_EDGES];
  cl_short index;
  cl_short n_inputs;
  cl_short n_checknodes;
}VariableNode;

// This structure holds all data required for the check nodes
typedef struct CheckNodes
{
  LLR inputs[MAX_CHECK_EDGES];
  cl_short variablenode_indexes[MAX_CHECK_EDGES];
  cl_short index;
  cl_short n_inputs;
}CheckNode;
#pragma endregion

#pragma region Function Declaractions
static void CheckPointerFunc(void * pointer,char const* const func,const int line);
static void ExitOnErrorFunc(char const* msg, char const* func, const int line);
static void readParityFile(VariableNode*&, CheckNode*&, int&, int&);
static bool checkIfSatisfied(LLR * parity);
static void updateParity(CheckNode *& c_nodes, const int n_checks);
static LLR * convertCodewordToBinary(VariableNode*&,const int n_vars);
static float* getCodeword(int size);
static LLR_BASE_TYPE * convertToLLR(const float * codeword, const int n_vars);
static void printCodeword(const int * codeword, const int n_vars);
static void printResults(LLR * results);
static void printResultsToFile(double overall, double variable, double check, double misc)
    ↪ ;
static void setFileNames();

double generate_random_gaussian();
double generate_random_number();
void introduceNoise(float*,int,int);

short * generate_VNtoCN_index_table(VariableNode* ,CheckNode* );
short * generate_CNtoVN_index_table(VariableNode* ,CheckNode* );



static void startOpenCL(cl_context,cl_device_id device, cl_program program);
static void startSimulationOpenCL(cl_context, cl_device_id,cl_program,VariableNode*&,
    ↪ CheckNode*&,const int, const int);
static void ldpcDecodeOpenCL(cl_context, cl_command_queue,cl_kernel,VariableNode*&,
    ↪ CheckNode*&,const int, const int,float*);
void initializeNodes(VariableNode*&, CheckNode*&,const LLR*,const int, const int);
void defineGlobalDefinitions();

void parseProgramInputs(int,char*[]);
#pragma endregion






int main(int argc, char * argv[])
{
  printf("%s\n",PROJECT_TITLE);
  printf("%s\n",PROJECT_DESCRIPTION);

  parseProgramInputs(argc,argv);

  // Let's do a quick check on the number
  if(LDPC_SIZE <=0) {
    printf("Codelength␣(number␣of␣variable␣nodes)␣is␣not␣valid\n");
    ExitOnError();
  }

  // Set the file names for the input and output
  setFileNames();


  // Set up the program
  cl_context context;
  cl_device_id device;
  cl_program program;
  cl_device_type type = CL_DEVICE_TYPE_ALL;

  printf("\nOpenCL␣API\n\n");
  printf("Max␣LLR␣=␣%d\n",MAX_LLR);


  // Let's do this
  AxCreateContext(type,AX_PLATFORM_STRING,&context,&device);
#ifdef ALTERA
  AxBuildProgramAltera(&program,device,context,KERNEL_FILENAME,COMPILER_OPTIONS);
#else
  AxBuildProgram(&program,device,context,KERNEL_FILENAME,COMPILER_OPTIONS);
#endif

  // Now let's start the rest of the program
  startOpenCL(context,device,program);

  clReleaseProgram(program);
  clReleaseContext(context);


}
```

```
// Start the main chunk of the host code
void startOpenCL(cl_context context, cl_device_id device, cl_program program)
{

  defineGlobalDefinitions();
  // Declare the variables
  // Declare an array to hold the checknodes
  CheckNode * c_nodes=NULL;

  // Declare an array to hold the variablenodes
  VariableNode * v_nodes=NULL;

  // The number of check nodes and variable nodes
  int n_checks=0;
  int n_vars=0;

  // Let's read the parity file to get the nodes
  readParityFile(v_nodes,c_nodes,n_vars,n_checks);
  printf("n_inputs␣=␣%d\n",c_nodes[6].n_inputs);

  // Let's just do a few quick checks to make sure that the read worked properly
  // We should check the c_nodes and v_nodes pointers
  // We also need to check that the number of variable nodes and check nodes are valid
  CheckPointer(c_nodes);CheckPointer(v_nodes);
  if(n_vars <= 0 || n_checks <= 0)
    ExitOnErrorWMsg("Number␣of␣variables␣and␣check␣nodes␣not␣valid.");

  printf("\n**********************************************\n");
  printf("Variable␣Nodes␣%d,␣Check␣Nodes␣%d\n",n_vars,n_checks);
  printf("%d␣Log␣Likelihood␣Ratio␣Bits\n",N_BITS_LLR);
  printf("Max␣Variable␣Edges␣%d,␣Max␣Check␣Edges␣%d\n",MAX_VARIABLE_EDGES,MAX_CHECK_EDGES)
      ↪ ;

  // Now we are ready to start our simulation
  startSimulationOpenCL(context,device,program,v_nodes,c_nodes,n_vars,n_checks);

}

void startSimulationOpenCL(cl_context context, cl_device_id device,cl_program program,
    ↪ VariableNode*& v_nodes,CheckNode*& c_nodes, const int n_vars, const int n_checks)
{

  // Firstly, we need to get our codeword to decode
  float * codeword = getCodeword(n_vars);


  cl_int err;
  // Let's create our openCL stuff

  // Now we need to create our command queues
  // We will create one command queue for each kernel
  cl_command_queue LDPCQueue = clCreateCommandQueue(context,device,
      ↪ CL_QUEUE_PROFILING_ENABLE,&err);AxCheckError(err);


  // Now we need to create the kernels themselves.
  cl_kernel LDPCKernel = clCreateKernel(program,"LDPCDecoder",&err);AxCheckError(err);


  // Now let's simply start the LDPC
  ldpcDecodeOpenCL(context, LDPCQueue, LDPCKernel,v_nodes,c_nodes,n_vars,n_checks,codeword
      ↪ );
}

void copyCodewordFloat(float *& source, float *& dest, int n_vars) {

  for(int i=0; i<n_vars; i++) {
    dest[i] = source[i];
  }
}

void copyCodewordLLRBase(LLR_BASE_TYPE *& source, LLR_BASE_TYPE *& dest, int n_vars) {
  for(int i=0; i<n_vars; i++) {
    dest[i] = source[i];
  }
}

void ldpcDecodeOpenCL(cl_context       context,
            cl_command_queue    LDPCQueue,
            cl_kernel         LDPCKernel,
            VariableNode*&     v_nodes,
            CheckNode*&       c_nodes,
            const int         n_vars,
            const int         n_checks,
            float*          codeword)
{
  cl_int err;


  // Note this is where I would initiate an LLR for fixed integer
```

```
LLR * codeword_LLR = (LLR*) _aligned_malloc(sizeof(LLR) * N_VARIABLE_NODES,
     ↪ AOCL_ALIGNMENT);


float * codeword0 = (float*) malloc(sizeof(float) * n_vars);
float * codeword1 = (float*) malloc(sizeof(float) * n_vars);
float * codeword2 = (float*) malloc(sizeof(float) * n_vars);
float * codeword3 = (float*) malloc(sizeof(float) * n_vars);
float * codeword4 = (float*) malloc(sizeof(float) * n_vars);
float * codeword5 = (float*) malloc(sizeof(float) * n_vars);
float * codeword6 = (float*) malloc(sizeof(float) * n_vars);
float * codeword7 = (float*) malloc(sizeof(float) * n_vars);

float * codeword8 = (float*) malloc(sizeof(float) * n_vars);
float * codeword9 = (float*) malloc(sizeof(float) * n_vars);
float * codeworda = (float*) malloc(sizeof(float) * n_vars);
float * codewordb = (float*) malloc(sizeof(float) * n_vars);
float * codewordc = (float*) malloc(sizeof(float) * n_vars);
float * codewordd = (float*) malloc(sizeof(float) * n_vars);
float * codeworde = (float*) malloc(sizeof(float) * n_vars);
float * codewordf = (float*) malloc(sizeof(float) * n_vars);


copyCodewordFloat(codeword,codeword0,n_vars);
copyCodewordFloat(codeword,codeword1,n_vars);
copyCodewordFloat(codeword,codeword2,n_vars);
copyCodewordFloat(codeword,codeword3,n_vars);
copyCodewordFloat(codeword,codeword4,n_vars);
copyCodewordFloat(codeword,codeword5,n_vars);
copyCodewordFloat(codeword,codeword6,n_vars);
copyCodewordFloat(codeword,codeword7,n_vars);

copyCodewordFloat(codeword,codeword8,n_vars);
copyCodewordFloat(codeword,codeword9,n_vars);
copyCodewordFloat(codeword,codeworda,n_vars);
copyCodewordFloat(codeword,codewordb,n_vars);
copyCodewordFloat(codeword,codewordc,n_vars);
copyCodewordFloat(codeword,codewordd,n_vars);
copyCodewordFloat(codeword,codeworde,n_vars);
copyCodewordFloat(codeword,codewordf,n_vars);






// Now we need to introduce some noise into our codeword
// Note that this function changes our codeword input variable
// But let's add noise individually to each one
introduceNoise(codeword0,5,n_vars);
introduceNoise(codeword1,20,n_vars);
introduceNoise(codeword2,-2,n_vars);
introduceNoise(codeword3,-12,n_vars);
introduceNoise(codeword4,10,n_vars);
introduceNoise(codeword5,20,n_vars);
introduceNoise(codeword6,-2,n_vars);
introduceNoise(codeword7,-12,n_vars);

introduceNoise(codeword8,10,n_vars);
introduceNoise(codeword9,20,n_vars);
introduceNoise(codeworda,-2,n_vars);
introduceNoise(codewordb,-12,n_vars);
introduceNoise(codewordc,10,n_vars);
introduceNoise(codewordd,20,n_vars);
introduceNoise(codeworde,-2,n_vars);
introduceNoise(codewordf,-12,n_vars);

for(int i=0; i<n_vars; i++) {
  printf("codeword[%d]␣=␣%f␣␣%f␣␣%f␣␣%f\n",i,codeword0[i],codeword1[i],codeword2[i],
       ↪ codeword3[i]);
}

// Now I need to add in the step to conver the number to an integer
char * llr0 = convertToLLR(codeword0,n_vars);
char * llr1 = convertToLLR(codeword1,n_vars);
char * llr2 = convertToLLR(codeword2,n_vars);
char * llr3 = convertToLLR(codeword3,n_vars);
char * llr4 = convertToLLR(codeword4,n_vars);
char * llr5 = convertToLLR(codeword5,n_vars);
char * llr6 = convertToLLR(codeword6,n_vars);
char * llr7 = convertToLLR(codeword7,n_vars);

char * llr8 = convertToLLR(codeword8,n_vars);
char * llr9 = convertToLLR(codeword9,n_vars);
char * llra = convertToLLR(codeworda,n_vars);
char * llrb = convertToLLR(codewordb,n_vars);
char * llrc = convertToLLR(codewordc,n_vars);
char * llrd = convertToLLR(codewordd,n_vars);
char * llre = convertToLLR(codeworde,n_vars);
char * llrf = convertToLLR(codewordf,n_vars);
```

```
  for(int i=0; i<n_vars; i++) {
#if MANUAL_VECTOR_WIDTH == 1
    codeword_LLR[i]= llr0[i];
#else
    for(int j=0; j<MANUAL_VECTOR_WIDTH; j++) {

      codeword_LLR[i].s[j] = llr0[i];
    }



    codeword_LLR[i].s0 = llr0[i];
    codeword_LLR[i].s1 = llr1[i];
    codeword_LLR[i].s2 = llr2[i];
    codeword_LLR[i].s3 = llr3[i];
    codeword_LLR[i].s4 = llr4[i];
    codeword_LLR[i].s5 = llr5[i];
    codeword_LLR[i].s6 = llr6[i];
    codeword_LLR[i].s7 = llr7[i];

    codeword_LLR[i].s8 = llr8[i];
    codeword_LLR[i].s9 = llr9[i];
    codeword_LLR[i].sa = llra[i];
    codeword_LLR[i].sb = llrb[i];
    codeword_LLR[i].sc = llrc[i];
    codeword_LLR[i].sd = llrd[i];
    codeword_LLR[i].se = llre[i];
    codeword_LLR[i].sf = llrf[i];
#endif


  }

  for(int i=0; i<n_vars; i++) {
#if MANUAL_VECTOR_WIDTH == 1
    printf("codeword[%d]␣=␣%d\n",i,codeword_LLR[i]);//.x,codeword_LLR[i].y,codeword_LLR[i
        ↪ ].z,codeword_LLR[i].w);
#else
    printf("codeword[%d]␣=␣%d\n",i,codeword_LLR[i].x,codeword_LLR[i].y,codeword_LLR[i].z,
        ↪ codeword_LLR[i].w);
#endif
  }

  // Let's start initializing the inputs
  // For the codewords starting at the VN we just need to create a buffer with the
        ↪ measurements






  // For the second version we need to start by doing the first VN portion
  //initializeNodes(v_nodes,c_nodes,codeword_LLR,n_vars,n_checks);

  // Let's get the connection tables
  size_t VNtoCNSize = sizeof(short)*N_VARIABLE_NODES*N_VARIABLE_INPUTS;
  size_t CNtoVNSize = sizeof(short)*N_CHECK_NODES*N_CHECK_INPUTS;


  short * VNtoCNLookup_h = generate_VNtoCN_index_table(v_nodes,c_nodes);
  short * CNtoVNLookup_h = generate_CNtoVN_index_table(v_nodes,c_nodes);




  cl_mem VNtoCNLookup = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
      ↪ VNtoCNSize,VNtoCNLookup_h,&err);AxCheckError(err);
  cl_mem CNtoVNLookup = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
      ↪ CNtoVNSize,CNtoVNLookup_h,&err);AxCheckError(err);




  // Make the constant memory info

  cl_mem vector_max_llr_d = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
      ↪ sizeof(vector_max_llr),&vector_max_llr,&err);AxCheckError(err);
  cl_mem vector_sign_bit_d = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
      ↪ sizeof(vector_sign_bit),&vector_sign_bit,&err);AxCheckError(err);
  cl_mem allones_d = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,sizeof(
      ↪ allones),&allones,&err);AxCheckError(err);
  cl_mem ones_d = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,sizeof(ones
      ↪ ),&ones,&err);AxCheckError(err);
  cl_mem zeros_d = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,sizeof(
      ↪ zeros),&zeros,&err);AxCheckError(err);
  cl_mem total_vector_max_llr_d = clCreateBuffer(context,CL_MEM_READ_ONLY|
      ↪ CL_MEM_COPY_HOST_PTR,sizeof(total_vector_max_llr),&total_vector_max_llr,&err);
      ↪ AxCheckError(err);
  cl_mem total_vector_min_neg_llr_d = clCreateBuffer(context,CL_MEM_READ_ONLY|
      ↪ CL_MEM_COPY_HOST_PTR,sizeof(total_vector_min_neg_llr),&total_vector_min_neg_llr
      ↪ ,&err);AxCheckError(err);
  size_t vnResultSize = sizeof(LLR) * N_VARIABLE_NODES;
  size_t cnResultSize = sizeof(LLR) * N_CHECK_NODES;

  LLR * vnResult = (LLR*) _aligned_malloc(vnResultSize,AOCL_ALIGNMENT);
  LLR * cnResult = (LLR*) _aligned_malloc(cnResultSize,AOCL_ALIGNMENT);

  cl_mem vnResult_d = clCreateBuffer(context,CL_MEM_READ_WRITE,vnResultSize,NULL,&err);
      ↪ AxCheckError(err);
  cl_mem cnResult_d = clCreateBuffer(context,CL_MEM_READ_WRITE,cnResultSize,NULL,&err);
      ↪ AxCheckError(err);

  size_t channelSize = sizeof(LLR)*N_VARIABLE_NODES;
  cl_mem initialChannelMeasurements = clCreateBuffer(context,CL_MEM_READ_WRITE |
      ↪ CL_MEM_COPY_HOST_PTR, channelSize,codeword_LLR,&err);AxCheckError(err);

  // Let's set up the kernel
  AxCheckError(clSetKernelArg(LDPCKernel,0,sizeof(VNtoCNLookup),&VNtoCNLookup));
  AxCheckError(clSetKernelArg(LDPCKernel,1,sizeof(CNtoVNLookup),&CNtoVNLookup));
  AxCheckError(clSetKernelArg(LDPCKernel,2,sizeof(vector_max_llr_d),&vector_max_llr_d));
  AxCheckError(clSetKernelArg(LDPCKernel,3,sizeof(vector_sign_bit_d),&vector_sign_bit_d));
  AxCheckError(clSetKernelArg(LDPCKernel,4,sizeof(allones_d),&allones_d));
  AxCheckError(clSetKernelArg(LDPCKernel,5,sizeof(ones_d),&ones_d));
  AxCheckError(clSetKernelArg(LDPCKernel,6,sizeof(zeros_d),&zeros_d));
  AxCheckError(clSetKernelArg(LDPCKernel,7,sizeof(total_vector_max_llr_d),&
      ↪ total_vector_max_llr_d));
  AxCheckError(clSetKernelArg(LDPCKernel,8,sizeof(total_vector_min_neg_llr_d),&
      ↪ total_vector_min_neg_llr_d));
  AxCheckError(clSetKernelArg(LDPCKernel,9,sizeof(initialChannelMeasurements),&
      ↪ initialChannelMeasurements));
  AxCheckError(clSetKernelArg(LDPCKernel,10,sizeof(vnResult_d),&vnResult_d));
  AxCheckError(clSetKernelArg(LDPCKernel,11,sizeof(cnResult_d),&cnResult_d));


  // Now let's run the kernel
  size_t global_size = N_CHECK_NODES;
  size_t local_size = N_CHECK_NODES;
  cl_event kernelTime;
  printf("Starting␣the␣kernel\n");
  AxCheckError(clEnqueueNDRangeKernel(LDPCQueue,LDPCKernel,1,NULL,&global_size,&local_size
      ↪ ,0,NULL,&kernelTime));

  clWaitForEvents(1,&kernelTime);
  clFinish(LDPCQueue);

  cl_ulong time_start,time_end;
  double total_time;

  AxCheckError(clGetEventProfilingInfo(kernelTime,CL_PROFILING_COMMAND_START,sizeof(
      ↪ time_start),&time_start,NULL));
  AxCheckError(clGetEventProfilingInfo(kernelTime,CL_PROFILING_COMMAND_END,sizeof(time_end
      ↪ ),&time_end,NULL));

  total_time = time_end-time_start;
  // Now we can just get the data
  AxCheckError(clEnqueueReadBuffer(LDPCQueue,vnResult_d,CL_TRUE,0,vnResultSize,vnResult,0,
      ↪ NULL,NULL));
  AxCheckError(clEnqueueReadBuffer(LDPCQueue,cnResult_d,CL_TRUE,0,cnResultSize,cnResult,0,
      ↪ NULL,NULL));

  // Let's check the accuracy
  // Ok let's print it out
  checkIfSatisfied(cnResult);
  printResults(vnResult);
  double milisecondTime = total_time/1000000.0;

  printf("The␣kernel␣execution␣time␣was␣%f␣ms\n",milisecondTime);
  double throughput = N_VARIABLE_NODES*MANUAL_VECTOR_WIDTH/((milisecondTime)/1000.0);
  printf("The␣throughput␣was␣%f␣bps\n",throughput);

  FILE * results = fopen("results_kernel.csv","w");

  fprintf(results,"%f,%f\n",milisecondTime,throughput);
  fclose(results);



}

void printResults(LLR * results) {

  // Need to get a value in binary first

#if MANUAL_VECTOR_WIDTH == 1
  printf("Vector␣0:␣");
  for(int i=0; i<N_VARIABLE_NODES; i++) {

    if(results[i] > 0)
      printf("0");
    else
      printf("1");
  }
  printf("\n");
#else
  for(int j=0; j<MANUAL_VECTOR_WIDTH; j++) {

    printf("Vector␣%d:␣",j);
    for(int i=0; i<N_VARIABLE_NODES; i++) {


      if(results[i].s[j] > 0)
        printf("0");
      else
        printf("1");
    }
    printf("\n");

  }
#endif
}

void setFileNames() {

  strncpy(INPUT_FILENAME,FILENAME,MAX_LINE);
  _snprintf(INPUT_FILENAME,MAX_LINE,"E:/Andrew/Projects/matrix_%d.txt",LDPC_SIZE);

  strncpy(OUTPUT_FILENAME,OUTPUT_FILE,MAX_LINE);
  _snprintf(OUTPUT_FILENAME,MAX_LINE,"%s%d.txt",OUTPUT_FILENAME,LDPC_SIZE);

#ifdef ALTERA
  _snprintf(KERNEL_FILENAME,MAX_LINE,"E:/Andrew/Projects/LocalMemoryOptimzed/autoSimd/
      ↪ localMemoryAutoSimd%d.aocx",LDPC_SIZE);
#else
  _snprintf(KERNEL_FILENAME,MAX_LINE,"../kernels_%d.cl",LDPC_SIZE);
#endif
}

void printResultsToFile(double overall, double variable, double check, double misc) {
  // Now let's open the file
  FILE * results = fopen(OUTPUT_FILENAME,"w"); CheckPointer(results);

  // Now let's write data to the file
  fprintf(results,"%f␣%f␣%f␣%f\n",overall,variable,check,misc);

  // Now we close the file
  fclose(results);

}





// Check if all of the check nodes are satisfied
bool checkIfSatisfied(LLR * parity)
{
#if MANUAL_VECTOR_WIDTH == 1
  // All we have to do use iterate over all and see if there is anything non zero
  for(int i=0; i<N_CHECK_NODES; i++) {
    if(parity[i]){
      printf("NOT␣SATISFIED\n");
      return false;
    }
  }
  printf("SATISFIED!\n");
  return true;
#else
  for(int j=0; j<MANUAL_VECTOR_WIDTH; j++){
    for(int i=0; i<N_CHECK_NODES; i++) {
      if(parity[i].s[j]){
        printf("NOT␣SATISFIED\n");
      }
    }
    printf("SATISFIED!\n");
  }

  return true;
#endif

}

#pragma region Codeword Operations
// Get the codeword in BPSK
float* getCodeword(int size)
{
  float *codeword = (float*)malloc(sizeof(float)*size);
  int * input = (int*) malloc(sizeof(int)*size);
  for(int i=0; i<size; i++)
    input[i] = 0;

  input[0] = 1;

  // Convert it to BPSK
  // 0 -> 1, 1 -> -1
  // Map binary 0 to 1
  // May binary 1 to -1
  for(int i=0; i<size; i++)
  {
    if(input[i])
      codeword[i] = -1;
    else
      codeword[i] = 1;
  }

  return codeword;
}


// Introduce gaussian noise to the input codeword based on the given SNR
void introduceNoise(float* codeword, int SNR, int n_vars)
{
  double sigma = pow(10,(double)-SNR/20);

  for(int i=0; i<n_vars; i++)
  {
    codeword[i] += generate_random_gaussian()*sigma;
    //printf("Codeword[%d] = %f\n",i,codeword[i]);
  }
}

void printCodeword(const int * codeword, const int n_vars) {
  for(int i=0; i<n_vars; i++) {
    printf("codeword[%d] = %d\n",i,codeword[i]);
  }

}

/* This function will take in the given float version of the codeword
and convert it to an integer representation LLR.  Please note that
it converts the LLR into a 2's complement number first
*/
LLR_BASE_TYPE * convertToLLR(const float *codeword, const int n_vars) {
  LLR_BASE_TYPE * output = (LLR_BASE_TYPE*) malloc(sizeof(LLR_BASE_TYPE)*n_vars);

  // Iterate over the entire codeword
  for(int i=0; i<n_vars; i++) {
    float word = abs(codeword[i]);
    output[i] = 0;


    // Now let's iterate over the possible bins and see where it fits
    float division_point = ((float)MAX_LLR)/SATURATION_POINT ;
    //printf("division_point = %f\n",division_point);
    for(int j=1; j<=(MAX_LLR); j++) {

      if(word <  j / division_point) {
        output[i] = j;

        break;
      }

    }
    //printf("codeword[%d] = %f output[%d] = %u     sign = %d\n",i,codeword[i],i,output[i
        ↪ ],(output[i]&SIGN_BIT)>>15);

    // Once we have the positioning we just need to make sure it's still within the valid
        ↪ LLR
    if(word >=  MAX_LLR / division_point) {
      output[i] = MAX_LLR;

    }

    // Now set the sign bit
    if(codeword[i] < 0) {
      output[i] *= -1;

    }

  }
  return output;
}
#pragma endregion


// This function will read the parity check matrix A-list file and get the following
     ↪ information:
// - number of variable nodes, number of check nodes
// - an array of variablenode structures
// - an array of checknode structures
// PLEASE NOTE ALL VARIABLES PASSED INTO THIS FUNCTION ARE PASSED BY REFERENCE AND WILL BE
     ↪  MODIFIED
// Please pass this function empty variables to store the result
void readParityFile(VariableNode *& v_nodes, CheckNode *& c_nodes,int &n_vars,int &
     ↪ n_checks)
{
#define MAX_LINE 1024
```

```
char line[MAX_LINE];

// Open the file
FILE * parity_matrix = fopen(INPUT_FILENAME,"r");
CheckPointer(parity_matrix);

// Now let's start reading
// The first two values in the text file will be the number of variable nodes and the
    ↪ number of check nodes
if(fscanf(parity_matrix,"%d␣%d",&n_vars,&n_checks)!=2)
  ExitOnErrorWMsg("Error␣with␣parity_matrix␣A-list␣file␣format,");

// Now that we know the number of variable nodes and checks we can allocate the amounts
v_nodes = (VariableNode*)malloc(sizeof(VariableNode)*n_vars);
c_nodes = (CheckNode*)malloc(sizeof(CheckNode)*n_checks);

// Throw away the next line.  This line tells us the approximate number of connections
    ↪ for each node, but we want the more accurate ones
int temp;
if(fscanf(parity_matrix,"%d␣%d",&temp,&temp)!=2)
  ExitOnErrorWMsg("Missing␣second␣line␣in␣A-list␣file.");

// Now let's get to the good stuff
// According to the A-list format the next line of text will be the list of check nodes
    ↪ for each variable node in order
// The next line will be a list of check nodes for each variable node
// Therefore we need to iterate n_vars times to get all of the connections
for(int i=0; i<n_vars; i++)
{
  int num_per_variable;
  if(fscanf(parity_matrix,"%d",&num_per_variable)!=1)
    ExitOnErrorWMsg("Error␣with␣parity_matrix␣A-list␣file␣format.");

  // Store the number of check nodes for each variable node and the number of inputs
  v_nodes[i].n_checknodes = num_per_variable;
  v_nodes[i].n_inputs = num_per_variable+1; // Add one here as there is a spot for the
      ↪ initial channel measurement

  // Let's also store the index
  v_nodes[i].index = i;
}

// The next line will be a list of variable nodes for each check node
// Therefore we need to iterate n_checks times to get all of the connections
for(int i=0; i<n_checks; i++)
{
  int num_per_check;
  if(fscanf(parity_matrix,"%d",&num_per_check)!=1)
    ExitOnErrorWMsg("Error␣with␣parity_matrix␣A-list␣file␣format.");

  // Store the number of inputs to the check node
  c_nodes[i].n_inputs = num_per_check; // Note there is not an extra here as there is no
      ↪  channel measurement

  // Also store the index
  c_nodes[i].index = i;

  // Set the satisifed bit to false
  //c_nodes[i].satisfied = false;
}

// Next we get the indexes for variable nodes (edges)
// This is the most important section where we determine which nodes get connected to
    ↪ each other
// First up are the variable node connections.  So we need to iterate over all the
    ↪ variable nodes
for(int i=0; i<n_vars; i++)
{
  int position;

  // Now we can use the number of checknodes we just received from the previous lines
  // We need to store which checknodes this variable node is connected to
  for(int j=0; j<v_nodes[i].n_checknodes; j++)
  {
    if(fscanf(parity_matrix,"%d",&position)!=1)
      ExitOnErrorWMsg("Error␣with␣parity_matrix␣A-list␣file␣format.");

    // Now store the position
    v_nodes[i].checknode_indexes[j] = position-1; // Note we need to subtract one here
        ↪ to start the position at 0

  }

}


// Same thing, but now for the check nodes.
// Next we get the indexes for the check nodes (edges)
for(int i=0; i<n_checks; i++)
{
  int position;
  for(int j=0; j<c_nodes[i].n_inputs; j++)
```

```
    {
      if(fscanf(parity_matrix,"%d",&position)!=1)
        ExitOnErrorWMsg("Error␣with␣parity_matrix␣A-list␣file␣format.");

      // Now store the position
      c_nodes[i].variablenode_indexes[j] = position-1; // Note we need to subtract one
           ↪ here to start the position at 0
    }
  }

  // That concludes the reading of the parity matrix file
  // All data is now stored within the v_nodes and c_nodes arrays
  fclose(parity_matrix);
  printf("Parity␣matrix␣file␣%s␣read␣successfully\n",INPUT_FILENAME);
  return;
}


//Initialize the nodes for the first run
// This is where we'll need to load up that there are 4 of the vector chars
void initializeNodes(VariableNode *& v_nodes, CheckNode *& c_nodes, const LLR * codeword,
    ↪ const int n_vars, const int n_checks)
{

  // First we load up the channel measurement into all of the variable nodes
  // Iterate over each variable node
  for(int i=0; i<n_vars; i++)
  {
    short channel_index = v_nodes[i].n_inputs-1;
    // For each of the variable nodes, we want to only load up the very last spot that
         ↪ deals with the channel input
    // All of the other inputs should be 0.  This will allows us to start the execution
         ↪ with the variable nodes instead of the check nodes


    for(int j=0; j<v_nodes[i].n_inputs; j++)
    {

      // We need to initialize all 4 of the chars in the vector
      // Note that we use codeword.x(y) etc however they are all equal to begin but it is
           ↪ possible for these to be different
#if MANUAL_VECTOR_WIDTH == 1
      v_nodes[i].inputs[j] = codeword[i];
#else
      for(int t=0; t<MANUAL_VECTOR_WIDTH; t++) {
        v_nodes[i].inputs[j].s[t] = codeword[i].s[t];
      }
#endif
    }
  }


  // Load up the inputs for the check nodes for the first iteration
  // This means we are now passing the specific channel measurements for each connection
  for(int i=0; i<n_checks; i++)
  {

    // For each one of the inputs in this check node, we need to load up the information
         ↪ from the variable node
    for(int j=0; j<c_nodes[i].n_inputs; j++)
    {
      int variable_index = c_nodes[i].variablenode_indexes[j];
      int index = v_nodes[variable_index].n_inputs-1; // All of the spots hold the same
           ↪ value currently

      // We have to convert this value into sign and mag
      LLR sign_and_mag_ver = v_nodes[variable_index].inputs[index];

      // Now we need to do the four comparisons individually
#if MANUAL_VECTOR_WIDTH == 1
      if(sign_and_mag_ver < 0) {
        sign_and_mag_ver *= -1;
        sign_and_mag_ver |= SIGN_BIT;
      }
#else
      for(int t=0; t<MANUAL_VECTOR_WIDTH; t++) {
        if(sign_and_mag_ver.s[t] < 0) {
        sign_and_mag_ver.s[t] *= -1;
        sign_and_mag_ver.s[t] |= SIGN_BIT;
        }
      }
#endif



      // Let's store the value in our input
      c_nodes[i].inputs[j] = sign_and_mag_ver;
    }
  }
}

#pragma region Error Checking Functions
```

```
// This function will check the given input pointer for validaty and fail exit the program
    ↪ if necessary
void CheckPointerFunc(void * pointer,char const* const func, const int line)
{
  if(pointer == NULL)
  {
    fprintf(stderr,"Error␣with␣pointer␣in␣%s␣line␣%d\n",func,line);
    exit(SYSTEM_ERROR);
  }
}

void ExitOnErrorFunc(char const* msg, char const* func, const int line)
{
  fprintf(stderr,"%s␣%s␣line␣%d\n",msg,func,line);
  exit(SYSTEM_ERROR);
}
#pragma endregion

#pragma region Random Number Code
double generate_random_gaussian()
{
#define PI 3.14159265358979323846
  double U1 = generate_random_number();//(double)rand()/((doule)RAND_MAX+1);
  double U2 = generate_random_number();//(double)rand()/((double)RAND_MAX+1);

  double Z0 = sqrt(-2*log(U1))*cos(2*PI*U2);
  return Z0;
}



short * generate_VNtoCN_index_table(VariableNode * v_nodes,CheckNode * c_nodes) {
  short * table = (short*)_aligned_malloc(sizeof(short)*N_VARIABLE_NODES*N_VARIABLE_INPUTS
      ↪ ,AOCL_ALIGNMENT);
  if(table == NULL)
    printf("err␣=␣%d\n",errno);
  // Iterate over the list of variable nodes
  for(int i=0; i<N_VARIABLE_NODES; i++) {

    // For each variable node we need to find the appropriate connections for each input
    for(int j=0; j<N_VARIABLE_INPUTS; j++) {
      int cnIndex = v_nodes[i].checknode_indexes[j];


      int cnInputIndex;
      // Now let's look inside that check node to find our appropriate location
      for(int n=0; n<N_CHECK_INPUTS; n++) {
        int tempIndex = c_nodes[cnIndex].variablenode_indexes[n];

        // If we found our variable node, make the table
        if(tempIndex == i) {
          cnInputIndex = n;
          break;
        }
      }

      // Now we can create our table entry
      table[i*N_VARIABLE_INPUTS + j] = (cnIndex*N_CHECK_INPUTS) + cnInputIndex;


    }



  }

  return table;
}

short * generate_CNtoVN_index_table(VariableNode * v_nodes,CheckNode * c_nodes) {
  short * table = (short*)_aligned_malloc(sizeof(short)*N_CHECK_NODES*N_CHECK_INPUTS,
      ↪ AOCL_ALIGNMENT);
  if(table == NULL)
    printf("err␣=␣%d\n",errno);
  // Iterate over the list of variable nodes
  for(int i=0; i<N_CHECK_NODES; i++) {

    // For each variable node we need to find the appropriate connections for each input
    for(int j=0; j<N_CHECK_INPUTS; j++) {
      int vnIndex = c_nodes[i].variablenode_indexes[j];

      int vnInputIndex = -100;
      // Now let's look inside that check node to find our appropriate location
      for(int n=0; n<N_VARIABLE_INPUTS; n++) {
        int tempIndex = v_nodes[vnIndex].checknode_indexes[n];

        // If we found our variable node, make the table
        if(tempIndex == i) {
          vnInputIndex = n;
          break;
        }
      }
      int foundIndex = (vnIndex*N_VARIABLE_INPUTS_W_CHANNEL) + vnInputIndex;

      // Populate the table
      table[i*N_CHECK_INPUTS + j] = foundIndex;

    }


  }

  return table;
}

double generate_random_number()
{
#define C1 4294967294
#define C2 4294967288
#define C3 4294967280
#define MAX 4294967295

  srand(45);
  static unsigned int s1 = rand();
  static unsigned int s2 = rand();
  static unsigned int s3 = rand();

  // Here's the first part
  // Now let's crank the tausworthe
  unsigned int xor1 = s1 << 13;

  // The first set of and/xor
  unsigned int left_temp = xor1 ^ s1;
  unsigned int right_temp = C1 & s1;

  // Shifts
  left_temp = left_temp >> 19;
  right_temp = right_temp << 12;

  s1 = left_temp ^ right_temp;

  // Second part
  xor1 = s2<<2;
  left_temp = xor1 ^ s2;
  right_temp = C2 & s2;

  left_temp = left_temp >> 25;
  right_temp = right_temp << 4;

  s2 = left_temp ^ right_temp;

  // Third part
  xor1 = s3 << 3;
  left_temp = xor1 ^ s3;
  right_temp = C3 & s3;

  left_temp = xor1 ^ s3;
  right_temp = C3 & s3;

  left_temp = left_temp >> 11;
  right_temp = right_temp << 17;

  s3 = left_temp ^ right_temp;

  // Now the return
  unsigned int output = s1 ^ s2 ^ s3;

  // Now just convert it into a double
  double last_value = (double)output /MAX;

  //cout<<"last value" << last_value<<endl;
  return last_value;

}
#pragma endregion

void defineGlobalDefinitions() {



#if MANUAL_VECTOR_WIDTH==1
  vector_max_llr = MAX_LLR;
  vector_min_neg_llr = MAX_NEG_LLR;
  vector_sign_bit = SIGN_BIT;
  allones = (LLR)(0xFF);
  ones = (LLR)(1);
  zeros = (LLR)(0);
  vector_too_neg_llr = MAX_NEG_LLR - ones;
  total_vector_max_llr = MAX_LLR;
  total_vector_min_neg_llr = MAX_NEG_LLR;
#else
  for(int i=0; i<MANUAL_VECTOR_WIDTH; i++)
  {
    vector_max_llr.s[i] = MAX_LLR;
    vector_min_neg_llr.s[i] = MAX_NEG_LLR;
    vector_sign_bit.s[i] = SIGN_BIT;
    allones.s[i] = (0xFF);
    ones.s[i] = (1);
    zeros.s[i] = (0);
    vector_too_neg_llr.s[i] = MAX_NEG_LLR - 1;
    total_vector_max_llr.s[i] = MAX_LLR;
    total_vector_min_neg_llr.s[i] = MAX_NEG_LLR;
  }
#endif
}

void parseProgramInputs(int n_args,char* args[]) {
  for(int i=0; i<n_args-1; i++) {
    if(strncmp(args[i],"-nVars",MAX_LINE)==0) {
      // If we found the n_vars just set the variable

      int new_value = atoi(args[i+1]);
      if(new_value <= 0) {
        ExitOnErrorWMsg("Not a valid LDPC size input\n");
      }
      N_VARIABLE_NODES = atoi(args[i+1]);
      LDPC_SIZE = N_VARIABLE_NODES;
      N_CHECK_NODES =(N_VARIABLE_NODES/2);
    }
  }

}
```

Appendix B

(3,6)-Regular LDPC Code Generator (C#)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CodewordGenerator
{
    /**
     * This class will generate an LDPC codeword based on the input size requirements.
     * In order to do this randomly, we must follow a few basic rules. Namely, each of the
     * variable nodes and check nodes must have exactly 3 and 6 connections respectively.
     * We must then keep track of how many have been filled through the random process and
     * assign connections accordingly.
     */
    class Generate
    {
        private int numVNodes;
        private int numCNodes;
        private String outputFilename;
        private System.IO.StreamWriter fileOutput;
        private Random rand;

        private const int vNodeFanout = 3;
        private const int cNodeFanout = 6;

        // Let's declare the internal arrays to keep track of how many connections are
            ↪ made
        // These are rectangular 2d arrays that will be of size numVNodes, fanout+1
        // Please note that the very first entry for each variablenode will be the current
            ↪  fanout achieved (or the length)
        // Internally all connections are 0 starting
        int[,] variableNodes;
        int[,] checkNodes;

        public Generate()
            : this(64, "matrix_64.txt")
        {
        }

        public Generate(int newNum, String output)
        {
            numVNodes = newNum;
            numCNodes = numVNodes / 2;
            outputFilename = output;
            fileOutput = new System.IO.StreamWriter(outputFilename);
            rand = new Random((int)DateTime.Now.Ticks & 0x0000FFFF);

            // Now let's initialize the arrays
            variableNodes = new int[numVNodes, vNodeFanout + 1];
            checkNodes = new int[numCNodes, cNodeFanout + 1];
        }

        public void SetVNodes(int newNum)
        {
            numVNodes = newNum;
            numCNodes = numVNodes / 2;
        }
        public int GetVNodes() { return numVNodes; }
        public int GetCNodes() { return numCNodes; }

        public void Create()
        {
            while (!StartProcess());

            // Let's start by writing what we know
            WriteInitialAListInfo();
            WriteAListData();
            fileOutput.Close();

        }

        private void WriteAListData()
        {
            // First it's the variable nodes
            for (int i = 0; i < variableNodes.GetLength(0); i++)
            {
                for (int j = 1; j < variableNodes.GetLength(1); j++)
                    fileOutput.Write(variableNodes[i, j] + "␣");
                fileOutput.Write("\r\n");
            }

            // Now the check nodes
            for (int i = 0; i < checkNodes.GetLength(0); i++)
            {
                for (int j = 1; j < checkNodes.GetLength(1); j++)
                    fileOutput.Write(checkNodes[i, j] + "␣");
                fileOutput.Write("\r\n");
            }
        }

        /**
         *
         * This method will create the LDPC codeword and output the matrix to the output
             ↪ file in A-List format.
         */
        public bool StartProcess()
        {
            System.Console.Write("Generating␣a␣(" + numVNodes + "," + numCNodes + ")␣
                ↪ regular␣" + vNodeFanout + "," + cNodeFanout + "␣codeword␣to␣" +
                ↪ outputFilename + "........");



            // Now to create it we need to use random numbers to figure out the
                ↪ connections
            // First let's initialize the arrays
            InitializeNodes();

            // Now we can start getting connections
            // We will do the same operation for all of the variable nodes
            // Iterate over all the variable Nodes
            for (int vNode = 0; vNode < variableNodes.GetLength(0); vNode++)
            {
                // Now let's just try and see if we can get random numbers for connections
                // First we need to check to see if this current node is full (already has
                    ↪ 3 connections)
                // If it does, we just simply skip over this node and move to the next one
                // Note this is just a precautionary thing, it should never happen
                if (variableNodes[vNode, 0] >= vNodeFanout) { continue; }

                // Now for each variable node, we need to do this calculation for each
                    ↪ entry
                for (int vInputIndex = variableNodes[vNode, 0] + 1; vInputIndex <=
                    ↪ vNodeFanout; vInputIndex++)
                {

                    // Now we can start with the process
                    // Let's get a random number between 0 and the number of check nodes
                    // This will give us an index to check
                    bool connectionSuccessful = false;
                    while (!connectionSuccessful)
                    {
                        int cNodeIndex = rand.Next(0, numCNodes);
                        int cNodeSize = checkNodes[cNodeIndex, 0];

                        // Great, now let's see if we can make this connection
                        // In order for this to happen, the connection must not currently
                            ↪ exist and the check node must have an empty slot
                        if (cNodeSize >= cNodeFanout)
                        {
                            connectionSuccessful = false;
                        }
                        else
                        {
                            bool connectionExists = false;
                            // Great it has an empty slot, let's see if the connection
                                ↪ currently exists
                            for (int i = 1; i <= cNodeSize; i++)
                            {
                                if (checkNodes[cNodeIndex, i] == vNode)
                                {
                                    connectionExists = true;
                                }
                            }

                            // Now let's see if it existed
                            if (!connectionExists)
                            {
                                // Great, now we've passed the tests and we can make the
                                    ↪ connection
                                // Let's start by placing the index in the check node
                                checkNodes[cNodeIndex, cNodeSize + 1] = vNode;

                                // Don't forget to increase the size of the checknode for
                                    ↪ future reference
                                checkNodes[cNodeIndex, 0]++;

                                // Now let's update the variablenodes
                                variableNodes[vNode, vInputIndex] = cNodeIndex;

                                // Don't forget to increase the size of the variable node
                                    ↪ internals
                                variableNodes[vNode, 0]++;

                                // Let's go ahead and set the connection successful
                                // This will let us move on to the next input
                                connectionSuccessful = true;
                            }
                            else
                            {
                                // Otherwise we need to check if we're in the deadlock
                                    ↪ state
                                // This deadlock state is when there's only one checknode
                                    ↪ left to connect, but we've already made a
                                    ↪ connection to this node
                                // In this case, we'll find a full node that we haven't
                                    ↪ connected, and trade out our connection
                                // First let's see if this is indeed the case
                                int numnonfull = 0;
                                for (int i = 0; i < checkNodes.GetLength(0); i++)
                                {
                                    // Let's see how many nodes aren't full yet
                                    if (checkNodes[i, 0] < cNodeFanout)
                                        numnonfull++;
                                }

                                // Let's see how many nodes that is. If it's only one,
                                    ↪ then we've got a problem
                                if (numnonfull <= 1)
                                {
                                    // This is where the bad things happen
                                    // If this part is true, we need to just start over
                                    System.Console.WriteLine("FAILED.␣RETYRING");
                                    return false;
                                }

                            }
                        }
                    }
                }


            }

            // Yay we're almost done
            // First let's index everything by 1 (the A-List standard)
            ChangeIndexing();

            // Finally let's just sort the Variable Node slots
            SortVariableNodeSlots();


            System.Console.WriteLine("SUCCEEDED");
            return true;
        }

        private void SortVariableNodeSlots()
        {
            // I'll have to do the sort myself due to the organization
            for (int c = 0; c < variableNodes.GetLength(0); c++)
            {
                // For each check node, let's sort the inputs
                for (int x = 1; x < variableNodes.GetLength(1); x++)
                {
                    for (int y = 1; y < variableNodes.GetLength(1) - 1; y++)
                    {
                        if (variableNodes[c, y] > variableNodes[c, y + 1])
                        {
                            // Swap
                            int temp = variableNodes[c, y + 1];
                            variableNodes[c, y + 1] = variableNodes[c, y];
                            variableNodes[c, y] = temp;
                        }
                    }
                }
            }
        }

        private void ChangeIndexing()
        {
            // We just need to change the indexing by adding one to all of the connections
            for (int i = 0; i < variableNodes.GetLength(0); i++)
                for (int j = 1; j < variableNodes.GetLength(1); j++)
                    variableNodes[i, j]++;

            for (int i = 0; i < checkNodes.GetLength(0); i++)
                for (int j = 1; j < checkNodes.GetLength(1); j++)
                    checkNodes[i, j]++;

        }

        private void InitializeNodes()
        {
            // Let's start with the variable nodes
            for (int i = 0; i < variableNodes.GetLength(0); i++)
                for (int j = 0; j < variableNodes.GetLength(1); j++)
                    variableNodes[i, j] = 0;

            for (int i = 0; i < checkNodes.GetLength(0); i++)
                for (int j = 0; j < checkNodes.GetLength(1); j++)
                    checkNodes[i, j] = 0;

        }

        private void WriteInitialAListInfo()
        {
            // The first line will simply be # #
            fileOutput.WriteLine(numVNodes + "␣" + numCNodes);

            // The next line will be the regular 3 6
            fileOutput.WriteLine("3␣6");

            // Now we have to write the specifics for each variable node and check node
            // As this is a regular code this is just a formality
            for (int i = 0; i < numVNodes; i++)
                fileOutput.Write("3␣");
            fileOutput.Write("\r\n");

            // Same with the next part
            for (int i = 0; i < numCNodes; i++)
                fileOutput.Write("6␣");
            fileOutput.Write("\r\n");
        }


    }
}
```

Appendix C

Branch Experiment Source Code (OpenCL)

```
#define N_VARIABLE_NODES 1024
#define N_CHECK_NODES 512
#define MAX_VARIABLE_EDGES 3
#define MAX_CHECK_EDGES 6

// This is for the local memory now
// This is a change
// Integer options
#define LLR_TYPE char16
#define LLR_BASE_TYPE char
#define N_BITS_LLR (sizeof(LLR_BASE_TYPE)*8) // 8 bits per byte
#define MAX_LLR ((1 << (N_BITS_LLR-1))-1)
#define SIGN_BIT (MAX_LLR+1)
#define MAX_NEG_LLR (0-MAX_LLR)

#define N_ITERATIONS 1000000 // Repeat the operation for 1 million iterations
#define M N_CHECK_NODES
#define N MAX_CHECK_EDGES

// We want to create a kernel that does 4 simultaneous codewords using the char4 operative
// There will just be one giant kernel that handles the entire execution path
__kernel __attribute__((reqd_work_group_size(M,1,1))) void LDPCDecoder(__global LLR_TYPE *
    ↪  restrict conditional, __global LLR_TYPE * restrict resulting) {

  // Input a is going to be the conditional value. 1 for the first section and 0 otherwise
  // b is going to be the resulting vector. We will add 1 if it went into the first and 0
      ↪ if it went into the second
  // a is going to be organized as a single dimension 2d array
  int workitem_id = get_global_id(0);

  // First let's calculate our start index
  int baseIndex = workitem_id*MAX_CHECK_EDGES;

  // Just to amplify the perfomance result, we'll iterate 1 million times
  for(int iterations = 0; iterations < N_ITERATIONS; iterations++)
  {
    // In order to create comparable hardware, we will unroll this loop
    // So now we'll want a for loop going over all of the inputs
    #pragma unroll
    for(int input=0; input<N; input++)
    {
      LLR_BASE_TYPE char_0 = conditional[baseIndex + input].s0;
      LLR_BASE_TYPE char_1 = conditional[baseIndex + input].s1;
      LLR_BASE_TYPE char_2 = conditional[baseIndex + input].s2;
      LLR_BASE_TYPE char_3 = conditional[baseIndex + input].s3;
      LLR_BASE_TYPE char_4 = conditional[baseIndex + input].s4;
      LLR_BASE_TYPE char_5 = conditional[baseIndex + input].s5;
      LLR_BASE_TYPE char_6 = conditional[baseIndex + input].s6;
      LLR_BASE_TYPE char_7 = conditional[baseIndex + input].s7;
      LLR_BASE_TYPE char_8 = conditional[baseIndex + input].s8;

      LLR_BASE_TYPE char_9 = conditional[baseIndex + input].s9;
      LLR_BASE_TYPE char_a = conditional[baseIndex + input].sa;
      LLR_BASE_TYPE char_b = conditional[baseIndex + input].sb;
      LLR_BASE_TYPE char_c = conditional[baseIndex + input].sc;
      LLR_BASE_TYPE char_d = conditional[baseIndex + input].sd;
      LLR_BASE_TYPE char_e = conditional[baseIndex + input].se;
      LLR_BASE_TYPE char_f = conditional[baseIndex + input].sf;
```

```
if(char_0)
{
  resulting[baseIndex + input].s0 = 1;
}
else
{
  resulting[baseIndex + input].s0 = -1;
}


if(char_1)
{
  resulting[baseIndex + input].s1 = 1;
}
else
{
  resulting[baseIndex + input].s1 = -1;
}


if(char_2)
{
  resulting[baseIndex + input].s2 = 1;
}
else
{
  resulting[baseIndex + input].s2 = -1;
}


if(char_3)
{
  resulting[baseIndex + input].s3 = 1;
}
else
{
  resulting[baseIndex + input].s3 = -1;
}


if(char_4)
{
  resulting[baseIndex + input].s4 = 1;
}
else
{
  resulting[baseIndex + input].s4 = -1;
}


if(char_5)
{
  resulting[baseIndex + input].s5 = 1;
}
else
{
  resulting[baseIndex + input].s5 = -1;
}


if(char_6)
{
  resulting[baseIndex + input].s6 = 1;
}
else
{
  resulting[baseIndex + input].s6 = -1;
}


if(char_7)
{
  resulting[baseIndex + input].s7 = 1;
}
else
{
  resulting[baseIndex + input].s7 = -1;
}


if(char_8)
{
  resulting[baseIndex + input].s8 = 1;
}
else
{
  resulting[baseIndex + input].s8 = -1;
}


if(char_9)
{
        resulting[baseIndex + input].s9 = 1;
      }
      else
      {
        resulting[baseIndex + input].s9 = -1;
      }


      if(char_a)
      {
        resulting[baseIndex + input].sa = 1;
      }
      else
      {
        resulting[baseIndex + input].sa = -1;
      }


      if(char_b)
      {
        resulting[baseIndex + input].sb = 1;
      }
      else
      {
        resulting[baseIndex + input].sb = -1;
      }


      if(char_c)
      {
        resulting[baseIndex + input].sc = 1;
      }
      else
      {
        resulting[baseIndex + input].sc = -1;
      }

      if(char_d)
      {
        resulting[baseIndex + input].sd = 1;
      }
      else
      {
        resulting[baseIndex + input].sd = -1;
      }

      if(char_e)
      {
        resulting[baseIndex + input].se = 1;
      }
      else
      {
        resulting[baseIndex + input].se = -1;
      }

      if(char_f)
      {
        resulting[baseIndex + input].sf = 1;
      }
      else
      {
        resulting[baseIndex + input].sf = -1;
      }

    }
  }

}
```

Appendix D

Vector Experiment Source Code (OpenCL)

```
#define N_VARIABLE_NODES 1024
#define N_CHECK_NODES 512
#define MAX_VARIABLE_EDGES 3
#define MAX_CHECK_EDGES 6

// This is for the local memory now
// This is a change
// Integer options
#define LLR_TYPE char16
#define LLR_BASE_TYPE char
#define N_BITS_LLR (sizeof(LLR_BASE_TYPE)*8) // 8 bits per byte
#define MAX_LLR ((1 << (N_BITS_LLR-1))-1)
#define SIGN_BIT (MAX_LLR+1)
#define MAX_NEG_LLR (0-MAX_LLR)

typedef LLR_TYPE LLR;

#define N_ITERATIONS 1000000 // Repeat the operation for 1 million iterations
#define M N_CHECK_NODES
#define N MAX_CHECK_EDGES

// We want to create a kernel that does 4 simultaneous codewords using the char4 operative
// There will just be one giant kernel that handles the entire execution path
__kernel __attribute__((reqd_work_group_size(M,1,1))) void LDPCDecoder(__global LLR_TYPE *
     ↪ restrict conditional, __global LLR_TYPE * restrict resulting) {

  // Input a is going to be the conditional value. 1 for the first section and 0 otherwise
  // b is going to be the resulting vector. We will add 1 if it went into the first and 0
       ↪ if it went into the second
  // a is going to be organized as a single dimension 2d array
  int workitem_id = get_global_id(0);

  // First let's calculate our start index
  int baseIndex = workitem_id*MAX_CHECK_EDGES;

  LLR allones = (char16)(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1);
  LLR allnegones = (char16)(-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1);

  // Just to amplify the perfomance result, we'll iterate 1 million times
  for(int iterations = 0; iterations < N_ITERATIONS; iterations++)
  {
    // So now we'll want a for loop going over all of the inputs
    #pragma unroll
    for(int input=0; input<N; input++)
    {
      // The first thing we're going to do is get the conditional
      LLR ifCondition = (conditional[baseIndex + input] == allones);

      // Now use the conditional to determine which one is placed
      resulting[baseIndex + input] = (allones & ifCondition) | (allnegones & (~ifCondition
          ↪ ));
    }
  }

}
```

Appendix E

Design A Altera Optimization Report

```
===============================================================================
|                                              *** Optimization Report ***
| Warning: Compile with "-g" to get line number and variable name information
===================================================================================
| Kernel: LDPCDecoder
====================================================================================
| Loop for.cond7.preheader
|   Pipelined execution inferred.
|   Successive iterations launched every 2 cycles due to:
|
|       Pipeline structure
|
|   Iterations will be executed serially across the following regions:
|
|       Loop for.body10
|       Loop for.body77
|       due to:
|       Memory dependency on Load Operation from:
|         Store Operation
|         Store Operation
|         Store Operation
|         Store Operation
|         Store Operation
|         Store Operation
|
|       Loop for.body10
|       Loop for.body77
|       due to:
|       Memory dependency on Store Operation from:
|         Store Operation
|         Store Operation
|         Load Operation
|
|       Loop for.body10
|       Loop for.body77
|       due to:
|       Memory dependency on Store Operation from:
|         Store Operation
|         Load Operation
|
|       Loop for.body10
|       Loop for.body77
|       due to:
|       Memory dependency on Store Operation from:
|         Store Operation
|         Load Operation
|
|       Loop for.body10
|       Loop for.body77
|       due to:
|       Memory dependency on Store Operation from:
|         Store Operation
|         Store Operation
|         Store Operation
|         Store Operation
|         Store Operation
|
|       Loop for.body10
|       Loop for.body77
|       due to:
|       Memory dependency on Store Operation from:
|         Store Operation
|
|         Store Operation
|         Store Operation
|         Store Operation
|
|       Loop for.body10
|       Loop for.body77
|       due to:
|       Memory dependency on Store Operation from:
|         Store Operation
|         Store Operation
|         Store Operation
|         Store Operation
|
|       Loop for.body10
|       Loop for.body77
|       due to:
|       Memory dependency on Store Operation from:
|         Store Operation
|         Store Operation
|         Store Operation
|         Store Operation
|
|       Loop for.body10
|       Loop for.body77
|       due to:
|       Memory dependency on Store Operation from:
|         Store Operation
|         Store Operation
|         Store Operation
|         Store Operation
|
|       Loop for.body10
|       Loop for.body77
|       due to:
|       Memory dependency on Store Operation from:
|         Store Operation
|         Store Operation
|         Store Operation
|         Store Operation
============================================================
| Loop for.body10
|   Pipelined execution inferred.
|   Successive iterations launched every 484 cycles due to:
|
|       Memory dependency on Store Operation from:
|         Store Operation
|       Largest Critical Path Contributors:
|           33%: Store Operation
|           33%: Store Operation
|           33%: Store Operation
|
|       Memory dependency on Store Operation from:
|         Store Operation
|       Largest Critical Path Contributors:
|           33%: Store Operation
|           33%: Store Operation
|           33%: Store Operation
|
|   Additional memory dependencies:
|
|       Memory dependency on Store Operation from:
|         Store Operation
|
|       Memory dependency on Store Operation from:
|         Store Operation
============================================================
| Loop for.body77
|   Pipelined execution inferred.
|   Successive iterations launched every 964 cycles due to:
|
|       Memory dependency on Store Operation from:
|         Store Operation
|       Largest Critical Path Contributors:
|           16%: Store Operation
|           16%: Store Operation
|           16%: Store Operation
|           16%: Store Operation
|           16%: Store Operation
|
|       Memory dependency on Store Operation from:
|         Store Operation
|       Largest Critical Path Contributors:
|           16%: Store Operation
|           16%: Store Operation
|           16%: Store Operation
|           16%: Store Operation
|           16%: Store Operation
|
|       Memory dependency on Store Operation from:
|         Store Operation
|       Largest Critical Path Contributors:
|           16%: Store Operation
|           16%: Store Operation
|           16%: Store Operation
|           16%: Store Operation
|           16%: Store Operation
|
|       Memory dependency on Store Operation from:
|         Store Operation
|       Largest Critical Path Contributors:
|           16%: Store Operation
|           16%: Store Operation
|           16%: Store Operation
|           16%: Store Operation
|           16%: Store Operation
|
|       Memory dependency on Store Operation from:
|         Store Operation
|       Largest Critical Path Contributors:
|           16%: Store Operation
|           16%: Store Operation
|           16%: Store Operation
|           16%: Store Operation
|           16%: Store Operation
|
|   Additional memory dependencies:
|
|       Memory dependency on Store Operation from:
|         Store Operation
|         Store Operation
|         Store Operation
|         Store Operation
|
|       Memory dependency on Store Operation from:
|         Store Operation
|         Store Operation
|         Store Operation
|
|       Memory dependency on Store Operation from:
|         Store Operation
|         Store Operation
|         Store Operation
|
|       Memory dependency on Store Operation from:
|         Store Operation
|         Store Operation
|         Store Operation
|
|       Memory dependency on Store Operation from:
|         Store Operation
|         Store Operation
|         Store Operation
|
|       Memory dependency on Store Operation from:
|         Store Operation
|         Store Operation
|         Store Operation
|         Store Operation
================================================================

+--------------------------------------------------------------------+
; Estimated Resource Usage Summary                                   ;
+----------------------------------------+---------------------------+
; Resource                               + Usage                     ;
+----------------------------------------+---------------------------+
; Logic utilization                      ;   39%                     ;
; Dedicated logic registers              ;   19%                     ;
; Memory blocks                          ;   32%                     ;
; DSP blocks                             ;    0%                     ;
+----------------------------------------+---------------------------;
System name: symmetricswsk
```

Appendix F

Single Work-Item, Single Kernel (Design A) Regular LDPC Kernel Source Code (OpenCL)

```
// Codeword specifications
#define N_VARIABLE_NODES 256
#define N_CHECK_NODES 128
#define MAX_VARIABLE_EDGES 3
#define MAX_CHECK_EDGES 6

#define N_VARIABLE_INPUTS (MAX_VARIABLE_EDGES)
#define N_CHECK_INPUTS (MAX_CHECK_EDGES)

#define N_VARIABLE_INPUTS_W_CHANNEL (N_VARIABLE_INPUTS+1)



// This is for the local memory now
// This is a change
// Integer options
// Define the LLR type

#define MANUAL_VECTOR_WIDTH (16)


#if MANUAL_VECTOR_WIDTH==2
#define LLR_TYPE char2
#define TOTAL_TYPE short2
#define CONVERT_TO_SHORT(a) convert_short2(a)
#define CONVERT_TO_CHAR(a) convert_char2(a)
#elif MANUAL_VECTOR_WIDTH==4
#define LLR_TYPE char4
#define TOTAL_TYPE short4
#define CONVERT_TO_SHORT(a) convert_short4(a)
#define CONVERT_TO_CHAR(a) convert_char4(a)
#elif MANUAL_VECTOR_WIDTH==8
#define LLR_TYPE char8
#define TOTAL_TYPE short8
#define CONVERT_TO_SHORT(a) convert_short8(a)
#define CONVERT_TO_CHAR(a) convert_char8(a)
#elif MANUAL_VECTOR_WIDTH==16
#define LLR_TYPE char16
#define TOTAL_TYPE short16
#define CONVERT_TO_SHORT(a) convert_short16(a)
#define CONVERT_TO_CHAR(a) convert_char16(a)
#else
#define LLR_TYPE char
#define TOTAL_TYPE short
#define CONVERT_TO_SHORT(a) convert_short(a)
#define CONVERT_TO_CHAR(a) convert_char(a)
#endif


typedef LLR_TYPE LLR;

// Number of decoding iterations
#define N_ITERATIONS (32)
```

```
// We want to create a kernel that does 4 simultaneous codewords using the char4 operative
// There will just be one giant kernel that handles the entire execution path
__kernel __attribute__((reqd_work_group_size(1,1,1))) void LDPCDecoder(
// Let's start with the constant inputs for the kernel

__global LLR * restrict vnInputs,
__global LLR * restrict cnInputs,
// We need a table for the variable nodes to determine their checknodes
// And then another table to determine the index into the check nodes
__constant short * restrict vntocnIndexLocation, // The size of this is num_VN by
    ↪ N_VN_INPUTS

// Now we need the data from the CN to the VN
__constant short * restrict cntovnIndexLocation, // The size of this is num_CN by
    ↪ N_CN_INPUTS

// Now that we have the connections we need other constants
// These are vector constants that are computed on the host
// These values are used in the vectored calculations throughout the operation
__constant LLR * restrict vector_max_llr,
__constant LLR * restrict vector_sign_bit,
__constant LLR * restrict allones,
__constant LLR * restrict ones,
__constant LLR * restrict zeros,
__constant TOTAL_TYPE * restrict total_vector_max_llr,
__constant TOTAL_TYPE * restrict total_vector_min_neg_llr,

// Now we just have the inputs releated to the initialization and outputs
// First we put in the channel measurements
// These are also passed via constant memory
__global LLR * restrict channelMeasurements, // This is of size num_VN by sizeof(LLR)
    ↪ which depends on the vectorization


// Now that all the constants are done, we just need an output
__global LLR * restrict vnResults, // Size is num_VN by LLR -> Simply the total of all
    ↪ inputs
__global LLR * restrict cnResults // Size is num_CN by LLR -> Simply the parity

) {

// This architecture is a single kernel design for all of the codes
// The VN operate and send data to the CN via the local memory
// Unfortunately we cannot pipeline the VN and CN calculations here


  // Now we need to define the internal storages (aka the interleaver)
  // The cnInputs variable is rounded up to the nearest power of two in terms of local
      ↪ memory storage
  // This is as per suggestion from the AOC optimization guide manual. The nearest value
      ↪ is always equal to the storage
  // produced by the Variable Nodes

  // Variable Node Private Variables
  // In terms of efficiency we will store the inputs into registers for easier access

  // We also need a short version of the LLR values to total
  TOTAL_TYPE vnInputsShort[N_VARIABLE_INPUTS_W_CHANNEL];


  // Variables for temporary values
  LLR vnCurrent;
  TOTAL_TYPE vnTempCurrent;
  LLR vnConditional;
  LLR vnSignB;
  LLR vnMag;



  // First we initialize for the Variable Nodes
  #pragma unroll
  for(int i=0; i<N_VARIABLE_NODES; i++)
    vnInputs[i*N_VARIABLE_INPUTS_W_CHANNEL + N_VARIABLE_INPUTS] = channelMeasurements[i];



  // Check Node Private Variables
  LLR cnFirstMin;
  LLR cnSecondMin;


  // And also the magnitudes
  LLR cnInputsMag[N_CHECK_INPUTS];
  LLR cnParity;



  // Now we step through the iterations (aka start the loop)
  for(char iter = 0; iter < N_ITERATIONS; iter++) {
```

```
// Variable Node
for(int node =0; node<N_VARIABLE_NODES; node++) {

  int vnInputsBaseIndex = node*N_VARIABLE_INPUTS_W_CHANNEL;
  int vntocnBaseIndex = node*N_VARIABLE_INPUTS;
  // This process starts with the Variable Nodes
  // First, we will load up our private values
  // At this point we will also total up a value
  TOTAL_TYPE total = (TOTAL_TYPE)(0);

  #pragma unroll
  for(char input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {

    // Store the short versions
    vnInputsShort[input] = CONVERT_TO_SHORT(vnInputs[vnInputsBaseIndex + input]);

    // Total up the values
    total += vnInputsShort[input];
  }

  /*for(short input = 0; input<N_VARIABLE_INPUTS_W_CHANNEL; input++) {
    //printf("Iteration,%d,VNode,%d,inputs,%d,%d,total,%d\n",iter,id,input,
        ↪ vnInputsPrivate[input],total);
    //printf("Iteration,%d,VNode,%d,inputs,%d,%d,total,%d\n",iter,id_second_variable,
        ↪ input,vnInputsPrivate_2[input],total_2);
  }*/

  // We now have the totals
  // Now we iterate over all the inputs and subtract their own value to get the result
  #pragma unroll
  for(char input = 0; input < N_VARIABLE_INPUTS; input++) {
    // Subtract the current from the total to get the output
    // We simply subtract from the large total and then saturate it down to the
        ↪ appropriate size
    vnTempCurrent = total - vnInputsShort[input];


    // Now saturate the value aka make sure it's MIN_LLR < n < MAX_LLR
    vnTempCurrent =   ((vnTempCurrent >= (*total_vector_max_llr)) ?
               (*total_vector_max_llr) :
               ((vnTempCurrent <= (*total_vector_min_neg_llr)) ?
               (*total_vector_min_neg_llr) :
               vnTempCurrent));

    // Now we reverse the procedure and store the data back into the LLR datatype
    vnCurrent = CONVERT_TO_CHAR(vnTempCurrent);

    // Now we can convert to sign and magnitude
    // Find out if it's negative
    vnConditional = ((vnCurrent & (*vector_sign_bit)) == (*vector_sign_bit)) ? (*
        ↪ allones) : (*zeros);


    // If it's negative we need to xor it and add one and mask out the sign bit
    vnMag = ((vnConditional & ((vnCurrent ^ (*allones)) + (*ones))) | ((vnConditional
        ↪ ^ (*allones)) & vnCurrent)) & (*vector_max_llr);

    // Now get the sign bit
    vnSignB = (vnConditional & (*vector_sign_bit));


    // Now we just need to find the location in the array to place the data
    short cnIndex = vntocnIndexLocation[vntocnBaseIndex + input];

    // Just put the data in the array for the check node
    cnInputs[cnIndex] = vnMag | vnSignB;

    //printf("Iteration,%d,VNode,%d,output,%d,sign,%d,mag,%d,2's,%d,cnIndex,%d\n",iter
        ↪ ,id,input,vnSignB,vnMag,vnCurrent,cnIndex);
    //printf("Iteration,%d,VNode,%d,output,%d,sign,%d,mag,%d,2's,%d,cnIndex,%d\n",iter
        ↪ ,id_second_variable,input,vnSignB_2,vnMag_2,vnCurrent_2,cnIndex_2);


  }

  if((iter+1) == N_ITERATIONS)
    vnResults[node] = vnCurrent;
}




// We have now completed the variable nodes
// Let's do the Check Nodes
//Check Node
for(int node=0; node<N_CHECK_NODES; node++) {

  int cnInputsBaseIndex = node*N_CHECK_INPUTS;
  // Now in each iteration we need to do the CN calculation first
```

```
cnFirstMin = (*vector_max_llr);
cnSecondMin = (*vector_max_llr);

// First let's do the minimum calculations

// Instead of the for loop we'll do a tournament structure
// There are 6 inputs total
// This means we'll need three rounds to get our minimum
// First round we break them into the following pairs
// 0 < 1    2  < 3    4 < 5
// with the winners advancing and the losers being sent to the second min
    ↪ calculation
// For ease let's assume that the 4 < 5 is done in the second round and is given a
    ↪ by as it would be a 3 - 1 comparison otherwise
// Here's an example:   0 < 1  True     2 < 3  True
//              0 < 2  True   4 < 5  False
//                  0 < 5 False
//                    5 is the min

// So here we go, round 1

// NOTE THAT I HAVE TO MULTIPLY BY NEGATIVE_(*ones) FOR EACH COMPARISON BECAUSE THE
    ↪ RESULT RETURNED IS 1111 1111 IN CHAR FORM
// THIS MEANS THAT WE CAN BITWISE AND THE RESULT TO GET OUR ANSWER

// But first, we need to make sure that we are taking the absolute value as we don't
    ↪  want to include that in the min calculation
// Therefore, we have to mask off the sign bits
#pragma unroll
for(short input = 0; input < N_CHECK_INPUTS; input++) {
  cnInputsMag[input] = cnInputs[cnInputsBaseIndex + input] & (*vector_max_llr);
}

// Now let's get the parity
cnParity =(cnInputs[cnInputsBaseIndex + 0] & (*vector_sign_bit)) ^
       (cnInputs[cnInputsBaseIndex + 1] & (*vector_sign_bit)) ^
       (cnInputs[cnInputsBaseIndex + 2] & (*vector_sign_bit)) ^
       (cnInputs[cnInputsBaseIndex + 3] & (*vector_sign_bit)) ^
       (cnInputs[cnInputsBaseIndex + 4] & (*vector_sign_bit)) ^
       (cnInputs[cnInputsBaseIndex + 5] & (*vector_sign_bit));

// If we're on the last iteration, just send the parity
if((iter+1) == N_ITERATIONS) {
  cnResults[node] = cnParity;
}


// ROUND 1
LLR zero_one_comp = (cnInputsMag[0] < cnInputsMag[1]) ? (*allones) : (*zeros); //
    ↪ Note this will return a value of all 1's
LLR two_three_comp = (cnInputsMag[2] < cnInputsMag[3]) ? (*allones): (*zeros);


// Now let's do some quick organizing to store the winners from this round as a
    ↪ reference (hopefully this will be taken out by the compiler)
// Note that we need to take the compliment using the xor
LLR zero_one_winner = (cnInputsMag[0] & zero_one_comp) | (cnInputsMag[1] & (
    ↪ zero_one_comp ^ (*allones)));
LLR two_three_winner = (cnInputsMag[2] & two_three_comp) | (cnInputsMag[3] & (
    ↪ two_three_comp ^ (*allones)));

//LLR zero_one_winner = (cnInputsMag[0] < cnInputsMag[1] ? cnInputsMag[0] :
    ↪ cnInputsMag[1]);
//LLR two_three_winner = (cnInputsMag[2] < cnInputsMag[3] ? cnInputsMag[2] :
    ↪ cnInputsMag[3]);



// ROUND 2
// In order to maintain ordering I will assume a,b,c,d as the inputs as I don't know
    ↪  the winners
// That is the winner of zero_one is a, two_three is b, 4 is c, and 5 is d
LLR a_b_comp = (zero_one_winner < two_three_winner) ? (*allones) : (*zeros);
LLR c_d_comp = (cnInputsMag[4] < cnInputsMag[5]) ? (*allones) : (*zeros);

// Now let's do some quick organizing again and store the results for the next round
LLR a_b_winner = (zero_one_winner & a_b_comp) | (two_three_winner & (a_b_comp ^ (*
    ↪ allones)));
LLR c_d_winner = (cnInputsMag[4] & c_d_comp) | (cnInputsMag[5] & (c_d_comp ^ (*
    ↪ allones)));

//LLR c_d_winner = (cnInputsMag[4] < cnInputsMag[5] ? cnInputsMag[4] : cnInputsMag
    ↪ [5]);
//LLR a_b_winner = (zero_one_winner < two_three_winner ? zero_one_winner :
    ↪ two_three_winner);

// ROUND 3
// Now this is the final round where we determine the ultimate minimum
// It is simply against the two winners
LLR first_min_comp = (a_b_winner < c_d_winner) ? (*allones) : (*zeros);

//LLR first_min_comp = (a_b_winner < c_d_winner ? a_b_winner : c_d_winner);
```

```
    // Let's store the result!
    cnFirstMin = (a_b_winner & first_min_comp) | (c_d_winner & (first_min_comp ^ (*
        ↪ allones)));



    // Now we need to get the second minimum
    // But we know the winners from the previous rounds and the losers
    // Let's gather all the losers and run it all again
    // First start by gathering all the losers
    // This is done by simply inverting the results
    LLR zero_second_min = (cnInputsMag[0] & (zero_one_comp ^ (*allones))) | (cnInputsMag
        ↪ [1] & zero_one_comp);
    LLR one_second_min = (cnInputsMag[2] & (two_three_comp ^ (*allones))) | (cnInputsMag
        ↪ [3] & two_three_comp);
    LLR two_second_min = (zero_one_winner & (a_b_comp ^ (*allones))) | (two_three_winner
        ↪  & a_b_comp);
    LLR three_second_min = (cnInputsMag[4] & (c_d_comp ^ (*allones))) | (cnInputsMag[5]
        ↪ & c_d_comp);
    LLR four_second_min = (a_b_winner & (first_min_comp ^ (*allones))) | (c_d_winner &
        ↪ first_min_comp);

    /*LLR zero_second_min = (cnInputsMag[0] < cnInputsMag[1] ? cnInputsMag[1] :
        ↪ cnInputsMag[0]);
    LLR one_second_min = (cnInputsMag[2] < cnInputsMag[3] ? cnInputsMag[3] : cnInputsMag
        ↪ [2]);
    LLR two_second_min = (zero_one_winner < two_three_winner ? two_three_winner :
        ↪ zero_one_winner);
    LLR three_second_min = (cnInputsMag[4] < cnInputsMag[5] ? cnInputsMag[5] :
        ↪ cnInputsMag[4]);
    LLR four_second_min = (a_b_winner < c_d_winner ? c_d_winner : a_b_winner);*/

    // Now that we have our 5 we can start the new tournament
    // This one will also be 3 rounds
    // Let's say the last one will get the pass
    // Here is the tournament structure
    // 0 < 1 True 2 < 3 True
    // 0 < 2 True 0 < 4 If false then 0 is the result
    // 0 < 4 True
    // 0 is the min

    // ROUND 1
    // Note there is an option in the above to compare 3 < 4 in the second round. If 3
        ↪ is less than 4 then we know the winner right away
    LLR zero_one_second_comp = (zero_second_min < one_second_min) ? (*allones) : (*zeros
        ↪ );
    LLR two_three_second_comp = (two_second_min < three_second_min) ? (*allones) : (*
        ↪ zeros);

    // Now determine the winners
    LLR zero_one_second_winner = (zero_second_min & zero_one_second_comp) | (
        ↪ one_second_min & (zero_one_second_comp ^ (*allones)));
    LLR two_three_second_winner = (two_second_min & two_three_second_comp) | (
        ↪ three_second_min & (two_three_second_comp ^ (*allones)));

    // ROUND 2
    // Just like with the first min, we will now switch over to using letters so we aren
        ↪ 't confused and don't have large names
    // 0 < 1 -> a  2 < 3 -> b  4 -> c
    LLR a_b_second_comp = (zero_one_second_winner < two_three_second_winner) ? (*allones
        ↪ ) : (*zeros);

    // Now store the winner
    LLR a_b_second_winner = (zero_one_second_winner & a_b_second_comp) | (
        ↪ two_three_second_winner & (a_b_second_comp ^ (*allones)));

    // FINAL ROUND
    // Now we get the second min
    LLR second_min_comp = (a_b_second_winner < four_second_min) ? (*allones) : (*zeros);

    // Store the result
    cnSecondMin = (a_b_second_winner & second_min_comp) | (four_second_min & (
        ↪ second_min_comp ^ (*allones)));

    // Yay we have both now



    // Now that we have the minimums and the parity, we should convert the first and
        ↪ second minimums to 2's complement assuming negative
    LLR first_min_neg = (((cnFirstMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));

    LLR second_min_neg = (((cnSecondMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));

    // Now we have positive and negative versions of all
    // Let's print out all the info

    /*for(short input = 0; input<N_CHECK_INPUTS; input++) {
      //printf("Iteration,%d,CNode,%d,inputs,%d,%d,first_min,%d,second_min,%d,parity,%d\
          ↪ n",iter,id,input,cnInputsPrivate[input],cnFirstMin,cnSecondMin,cnParity);
    }*/
```

```
      // Now let's start the check nodes
      // We only need to execute one check node for each calculation as there are
          ↪ N_CHECK_NODE number of work-items
      #pragma unroll
      for(char input = 0; input < N_CHECK_INPUTS; input++) {

        // For each of these we need to determine if it was the first minimum
        // We just need a conditional that has either 1 or 0 depending if it's min or
            ↪ second min
        // And then we can make the expression LLR to_vnode = cnFirstMin*condition1 +
            ↪ cnSecondMin*inv(condition1)
        LLR min_conditional = ((cnInputs[cnInputsBaseIndex + input] & (*vector_max_llr))
            ↪ == cnFirstMin) ? (*allones) : (*zeros);

        // the LLR conditional variable will now have a vector of either 0 or 1.
        // Now we can just use the expression

        // Now we can move on to calculating the output
        LLR sign_b = cnParity;

        // The sign bit is assigned to complete the parity compared to the other inputs
        sign_b ^= cnInputs[cnInputsBaseIndex + input] & (*vector_sign_bit);

        // Now we just send the data in 2's complement which was already calculated
        LLR sign_conditional = (sign_b == (*vector_sign_bit)) ? (*allones) : (*zeros);

        // If it's negative, give it the negative value, if not then the positive
        // Start with cnSecondMin, second_min_neg, cnFirstMin, first_min_neg
        LLR to_vnode =  ((min_conditional & sign_conditional & second_min_neg)) |
              ((min_conditional & (sign_conditional ^ (*allones)) & cnSecondMin)) |
              ((min_conditional ^ (*allones)) & sign_conditional & first_min_neg) |
              ((min_conditional ^ (*allones)) & (sign_conditional ^ (*allones)) &
                  ↪ cnFirstMin);



        // Now we just need to find the variable node index
        short vnIndex = cntovnIndexLocation[cnInputsBaseIndex + input];


        vnInputs[vnIndex] = to_vnode;

        //printf("Iteration,%d,CNode,%d,output,%d,%d,vnIndex,%d\n",iter,id,input,to_vnode,
            ↪ vnIndex);

      }
    }

  }



}
```

Appendix G

NDRange Decoder Regular LDPC Kernel (Design B) Source Code (OpenCL)

```
#ifndef N_VARIABLE_NODES
// Codeword specifications
#define N_VARIABLE_NODES 32
#endif

#define N_CHECK_NODES (N_VARIABLE_NODES/2)
#define MAX_VARIABLE_EDGES 3
#define MAX_CHECK_EDGES 6


#define N_VARIABLE_INPUTS (MAX_VARIABLE_EDGES)
#define N_CHECK_INPUTS (MAX_CHECK_EDGES)

#define N_VARIABLE_INPUTS_W_CHANNEL (N_VARIABLE_INPUTS+1)




// This is for the local memory now
// This is a change
// Integer options
// Define the LLR type

#define MANUAL_VECTOR_WIDTH (16)



#if MANUAL_VECTOR_WIDTH==2
#define LLR_TYPE char2
#define TOTAL_TYPE short2
#define CONVERT_TO_SHORT(a) convert_short2(a)
#define CONVERT_TO_CHAR(a) convert_char2(a)
#elif MANUAL_VECTOR_WIDTH==4
#define LLR_TYPE char4
#define TOTAL_TYPE short4
#define CONVERT_TO_SHORT(a) convert_short4(a)
#define CONVERT_TO_CHAR(a) convert_char4(a)
#elif MANUAL_VECTOR_WIDTH==8
#define LLR_TYPE char8
#define TOTAL_TYPE short8
#define CONVERT_TO_SHORT(a) convert_short8(a)
#define CONVERT_TO_CHAR(a) convert_char8(a)
#elif MANUAL_VECTOR_WIDTH==16
#define LLR_TYPE char16
#define TOTAL_TYPE short16
#define CONVERT_TO_SHORT(a) convert_short16(a)
#define CONVERT_TO_CHAR(a) convert_char16(a)
#else
#define LLR_TYPE char
#define TOTAL_TYPE short
#define CONVERT_TO_SHORT(a) convert_short(a)
#define CONVERT_TO_CHAR(a) convert_char(a)
#endif


typedef LLR_TYPE LLR;
```

```
// Number of decoding iterations
#define N_ITERATIONS (32)




// We want to create a kernel that does 4 simultaneous codewords using the char4 operative
// There will just be one giant kernel that handles the entire execution path
__kernel __attribute__((reqd_work_group_size(N_CHECK_NODES,1,1))) void LDPCDecoder(
// Let's start with the constant inputs for the kernel
// We need a table for the variable nodes to determine their checknodes
// And then another table to determine the index into the check nodes
__constant short * restrict vntocnIndexLocation, // The size of this is num_VN by
    ↪ N_VN_INPUTS

// Now we need the data from the CN to the VN
__constant short * restrict cntovnIndexLocation, // The size of this is num_CN by
    ↪ N_CN_INPUTS

// Now that we have the connections we need other constants
// These are vector constants that are computed on the host
// These values are used in the vectored calculations throughout the operation
__constant LLR * restrict vector_max_llr,
__constant LLR * restrict vector_sign_bit,
__constant LLR * restrict allones,
__constant LLR * restrict ones,
__constant LLR * restrict zeros,
__constant TOTAL_TYPE * restrict total_vector_max_llr,
__constant TOTAL_TYPE * restrict total_vector_min_neg_llr,

// Now we just have the inputs releated to the initialization and outputs
// First we put in the channel measurements
// These are also passed via constant memory
__global LLR * restrict channelMeasurements, // This is of size num_VN by sizeof(LLR)
    ↪ which depends on the vectorization


// Now that all the constants are done, we just need an output
__global LLR * restrict vnResults, // Size is num_VN by LLR -> Simply the total of all
    ↪ inputs
__global LLR * restrict cnResults // Size is num_CN by LLR -> Simply the parity

) {

// This architecture is a single kernel design for all of the codes
// The VN operate and send data to the CN via the local memory
// Unfortunately we cannot pipeline the VN and CN calculations here

  // First we need to get our global id
  int id = get_global_id(0);

  // Get the second vnIndex
  int id_second_variable = id + N_CHECK_NODES;

  // Now we need to define the internal storages (aka the interleaver)
  __local LLR vnInputs[N_VARIABLE_NODES * N_VARIABLE_INPUTS_W_CHANNEL]; // VN Storage
  __local LLR cnInputs[N_CHECK_NODES * N_CHECK_INPUTS]; // CN Storage
  // The cnInputs variable is rounded up to the nearest power of two in terms of local
      ↪ memory storage
  // This is as per suggestion from the AOC optimization guide manual. The nearest value
      ↪ is always equal to the storage
  // produced by the Variable Nodes

  // Variable Node Private Variables
  // In terms of efficiency we will store the inputs into registers for easier access
  LLR vnInputsPrivate[N_VARIABLE_INPUTS_W_CHANNEL];
  LLR vnInputsPrivate_2[N_VARIABLE_INPUTS_W_CHANNEL];

  // We also need a short version of the LLR values to total
  TOTAL_TYPE vnInputsShort[N_VARIABLE_INPUTS_W_CHANNEL];
  TOTAL_TYPE vnInputsShort_2[N_VARIABLE_INPUTS_W_CHANNEL];

  // Variables for temporary values
  LLR vnCurrent;
  LLR vnCurrent_2;
  TOTAL_TYPE vnTempCurrent;
  TOTAL_TYPE vnTempCurrent_2;
  LLR vnConditional;
  LLR vnConditional_2;
  LLR vnSignB;
  LLR vnSignB_2;
  LLR vnMag;
  LLR vnMag_2;
  int vnInputsBaseIndex = id * N_VARIABLE_INPUTS_W_CHANNEL;
  int vnInputsBaseIndex_2 = id_second_variable * N_VARIABLE_INPUTS_W_CHANNEL;
  int vntocnBaseIndex = id * N_VARIABLE_INPUTS;
  int vntocnBaseIndex_2 = id_second_variable * N_VARIABLE_INPUTS;


  // First we initialize for the Variable Nodes
  // Note here that there are only N_CHECK_NODES number of work-items
  // So each work-item needs to initialize two values
```

```
vnInputs[vnInputsBaseIndex + N_VARIABLE_INPUTS] = channelMeasurements[id];
vnInputs[vnInputsBaseIndex_2 + N_VARIABLE_INPUTS] = channelMeasurements[
    ↪ id_second_variable];



// Check Node Private Variables
LLR cnFirstMin;
LLR cnSecondMin;

// Let's store the private inputs
LLR cnInputsPrivate[N_CHECK_INPUTS];

// And also the magnitudes
LLR cnInputsMag[N_CHECK_INPUTS];

int cnInputsBaseIndex = id*N_CHECK_INPUTS;
LLR cnParity;



// Now we step through the iterations (aka start the loop)
for(char iter =  0; iter < N_ITERATIONS; iter++) {



  // Variable Node

  // This process starts with the Variable Nodes
  // First, we will load up our private values
  // At this point we will also total up a value
  TOTAL_TYPE total = (TOTAL_TYPE)(0);
  TOTAL_TYPE total_2 = (TOTAL_TYPE)(0);
  #pragma unroll
  for(char input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {

    // Store the data
    vnInputsPrivate[input] = vnInputs[vnInputsBaseIndex + input];
    vnInputsPrivate_2[input] = vnInputs[vnInputsBaseIndex_2 + input];

    // Store the short versions
    vnInputsShort[input] = CONVERT_TO_SHORT(vnInputsPrivate[input]);
    vnInputsShort_2[input] = CONVERT_TO_SHORT(vnInputsPrivate_2[input]);

    // Total up the values
    total += vnInputsShort[input];
    total_2 += vnInputsShort_2[input];
  }

  /*for(short input = 0; input<N_VARIABLE_INPUTS_W_CHANNEL; input++) {
    //printf("Iteration,%d,VNode,%d,inputs,%d,%d,total,%d\n",iter,id,input,
        ↪ vnInputsPrivate[input],total);
    //printf("Iteration,%d,VNode,%d,inputs,%d,%d,total,%d\n",iter,id_second_variable,
        ↪ input,vnInputsPrivate_2[input],total_2);
  }*/

  // We now have the totals
  // Now we iterate over all the inputs and subtract their own value to get the result
  #pragma unroll
  for(char input = 0; input < N_VARIABLE_INPUTS; input++) {
    // Subtract the current from the total to get the output
    // We simply subtract from the large total and then saturate it down to the
        ↪ appropriate size
    vnTempCurrent = total - vnInputsShort[input];
    vnTempCurrent_2 = total_2 - vnInputsShort_2[input];

    // Now saturate the value aka make sure it's MIN_LLR < n < MAX_LLR
    vnTempCurrent =   ((vnTempCurrent >= (*total_vector_max_llr)) ?
              (*total_vector_max_llr) :
              ((vnTempCurrent <= (*total_vector_min_neg_llr)) ?
              (*total_vector_min_neg_llr) :
              vnTempCurrent));
    vnTempCurrent_2 =   ((vnTempCurrent_2 >= (*total_vector_max_llr)) ?
              (*total_vector_max_llr) :
              ((vnTempCurrent_2 <= (*total_vector_min_neg_llr)) ?
              (*total_vector_min_neg_llr) :
              vnTempCurrent_2));

    // Now we reverse the procedure and store the data back into the LLR datatype
    vnCurrent = CONVERT_TO_CHAR(vnTempCurrent);
    vnCurrent_2 = CONVERT_TO_CHAR(vnTempCurrent_2);

    // Now we can convert to sign and magnitude
    // Find out if it's negative
    vnConditional = ((vnCurrent & (*vector_sign_bit)) == (*vector_sign_bit)) ? (*allones
        ↪ ) : (*zeros);
    vnConditional_2 = ((vnCurrent_2 & (*vector_sign_bit)) == (*vector_sign_bit)) ? (*
        ↪ allones) : (*zeros);


    // If it's negative we need to xor it and add one and mask out the sign bit
```

```
      vnMag = ((vnConditional & ((vnCurrent ^ (*allones)) + (*ones))) | ((vnConditional ^
           ↪ (*allones)) & vnCurrent)) & (*vector_max_llr);
      vnMag_2 = ((vnConditional_2 & ((vnCurrent_2 ^ (*allones)) + (*ones))) | ((
           ↪ vnConditional_2 ^ (*allones)) & vnCurrent_2)) & (*vector_max_llr);

      // Now get the sign bit
      vnSignB = (vnConditional & (*vector_sign_bit));
      vnSignB_2 = (vnConditional_2 & (*vector_sign_bit));


      // Now we just need to find the location in the array to place the data
      short cnIndex = vntocnIndexLocation[vntocnBaseIndex + input];
      short cnIndex_2 = vntocnIndexLocation[vntocnBaseIndex_2 + input];

      // Just put the data in the array for the check node
      cnInputs[cnIndex] = vnMag | vnSignB;
      cnInputs[cnIndex_2] = vnMag_2 | vnSignB_2;

      //printf("Iteration,%d,VNode,%d,output,%d,sign,%d,mag,%d,2's,%d,cnIndex,%d\n",iter,
           ↪ id,input,vnSignB,vnMag,vnCurrent,cnIndex);
      //printf("Iteration,%d,VNode,%d,output,%d,sign,%d,mag,%d,2's,%d,cnIndex,%d\n",iter,
           ↪ id_second_variable,input,vnSignB_2,vnMag_2,vnCurrent_2,cnIndex_2);


    }



    // We have now completed the variable nodes
    // However, we must wait for all the variable nodes to completely finish before
         ↪ continuing
    barrier(CLK_LOCAL_MEM_FENCE);


    //Check Node

    // Now in each iteration we need to do the CN calculation first
    cnFirstMin = (*vector_max_llr);
    cnSecondMin = (*vector_max_llr);

    // First let's do the minimum calculations

    // Instead of the for loop we'll do a tournament structure
    // There are 6 inputs total
    // This means we'll need three rounds to get our minimum
    // First round we break them into the following pairs
    // 0 < 1    2  < 3    4 < 5
    // with the winners advancing and the losers being sent to the second min calculation
    // For ease let's assume that the 4 < 5 is done in the second round and is given a by
         ↪ as it would be a 3 - 1 comparison otherwise
    // Here's an example:   0 < 1  True     2 < 3  True
    //            0 < 2  True    4 < 5  False
    //                0 < 5 False
    //                  5 is the min

    // So here we go, round 1

    // NOTE THAT I HAVE TO MULTIPLY BY NEGATIVE_(*ones) FOR EACH COMPARISON BECAUSE THE
         ↪ RESULT RETURNED IS 1111 1111 IN CHAR FORM
    // THIS MEANS THAT WE CAN BITWISE AND THE RESULT TO GET OUR ANSWER

    // But first, we need to make sure that we are taking the absolute value as we don't
         ↪ want to include that in the min calculation
    // Therefore, we have to mask off the sign bits
    #pragma unroll
    for(short input = 0; input < N_CHECK_INPUTS; input++) {
      cnInputsPrivate[input] = cnInputs[cnInputsBaseIndex + input];
      cnInputsMag[input] = cnInputsPrivate[input] & (*vector_max_llr);
    }

    // Now let's get the parity
    cnParity =(cnInputsPrivate[0] & (*vector_sign_bit)) ^
          (cnInputsPrivate[1] & (*vector_sign_bit)) ^
          (cnInputsPrivate[2] & (*vector_sign_bit)) ^
          (cnInputsPrivate[3] & (*vector_sign_bit)) ^
          (cnInputsPrivate[4] & (*vector_sign_bit)) ^
          (cnInputsPrivate[5] & (*vector_sign_bit));

    // If we're on the last iteration, just send the parity
    if((iter+1) == N_ITERATIONS) {
      cnResults[id] = cnParity;
    }


    // ROUND 1
    LLR zero_one_comp = (cnInputsMag[0] < cnInputsMag[1]) ? (*allones) : (*zeros); // Note
         ↪  this will return a value of all 1's
    LLR two_three_comp = (cnInputsMag[2] < cnInputsMag[3]) ? (*allones): (*zeros);


    // Now let's do some quick organizing to store the winners from this round as a
         ↪ reference (hopefully this will be taken out by the compiler)
```

```
// Note that we need to take the compliment using the xor
LLR zero_one_winner = (cnInputsMag[0] & zero_one_comp) | (cnInputsMag[1] & (
    ↪ zero_one_comp ^ (*allones)));
LLR two_three_winner = (cnInputsMag[2] & two_three_comp) | (cnInputsMag[3] & (
    ↪ two_three_comp ^ (*allones)));

//LLR zero_one_winner = (cnInputsMag[0] < cnInputsMag[1] ? cnInputsMag[0] :
    ↪ cnInputsMag[1]);
//LLR two_three_winner = (cnInputsMag[2] < cnInputsMag[3] ? cnInputsMag[2] :
    ↪ cnInputsMag[3]);



// ROUND 2
// In order to maintain ordering I will assume a,b,c,d as the inputs as I don't know
    ↪ the winners
// That is the winner of zero_one is a, two_three is b, 4 is c, and 5 is d
LLR a_b_comp = (zero_one_winner < two_three_winner) ? (*allones) : (*zeros);
LLR c_d_comp = (cnInputsMag[4] < cnInputsMag[5]) ? (*allones) : (*zeros);

// Now let's do some quick organizing again and store the results for the next round
LLR a_b_winner = (zero_one_winner & a_b_comp) | (two_three_winner & (a_b_comp ^ (*
    ↪ allones)));
LLR c_d_winner = (cnInputsMag[4] & c_d_comp) | (cnInputsMag[5] & (c_d_comp ^ (*allones
    ↪ )));

//LLR c_d_winner = (cnInputsMag[4] < cnInputsMag[5] ? cnInputsMag[4] : cnInputsMag[5])
    ↪ ;
//LLR a_b_winner = (zero_one_winner < two_three_winner ? zero_one_winner :
    ↪ two_three_winner);

// ROUND 3
// Now this is the final round where we determine the ultimate minimum
// It is simply against the two winners
LLR first_min_comp = (a_b_winner < c_d_winner) ? (*allones) : (*zeros);

//LLR first_min_comp = (a_b_winner < c_d_winner ? a_b_winner : c_d_winner);

// Let's store the result!
cnFirstMin = (a_b_winner & first_min_comp) | (c_d_winner & (first_min_comp ^ (*allones
    ↪ )));



// Now we need to get the second minimum
// But we know the winners from the previous rounds and the losers
// Let's gather all the losers and run it all again
// First start by gathering all the losers
// This is done by simply inverting the results
LLR zero_second_min = (cnInputsMag[0] & (zero_one_comp ^ (*allones))) | (cnInputsMag
    ↪ [1] & zero_one_comp);
LLR one_second_min = (cnInputsMag[2] & (two_three_comp ^ (*allones))) | (cnInputsMag
    ↪ [3] & two_three_comp);
LLR two_second_min = (zero_one_winner & (a_b_comp ^ (*allones))) | (two_three_winner &
    ↪  a_b_comp);
LLR three_second_min = (cnInputsMag[4] & (c_d_comp ^ (*allones))) | (cnInputsMag[5] &
    ↪ c_d_comp);
LLR four_second_min = (a_b_winner & (first_min_comp ^ (*allones))) | (c_d_winner &
    ↪ first_min_comp);

/*LLR zero_second_min = (cnInputsMag[0] < cnInputsMag[1] ? cnInputsMag[1] :
    ↪ cnInputsMag[0]);
LLR one_second_min = (cnInputsMag[2] < cnInputsMag[3] ? cnInputsMag[3] : cnInputsMag
    ↪ [2]);
LLR two_second_min = (zero_one_winner < two_three_winner ? two_three_winner :
    ↪ zero_one_winner);
LLR three_second_min = (cnInputsMag[4] < cnInputsMag[5] ? cnInputsMag[5] : cnInputsMag
    ↪ [4]);
LLR four_second_min = (a_b_winner < c_d_winner ? c_d_winner : a_b_winner);*/

// Now that we have our 5 we can start the new tournament
// This one will also be 3 rounds
// Let's say the last one will get the pass
// Here is the tournament structure
// 0 < 1 True 2 < 3 True
// 0 < 2 True 0 < 4 If false then 0 is the result
// 0 < 4 True
// 0 is the min

// ROUND 1
// Note there is an option in the above to compare 3 < 4 in the second round. If 3 is
    ↪ less than 4 then we know the winner right away
LLR zero_one_second_comp = (zero_second_min < one_second_min) ? (*allones) : (*zeros);
LLR two_three_second_comp = (two_second_min < three_second_min) ? (*allones) : (*zeros
    ↪ );

// Now determine the winners
LLR zero_one_second_winner = (zero_second_min & zero_one_second_comp) | (
    ↪ one_second_min & (zero_one_second_comp ^ (*allones)));
LLR two_three_second_winner = (two_second_min & two_three_second_comp) | (
    ↪ three_second_min & (two_three_second_comp ^ (*allones)));
    // ROUND 2
    // Just like with the first min, we will now switch over to using letters so we aren't
        ↪  confused and don't have large names
    // 0 < 1 -> a  2 < 3 -> b  4 -> c
    LLR a_b_second_comp = (zero_one_second_winner < two_three_second_winner) ? (*allones)
        ↪ : (*zeros);

    // Now store the winner
    LLR a_b_second_winner = (zero_one_second_winner & a_b_second_comp) | (
        ↪ two_three_second_winner & (a_b_second_comp ^ (*allones)));

    // FINAL ROUND
    // Now we get the second min
    LLR second_min_comp = (a_b_second_winner < four_second_min) ? (*allones) : (*zeros);

    // Store the result
    cnSecondMin = (a_b_second_winner & second_min_comp) | (four_second_min & (
        ↪ second_min_comp ^ (*allones)));

    // Yay we have both now



    // Now that we have the minimums and the parity, we should convert the first and
        ↪ second minimums to 2's complement assuming negative
    LLR first_min_neg = (((cnFirstMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));

    LLR second_min_neg = (((cnSecondMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));

    // Now we have positive and negative versions of all
    // Let's print out all the info

    /*for(short input = 0; input<N_CHECK_INPUTS; input++) {
      //printf("Iteration,%d,CNode,%d,inputs,%d,%d,first_min,%d,second_min,%d,parity,%d\n
          ↪ ",iter,id,input,cnInputsPrivate[input],cnFirstMin,cnSecondMin,cnParity);
    }*/

    // Now let's start the check nodes
    // We only need to execute one check node for each calculation as there are
        ↪ N_CHECK_NODE number of work-items
   #pragma unroll
    for(char input = 0; input < N_CHECK_INPUTS; input++) {

      // For each of these we need to determine if it was the first minimum
      // We just need a conditional that has either 1 or 0 depending if it's min or second
          ↪  min
      // And then we can make the expression LLR to_vnode = cnFirstMin*condition1 +
          ↪ cnSecondMin*inv(condition1)
      LLR min_conditional = ((cnInputsPrivate[input] & (*vector_max_llr)) == cnFirstMin) ?
          ↪  (*allones) : (*zeros);

      // the LLR conditional variable will now have a vector of either 0 or 1.
      // Now we can just use the expression

      // Now we can move on to calculating the output
      LLR sign_b = cnParity;

      // The sign bit is assigned to complete the parity compared to the other inputs
      sign_b ^= cnInputsPrivate[input] & (*vector_sign_bit);

      // Now we just send the data in 2's complement which was already calculated
      LLR sign_conditional = (sign_b == (*vector_sign_bit)) ? (*allones) : (*zeros);

      // If it's negative, give it the negative value, if not then the positive
      // Start with cnSecondMin, second_min_neg, cnFirstMin, first_min_neg
      LLR to_vnode =  ((min_conditional & sign_conditional & second_min_neg)) |
            ((min_conditional & (sign_conditional ^ (*allones)) & cnSecondMin)) |
            ((min_conditional ^ (*allones)) & sign_conditional & first_min_neg) |
            ((min_conditional ^ (*allones)) & (sign_conditional ^ (*allones)) & cnFirstMin
                ↪ );



      // Now we just need to find the variable node index
      short vnIndex = cntovnIndexLocation[cnInputsBaseIndex + input];

      vnInputs[vnIndex] = to_vnode;

      //printf("Iteration,%d,CNode,%d,output,%d,%d,vnIndex,%d\n",iter,id,input,to_vnode,
          ↪ vnIndex);

    }

    // Finally we just need to put a mem_fence so that no data gets lost
    barrier(CLK_LOCAL_MEM_FENCE);

  }

  // After all the iterations we just need to send back the data
  // For the variable nodes, that means sending the values stored in the 'vnCurrent'
      ↪ variables as these are the totals
  vnResults[id] = vnCurrent;
  vnResults[id + N_CHECK_NODES] = vnCurrent_2;


}
```

Appendix H

Replication of the NDRange Decoder (Design C) Source Code (OpenCL)

```
#ifndef N_VARIABLE_NODES
// Codeword specifications
#define N_VARIABLE_NODES 32
#endif

#define N_CHECK_NODES (N_VARIABLE_NODES/2)
#define MAX_VARIABLE_EDGES 3
#define MAX_CHECK_EDGES 6

#ifndef SIMD
#define SIMD 4
#endif


#define N_VARIABLE_INPUTS (MAX_VARIABLE_EDGES)
#define N_CHECK_INPUTS (MAX_CHECK_EDGES)

#define N_VARIABLE_INPUTS_W_CHANNEL (N_VARIABLE_INPUTS+1)




// This is for the local memory now
// This is a change
// Integer options
// Define the LLR type

#define MANUAL_VECTOR_WIDTH (16)



#if MANUAL_VECTOR_WIDTH==2
#define LLR_TYPE char2
#define TOTAL_TYPE short2
#define CONVERT_TO_SHORT(a) convert_short2(a)
#define CONVERT_TO_CHAR(a) convert_char2(a)
#elif MANUAL_VECTOR_WIDTH==4
#define LLR_TYPE char4
#define TOTAL_TYPE short4
#define CONVERT_TO_SHORT(a) convert_short4(a)
#define CONVERT_TO_CHAR(a) convert_char4(a)
#elif MANUAL_VECTOR_WIDTH==8
#define LLR_TYPE char8
#define TOTAL_TYPE short8
#define CONVERT_TO_SHORT(a) convert_short8(a)
#define CONVERT_TO_CHAR(a) convert_char8(a)
#elif MANUAL_VECTOR_WIDTH==16
#define LLR_TYPE char16
#define TOTAL_TYPE short16
#define CONVERT_TO_SHORT(a) convert_short16(a)
#define CONVERT_TO_CHAR(a) convert_char16(a)
#else
#define LLR_TYPE char
#define TOTAL_TYPE short
#define CONVERT_TO_SHORT(a) convert_short(a)
#define CONVERT_TO_CHAR(a) convert_char(a)
#endif
```

```
typedef LLR_TYPE LLR;

// Number of decoding iterations
#define N_ITERATIONS (32)




// We want to create a kernel that does 4 simultaneous codewords using the char4 operative
// There will just be one giant kernel that handles the entire execution path
__kernel
__attribute__((reqd_work_group_size(N_CHECK_NODES,1,1)))
__attribute__((num_compute_units(1)))
__attribute__((num_simd_work_items(SIMD)))
void LDPCDecoder(
// Let's start with the constant inputs for the kernel
// We need a table for the variable nodes to determine their checknodes
// And then another table to determine the index into the check nodes
__constant short * restrict vntocnIndexLocation, // The size of this is num_VN by
     ↪ N_VN_INPUTS

// Now we need the data from the CN to the VN
__constant short * restrict cntovnIndexLocation, // The size of this is num_CN by
     ↪ N_CN_INPUTS

// Now that we have the connections we need other constants
// These are vector constants that are computed on the host
// These values are used in the vectored calculations throughout the operation
__constant LLR * restrict vector_max_llr,
__constant LLR * restrict vector_sign_bit,
__constant LLR * restrict allones,
__constant LLR * restrict ones,
__constant LLR * restrict zeros,
__constant TOTAL_TYPE * restrict total_vector_max_llr,
__constant TOTAL_TYPE * restrict total_vector_min_neg_llr,

// Now we just have the inputs releated to the initialization and outputs
// First we put in the channel measurements
// These are also passed via constant memory
__global LLR * restrict channelMeasurements, // This is of size num_VN by sizeof(LLR)
     ↪ which depends on the vectorization


// Now that all the constants are done, we just need an output
__global LLR * restrict vnResults, // Size is num_VN by LLR -> Simply the total of all
     ↪ inputs
__global LLR * restrict cnResults // Size is num_CN by LLR -> Simply the parity

) {

// This architecture is a single kernel design for all of the codes
// The VN operate and send data to the CN via the local memory
// Unfortunately we cannot pipeline the VN and CN calculations here

  // First we need to get our global id
  int id = get_global_id(0);

  // Get the second vnIndex
  int id_second_variable = id + N_CHECK_NODES;

  // Now we need to define the internal storages (aka the interleaver)
  __local LLR vnInputs[N_VARIABLE_NODES * N_VARIABLE_INPUTS_W_CHANNEL]; // VN Storage
  __local LLR cnInputs[N_CHECK_NODES * N_CHECK_INPUTS]; // CN Storage
  // The cnInputs variable is rounded up to the nearest power of two in terms of local
       ↪ memory storage
  // This is as per suggestion from the AOC optimization guide manual. The nearest value
       ↪ is always equal to the storage
  // produced by the Variable Nodes

  // Variable Node Private Variables
  // In terms of efficiency we will store the inputs into registers for easier access
  LLR vnInputsPrivate[N_VARIABLE_INPUTS_W_CHANNEL];
  LLR vnInputsPrivate_2[N_VARIABLE_INPUTS_W_CHANNEL];

  // We also need a short version of the LLR values to total
  TOTAL_TYPE vnInputsShort[N_VARIABLE_NODES * N_VARIABLE_INPUTS_W_CHANNEL];
  TOTAL_TYPE vnInputsShort_2[N_VARIABLE_NODES *N_VARIABLE_INPUTS_W_CHANNEL];

  // Variables for temporary values
  LLR vnCurrent;
  LLR vnCurrent_2;
  TOTAL_TYPE vnTempCurrent;
  TOTAL_TYPE vnTempCurrent_2;
  LLR vnConditional;
  LLR vnConditional_2;
  LLR vnSignB;
  LLR vnSignB_2;
  LLR vnMag;
  LLR vnMag_2;
  int vnInputsBaseIndex = id * N_VARIABLE_INPUTS_W_CHANNEL;
  int vnInputsBaseIndex_2 = id_second_variable * N_VARIABLE_INPUTS_W_CHANNEL;
  int vntocnBaseIndex = id * N_VARIABLE_INPUTS;
  int vntocnBaseIndex_2 = id_second_variable * N_VARIABLE_INPUTS;


  // First we initialize for the Variable Nodes
  // Note here that there are only N_CHECK_NODES number of work-items
  // So each work-item needs to initialize two values
  vnInputs[vnInputsBaseIndex + N_VARIABLE_INPUTS] = channelMeasurements[id];
  vnInputs[vnInputsBaseIndex_2 + N_VARIABLE_INPUTS] = channelMeasurements[
      ↪ id_second_variable];



  // Check Node Private Variables
  LLR cnFirstMin;
  LLR cnSecondMin;

  // Let's store the private inputs
  LLR cnInputsPrivate[N_CHECK_INPUTS];

  // And also the magnitudes
  LLR cnInputsMag[N_CHECK_INPUTS];

  int cnInputsBaseIndex = id*N_CHECK_INPUTS;
  LLR cnParity;



  // Now we step through the iterations (aka start the loop)
  for(char iter =  0; iter < N_ITERATIONS; iter++) {



    // Variable Node

    // This process starts with the Variable Nodes
    // First, we will load up our private values
    // At this point we will also total up a value
    TOTAL_TYPE total = (TOTAL_TYPE)(0);
    TOTAL_TYPE total_2 = (TOTAL_TYPE)(0);
    #pragma unroll
    for(char input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {

      // Store the data
      vnInputsPrivate[input] = vnInputs[vnInputsBaseIndex + input];
      vnInputsPrivate_2[input] = vnInputs[vnInputsBaseIndex_2 + input];

      // Store the short versions
      vnInputsShort[input] = CONVERT_TO_SHORT(vnInputsPrivate[input]);
      vnInputsShort_2[input] = CONVERT_TO_SHORT(vnInputsPrivate_2[input]);

      // Total up the values
      total += vnInputsShort[input];
      total_2 += vnInputsShort_2[input];
    }

    /*for(short input = 0; input<N_VARIABLE_INPUTS_W_CHANNEL; input++) {
      //printf("Iteration,%d,VNode,%d,inputs,%d,%d,total,%d\n",iter,id,input,
          ↪ vnInputsPrivate[input],total);
      //printf("Iteration,%d,VNode,%d,inputs,%d,%d,total,%d\n",iter,id_second_variable,
          ↪ input,vnInputsPrivate_2[input],total_2);
    }*/

    // We now have the totals
    // Now we iterate over all the inputs and subtract their own value to get the result
    #pragma unroll
    for(char input = 0; input < N_VARIABLE_INPUTS; input++) {
      // Subtract the current from the total to get the output
      // We simply subtract from the large total and then saturate it down to the
          ↪ appropriate size
      vnTempCurrent = total - vnInputsShort[input];
      vnTempCurrent_2 = total_2 - vnInputsShort_2[input];

      // Now saturate the value aka make sure it's MIN_LLR < n < MAX_LLR
      vnTempCurrent =   ((vnTempCurrent >= (*total_vector_max_llr)) ?
                (*total_vector_max_llr) :
                ((vnTempCurrent <= (*total_vector_min_neg_llr)) ?
                (*total_vector_min_neg_llr) :
                vnTempCurrent));
      vnTempCurrent_2 =   ((vnTempCurrent_2 >= (*total_vector_max_llr)) ?
                (*total_vector_max_llr) :
                ((vnTempCurrent_2 <= (*total_vector_min_neg_llr)) ?
                (*total_vector_min_neg_llr) :
                vnTempCurrent_2));

      // Now we reverse the procedure and store the data back into the LLR datatype
      vnCurrent = CONVERT_TO_CHAR(vnTempCurrent);
      vnCurrent_2 = CONVERT_TO_CHAR(vnTempCurrent_2);

      // Now we can convert to sign and magnitude
      // Find out if it's negative
```

```
      vnConditional = ((vnCurrent & (*vector_sign_bit)) == (*vector_sign_bit)) ? (*allones
          ↪ ) : (*zeros);
      vnConditional_2 = ((vnCurrent_2 & (*vector_sign_bit)) == (*vector_sign_bit)) ? (*
          ↪ allones) : (*zeros);


      // If it's negative we need to xor it and add one and mask out the sign bit
      vnMag = ((vnConditional & ((vnCurrent ^ (*allones)) + (*ones))) | ((vnConditional ^
          ↪ (*allones)) & vnCurrent)) & (*vector_max_llr);
      vnMag_2 = ((vnConditional_2 & ((vnCurrent_2 ^ (*allones)) + (*ones))) | ((
          ↪ vnConditional_2 ^ (*allones)) & vnCurrent_2)) & (*vector_max_llr);

      // Now get the sign bit
      vnSignB = (vnConditional & (*vector_sign_bit));
      vnSignB_2 = (vnConditional_2 & (*vector_sign_bit));


      // Now we just need to find the location in the array to place the data
      short cnIndex = vntocnIndexLocation[vntocnBaseIndex + input];
      short cnIndex_2 = vntocnIndexLocation[vntocnBaseIndex_2 + input];

      // Just put the data in the array for the check node
      cnInputs[cnIndex] = vnMag | vnSignB;
      cnInputs[cnIndex_2] = vnMag_2 | vnSignB_2;

      //printf("Iteration,%d,VNode,%d,output,%d,sign,%d,mag,%d,2's,%d,cnIndex,%d\n",iter,
          ↪ id,input,vnSignB,vnMag,vnCurrent,cnIndex);
      //printf("Iteration,%d,VNode,%d,output,%d,sign,%d,mag,%d,2's,%d,cnIndex,%d\n",iter,
          ↪ id_second_variable,input,vnSignB_2,vnMag_2,vnCurrent_2,cnIndex_2);


    }



    // We have now completed the variable nodes
    // However, we must wait for all the variable nodes to completely finish before
        ↪ continuing
    barrier(CLK_LOCAL_MEM_FENCE);


    //Check Node

    // Now in each iteration we need to do the CN calculation first
    cnFirstMin = (*vector_max_llr);
    cnSecondMin = (*vector_max_llr);

    // First let's do the minimum calculations

    // Instead of the for loop we'll do a tournament structure
    // There are 6 inputs total
    // This means we'll need three rounds to get our minimum
    // First round we break them into the following pairs
    // 0 < 1    2  < 3    4 < 5
    // with the winners advancing and the losers being sent to the second min calculation
    // For ease let's assume that the 4 < 5 is done in the second round and is given a by
        ↪ as it would be a 3 - 1 comparison otherwise
    // Here's an example:   0 < 1  True      2 < 3  True
    //             0  < 2  True    4 < 5  False
    //                 0  < 5 False
    //                   5 is the min

    // So here we go, round 1

    // NOTE THAT I HAVE TO MULTIPLY BY NEGATIVE_(*ones) FOR EACH COMPARISON BECAUSE THE
        ↪ RESULT RETURNED IS 1111 1111 IN CHAR FORM
    // THIS MEANS THAT WE CAN BITWISE AND THE RESULT TO GET OUR ANSWER

    // But first, we need to make sure that we are taking the absolute value as we don't
        ↪ want to include that in the min calculation
    // Therefore, we have to mask off the sign bits
    #pragma unroll
    for(short input = 0; input < N_CHECK_INPUTS; input++) {
      cnInputsPrivate[input] = cnInputs[cnInputsBaseIndex + input];
      cnInputsMag[input] = cnInputsPrivate[input] & (*vector_max_llr);
    }

    // Now let's get the parity
    cnParity =(cnInputsPrivate[0] & (*vector_sign_bit)) ^
          (cnInputsPrivate[1] & (*vector_sign_bit)) ^
          (cnInputsPrivate[2] & (*vector_sign_bit)) ^
          (cnInputsPrivate[3] & (*vector_sign_bit)) ^
          (cnInputsPrivate[4] & (*vector_sign_bit)) ^
          (cnInputsPrivate[5] & (*vector_sign_bit));

    // If we're on the last iteration, just send the parity
    if((iter+1) == N_ITERATIONS) {
      cnResults[id] = cnParity;
    }


    // ROUND 1
    LLR zero_one_comp = (cnInputsMag[0] < cnInputsMag[1]) ? (*allones) : (*zeros); // Note
        ↪  this will return a value of all 1's
    LLR two_three_comp = (cnInputsMag[2] < cnInputsMag[3]) ? (*allones): (*zeros);


    // Now let's do some quick organizing to store the winners from this round as a
        ↪ reference (hopefully this will be taken out by the compiler)
    // Note that we need to take the compliment using the xor
    LLR zero_one_winner = (cnInputsMag[0] & zero_one_comp) | (cnInputsMag[1] & (
        ↪ zero_one_comp ^ (*allones)));
    LLR two_three_winner = (cnInputsMag[2] & two_three_comp) | (cnInputsMag[3] & (
        ↪ two_three_comp ^ (*allones)));

    //LLR zero_one_winner = (cnInputsMag[0] < cnInputsMag[1] ? cnInputsMag[0] :
        ↪ cnInputsMag[1]);
    //LLR two_three_winner = (cnInputsMag[2] < cnInputsMag[3] ? cnInputsMag[2] :
        ↪ cnInputsMag[3]);



    // ROUND 2
    // In order to maintain ordering I will assume a,b,c,d as the inputs as I don't know
        ↪ the winners
    // That is the winner of zero_one is a, two_three is b, 4 is c, and 5 is d
    LLR a_b_comp = (zero_one_winner < two_three_winner) ? (*allones) : (*zeros);
    LLR c_d_comp = (cnInputsMag[4] < cnInputsMag[5]) ? (*allones) : (*zeros);

    // Now let's do some quick organizing again and store the results for the next round
    LLR a_b_winner = (zero_one_winner & a_b_comp) | (two_three_winner & (a_b_comp ^ (*
        ↪ allones)));
    LLR c_d_winner = (cnInputsMag[4] & c_d_comp) | (cnInputsMag[5] & (c_d_comp ^ (*allones
        ↪ )));

    //LLR c_d_winner = (cnInputsMag[4] < cnInputsMag[5] ? cnInputsMag[4] : cnInputsMag[5])
        ↪ ;
    //LLR a_b_winner = (zero_one_winner < two_three_winner ? zero_one_winner :
        ↪ two_three_winner);

    // ROUND 3
    // Now this is the final round where we determine the ultimate minimum
    // It is simply against the two winners
    LLR first_min_comp = (a_b_winner < c_d_winner) ? (*allones) : (*zeros);

    //LLR first_min_comp = (a_b_winner < c_d_winner ? a_b_winner : c_d_winner);

    // Let's store the result!
    cnFirstMin = (a_b_winner & first_min_comp) | (c_d_winner & (first_min_comp ^ (*allones
        ↪ )));



    // Now we need to get the second minimum
    // But we know the winners from the previous rounds and the losers
    // Let's gather all the losers and run it all again
    // First start by gathering all the losers
    // This is done by simply inverting the results
    LLR zero_second_min = (cnInputsMag[0] & (zero_one_comp ^ (*allones))) | (cnInputsMag
        ↪ [1] & zero_one_comp);
    LLR one_second_min = (cnInputsMag[2] & (two_three_comp ^ (*allones))) | (cnInputsMag
        ↪ [3] & two_three_comp);
    LLR two_second_min = (zero_one_winner & (a_b_comp ^ (*allones))) | (two_three_winner &
        ↪  a_b_comp);
    LLR three_second_min = (cnInputsMag[4] & (c_d_comp ^ (*allones))) | (cnInputsMag[5] &
        ↪ c_d_comp);
    LLR four_second_min = (a_b_winner & (first_min_comp ^ (*allones))) | (c_d_winner &
        ↪ first_min_comp);

    /*LLR zero_second_min = (cnInputsMag[0] < cnInputsMag[1] ? cnInputsMag[1] :
        ↪ cnInputsMag[0]);
    LLR one_second_min = (cnInputsMag[2] < cnInputsMag[3] ? cnInputsMag[3] : cnInputsMag
        ↪ [2]);
    LLR two_second_min = (zero_one_winner < two_three_winner ? two_three_winner :
        ↪ zero_one_winner);
    LLR three_second_min = (cnInputsMag[4] < cnInputsMag[5] ? cnInputsMag[5] : cnInputsMag
        ↪ [4]);
    LLR four_second_min = (a_b_winner < c_d_winner ? c_d_winner : a_b_winner);*/

    // Now that we have our 5 we can start the new tournament
    // This one will also be 3 rounds
    // Let's say the last one will get the pass
    // Here is the tournament structure
    // 0 < 1 True 2 < 3 True
    // 0 < 2 True 0 < 4 If false then 0 is the result
    // 0 < 4 True
    // 0 is the min

    // ROUND 1
    // Note there is an option in the above to compare 3 < 4 in the second round. If 3 is
        ↪ less than 4 then we know the winner right away
    LLR zero_one_second_comp = (zero_second_min < one_second_min) ? (*allones) : (*zeros);
    LLR two_three_second_comp = (two_second_min < three_second_min) ? (*allones) : (*zeros
        ↪ );
```

```
// Now determine the winners
LLR zero_one_second_winner = (zero_second_min & zero_one_second_comp) | (
     ↪ one_second_min & (zero_one_second_comp ^ (*allones)));
LLR two_three_second_winner = (two_second_min & two_three_second_comp) | (
     ↪ three_second_min & (two_three_second_comp ^ (*allones)));

// ROUND 2
// Just like with the first min, we will now switch over to using letters so we aren't
     ↪  confused and don't have large names
// 0 < 1 -> a  2 < 3 -> b  4 -> c
LLR a_b_second_comp = (zero_one_second_winner < two_three_second_winner) ? (*allones)
     ↪ : (*zeros);

// Now store the winner
LLR a_b_second_winner = (zero_one_second_winner & a_b_second_comp) | (
     ↪ two_three_second_winner & (a_b_second_comp ^ (*allones)));

// FINAL ROUND
// Now we get the second min
LLR second_min_comp = (a_b_second_winner < four_second_min) ? (*allones) : (*zeros);

// Store the result
cnSecondMin = (a_b_second_winner & second_min_comp) | (four_second_min & (
     ↪ second_min_comp ^ (*allones)));

// Yay we have both now



// Now that we have the minimums and the parity, we should convert the first and
     ↪ second minimums to 2's complement assuming negative
LLR first_min_neg = (((cnFirstMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));

LLR second_min_neg = (((cnSecondMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));

// Now we have positive and negative versions of all
// Let's print out all the info

/*for(short input = 0; input<N_CHECK_INPUTS; input++) {
  //printf("Iteration,%d,CNode,%d,inputs,%d,%d,first_min,%d,second_min,%d,parity,%d\n
       ↪ ",iter,id,input,cnInputsPrivate[input],cnFirstMin,cnSecondMin,cnParity);
}*/

// Now let's start the check nodes
// We only need to execute one check node for each calculation as there are
     ↪ N_CHECK_NODE number of work-items
#pragma unroll
for(char input = 0; input < N_CHECK_INPUTS; input++) {

  // For each of these we need to determine if it was the first minimum
  // We just need a conditional that has either 1 or 0 depending if it's min or second
       ↪  min
  // And then we can make the expression LLR to_vnode = cnFirstMin*condition1 +
       ↪ cnSecondMin*inv(condition1)
  LLR min_conditional = ((cnInputsPrivate[input] & (*vector_max_llr)) == cnFirstMin) ?
       ↪  (*allones) : (*zeros);

  // the LLR conditional variable will now have a vector of either 0 or 1.
  // Now we can just use the expression

  // Now we can move on to calculating the output
  LLR sign_b = cnParity;

  // The sign bit is assigned to complete the parity compared to the other inputs
  sign_b ^= cnInputsPrivate[input] & (*vector_sign_bit);

  // Now we just send the data in 2's complement which was already calculated
  LLR sign_conditional = (sign_b == (*vector_sign_bit)) ? (*allones) : (*zeros);

  // If it's negative, give it the negative value, if not then the positive
  // Start with cnSecondMin, second_min_neg, cnFirstMin, first_min_neg
  LLR to_vnode =  ((min_conditional & sign_conditional & second_min_neg)) |
        ((min_conditional & (sign_conditional ^ (*allones)) & cnSecondMin)) |
        ((min_conditional ^ (*allones)) & sign_conditional & first_min_neg) |
        ((min_conditional ^ (*allones)) & (sign_conditional ^ (*allones)) & cnFirstMin
            ↪ );



  // Now we just need to find the variable node index
  short vnIndex = cntovnIndexLocation[cnInputsBaseIndex + input];

  vnInputs[vnIndex] = to_vnode;

  //printf("Iteration,%d,CNode,%d,output,%d,%d,vnIndex,%d\n",iter,id,input,to_vnode,
       ↪ vnIndex);

}

// Finally we just need to put a mem_fence so that no data gets lost
barrier(CLK_LOCAL_MEM_FENCE);
```

```
  }

  // After all the iterations we just need to send back the data
  // For the variable nodes, that means sending the values stored in the 'vnCurrent'
      ↪ variables as these are the totals
  vnResults[id] = vnCurrent;
  vnResults[id + N_CHECK_NODES] = vnCurrent_2;



}
```

Appendix I

Circular LDPC Code A-List Converter

```
 using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;

namespace CircularMatrixConverter
{

    class CircularConverter
    {
        private string _inputFilename;
        private string _outputFilename;
        private int cols;
        private int rows;
        private int L;
        private int numColumns;
        private int numRows;
        private int[][] matrixH;

        public string outputFilename
        {
            get
            {
                return _outputFilename;
            }
            set
            {
                if (value != "")
                {
                    _outputFilename = value;
                }
                else
                {
                    throw new Exception("Not␣a␣valid␣output␣filename");
                    //Console.WriteLine("Not a valid output filename");
                }
            }
        }

        public string inputFilename
        {
            get
            {
                return _inputFilename;
            }
            set
            {
                if (!File.Exists(value))
                    throw new Exception("Can't␣find␣input␣file");
                    //Console.WriteLine("File " + value + " does not exist.");
                else
                    _inputFilename = value;
            }
        }

        public CircularConverter(string filename,string output)
        {
            inputFilename = filename;
            outputFilename = output;
        }
        public CircularConverter(string input)
        {
            inputFilename = input;
            outputFilename = "outputFile.txt";
        }

        public void convert()
        {
            string[] lines = initialize();

            // Now we have just the matrix definitions left
            createMatrixH(lines);

            // Now output the A-list file
            writeAList();


        }

        private void writeAList()
        {
            System.IO.StreamWriter file = new System.IO.StreamWriter(outputFilename);

            int[] colCount = new int[cols];

            for (int c = 0; c < cols; c++)
            {
                colCount[c] = 0;
                for (int r = 0; r < rows; r++)
                {
                    if (matrixH[r][c] == 1)
                        colCount[c]++;
                }
            }

            int[] rowCount = new int[rows];

            for(int r=0; r < rows; r++)
            {
                rowCount[r] = 0;
                for (int c = 0; c < cols; c++)
                {
                    if (matrixH[r][c] == 1)
                        rowCount[r]++;
                }
            }

            file.WriteLine(cols + "␣" + rows);

            // Now output the row and column counts
            file.WriteLine(colCount[0] + "␣" + rowCount[0]);

            // Now output the details
            foreach (int c in colCount)
            {
                file.Write(c + "␣");
            }
            file.WriteLine();
            foreach (int r in rowCount)
            {
                file.Write(r + "␣");
            }
            file.WriteLine();

            // Now let's write out the individuals
            for (int c = 0; c < cols; c++)
            {
                for (int r = 0; r < rows; r++)
                {
                    if (matrixH[r][c] == 1)
                    {
                        file.Write((r + 1) + "␣");
                    }
                }
                file.WriteLine();
            }

            for (int r = 0; r < rows; r++)
            {
                for (int c = 0; c < cols; c++)
                {
                    if (matrixH[r][c] == 1)
                    {
                        file.Write((c + 1) + "␣");
                    }
                }
                file.WriteLine();
            }

            file.Close();
```

```
    }

    private void createMatrixH(string[] lines)
    {
        var r = 0;

        Console.WriteLine("numRows␣=␣" + numRows);
        Console.WriteLine("Lines␣length␣=␣" +lines.Length);
        foreach (string line in lines)
        {
            // Now let's split the line into the specific numbers
            string[] elements = line.Split('␣');

            // Double check the size of elements to make sure it's right
            if (elements.Length != numColumns)
                throw new Exception("Invalid␣input␣file␣format");
            var c = 0;
            foreach (string specific in elements)
            {
                try
                {
                    int shift = Int32.Parse(specific);

                    int x = shift + c * L;

                    for (int y = r * L; y < (r * L + L); y++)
                    {
                        matrixH[y][x] = 1;

                        x++;
                        // Check if we need to wrap around
                        if (x >= (c * L + L))
                            x = c * L;
                    }

                }
                catch (Exception e)
                {
                    // This means it was a dash which means we do nothing
                    continue;
                }
                finally
                {
                    c++;
                }
            }



            r++;
        }

    }


    private string[] initialize()
    {
        // First let's read in the input file
        string[] lines = System.IO.File.ReadAllLines(inputFilename);

        // Now let's parse one by one

        // The first line will be the sizes
        string[] parts = lines[0].Split('␣');
        cols = Int32.Parse(parts[0]);
        rows = Int32.Parse(parts[1]);

        L = Int32.Parse(lines[1]);

        numColumns = cols / L;
        numRows = rows / L;

        matrixH = new int[rows][];


        for (int i = 0; i < rows; i++)
        {
            matrixH[i] = new int[cols];
            for (int j = 0; j < cols; j++)
            {
                matrixH[i][j] = 0;
            }
        }

        // Now let's remove the top to lines
        lines = lines.Where(w => w != lines[0]).ToArray();
        lines = lines.Where(w => w != lines[0]).ToArray();
```

```
            return lines;
        }
    }
}
```

Appendix J

Irregular Design D Host Code (C)

```
// This is the start of the Channel LDPC implementation
#include <iostream>
#include <fstream>
#include <ctime>
#include <math.h>
#include <malloc.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <Windows.h>
#include "E:\Andrew\Projects\common\ax_common.h"



#define ALTERA

#define MANUAL_VECTOR_WIDTH 16

int N_VARIABLE_NODES = 16200;
int N_CHECK_NODES =1800;
#define MAX_VARIABLE_EDGES 5
#define MAX_CHECK_EDGES 34

#define PROJECT_TITLE "Irregular Global Memory LDPC"
#define PROJECT_DESCRIPTION "This host code supports the multi kernel implementation of
    ↪ the LDPC decoder using the char16 data type. Within the char16 data type, each of
    ↪ the chars will contain a different codeword to decode"


// These options apply to the code being used as well as the number of iterations to do
int LDPC_SIZE = N_VARIABLE_NODES; // Note this is overwritten when a value is passed in
    ↪ via command line

#define FILENAME "matrix_"
#define OUTPUT_FILE "results_"
#define N_ITERATIONS 32

#define N_VARIABLE_INPUTS (MAX_VARIABLE_EDGES)
#define N_CHECK_INPUTS (MAX_CHECK_EDGES)


#define N_VARIABLE_INPUTS_W_CHANNEL (N_VARIABLE_INPUTS+1)


#define MAX_LINE (1024*3)

static char * INPUT_FILENAME = (char*) malloc(sizeof(char)*MAX_LINE);
static char * OUTPUT_FILENAME = (char*) malloc(sizeof(char)*MAX_LINE);

#if MANUAL_VECTOR_WIDTH==2
#define LLR_TYPE cl_char2
#define TOTAL_TYPE cl_short2
#define CONVERT_TO_SHORT(a) convert_short2(a)
#define CONVERT_TO_CHAR(a) convert_char2(a)
#elif MANUAL_VECTOR_WIDTH==4
#define LLR_TYPE cl_char4
#define TOTAL_TYPE cl_short4
#define CONVERT_TO_SHORT(a) convert_short4(a)
#define CONVERT_TO_CHAR(a) convert_char4(a)
#elif MANUAL_VECTOR_WIDTH==8
#define LLR_TYPE cl_char8
```

```
#define TOTAL_TYPE cl_short8
#define CONVERT_TO_SHORT(a) convert_short8(a)
#define CONVERT_TO_CHAR(a) convert_char8(a)
#elif MANUAL_VECTOR_WIDTH==16
#define LLR_TYPE cl_char16
#define TOTAL_TYPE cl_short16
#define CONVERT_TO_SHORT(a) convert_short16(a)
#define CONVERT_TO_CHAR(a) convert_char16(a)
#else
#define LLR_TYPE cl_char
#define TOTAL_TYPE cl_short
#define CONVERT_TO_SHORT(a) convert_short(a)
#define CONVERT_TO_CHAR(a) convert_char(a)
#endif


// Integer options
#define LLR_BASE_TYPE char
#define N_BITS_LLR (sizeof(LLR_BASE_TYPE)*8) // 8 bits per byte
#define MAX_LLR ((1 << (N_BITS_LLR-1))-1)
#define SIGN_BIT (MAX_LLR+1)
#define MAX_NEG_LLR (0-MAX_LLR)
typedef LLR_TYPE LLR;

LLR *vector_max_llr;
LLR *vector_min_neg_llr;
LLR *vector_n_bits_llr;
LLR *vector_sign_bit;
LLR *vector_too_neg_llr;
LLR *allones;
LLR *ones;
LLR *zeros;
TOTAL_TYPE *total_vector_max_llr;
TOTAL_TYPE *total_vector_min_neg_llr;


#define BINS_PER_ONE_STEP 4
#define SATURATION_POINT int(MAX_LLR/BINS_PER_ONE_STEP)



//#define PRINT_DEBUG
//#define QUICK_DEBUG

#ifdef PRINT_DEBUG
#undef QUICK_DEBUG
#endif

#define DEBUG_FILENAME "output_debug_integer.csv"
#define CHECKNODE_DEBUG_FILENAME "checknode_debug.csv"
#define VARIABLENODE_DEBUG_FILENAME "variablenode_debug.csv"




#pragma region OpenCL Specific Declarations
static char * KERNEL_FILENAME  = (char*) malloc(sizeof(char)*MAX_LINE);
static char const* COMPILER_OPTIONS = "-w";
#pragma endregion

#define DEBUG 1
#define AOCL_ALIGNMENT 64





#pragma region Error Checking Macros
#define SYSTEM_ERROR -1
#define CheckPointer(pointer) CheckPointerFunc(pointer,__FUNCTION__, __LINE__)
#define ExitOnErrorWMsg(msg) ExitOnErrorFunc(msg,__FUNCTION__,__LINE__)
#define ExitOnError() ExitOnErrorWMsg("Error! Please refer to")
#pragma endregion

#pragma region Structure Defintions
// This structure holds all data required for the variable nodes.
// Note that the last input will be the channel input for each variable node
// the n_inputs will always be n_checknodes+1
typedef struct VariableNodes
{
  LLR inputs[MAX_VARIABLE_EDGES+1];
  cl_short checknode_indexes[MAX_VARIABLE_EDGES];
  cl_short index;
  cl_short n_inputs;
  cl_short n_checknodes;
}VariableNode;

// This structure holds all data required for the check nodes
typedef struct CheckNodes
{
  LLR inputs[MAX_CHECK_EDGES];
  cl_short variablenode_indexes[MAX_CHECK_EDGES];
  cl_short index;
  cl_short n_inputs;
}CheckNode;
#pragma endregion

#pragma region Function Declaractions
static void CheckPointerFunc(void * pointer,char const* const func,const int line);
static void ExitOnErrorFunc(char const* msg, char const* func, const int line);
static void readParityFile(VariableNode*&, CheckNode*&, int&, int&);
static bool checkIfSatisfied(LLR * parity);
static void updateParity(CheckNode *& c_nodes, const int n_checks);
static LLR * convertCodewordToBinary(VariableNode*&,const int n_vars);
static float* getCodeword(int size);
static LLR_BASE_TYPE * convertToLLR(const float * codeword, const int n_vars);
static void printCodeword(const int * codeword, const int n_vars);
static void printResults(LLR * results);
static void printResultsToFile(double overall, double variable, double check, double misc)
     ↪ ;
static void setFileNames();

double generate_random_gaussian();
double generate_random_number();
void introduceNoise(float*,int,int);

int * generate_VNtoCN_index_table(VariableNode* ,CheckNode* );
int * generate_CNtoVN_index_table(VariableNode* ,CheckNode* );



static void startOpenCL(cl_context,cl_device_id device, cl_program program);
static void startSimulationOpenCL(cl_context, cl_device_id,cl_program,VariableNode*&,
     ↪ CheckNode*&,const int, const int);
static void ldpcDecodeOpenCL(cl_context, cl_command_queue,cl_command_queue,cl_kernel,
     ↪ cl_kernel,VariableNode*&,CheckNode*&,const int, const int,float*);
void initializeNodes(VariableNode*&, CheckNode*&,const LLR*,const int, const int);
void defineGlobalDefinitions();

void parseProgramInputs(int,char*[]);
#pragma endregion






int main(int argc, char * argv[])
{
  printf("%s\n",PROJECT_TITLE);
  printf("%s\n",PROJECT_DESCRIPTION);

  parseProgramInputs(argc,argv);

  // Let's do a quick check on the number
  if(LDPC_SIZE <=0) {
    printf("Codelength␣(number␣of␣variable␣nodes)␣is␣not␣valid\n");
    ExitOnError();
  }

  // Set the file names for the input and output
  setFileNames();


  // Set up the program
  cl_context context;
  cl_device_id device;
  cl_program program;
  cl_device_type type = CL_DEVICE_TYPE_ALL;

  printf("\nOpenCL␣API\n\n");
  printf("Max␣LLR␣=␣%d\n",MAX_LLR);


  // Let's do this
  AxCreateContext(type,AX_PLATFORM_STRING,&context,&device);
#ifdef ALTERA
  AxBuildProgramAltera(&program,device,context,KERNEL_FILENAME,COMPILER_OPTIONS);
#else
  AxBuildProgram(&program,device,context,KERNEL_FILENAME,COMPILER_OPTIONS);
#endif

  // Now let's start the rest of the program
  startOpenCL(context,device,program);

  clReleaseProgram(program);
  clReleaseContext(context);


}


// Start the main chunk of the host code
void startOpenCL(cl_context context, cl_device_id device, cl_program program)
```

```
{

  defineGlobalDefinitions();
  // Declare the variables
  // Declare an array to hold the checknodes
  CheckNode * c_nodes=NULL;

  // Declare an array to hold the variablenodes
  VariableNode * v_nodes=NULL;

  // The number of check nodes and variable nodes
  int n_checks=0;
  int n_vars=0;

  // Let's read the parity file to get the nodes
  readParityFile(v_nodes,c_nodes,n_vars,n_checks);
  printf("Testing␣file␣for␣accuracy....");
  for(int i=0; i<n_vars; i++) {
    // Check that every fanout isn't greater than the max
    if(v_nodes[i].n_inputs > N_VARIABLE_INPUTS_W_CHANNEL)
      ExitOnErrorWMsg("Software␣defined␣N_VARIABLE_INPUTS␣too␣small");

  }
  for(int i=0; i<n_checks; i++) {

    if(c_nodes[i].n_inputs > N_CHECK_INPUTS)
      ExitOnErrorWMsg("Software␣defined␣N_CHECK_INPUTS␣too␣small");
  }
  printf("Successful\n");

  // Let's just do a few quick checks to make sure that the read worked properly
  // We should check the c_nodes and v_nodes pointers
  // We also need to check that the number of variable nodes and check nodes are valid
  CheckPointer(c_nodes);CheckPointer(v_nodes);
  if(n_vars <= 0 || n_checks <= 0)
    ExitOnErrorWMsg("Number␣of␣variables␣and␣check␣nodes␣not␣valid.");

  printf("\n********************************************\n");
  printf("Variable␣Nodes␣%d,␣Check␣Nodes␣%d\n",n_vars,n_checks);
  printf("%d␣Log␣Likelihood␣Ratio␣Bits\n",N_BITS_LLR);
  printf("Max␣Variable␣Edges␣%d,␣Max␣Check␣Edges␣%d\n",MAX_VARIABLE_EDGES,MAX_CHECK_EDGES)
      ↪ ;

  // Now we are ready to start our simulation
  startSimulationOpenCL(context,device,program,v_nodes,c_nodes,n_vars,n_checks);

}

void startSimulationOpenCL(cl_context context, cl_device_id device,cl_program program,
    ↪ VariableNode*& v_nodes,CheckNode*& c_nodes, const int n_vars, const int n_checks)
{

  // Firstly, we need to get our codeword to decode
  float * codeword = getCodeword(n_vars);


  cl_int err;
  // Let's create our openCL stuff

  size_t output;
  AxCheckError(clGetDeviceInfo(device,CL_DEVICE_MAX_WORK_GROUP_SIZE,sizeof(output),&output
      ↪ ,0));
  printf("It␣is␣=␣%d\n",output);

  // Now we need to create our command queues
  // We will create one command queue for each kernel
  cl_command_queue VariableQueue = clCreateCommandQueue(context,device,
      ↪ CL_QUEUE_PROFILING_ENABLE,&err);AxCheckError(err);
  cl_command_queue CheckQueue = clCreateCommandQueue(context,device,
      ↪ CL_QUEUE_PROFILING_ENABLE,&err);AxCheckError(err);



  // Now we need to create the kernels themselves.
  cl_kernel VariableKernel = clCreateKernel(program,"VariableNode",&err);AxCheckError(err)
      ↪ ;
  cl_kernel CheckKernel = clCreateKernel(program,"CheckNode",&err);AxCheckError(err);

  // Now let's simply start the LDPC
  ldpcDecodeOpenCL(context, VariableQueue,CheckQueue,VariableKernel,CheckKernel,v_nodes,
      ↪ c_nodes,n_vars,n_checks,codeword);
}

void copyCodewordFloat(float *& source, float *& dest, int n_vars) {

  for(int i=0; i<n_vars; i++) {
    dest[i] = source[i];
  }
}

void copyCodewordLLRBase(LLR_BASE_TYPE *& source, LLR_BASE_TYPE *& dest, int n_vars) {
  for(int i=0; i<n_vars; i++) {
    dest[i] = source[i];
  }
}

void ldpcDecodeOpenCL(cl_context       context,
            cl_command_queue     VariableQueue,
            cl_command_queue     CheckQueue,
            cl_kernel        VariableKernel,
            cl_kernel        CheckKernel,
            VariableNode*&     v_nodes,
            CheckNode*&      c_nodes,
            const int        n_vars,
            const int        n_checks,
            float*         codeword)
{
  cl_int err;


  // Note this is where I would initiate an LLR for fixed integer
  size_t channelSize = sizeof(LLR) * N_VARIABLE_NODES*2;
  LLR * codeword_LLR = (LLR*) _aligned_malloc(channelSize,AOCL_ALIGNMENT);


  float * codeword0 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword1 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword2 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword3 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword4 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword5 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword6 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword7 = (float*) malloc(sizeof(float) * n_vars);

  float * codeword8 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword9 = (float*) malloc(sizeof(float) * n_vars);
  float * codeworda = (float*) malloc(sizeof(float) * n_vars);
  float * codewordb = (float*) malloc(sizeof(float) * n_vars);
  float * codewordc = (float*) malloc(sizeof(float) * n_vars);
  float * codewordd = (float*) malloc(sizeof(float) * n_vars);
  float * codeworde = (float*) malloc(sizeof(float) * n_vars);
  float * codewordf = (float*) malloc(sizeof(float) * n_vars);


  copyCodewordFloat(codeword,codeword0,n_vars);
  copyCodewordFloat(codeword,codeword1,n_vars);
  copyCodewordFloat(codeword,codeword2,n_vars);
  copyCodewordFloat(codeword,codeword3,n_vars);
  copyCodewordFloat(codeword,codeword4,n_vars);
  copyCodewordFloat(codeword,codeword5,n_vars);
  copyCodewordFloat(codeword,codeword6,n_vars);
  copyCodewordFloat(codeword,codeword7,n_vars);

  copyCodewordFloat(codeword,codeword8,n_vars);
  copyCodewordFloat(codeword,codeword9,n_vars);
  copyCodewordFloat(codeword,codeworda,n_vars);
  copyCodewordFloat(codeword,codewordb,n_vars);
  copyCodewordFloat(codeword,codewordc,n_vars);
  copyCodewordFloat(codeword,codewordd,n_vars);
  copyCodewordFloat(codeword,codeworde,n_vars);
  copyCodewordFloat(codeword,codewordf,n_vars);






  // Now we need to introduce some noise into our codeword
  // Note that this function changes our codeword input variable
  // But let's add noise individually to each one
  introduceNoise(codeword0,1000,n_vars);
  introduceNoise(codeword1,-20,n_vars);
  introduceNoise(codeword2,-20,n_vars);
  introduceNoise(codeword3,-12,n_vars);
  introduceNoise(codeword4,-10,n_vars);
  introduceNoise(codeword5,-20,n_vars);
  introduceNoise(codeword6,-20,n_vars);
  introduceNoise(codeword7,-12,n_vars);

  introduceNoise(codeword8,-10,n_vars);
  introduceNoise(codeword9,-20,n_vars);
  introduceNoise(codeworda,-20,n_vars);
  introduceNoise(codewordb,-12,n_vars);
  introduceNoise(codewordc,10,n_vars);
  introduceNoise(codewordd,20,n_vars);
  introduceNoise(codeworde,-20,n_vars);
  introduceNoise(codewordf,-12,n_vars);

  /*for(int i=0; i<n_vars; i++) {
    printf("codeword[%d] = %f\n",i,codeword0[i]);
  }*/

  // Now I need to add in the step to conver the number to an integer
  char * llr0 = convertToLLR(codeword0,n_vars);
  char * llr1 = convertToLLR(codeword1,n_vars);
  char * llr2 = convertToLLR(codeword2,n_vars);
  char * llr3 = convertToLLR(codeword3,n_vars);
  char * llr4 = convertToLLR(codeword4,n_vars);
  char * llr5 = convertToLLR(codeword5,n_vars);
  char * llr6 = convertToLLR(codeword6,n_vars);
  char * llr7 = convertToLLR(codeword7,n_vars);

  char * llr8 = convertToLLR(codeword8,n_vars);
  char * llr9 = convertToLLR(codeword9,n_vars);
  char * llra = convertToLLR(codeworda,n_vars);
  char * llrb = convertToLLR(codewordb,n_vars);
  char * llrc = convertToLLR(codewordc,n_vars);
  char * llrd = convertToLLR(codewordd,n_vars);
  char * llre = convertToLLR(codeworde,n_vars);
  char * llrf = convertToLLR(codewordf,n_vars);


  for(int i=0; i<n_vars; i++) {
#if MANUAL_VECTOR_WIDTH == 1
    codeword_LLR[i]= llr0[i];
#else

    for(int j=0; j <MANUAL_VECTOR_WIDTH; j++) {
      codeword_LLR[i].s[j] = llr0[i];
    }


#endif


  }

  int k=0;
  for(int i=n_vars; i<n_vars*2;i++) {

    for(int j=0; j<MANUAL_VECTOR_WIDTH; j++) {
      codeword_LLR[i].s[j] = llr0[k];
    }
    k++;
  }

  /*for(int i=0; i<n_vars; i++) {
#if MANUAL_VECTOR_WIDTH == 1
    printf("codeword[%d] = %d\n",i,codeword_LLR[i]);//.x,codeword_LLR[i].y,codeword_LLR[i
        ↪ ].z,codeword_LLR[i].w);
#else
    printf("codeword[%d] = %d\n",i,codeword_LLR[i].s0);//codeword_LLR[i].y,codeword_LLR[i
        ↪ ].z,codeword_LLR[i].w);
#endif
  }*/

  // Let's start initializing the inputs
  // For the codewords starting at the VN we just need to create a buffer with the
      ↪ measurements





  // For the second version we need to start by doing the first VN portion
  //initializeNodes(v_nodes,c_nodes,codeword_LLR,n_vars,n_checks);

  // Let's get the connection tables
  size_t VNtoCNSize = sizeof(int)*N_VARIABLE_NODES*N_VARIABLE_INPUTS;
  size_t CNtoVNSize = sizeof(int)*N_CHECK_NODES*N_CHECK_INPUTS;
  printf("Generating␣VntoCN␣lookup␣table....");

  int * VNtoCNLookup_h = generate_VNtoCN_index_table(v_nodes,c_nodes);
  printf("Successful\n");


  printf("Generating␣CntoVN␣lookup␣table....");
  int * CNtoVNLookup_h = generate_CNtoVN_index_table(v_nodes,c_nodes);

  printf("Successful\n");




  cl_mem VNtoCNLookup = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
      ↪ VNtoCNSize,VNtoCNLookup_h,&err);AxCheckError(err);
  cl_mem CNtoVNLookup = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
      ↪ CNtoVNSize,CNtoVNLookup_h,&err);AxCheckError(err);




  // Make the constant memory info
```

```
cl_mem vector_max_llr_d = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
     ↪ sizeof(LLR),vector_max_llr,&err);AxCheckError(err);
cl_mem vector_sign_bit_d = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
     ↪ sizeof(LLR),vector_sign_bit,&err);AxCheckError(err);
cl_mem allones_d = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,sizeof(
     ↪ LLR),allones,&err);AxCheckError(err);
cl_mem ones_d = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,sizeof(LLR)
     ↪ ,ones,&err);AxCheckError(err);
cl_mem zeros_d = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,sizeof(LLR
     ↪ ),zeros,&err);AxCheckError(err);
cl_mem total_vector_max_llr_d = clCreateBuffer(context,CL_MEM_READ_ONLY|
     ↪ CL_MEM_COPY_HOST_PTR,sizeof(TOTAL_TYPE),total_vector_max_llr,&err);AxCheckError(
     ↪ err);
cl_mem total_vector_min_neg_llr_d = clCreateBuffer(context,CL_MEM_READ_ONLY|
     ↪ CL_MEM_COPY_HOST_PTR,sizeof(TOTAL_TYPE),total_vector_min_neg_llr,&err);
     ↪ AxCheckError(err);

size_t vnResultSize = sizeof(LLR) * N_VARIABLE_NODES*2;
size_t cnResultSize = sizeof(LLR) * N_CHECK_NODES*2;

LLR * vnResult = (LLR*) _aligned_malloc(vnResultSize,AOCL_ALIGNMENT);
LLR * cnResult = (LLR*) _aligned_malloc(cnResultSize,AOCL_ALIGNMENT);

cl_mem vnResult_d = clCreateBuffer(context,CL_MEM_READ_WRITE,vnResultSize,NULL,&err);
     ↪ AxCheckError(err);
cl_mem cnResult_d = clCreateBuffer(context,CL_MEM_READ_WRITE,cnResultSize,NULL,&err);
     ↪ AxCheckError(err);


cl_mem initialChannelMeasurements = clCreateBuffer(context,CL_MEM_READ_WRITE |
     ↪ CL_MEM_COPY_HOST_PTR, channelSize,codeword_LLR,&err);AxCheckError(err);

// We now need to create the buffers for each node in global memory (for data transfer
     ↪ between the nodes)
size_t vnInputsGMSize = sizeof(LLR)*N_VARIABLE_NODES*(N_VARIABLE_INPUTS+1)*2;
size_t cnInputsGMSize = sizeof(LLR)*N_CHECK_NODES*(N_CHECK_INPUTS+1)*2;

// We need to initialize them to 0 which means we need to create and send that data
LLR * initializationVNBuffer = (LLR*) _aligned_malloc(vnInputsGMSize,AOCL_ALIGNMENT);
LLR * initializationCNBuffer = (LLR*) _aligned_malloc(cnInputsGMSize,AOCL_ALIGNMENT);

CheckPointer(initializationVNBuffer);
CheckPointer(initializationCNBuffer);


printf("Initializing␣VN␣Buffers....");
for(int i=0; i < n_vars*2; i++) {

  // For each one let's look at the inputs
  int j;
  for(j = 0; j< (N_VARIABLE_INPUTS + 1); j++) {
    for(int k=0; k<MANUAL_VECTOR_WIDTH; k++) {
      initializationVNBuffer[i*(N_VARIABLE_INPUTS+1) + j].s[k]= (LLR_BASE_TYPE) 0;
    }
  }
}


printf("Successful\n");
printf("Initializing␣CN␣Buffers....");
for(int i=0; i< n_checks * 2; i++) {
  int c = i % n_checks;

  int j;
  for(j = 0; j < c_nodes[c].n_inputs; j++) {
    for(int k=0; k<MANUAL_VECTOR_WIDTH; k++) {
      initializationCNBuffer[i*(N_CHECK_INPUTS +1) + j].s[k]= (LLR_BASE_TYPE) 0;
    }
  }

  for(j = c_nodes[c].n_inputs; j < (N_CHECK_INPUTS + 1); j++) {
    for(int k=0; k<MANUAL_VECTOR_WIDTH; k++) {
      initializationCNBuffer[i*(N_CHECK_INPUTS +1) + j].s[k]= (LLR_BASE_TYPE) MAX_LLR;
    }
  }

}

printf("Successful\n");

// Start with the Variable Node buffer
cl_mem vnInputsBuffer = clCreateBuffer(context,CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR
     ↪ ,vnInputsGMSize,initializationVNBuffer,&err);AxCheckError(err);
cl_mem cnInputsBuffer = clCreateBuffer(context,CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR,
     ↪  cnInputsGMSize, initializationCNBuffer, &err); AxCheckError(err);

// Make the sync buffers
char * VNSync_h = (char*) _aligned_malloc(1,AOCL_ALIGNMENT);
char * CNSync_h = (char*) _aligned_malloc(1,AOCL_ALIGNMENT);

(*VNSync_h) = -1;
(*CNSync_h) = -1;
```

```
cl_mem VNSync = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, 1,
    ↪ VNSync_h,&err);AxCheckError(err);
cl_mem CNSync = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, 1,
    ↪ CNSync_h,&err);AxCheckError(err);

printf("Host␣pointers\n");
printf("VNtoCNLookup␣%lx\n",VNtoCNLookup_h);
printf("CNtoVNLookup␣%lx\n",CNtoVNLookup_h);
printf("codeword_LLR␣%lx\n",codeword_LLR);
printf("initializationVNBuffer␣%lx\n",initializationVNBuffer);
printf("initiailzationCNBuffer␣%lx\n",initializationCNBuffer);

printf("VNSync␣%lx\n",VNSync_h);
printf("CNSync␣%lx\n",CNSync_h);
printf("vector_max_llr␣%lx\n",vector_max_llr);
printf("vector_sign_bit␣%lx\n",vector_sign_bit);
printf("allones␣%lx\n",allones);
printf("ones␣%lx\n",ones);
printf("total_vector_max_llr␣%lx\n",total_vector_max_llr);
printf("total_vector_min_neg_llr␣%lx\n",total_vector_min_neg_llr);
printf("VNResult␣%lx\n",vnResult);
printf("CNResult␣%lx\n",cnResult);




// Let's set up the kernels
// Starting with the Variable Node
AxCheckError(clSetKernelArg(VariableKernel,0,sizeof(VNtoCNLookup),&VNtoCNLookup));
AxCheckError(clSetKernelArg(VariableKernel,1,sizeof(initialChannelMeasurements),&
    ↪ initialChannelMeasurements));
AxCheckError(clSetKernelArg(VariableKernel,2,sizeof(vnInputsBuffer),&vnInputsBuffer));
AxCheckError(clSetKernelArg(VariableKernel,3,sizeof(cnInputsBuffer),&cnInputsBuffer));
AxCheckError(clSetKernelArg(VariableKernel,4,sizeof(vnResult_d),&vnResult_d));
AxCheckError(clSetKernelArg(VariableKernel,5,sizeof(vector_max_llr_d),&vector_max_llr_d)
    ↪ );
AxCheckError(clSetKernelArg(VariableKernel,6,sizeof(vector_sign_bit_d),&
    ↪ vector_sign_bit_d));
AxCheckError(clSetKernelArg(VariableKernel,7,sizeof(allones_d),&allones_d));
AxCheckError(clSetKernelArg(VariableKernel,8,sizeof(ones_d),&ones_d));
AxCheckError(clSetKernelArg(VariableKernel,9,sizeof(zeros_d),&zeros_d));
AxCheckError(clSetKernelArg(VariableKernel,10,sizeof(total_vector_max_llr_d),&
    ↪ total_vector_max_llr_d));
AxCheckError(clSetKernelArg(VariableKernel,11,sizeof(total_vector_min_neg_llr_d),&
    ↪ total_vector_min_neg_llr_d));

// Check Node
AxCheckError(clSetKernelArg(CheckKernel,0,sizeof(CNtoVNLookup),&CNtoVNLookup));
AxCheckError(clSetKernelArg(CheckKernel,1,sizeof(vector_max_llr_d),&vector_max_llr_d));
AxCheckError(clSetKernelArg(CheckKernel,2,sizeof(vector_sign_bit_d),&vector_sign_bit_d))
    ↪ ;
AxCheckError(clSetKernelArg(CheckKernel,3,sizeof(allones_d),&allones_d));
AxCheckError(clSetKernelArg(CheckKernel,4,sizeof(ones_d),&ones_d));
AxCheckError(clSetKernelArg(CheckKernel,5,sizeof(zeros_d),&zeros_d));
AxCheckError(clSetKernelArg(CheckKernel,6,sizeof(cnInputsBuffer),&cnInputsBuffer));
AxCheckError(clSetKernelArg(CheckKernel,7,sizeof(vnInputsBuffer),&vnInputsBuffer));
AxCheckError(clSetKernelArg(CheckKernel,8,sizeof(cnResult_d),&cnResult_d));

clock_t start_t, end_t;
double total_t;

unsigned _int64 freq;
QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
double timerFrequency = (1.0/freq);

start_t = clock();

unsigned __int64 startTime;
QueryPerformanceCounter((LARGE_INTEGER *)&startTime);


// Now let's run the kernels
size_t global_size_vn= 1;
size_t local_size_vn=  1;

// Now we start the Check Node Kernel
size_t global_size_cn = 1;
size_t local_size_cn = 1;


cl_event kernelTimeVN;
//printf("Starting the Variable Node Kernel\n");
AxCheckError(clEnqueueNDRangeKernel(VariableQueue,VariableKernel,1,NULL,&global_size_vn
    ↪ ,&local_size_vn,0,NULL,&kernelTimeVN));

//printf("Starting the Check Node Kernel\n");
cl_event kernelTimeCN;
AxCheckError(clEnqueueNDRangeKernel(CheckQueue,CheckKernel,1,NULL,&global_size_cn,&
    ↪ local_size_cn,0,NULL,&kernelTimeCN));
```

```
  clFinish(CheckQueue);
  clFinish(VariableQueue);

  end_t = clock();

  unsigned __int64 endTime;
  QueryPerformanceCounter((LARGE_INTEGER *)&endTime);
  double timeDifferenceInMilliseconds = ((endTime-startTime) * timerFrequency)*1000;

  total_t = (double)(end_t - start_t)/CLOCKS_PER_SEC;
  printf("Clock␣says␣%f\n",total_t);

  printf("Timemilliseconds␣=␣%f\n",timeDifferenceInMilliseconds);
  printf("frequency␣=␣%f\n",timerFrequency);
  printf("LArger␣start␣=␣%llu\n",startTime);
  printf("Larger␣end␣␣%llu\n",endTime);

  clWaitForEvents(1,&kernelTimeVN);
  printf("Variable␣Node␣kernel␣has␣finished␣all␣%d␣iterations\n", N_ITERATIONS);

  clWaitForEvents(1,&kernelTimeCN);
  printf("Check␣Node␣kernel␣has␣finished␣all␣%d␣iterations\n", N_ITERATIONS);




  // Now we can just get the data
  //AxCheckError(clEnqueueReadBuffer(LDPCQueue,vnResult_d,CL_TRUE,0,vnResultSize,vnResult
      ↪ ,0,NULL,NULL));
  //AxCheckError(clEnqueueReadBuffer(LDPCQueue,cnResult_d,CL_TRUE,0,cnResultSize,cnResult
      ↪ ,0,NULL,NULL));

  // Let's check the accuracy
  // Ok let's print it out
  //checkIfSatisfied(cnResult);
  //printResults(vnResult);
  double milisecondTime = timeDifferenceInMilliseconds;

  printf("The␣kernel␣execution␣time␣was␣%f␣ms\n",milisecondTime);
  double throughput = N_VARIABLE_NODES*MANUAL_VECTOR_WIDTH*2/((milisecondTime)/1000.0);
  printf("The␣throughput␣was␣%f␣bps\n",throughput);

  /*cl_ulong time_start,time_end;
  double total_time;

  AxCheckError(clGetEventProfilingInfo(kernelTime,CL_PROFILING_COMMAND_START,sizeof(
      ↪ time_start),&time_start,NULL));
  AxCheckError(clGetEventProfilingInfo(kernelTime,CL_PROFILING_COMMAND_END,sizeof(time_end
      ↪ ),&time_end,NULL));

  total_time = time_end-time_start;
  // Now we can just get the data
  AxCheckError(clEnqueueReadBuffer(LDPCQueue,vnResult_d,CL_TRUE,0,vnResultSize,vnResult,0,
      ↪ NULL,NULL));
  AxCheckError(clEnqueueReadBuffer(LDPCQueue,cnResult_d,CL_TRUE,0,cnResultSize,cnResult,0,
      ↪ NULL,NULL));
  */
  // Let's check the accuracy
  // Ok let's print it out
  //checkIfSatisfied(cnResult);
  //printResults(vnResult);
  //double milisecondTime = total_time/1000000.0;

  /*printf("The kernel execution time was %f ms\n",milisecondTime);
  double throughput = N_VARIABLE_NODES*MANUAL_VECTOR_WIDTH/((milisecondTime)/1000.0);
  printf("The throughput was %f bps\n",throughput);

  FILE * results = fopen("results_kernel.csv","w");

  fprintf(results,"%f,%f\n",milisecondTime,throughput);
  fclose(results);*/



}

void printResults(LLR * results) {

  // Need to get a value in binary first

#if MANUAL_VECTOR_WIDTH == 1
  printf("Vector␣0:␣");
  for(int i=0; i<N_VARIABLE_NODES; i++) {

    if(results[i] > 0)
      printf("0");
    else
      printf("1");
  }
  printf("\n");
#else
```

```
  for(int j=0; j<MANUAL_VECTOR_WIDTH; j++) {

    printf("Vector␣%d:␣",j);
    for(int i=0; i<N_VARIABLE_NODES; i++) {


      if(results[i].s[j] > 0)
        printf("0");
      else
        printf("1");
    }
    printf("\n");

  }
#endif
}

void setFileNames() {
  char temp[MAX_LINE];
  strncpy(temp,FILENAME,MAX_LINE);
  _snprintf(INPUT_FILENAME,MAX_LINE,"E:\\Andrew\\Projects\\%s%d.txt",temp,LDPC_SIZE);

  strncpy(temp,OUTPUT_FILE,MAX_LINE);
  _snprintf(OUTPUT_FILENAME,MAX_LINE,"E:\Andrew\Projects\IrregularCodewordAttempt\%s%d.txt
      ↪ ",temp,LDPC_SIZE);

#ifdef ALTERA
  _snprintf(KERNEL_FILENAME,MAX_LINE,"E:\\Andrew\\Projects\\IrregularCodewordAttempt\\
      ↪ SingleThreadTry\\kernels%d.aocx",LDPC_SIZE);
#else
  _snprintf(KERNEL_FILENAME,MAX_LINE,"E:\\Andrew\\Projects\\IrregularCodewordAttempt\\
      ↪ kernels%dtrygmcommunication.cl",LDPC_SIZE);
#endif
}

void printResultsToFile(double overall, double variable, double check, double misc) {
  // Now let's open the file
  FILE * results = fopen(OUTPUT_FILENAME,"w"); CheckPointer(results);

  // Now let's write data to the file
  fprintf(results,"%f␣%f␣%f␣%f\n",overall,variable,check,misc);

  // Now we close the file
  fclose(results);

}





// Check if all of the check nodes are satisfied
bool checkIfSatisfied(LLR * parity)
{
#if MANUAL_VECTOR_WIDTH == 1
  // All we have to do use iterate over all and see if there is anything non zero
  for(int i=0; i<N_CHECK_NODES; i++) {
    if(parity[i]){
      printf("NOT␣SATISFIED\n");
      return false;
    }
  }
  printf("SATISFIED!\n");
  return true;
#else
  for(int j=0; j<MANUAL_VECTOR_WIDTH; j++){
    for(int i=0; i<N_CHECK_NODES; i++) {
      if(parity[i].s[j]){
        printf("NOT␣SATISFIED\n");
      }
    }
    printf("SATISFIED!\n");
  }

  return true;
#endif

}

#pragma region Codeword Operations
// Get the codeword in BPSK
float* getCodeword(int size)
{
  float *codeword = (float*)malloc(sizeof(float)*size);
  int * input = (int*) malloc(sizeof(int)*size);
  for(int i=0; i<size; i++)
    input[i] = 0;

  //input[0] = 1;

  // Convert it to BPSK
  // 0 -> 1, 1 -> -1
  // Map binary 0 to 1
  // May binary 1 to -1
  for(int i=0; i<size; i++)
  {
    if(input[i])
      codeword[i] = -1;
    else
      codeword[i] = 1;
  }

  return codeword;
}


// Introduce gaussian noise to the input codeword based on the given SNR
void introduceNoise(float* codeword, int SNR, int n_vars)
{
  double sigma = pow(10,(double)-SNR/20);

  for(int i=0; i<n_vars; i++)
  {
    codeword[i] += generate_random_gaussian()*sigma;
    //printf("Codeword[%d] = %f\n",i,codeword[i]);
  }
}

void printCodeword(const int * codeword, const int n_vars) {
  for(int i=0; i<n_vars; i++) {
    printf("codeword[%d] = %d\n",i,codeword[i]);
  }

}

/* This function will take in the given float version of the codeword
and convert it to an integer representation LLR.  Please note that
it converts the LLR into a 2's complement number first
*/
LLR_BASE_TYPE * convertToLLR(const float *codeword, const int n_vars) {
  LLR_BASE_TYPE * output = (LLR_BASE_TYPE*) malloc(sizeof(LLR_BASE_TYPE)*n_vars);

  // Iterate over the entire codeword
  for(int i=0; i<n_vars; i++) {
    float word = abs(codeword[i]);
    output[i] = 0;


    // Now let's iterate over the possible bins and see where it fits
    float division_point = ((float)MAX_LLR)/SATURATION_POINT ;
    //printf("division_point = %f\n",division_point);
    for(int j=1; j<=(MAX_LLR); j++) {

      if(word <  j / division_point) {
        output[i] = j;

        break;
      }

    }
    //printf("codeword[%d] = %f output[%d] = %u     sign = %d\n",i,codeword[i],i,output[i
        ↪ ],(output[i]&SIGN_BIT)>>15);

    // Once we have the positioning we just need to make sure it's still within the valid
        ↪ LLR
    if(word >=  MAX_LLR / division_point) {
      output[i] = MAX_LLR;

    }

    // Now set the sign bit
    if(codeword[i] < 0) {
      output[i] *= -1;

    }

  }
  return output;
}
#pragma endregion


// This function will read the parity check matrix A-list file and get the following
    ↪ information:
// - number of variable nodes, number of check nodes
// - an array of variablenode structures
// - an array of checknode structures
// PLEASE NOTE ALL VARIABLES PASSED INTO THIS FUNCTION ARE PASSED BY REFERENCE AND WILL BE
    ↪  MODIFIED
// Please pass this function empty variables to store the result
void readParityFile(VariableNode *& v_nodes, CheckNode *& c_nodes,int &n_vars,int &
    ↪ n_checks)
{
  char line[MAX_LINE];
```

```
// Open the file
FILE * parity_matrix = fopen(INPUT_FILENAME,"r");
CheckPointer(parity_matrix);

// Now let's start reading
// The first two values in the text file will be the number of variable nodes and the
    ↪ number of check nodes
if(fscanf(parity_matrix,"%d␣%d",&n_vars,&n_checks)!=2)
  ExitOnErrorWMsg("Error␣with␣parity_matrix␣A-list␣file␣format,");

// Now that we know the number of variable nodes and checks we can allocate the amounts
v_nodes = (VariableNode*)malloc(sizeof(VariableNode)*n_vars);
c_nodes = (CheckNode*)malloc(sizeof(CheckNode)*n_checks);

// Throw away the next line.  This line tells us the approximate number of connections
    ↪ for each node, but we want the more accurate ones
int temp;
if(fscanf(parity_matrix,"%d␣%d",&temp,&temp)!=2)
  ExitOnErrorWMsg("Missing␣second␣line␣in␣A-list␣file.");

// Now let's get to the good stuff
// According to the A-list format the next line of text will be the list of check nodes
    ↪ for each variable node in order
// The next line will be a list of check nodes for each variable node
// Therefore we need to iterate n_vars times to get all of the connections
for(int i=0; i<n_vars; i++)
{
  int num_per_variable;
  if(fscanf(parity_matrix,"%d",&num_per_variable)!=1)
    ExitOnErrorWMsg("Error␣with␣parity_matrix␣A-list␣file␣format.");

  // Store the number of check nodes for each variable node and the number of inputs
  v_nodes[i].n_checknodes = num_per_variable;
  v_nodes[i].n_inputs = num_per_variable+1; // Add one here as there is a spot for the
      ↪ initial channel measurement

  // Let's also store the index
  v_nodes[i].index = i;
}

// The next line will be a list of variable nodes for each check node
// Therefore we need to iterate n_checks times to get all of the connections
for(int i=0; i<n_checks; i++)
{
  int num_per_check;
  if(fscanf(parity_matrix,"%d",&num_per_check)!=1)
    ExitOnErrorWMsg("Error␣with␣parity_matrix␣A-list␣file␣format.");

  // Store the number of inputs to the check node
  c_nodes[i].n_inputs = num_per_check; // Note there is not an extra here as there is no
      ↪  channel measurement

  // Also store the index
  c_nodes[i].index = i;

  // Set the satisifed bit to false
  //c_nodes[i].satisfied = false;
}

// Next we get the indexes for variable nodes (edges)
// This is the most important section where we determine which nodes get connected to
    ↪ each other
// First up are the variable node connections.  So we need to iterate over all the
    ↪ variable nodes
for(int i=0; i<n_vars; i++)
{
  int position;

  // Now we can use the number of checknodes we just received from the previous lines
  // We need to store which checknodes this variable node is connected to
  int j;
  for(j=0; j<v_nodes[i].n_checknodes; j++)
  {
    if(fscanf(parity_matrix,"%d",&position)!=1)
      ExitOnErrorWMsg("Error␣with␣parity_matrix␣A-list␣file␣format.");

    // Now store the position
    v_nodes[i].checknode_indexes[j] = position-1; // Note we need to subtract one here
        ↪ to start the position at 0

  }

  // Get the rest of it up to the max
  for(j = v_nodes[i].n_checknodes; j < N_VARIABLE_INPUTS; j++) {
    // So for the rest of the connections, we actually want to assign one of the garbage
        ↪  locations
    int randomCheck = rand() % n_checks;

    // Now we have a random check node
    v_nodes[i].checknode_indexes[j] = randomCheck;
    }

  }


  // Same thing, but now for the check nodes.
  // Next we get the indexes for the check nodes (edges)
  for(int i=0; i<n_checks; i++)
  {
    int position;
    int j;
    for(j=0; j<c_nodes[i].n_inputs; j++)
    {
      if(fscanf(parity_matrix,"%d",&position)!=1)
        ExitOnErrorWMsg("Error␣with␣parity_matrix␣A-list␣file␣format.");

      // Now store the position
      c_nodes[i].variablenode_indexes[j] = position-1; // Note we need to subtract one
           ↪ here to start the position at 0
    }

    // Get the rest of it up to the max
    for(j = c_nodes[i].n_inputs; j < N_CHECK_INPUTS; j++) {
      // So for the rest of the connections, we actually want to assign one of the garbage
           ↪  locations
      int randomCheck = rand() % n_vars;

      // Now we have a random check node
      c_nodes[i].variablenode_indexes[j] = randomCheck;


    }
  }

  // That concludes the reading of the parity matrix file
  // All data is now stored within the v_nodes and c_nodes arrays
  fclose(parity_matrix);
  printf("Parity␣matrix␣file␣%s␣read␣successfully\n",INPUT_FILENAME);
  return;
}


//Initialize the nodes for the first run
// This is where we'll need to load up that there are 4 of the vector chars
void initializeNodes(VariableNode *& v_nodes, CheckNode *& c_nodes, const LLR * codeword,
     ↪ const int n_vars, const int n_checks)
{

  // First we load up the channel measurement into all of the variable nodes
  // Iterate over each variable node
  for(int i=0; i<n_vars; i++)
  {
    short channel_index = v_nodes[i].n_inputs-1;
    // For each of the variable nodes, we want to only load up the very last spot that
         ↪ deals with the channel input
    // All of the other inputs should be 0.  This will allows us to start the execution
         ↪ with the variable nodes instead of the check nodes


    for(int j=0; j<v_nodes[i].n_inputs; j++)
    {

      // We need to initialize all 4 of the chars in the vector
      // Note that we use codeword.x(y) etc however they are all equal to begin but it is
           ↪ possible for these to be different
#if MANUAL_VECTOR_WIDTH == 1
      v_nodes[i].inputs[j] = codeword[i];
#else
      for(int t=0; t<MANUAL_VECTOR_WIDTH; t++) {
        v_nodes[i].inputs[j].s[t] = codeword[i].s[t];
      }
#endif
    }
  }


  // Load up the inputs for the check nodes for the first iteration
  // This means we are now passing the specific channel measurements for each connection
  for(int i=0; i<n_checks; i++)
  {

    // For each one of the inputs in this check node, we need to load up the information
         ↪ from the variable node
    for(int j=0; j<c_nodes[i].n_inputs; j++)
    {
      int variable_index = c_nodes[i].variablenode_indexes[j];
      int index = v_nodes[variable_index].n_inputs-1; // All of the spots hold the same
           ↪ value currently

      // We have to convert this value into sign and mag
      LLR sign_and_mag_ver = v_nodes[variable_index].inputs[index];
      // Now we need to do the four comparisons individually
#if MANUAL_VECTOR_WIDTH == 1
      if(sign_and_mag_ver < 0) {
        sign_and_mag_ver *= -1;
        sign_and_mag_ver |= SIGN_BIT;
      }
#else
      for(int t=0; t<MANUAL_VECTOR_WIDTH; t++) {
        if(sign_and_mag_ver.s[t] < 0) {
        sign_and_mag_ver.s[t] *= -1;
        sign_and_mag_ver.s[t] |= SIGN_BIT;
        }
      }
#endif



      // Let's store the value in our input
      c_nodes[i].inputs[j] = sign_and_mag_ver;
    }
  }
}

#pragma region Error Checking Functions
// This function will check the given input pointer for validaty and fail exit the program
    ↪  if necessary
void CheckPointerFunc(void * pointer,char const* const func, const int line)
{
  if(pointer == NULL)
  {
    fprintf(stderr,"Error␣with␣pointer␣in␣%s␣line␣%d\n",func,line);
    exit(SYSTEM_ERROR);
  }
}

void ExitOnErrorFunc(char const* msg, char const* func, const int line)
{
  fprintf(stderr,"%s␣%s␣line␣%d\n",msg,func,line);
  exit(SYSTEM_ERROR);
}
#pragma endregion

#pragma region Random Number Code
double generate_random_gaussian()
{
#define PI 3.14159265358979323846
  double U1 = generate_random_number();//(double)rand()/((doule)RAND_MAX+1);
  double U2 = generate_random_number();//(double)rand()/((double)RAND_MAX+1);

  double Z0 = sqrt(-2*log(U1))*cos(2*PI*U2);
  return Z0;
}



int * generate_VNtoCN_index_table(VariableNode * v_nodes,CheckNode * c_nodes) {
  int * table = (int*)_aligned_malloc(sizeof(int)*N_VARIABLE_NODES*N_VARIABLE_INPUTS,
      ↪ AOCL_ALIGNMENT);
  if(table == NULL)
    printf("err␣=␣%d\n",errno);

  // Iterate over the list of variable nodes
  for(int i=0; i<N_VARIABLE_NODES; i++) {

    int j;
    // For each variable node we need to find the appropriate connections for each input
    for(j=0; j<v_nodes[i].n_checknodes; j++) {
      int cnIndex = v_nodes[i].checknode_indexes[j];


      int cnInputIndex;
      // Now let's look inside that check node to find our appropriate location
      for(int n=0; n<N_CHECK_INPUTS; n++) {
        int tempIndex = c_nodes[cnIndex].variablenode_indexes[n];

        // If we found our variable node, make the table
        if(tempIndex == i) {
          cnInputIndex = n;
          break;
        }
      }

      // Now we can create our table entry
      table[i*N_VARIABLE_INPUTS + j] = (cnIndex*(N_CHECK_INPUTS+1)) + cnInputIndex;




    }

    for(j = v_nodes[i].n_checknodes; j < N_VARIABLE_INPUTS; j++) {
      // For the rest of them we want to assign the random check node at the end
      int cnIndex = v_nodes[i].checknode_indexes[j];

      // Alright so we just need to assign the last position
      table[i*N_VARIABLE_INPUTS + j] = (cnIndex * (N_CHECK_INPUTS+1)) + N_CHECK_INPUTS;
    }







  }
  return table;
}

int * generate_CNtoVN_index_table(VariableNode * v_nodes,CheckNode * c_nodes) {
  int * table = (int*)_aligned_malloc(sizeof(int)*N_CHECK_NODES*N_CHECK_INPUTS,
      ↪ AOCL_ALIGNMENT);
  if(table == NULL)
    printf("err␣=␣%d\n",errno);
  // Iterate over the list of variable nodes
  for(int i=0; i<N_CHECK_NODES; i++) {

    int j;
    // For each variable node we need to find the appropriate connections for each input
    for(j=0; j<c_nodes[i].n_inputs; j++) {
      int vnIndex = c_nodes[i].variablenode_indexes[j];

      int vnInputIndex = -100;
      // Now let's look inside that check node to find our appropriate location
      for(int n=0; n<N_VARIABLE_INPUTS; n++) {
        int tempIndex = v_nodes[vnIndex].checknode_indexes[n];

        // If we found our variable node, make the table
        if(tempIndex == i) {
          vnInputIndex = n;
          break;
        }
      }

      int foundIndex = (vnIndex * (N_VARIABLE_INPUTS+1)) + vnInputIndex;

      // Populate the table
      table[i*N_CHECK_INPUTS + j] = foundIndex;

    }

    // Alright for the rest all we need to do is give it the garbage address
    for(j = c_nodes[i].n_inputs; j < N_CHECK_INPUTS; j++) {
      int vnIndex = c_nodes[i].variablenode_indexes[j];


      table[i*N_CHECK_INPUTS + j] = (vnIndex * (N_VARIABLE_INPUTS+1)) + N_VARIABLE_INPUTS;



    }


  }

  printf("one␣in␣question␣=␣%d\n",table[1*N_CHECK_INPUTS + 15]);

  return table;
}

double generate_random_number()
{
#define C1 4294967294
#define C2 4294967288
#define C3 4294967280
#define MAX 4294967295

  srand(45);
  static unsigned int s1 = rand();
  static unsigned int s2 = rand();
  static unsigned int s3 = rand();

  // Here's the first part
  // Now let's crank the tausworthe
  unsigned int xor1 = s1 << 13;

  // The first set of and/xor
  unsigned int left_temp = xor1 ^ s1;
  unsigned int right_temp = C1 & s1;

  // Shifts
  left_temp = left_temp >> 19;
  right_temp = right_temp << 12;

  s1 = left_temp ^ right_temp;
  // Second part
  xor1 = s2<<2;
  left_temp = xor1 ^ s2;
  right_temp = C2 & s2;

  left_temp = left_temp >> 25;
  right_temp = right_temp << 4;

  s2 = left_temp ^ right_temp;

  // Third part
  xor1 = s3 << 3;
  left_temp = xor1 ^ s3;
  right_temp = C3 & s3;

  left_temp = xor1 ^ s3;
  right_temp = C3 & s3;

  left_temp = left_temp >> 11;
  right_temp = right_temp << 17;

  s3 = left_temp ^ right_temp;

  // Now the return
  unsigned int output = s1 ^ s2 ^ s3;

  // Now just convert it into a double
  double last_value = (double)output /MAX;

  //cout<<"last value" << last_value<<endl;
  return last_value;

}
#pragma endregion

void defineGlobalDefinitions() {

  vector_max_llr = (LLR*) _aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  vector_min_neg_llr = (LLR*)_aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  vector_sign_bit = (LLR*)_aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  allones = (LLR*)_aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  ones = (LLR*)_aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  zeros = (LLR*)_aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  vector_too_neg_llr = (LLR*)_aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  total_vector_max_llr = (TOTAL_TYPE*)_aligned_malloc(sizeof(TOTAL_TYPE),AOCL_ALIGNMENT);
  total_vector_min_neg_llr = (TOTAL_TYPE*)_aligned_malloc(sizeof(TOTAL_TYPE),
      ↪ AOCL_ALIGNMENT);



#if MANUAL_VECTOR_WIDTH==1
  vector_max_llr = MAX_LLR;
  vector_min_neg_llr = MAX_NEG_LLR;
  vector_sign_bit = SIGN_BIT;
  allones = (LLR)(0xFF);
  ones = (LLR)(1);
  zeros = (LLR)(0);
  vector_too_neg_llr = MAX_NEG_LLR - ones;
  total_vector_max_llr = MAX_LLR;
  total_vector_min_neg_llr = MAX_NEG_LLR;
#else
  for(int i=0; i<MANUAL_VECTOR_WIDTH; i++)
  {
    (*vector_max_llr).s[i] = MAX_LLR;
    (*vector_min_neg_llr).s[i] = MAX_NEG_LLR;
    (*vector_sign_bit).s[i] = SIGN_BIT;
    (*allones).s[i] = (0xFF);
    (*ones).s[i] = (1);
    (*zeros).s[i] = (0);
    (*vector_too_neg_llr).s[i] = MAX_NEG_LLR - 1;
    (*total_vector_max_llr).s[i] = MAX_LLR;
    (*total_vector_min_neg_llr).s[i] = MAX_NEG_LLR;
  }
#endif
}

void parseProgramInputs(int n_args,char* args[]) {
  for(int i=0; i<n_args-1; i++) {
    if(strncmp(args[i],"-nVars",MAX_LINE)==0) {
      // If we found the n_vars just set the variable

      int new_value = atoi(args[i+1]);
      if(new_value <= 0) {
        ExitOnErrorWMsg("Not␣a␣valid␣LDPC␣size␣input\n");
      }
      N_VARIABLE_NODES = atoi(args[i+1]);
      LDPC_SIZE = N_VARIABLE_NODES;
    }
    else if (strncmp(args[i], "-nCheck",MAX_LINE)==0) {
      int new_value = atoi(args[i+1]);
      if(new_value <=0 ) {
        ExitOnErrorWMsg("Not a valid LDPC size input\n");
      }
      N_CHECK_NODES = atoi(args[i+1]);
    }
  }

}
```

Appendix K

Irregular Design E Host Code (C)

```
// This is the start of the Channel LDPC implementation
#include <iostream>
#include <fstream>
#include <ctime>
#include <math.h>
#include <malloc.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <Windows.h>
#include "E:\Andrew\Projects\common\ax_common.h"



//#define ALTERA

#define MANUAL_VECTOR_WIDTH 16

int N_VARIABLE_NODES = 1120;
int N_CHECK_NODES =280;
#define MAX_VARIABLE_EDGES 5
#define MAX_CHECK_EDGES 16

#define PROJECT_TITLE "Irregular Global Memory LDPC"
#define PROJECT_DESCRIPTION "This host code supports the multi kernel implementation of
    ↪ the LDPC decoder using the char16 data type. Within the char16 data type, each of
    ↪ the chars will contain a different codeword to decode"


// These options apply to the code being used as well as the number of iterations to do
int LDPC_SIZE = N_VARIABLE_NODES; // Note this is overwritten when a value is passed in
    ↪ via command line

#define FILENAME "matrix_"
#define OUTPUT_FILE "results_"
#define N_ITERATIONS 32

#define N_VARIABLE_INPUTS (MAX_VARIABLE_EDGES)
#define N_CHECK_INPUTS (MAX_CHECK_EDGES)


#define N_VARIABLE_INPUTS_W_CHANNEL (N_VARIABLE_INPUTS+1)


#define MAX_LINE (1024*3)

static char * INPUT_FILENAME = (char*) malloc(sizeof(char)*MAX_LINE);
static char * OUTPUT_FILENAME = (char*) malloc(sizeof(char)*MAX_LINE);

#if MANUAL_VECTOR_WIDTH==2
#define LLR_TYPE cl_char2
#define TOTAL_TYPE cl_short2
#define CONVERT_TO_SHORT(a) convert_short2(a)
#define CONVERT_TO_CHAR(a) convert_char2(a)
#elif MANUAL_VECTOR_WIDTH==4
#define LLR_TYPE cl_char4
#define TOTAL_TYPE cl_short4
#define CONVERT_TO_SHORT(a) convert_short4(a)
#define CONVERT_TO_CHAR(a) convert_char4(a)
#elif MANUAL_VECTOR_WIDTH==8
#define LLR_TYPE cl_char8
```

```
#define TOTAL_TYPE cl_short8
#define CONVERT_TO_SHORT(a) convert_short8(a)
#define CONVERT_TO_CHAR(a) convert_char8(a)
#elif MANUAL_VECTOR_WIDTH==16
#define LLR_TYPE cl_char16
#define TOTAL_TYPE cl_short16
#define CONVERT_TO_SHORT(a) convert_short16(a)
#define CONVERT_TO_CHAR(a) convert_char16(a)
#else
#define LLR_TYPE cl_char
#define TOTAL_TYPE cl_short
#define CONVERT_TO_SHORT(a) convert_short(a)
#define CONVERT_TO_CHAR(a) convert_char(a)
#endif


// Integer options
#define LLR_BASE_TYPE char
#define N_BITS_LLR (sizeof(LLR_BASE_TYPE)*8) // 8 bits per byte
#define MAX_LLR ((1 << (N_BITS_LLR-1))-1)
#define SIGN_BIT (MAX_LLR+1)
#define MAX_NEG_LLR (0-MAX_LLR)
typedef LLR_TYPE LLR;

LLR *vector_max_llr;
LLR *vector_min_neg_llr;
LLR *vector_n_bits_llr;
LLR *vector_sign_bit;
LLR *vector_too_neg_llr;
LLR *allones;
LLR *ones;
LLR *zeros;
TOTAL_TYPE *total_vector_max_llr;
TOTAL_TYPE *total_vector_min_neg_llr;


#define BINS_PER_ONE_STEP 4
#define SATURATION_POINT int(MAX_LLR/BINS_PER_ONE_STEP)



//#define PRINT_DEBUG
//#define QUICK_DEBUG

#ifdef PRINT_DEBUG
#undef QUICK_DEBUG
#endif

#define DEBUG_FILENAME "output_debug_integer.csv"
#define CHECKNODE_DEBUG_FILENAME "checknode_debug.csv"
#define VARIABLENODE_DEBUG_FILENAME "variablenode_debug.csv"




#pragma region OpenCL Specific Declarations
static char * KERNEL_FILENAME  = (char*) malloc(sizeof(char)*MAX_LINE);
static char const* COMPILER_OPTIONS = "-w";
#pragma endregion

#define DEBUG 1
#define AOCL_ALIGNMENT 64





#pragma region Error Checking Macros
#define SYSTEM_ERROR -1
#define CheckPointer(pointer) CheckPointerFunc(pointer,__FUNCTION__, __LINE__)
#define ExitOnErrorWMsg(msg) ExitOnErrorFunc(msg,__FUNCTION__,__LINE__)
#define ExitOnError() ExitOnErrorWMsg("Error! Please refer to")
#pragma endregion

#pragma region Structure Defintions
// This structure holds all data required for the variable nodes.
// Note that the last input will be the channel input for each variable node
// the n_inputs will always be n_checknodes+1
typedef struct VariableNodes
{
  LLR inputs[MAX_VARIABLE_EDGES+1];
  cl_short checknode_indexes[MAX_VARIABLE_EDGES];
  cl_short index;
  cl_short n_inputs;
  cl_short n_checknodes;
}VariableNode;

// This structure holds all data required for the check nodes
typedef struct CheckNodes
{
  LLR inputs[MAX_CHECK_EDGES];
  cl_short variablenode_indexes[MAX_CHECK_EDGES];
  cl_short index;
  cl_short n_inputs;
}CheckNode;
#pragma endregion

#pragma region Function Declaractions
static void CheckPointerFunc(void * pointer,char const* const func,const int line);
static void ExitOnErrorFunc(char const* msg, char const* func, const int line);
static void readParityFile(VariableNode*&, CheckNode*&, int&, int&);
static bool checkIfSatisfied(LLR * parity);
static void updateParity(CheckNode *& c_nodes, const int n_checks);
static LLR * convertCodewordToBinary(VariableNode*&,const int n_vars);
static float* getCodeword(int size);
static LLR_BASE_TYPE * convertToLLR(const float * codeword, const int n_vars);
static void printCodeword(const int * codeword, const int n_vars);
static void printResults(LLR * results);
static void printResultsToFile(double overall, double variable, double check, double misc)
     ↪ ;
static void setFileNames();

double generate_random_gaussian();
double generate_random_number();
void introduceNoise(float*,int,int);

int * generate_VNtoCN_index_table(VariableNode* ,CheckNode* );
int * generate_CNtoVN_index_table(VariableNode* ,CheckNode* );



static void startOpenCL(cl_context,cl_device_id device, cl_program program);
static void startSimulationOpenCL(cl_context, cl_device_id,cl_program,VariableNode*&,
     ↪ CheckNode*&,const int, const int);
static void ldpcDecodeOpenCL(cl_context, cl_command_queue,cl_command_queue,cl_kernel,
     ↪ cl_kernel,VariableNode*&,CheckNode*&,const int, const int,float*);
void initializeNodes(VariableNode*&, CheckNode*&,const LLR*,const int, const int);
void defineGlobalDefinitions();

void parseProgramInputs(int,char*[]);
#pragma endregion






int main(int argc, char * argv[])
{
  printf("%s\n",PROJECT_TITLE);
  printf("%s\n",PROJECT_DESCRIPTION);

  parseProgramInputs(argc,argv);

  // Let's do a quick check on the number
  if(LDPC_SIZE <=0) {
    printf("Codelength␣(number␣of␣variable␣nodes)␣is␣not␣valid\n");
    ExitOnError();
  }

  // Set the file names for the input and output
  setFileNames();


  // Set up the program
  cl_context context;
  cl_device_id device;
  cl_program program;
  cl_device_type type = CL_DEVICE_TYPE_ALL;

  printf("\nOpenCL␣API\n\n");
  printf("Max␣LLR␣=␣%d\n",MAX_LLR);


  // Let's do this
  AxCreateContext(type,AX_PLATFORM_STRING,&context,&device);
#ifdef ALTERA
  AxBuildProgramAltera(&program,device,context,KERNEL_FILENAME,COMPILER_OPTIONS);
#else
  AxBuildProgram(&program,device,context,KERNEL_FILENAME,COMPILER_OPTIONS);
#endif

  // Now let's start the rest of the program
  startOpenCL(context,device,program);

  clReleaseProgram(program);
  clReleaseContext(context);


}


// Start the main chunk of the host code
void startOpenCL(cl_context context, cl_device_id device, cl_program program)
```

```
{

  defineGlobalDefinitions();
  // Declare the variables
  // Declare an array to hold the checknodes
  CheckNode * c_nodes=NULL;

  // Declare an array to hold the variablenodes
  VariableNode * v_nodes=NULL;

  // The number of check nodes and variable nodes
  int n_checks=0;
  int n_vars=0;

  // Let's read the parity file to get the nodes
  readParityFile(v_nodes,c_nodes,n_vars,n_checks);
  printf("Testing␣file␣for␣accuracy....");
  for(int i=0; i<n_vars; i++) {
    // Check that every fanout isn't greater than the max
    if(v_nodes[i].n_inputs > N_VARIABLE_INPUTS_W_CHANNEL)
      ExitOnErrorWMsg("Software␣defined␣N_VARIABLE_INPUTS␣too␣small");

  }
  for(int i=0; i<n_checks; i++) {

    if(c_nodes[i].n_inputs > N_CHECK_INPUTS)
      ExitOnErrorWMsg("Software␣defined␣N_CHECK_INPUTS␣too␣small");
  }
  printf("Successful\n");

  // Let's just do a few quick checks to make sure that the read worked properly
  // We should check the c_nodes and v_nodes pointers
  // We also need to check that the number of variable nodes and check nodes are valid
  CheckPointer(c_nodes);CheckPointer(v_nodes);
  if(n_vars <= 0 || n_checks <= 0)
    ExitOnErrorWMsg("Number␣of␣variables␣and␣check␣nodes␣not␣valid.");

  printf("\n**********************************************\n");
  printf("Variable␣Nodes␣%d,␣Check␣Nodes␣%d\n",n_vars,n_checks);
  printf("%d␣Log␣Likelihood␣Ratio␣Bits\n",N_BITS_LLR);
  printf("Max␣Variable␣Edges␣%d,␣Max␣Check␣Edges␣%d\n",MAX_VARIABLE_EDGES,MAX_CHECK_EDGES)
      ↪ ;

  // Now we are ready to start our simulation
  startSimulationOpenCL(context,device,program,v_nodes,c_nodes,n_vars,n_checks);

}

void startSimulationOpenCL(cl_context context, cl_device_id device,cl_program program,
    ↪ VariableNode*& v_nodes,CheckNode*& c_nodes, const int n_vars, const int n_checks)
{

  // Firstly, we need to get our codeword to decode
  float * codeword = getCodeword(n_vars);


  cl_int err;
  // Let's create our openCL stuff

  size_t output;
  AxCheckError(clGetDeviceInfo(device,CL_DEVICE_MAX_WORK_GROUP_SIZE,sizeof(output),&output
      ↪ ,0));
  printf("It␣is␣=␣%d\n",output);

  // Now we need to create our command queues
  // We will create one command queue for each kernel
  cl_command_queue VariableQueue = clCreateCommandQueue(context,device,
      ↪ CL_QUEUE_PROFILING_ENABLE,&err);AxCheckError(err);
  cl_command_queue CheckQueue = clCreateCommandQueue(context,device,
      ↪ CL_QUEUE_PROFILING_ENABLE,&err);AxCheckError(err);



  // Now we need to create the kernels themselves.
  cl_kernel VariableKernel = clCreateKernel(program,"VariableNode",&err);AxCheckError(err)
      ↪ ;
  cl_kernel CheckKernel = clCreateKernel(program,"CheckNode",&err);AxCheckError(err);

  // Now let's simply start the LDPC
  ldpcDecodeOpenCL(context, VariableQueue,CheckQueue,VariableKernel,CheckKernel,v_nodes,
      ↪ c_nodes,n_vars,n_checks,codeword);
}

void copyCodewordFloat(float *& source, float *& dest, int n_vars) {

  for(int i=0; i<n_vars; i++) {
    dest[i] = source[i];
  }
}

void copyCodewordLLRBase(LLR_BASE_TYPE *& source, LLR_BASE_TYPE *& dest, int n_vars) {
  for(int i=0; i<n_vars; i++) {
    dest[i] = source[i];
  }
}

void ldpcDecodeOpenCL(cl_context      context,
            cl_command_queue    VariableQueue,
            cl_command_queue    CheckQueue,
            cl_kernel       VariableKernel,
            cl_kernel       CheckKernel,
            VariableNode*&    v_nodes,
            CheckNode*&     c_nodes,
            const int       n_vars,
            const int       n_checks,
            float*        codeword)
{
  cl_int err;


  // Note this is where I would initiate an LLR for fixed integer
  size_t channelSize = sizeof(LLR) * N_VARIABLE_NODES*2;
  LLR * codeword_LLR = (LLR*) _aligned_malloc(channelSize,AOCL_ALIGNMENT);


  float * codeword0 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword1 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword2 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword3 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword4 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword5 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword6 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword7 = (float*) malloc(sizeof(float) * n_vars);

  float * codeword8 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword9 = (float*) malloc(sizeof(float) * n_vars);
  float * codeworda = (float*) malloc(sizeof(float) * n_vars);
  float * codewordb = (float*) malloc(sizeof(float) * n_vars);
  float * codewordc = (float*) malloc(sizeof(float) * n_vars);
  float * codewordd = (float*) malloc(sizeof(float) * n_vars);
  float * codeworde = (float*) malloc(sizeof(float) * n_vars);
  float * codewordf = (float*) malloc(sizeof(float) * n_vars);


  copyCodewordFloat(codeword,codeword0,n_vars);
  copyCodewordFloat(codeword,codeword1,n_vars);
  copyCodewordFloat(codeword,codeword2,n_vars);
  copyCodewordFloat(codeword,codeword3,n_vars);
  copyCodewordFloat(codeword,codeword4,n_vars);
  copyCodewordFloat(codeword,codeword5,n_vars);
  copyCodewordFloat(codeword,codeword6,n_vars);
  copyCodewordFloat(codeword,codeword7,n_vars);

  copyCodewordFloat(codeword,codeword8,n_vars);
  copyCodewordFloat(codeword,codeword9,n_vars);
  copyCodewordFloat(codeword,codeworda,n_vars);
  copyCodewordFloat(codeword,codewordb,n_vars);
  copyCodewordFloat(codeword,codewordc,n_vars);
  copyCodewordFloat(codeword,codewordd,n_vars);
  copyCodewordFloat(codeword,codeworde,n_vars);
  copyCodewordFloat(codeword,codewordf,n_vars);






  // Now we need to introduce some noise into our codeword
  // Note that this function changes our codeword input variable
  // But let's add noise individually to each one
  introduceNoise(codeword0,1000,n_vars);
  introduceNoise(codeword1,-20,n_vars);
  introduceNoise(codeword2,-20,n_vars);
  introduceNoise(codeword3,-12,n_vars);
  introduceNoise(codeword4,-10,n_vars);
  introduceNoise(codeword5,-20,n_vars);
  introduceNoise(codeword6,-20,n_vars);
  introduceNoise(codeword7,-12,n_vars);

  introduceNoise(codeword8,-10,n_vars);
  introduceNoise(codeword9,-20,n_vars);
  introduceNoise(codeworda,-20,n_vars);
  introduceNoise(codewordb,-12,n_vars);
  introduceNoise(codewordc,10,n_vars);
  introduceNoise(codewordd,20,n_vars);
  introduceNoise(codeworde,-20,n_vars);
  introduceNoise(codewordf,-12,n_vars);

  /*for(int i=0; i<n_vars; i++) {
    printf("codeword[%d] = %f\n",i,codeword0[i]);
  }*/

  // Now I need to add in the step to conver the number to an integer
  char * llr0 = convertToLLR(codeword0,n_vars);
  char * llr1 = convertToLLR(codeword1,n_vars);
  char * llr2 = convertToLLR(codeword2,n_vars);
  char * llr3 = convertToLLR(codeword3,n_vars);
  char * llr4 = convertToLLR(codeword4,n_vars);
  char * llr5 = convertToLLR(codeword5,n_vars);
  char * llr6 = convertToLLR(codeword6,n_vars);
  char * llr7 = convertToLLR(codeword7,n_vars);

  char * llr8 = convertToLLR(codeword8,n_vars);
  char * llr9 = convertToLLR(codeword9,n_vars);
  char * llra = convertToLLR(codeworda,n_vars);
  char * llrb = convertToLLR(codewordb,n_vars);
  char * llrc = convertToLLR(codewordc,n_vars);
  char * llrd = convertToLLR(codewordd,n_vars);
  char * llre = convertToLLR(codeworde,n_vars);
  char * llrf = convertToLLR(codewordf,n_vars);


  for(int i=0; i<n_vars; i++) {
#if MANUAL_VECTOR_WIDTH == 1
    codeword_LLR[i]= llr0[i];
#else

    for(int j=0; j <MANUAL_VECTOR_WIDTH; j++) {
      codeword_LLR[i].s[j] = llr0[i];
    }


#endif


  }

  int k=0;
  for(int i=n_vars; i<n_vars*2;i++) {

    for(int j=0; j<MANUAL_VECTOR_WIDTH; j++) {
      codeword_LLR[i].s[j] = llr0[k];
    }
    k++;
  }

  /*for(int i=0; i<n_vars; i++) {
#if MANUAL_VECTOR_WIDTH == 1
    printf("codeword[%d] = %d\n",i,codeword_LLR[i]);//.x,codeword_LLR[i].y,codeword_LLR[i
        ↪ ].z,codeword_LLR[i].w);
#else
    printf("codeword[%d] = %d\n",i,codeword_LLR[i].s0);//codeword_LLR[i].y,codeword_LLR[i
        ↪ ].z,codeword_LLR[i].w);
#endif
  }*/

  // Let's start initializing the inputs
  // For the codewords starting at the VN we just need to create a buffer with the
      ↪ measurements





  // For the second version we need to start by doing the first VN portion
  //initializeNodes(v_nodes,c_nodes,codeword_LLR,n_vars,n_checks);

  // Let's get the connection tables
  size_t VNtoCNSize = sizeof(int)*N_VARIABLE_NODES*N_VARIABLE_INPUTS;
  size_t CNtoVNSize = sizeof(int)*N_CHECK_NODES*N_CHECK_INPUTS;
  printf("Generating␣VntoCN␣lookup␣table....");

  int * VNtoCNLookup_h = generate_VNtoCN_index_table(v_nodes,c_nodes);
  printf("Successful\n");


  printf("Generating␣CntoVN␣lookup␣table....");
  int * CNtoVNLookup_h = generate_CNtoVN_index_table(v_nodes,c_nodes);

  printf("Successful\n");




  cl_mem VNtoCNLookup = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
      ↪ VNtoCNSize,VNtoCNLookup_h,&err);AxCheckError(err);
  cl_mem CNtoVNLookup = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
      ↪ CNtoVNSize,CNtoVNLookup_h,&err);AxCheckError(err);




  // Make the constant memory info
```

```
cl_mem vector_max_llr_d = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
     ↪ sizeof(LLR),vector_max_llr,&err);AxCheckError(err);
cl_mem vector_sign_bit_d = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
     ↪ sizeof(LLR),vector_sign_bit,&err);AxCheckError(err);
cl_mem allones_d = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,sizeof(
     ↪ LLR),allones,&err);AxCheckError(err);
cl_mem ones_d = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,sizeof(LLR)
     ↪ ,ones,&err);AxCheckError(err);
cl_mem zeros_d = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,sizeof(LLR
     ↪ ),zeros,&err);AxCheckError(err);
cl_mem total_vector_max_llr_d = clCreateBuffer(context,CL_MEM_READ_ONLY|
     ↪ CL_MEM_COPY_HOST_PTR,sizeof(TOTAL_TYPE),total_vector_max_llr,&err);AxCheckError(
     ↪ err);
cl_mem total_vector_min_neg_llr_d = clCreateBuffer(context,CL_MEM_READ_ONLY|
     ↪ CL_MEM_COPY_HOST_PTR,sizeof(TOTAL_TYPE),total_vector_min_neg_llr,&err);
     ↪ AxCheckError(err);

size_t vnResultSize = sizeof(LLR) * N_VARIABLE_NODES*2;
size_t cnResultSize = sizeof(LLR) * N_CHECK_NODES*2;

LLR * vnResult = (LLR*) _aligned_malloc(vnResultSize,AOCL_ALIGNMENT);
LLR * cnResult = (LLR*) _aligned_malloc(cnResultSize,AOCL_ALIGNMENT);

cl_mem vnResult_d = clCreateBuffer(context,CL_MEM_READ_WRITE,vnResultSize,NULL,&err);
     ↪ AxCheckError(err);
cl_mem cnResult_d = clCreateBuffer(context,CL_MEM_READ_WRITE,cnResultSize,NULL,&err);
     ↪ AxCheckError(err);


cl_mem initialChannelMeasurements = clCreateBuffer(context,CL_MEM_READ_WRITE |
     ↪ CL_MEM_COPY_HOST_PTR, channelSize,codeword_LLR,&err);AxCheckError(err);

// We now need to create the buffers for each node in global memory (for data transfer
     ↪ between the nodes)
size_t vnInputsGMSize = sizeof(LLR)*N_VARIABLE_NODES*(N_VARIABLE_INPUTS+1)*2;
size_t cnInputsGMSize = sizeof(LLR)*N_CHECK_NODES*(N_CHECK_INPUTS+1)*2;

// We need to initialize them to 0 which means we need to create and send that data
LLR * initializationVNBuffer = (LLR*) _aligned_malloc(vnInputsGMSize,AOCL_ALIGNMENT);
LLR * initializationCNBuffer = (LLR*) _aligned_malloc(cnInputsGMSize,AOCL_ALIGNMENT);

CheckPointer(initializationVNBuffer);
CheckPointer(initializationCNBuffer);


printf("Initializing␣VN␣Buffers....");
for(int i=0; i < n_vars*2; i++) {

  // For each one let's look at the inputs
  int j;
  for(j = 0; j< (N_VARIABLE_INPUTS + 1); j++) {
    for(int k=0; k<MANUAL_VECTOR_WIDTH; k++) {
      initializationVNBuffer[i*(N_VARIABLE_INPUTS+1) + j].s[k]= (LLR_BASE_TYPE) 0;
    }
  }
}


printf("Successful\n");
printf("Initializing␣CN␣Buffers....");
for(int i=0; i< n_checks * 2; i++) {
  int c = i % n_checks;

  int j;
  for(j = 0; j < c_nodes[c].n_inputs; j++) {
    for(int k=0; k<MANUAL_VECTOR_WIDTH; k++) {
      initializationCNBuffer[i*(N_CHECK_INPUTS +1) + j].s[k]= (LLR_BASE_TYPE) 0;
    }
  }

  for(j = c_nodes[c].n_inputs; j < (N_CHECK_INPUTS + 1); j++) {
    for(int k=0; k<MANUAL_VECTOR_WIDTH; k++) {
      initializationCNBuffer[i*(N_CHECK_INPUTS +1) + j].s[k]= (LLR_BASE_TYPE) MAX_LLR;
    }
  }

}

printf("Successful\n");

// Start with the Variable Node buffer
cl_mem vnInputsBuffer = clCreateBuffer(context,CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR
     ↪ ,vnInputsGMSize,initializationVNBuffer,&err);AxCheckError(err);
cl_mem cnInputsBuffer = clCreateBuffer(context,CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR,
     ↪  cnInputsGMSize, initializationCNBuffer, &err); AxCheckError(err);

// Make the sync buffers
char * VNSync_h = (char*) _aligned_malloc(1,AOCL_ALIGNMENT);
char * CNSync_h = (char*) _aligned_malloc(1,AOCL_ALIGNMENT);

(*VNSync_h) = -1;
(*CNSync_h) = -1;
```

```
cl_mem VNSync = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, 1,
     ↪ VNSync_h,&err);AxCheckError(err);
cl_mem CNSync = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, 1,
     ↪ CNSync_h,&err);AxCheckError(err);

printf("Host␣pointers\n");
printf("VNtoCNLookup␣%lx\n",VNtoCNLookup_h);
printf("CNtoVNLookup␣%lx\n",CNtoVNLookup_h);
printf("codeword_LLR␣%lx\n",codeword_LLR);
printf("initializationVNBuffer␣%lx\n",initializationVNBuffer);
printf("initiailzationCNBuffer␣%lx\n",initializationCNBuffer);

printf("VNSync␣%lx\n",VNSync_h);
printf("CNSync␣%lx\n",CNSync_h);
printf("vector_max_llr␣%lx\n",vector_max_llr);
printf("vector_sign_bit␣%lx\n",vector_sign_bit);
printf("allones␣%lx\n",allones);
printf("ones␣%lx\n",ones);
printf("total_vector_max_llr␣%lx\n",total_vector_max_llr);
printf("total_vector_min_neg_llr␣%lx\n",total_vector_min_neg_llr);
printf("VNResult␣%lx\n",vnResult);
printf("CNResult␣%lx\n",cnResult);


int vnargs = 0;
int cnargs = 0;

// Let's set up the kernels
// Starting with the Variable Node
AxCheckError(clSetKernelArg(VariableKernel,vnargs++,sizeof(VNtoCNLookup),&VNtoCNLookup))
     ↪ ;
AxCheckError(clSetKernelArg(VariableKernel,vnargs++,sizeof(initialChannelMeasurements),&
     ↪ initialChannelMeasurements));
AxCheckError(clSetKernelArg(VariableKernel,vnargs++,sizeof(vnInputsBuffer),&
     ↪ vnInputsBuffer));
AxCheckError(clSetKernelArg(VariableKernel,vnargs++,sizeof(cnInputsBuffer),&
     ↪ cnInputsBuffer));
AxCheckError(clSetKernelArg(VariableKernel,vnargs++,sizeof(vnResult_d),&vnResult_d));
AxCheckError(clSetKernelArg(VariableKernel,vnargs++,sizeof(VNSync),&VNSync));
AxCheckError(clSetKernelArg(VariableKernel,vnargs++,sizeof(CNSync),&CNSync));
AxCheckError(clSetKernelArg(VariableKernel,vnargs++,sizeof(vector_max_llr_d),&
     ↪ vector_max_llr_d));
AxCheckError(clSetKernelArg(VariableKernel,vnargs++,sizeof(vector_sign_bit_d),&
     ↪ vector_sign_bit_d));
AxCheckError(clSetKernelArg(VariableKernel,vnargs++,sizeof(allones_d),&allones_d));
AxCheckError(clSetKernelArg(VariableKernel,vnargs++,sizeof(ones_d),&ones_d));
AxCheckError(clSetKernelArg(VariableKernel,vnargs++,sizeof(zeros_d),&zeros_d));
AxCheckError(clSetKernelArg(VariableKernel,vnargs++,sizeof(total_vector_max_llr_d),&
     ↪ total_vector_max_llr_d));
AxCheckError(clSetKernelArg(VariableKernel,vnargs++,sizeof(total_vector_min_neg_llr_d),&
     ↪ total_vector_min_neg_llr_d));

// Check Node
AxCheckError(clSetKernelArg(CheckKernel,cnargs++,sizeof(CNtoVNLookup),&CNtoVNLookup));
AxCheckError(clSetKernelArg(CheckKernel,cnargs++,sizeof(vector_max_llr_d),&
     ↪ vector_max_llr_d));
AxCheckError(clSetKernelArg(CheckKernel,cnargs++,sizeof(vector_sign_bit_d),&
     ↪ vector_sign_bit_d));
AxCheckError(clSetKernelArg(CheckKernel,cnargs++,sizeof(allones_d),&allones_d));
AxCheckError(clSetKernelArg(CheckKernel,cnargs++,sizeof(ones_d),&ones_d));
AxCheckError(clSetKernelArg(CheckKernel,cnargs++,sizeof(zeros_d),&zeros_d));
AxCheckError(clSetKernelArg(CheckKernel,cnargs++,sizeof(cnInputsBuffer),&cnInputsBuffer)
     ↪ );
AxCheckError(clSetKernelArg(CheckKernel,cnargs++,sizeof(vnInputsBuffer),&vnInputsBuffer)
     ↪ );
AxCheckError(clSetKernelArg(CheckKernel,cnargs++,sizeof(cnResult_d),&cnResult_d));

clock_t start_t, end_t;
double total_t;

unsigned _int64 freq;
QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
double timerFrequency = (1.0/freq);

start_t = clock();

unsigned __int64 startTime;
QueryPerformanceCounter((LARGE_INTEGER *)&startTime);


// Now let's run the kernels
size_t global_size_vn= N_VARIABLE_NODES;
size_t local_size_vn=  N_VARIABLE_NODES/2;

// Now we start the Check Node Kernel
size_t global_size_cn = N_CHECK_NODES;
size_t local_size_cn = N_CHECK_NODES;


cl_event kernelTimeVN;
printf("Starting␣the␣Variable␣Node␣Kernel\n");
```

```
  AxCheckError(clEnqueueNDRangeKernel(VariableQueue,VariableKernel,1,NULL,&global_size_vn
      ↪ ,&local_size_vn,0,NULL,&kernelTimeVN));

  printf("Starting␣the␣Check␣Node␣Kernel\n");
  cl_event kernelTimeCN;
  AxCheckError(clEnqueueNDRangeKernel(CheckQueue,CheckKernel,1,NULL,&global_size_cn,&
      ↪ local_size_cn,0,NULL,&kernelTimeCN));



  clFinish(CheckQueue);
  clFinish(VariableQueue);

  end_t = clock();

  unsigned __int64 endTime;
  QueryPerformanceCounter((LARGE_INTEGER *)&endTime);
  double timeDifferenceInMilliseconds = ((endTime-startTime) * timerFrequency)*1000;

  total_t = (double)(end_t - start_t)/CLOCKS_PER_SEC;
  printf("Clock␣says␣%f\n",total_t);

  printf("Timemilliseconds␣=␣%f\n",timeDifferenceInMilliseconds);
  printf("frequency␣=␣%f\n",timerFrequency);
  printf("LArger␣start␣=␣%llu\n",startTime);
  printf("Larger␣end␣␣%llu\n",endTime);

  clWaitForEvents(1,&kernelTimeVN);
  printf("Variable␣Node␣kernel␣has␣finished␣all␣%d␣iterations\n", N_ITERATIONS);

  clWaitForEvents(1,&kernelTimeCN);
  printf("Check␣Node␣kernel␣has␣finished␣all␣%d␣iterations\n", N_ITERATIONS);




  // Now we can just get the data
  //AxCheckError(clEnqueueReadBuffer(LDPCQueue,vnResult_d,CL_TRUE,0,vnResultSize,vnResult
      ↪ ,0,NULL,NULL));
  //AxCheckError(clEnqueueReadBuffer(LDPCQueue,cnResult_d,CL_TRUE,0,cnResultSize,cnResult
      ↪ ,0,NULL,NULL));

  // Let's check the accuracy
  // Ok let's print it out
  //checkIfSatisfied(cnResult);
  //printResults(vnResult);
  double milisecondTime = timeDifferenceInMilliseconds;

  printf("The␣kernel␣execution␣time␣was␣%f␣ms\n",milisecondTime);
  double throughput = N_VARIABLE_NODES*MANUAL_VECTOR_WIDTH*2/((milisecondTime)/1000.0);
  printf("The␣throughput␣was␣%f␣bps\n",throughput);

  /*cl_ulong time_start,time_end;
  double total_time;

  AxCheckError(clGetEventProfilingInfo(kernelTime,CL_PROFILING_COMMAND_START,sizeof(
      ↪ time_start),&time_start,NULL));
  AxCheckError(clGetEventProfilingInfo(kernelTime,CL_PROFILING_COMMAND_END,sizeof(time_end
      ↪ ),&time_end,NULL));

  total_time = time_end-time_start;
  // Now we can just get the data
  AxCheckError(clEnqueueReadBuffer(LDPCQueue,vnResult_d,CL_TRUE,0,vnResultSize,vnResult,0,
      ↪ NULL,NULL));
  AxCheckError(clEnqueueReadBuffer(LDPCQueue,cnResult_d,CL_TRUE,0,cnResultSize,cnResult,0,
      ↪ NULL,NULL));
  */
  // Let's check the accuracy
  // Ok let's print it out
  //checkIfSatisfied(cnResult);
  //printResults(vnResult);
  //double milisecondTime = total_time/1000000.0;

  /*printf("The kernel execution time was %f ms\n",milisecondTime);
  double throughput = N_VARIABLE_NODES*MANUAL_VECTOR_WIDTH/((milisecondTime)/1000.0);
  printf("The throughput was %f bps\n",throughput);

  FILE * results = fopen("results_kernel.csv","w");

  fprintf(results,"%f,%f\n",milisecondTime,throughput);
  fclose(results);*/



}

void printResults(LLR * results) {

  // Need to get a value in binary first

#if MANUAL_VECTOR_WIDTH == 1
  printf("Vector␣0:␣");
  for(int i=0; i<N_VARIABLE_NODES; i++) {

    if(results[i] > 0)
      printf("0");
    else
      printf("1");
  }
  printf("\n");
#else
  for(int j=0; j<MANUAL_VECTOR_WIDTH; j++) {

    printf("Vector␣%d:␣",j);
    for(int i=0; i<N_VARIABLE_NODES; i++) {


      if(results[i].s[j] > 0)
        printf("0");
      else
        printf("1");
    }
    printf("\n");

  }
#endif
}

void setFileNames() {
  char temp[MAX_LINE];
  strncpy(temp,FILENAME,MAX_LINE);
  _snprintf(INPUT_FILENAME,MAX_LINE,"E:\\Andrew\\Projects\\%s%d.txt",temp,LDPC_SIZE);

  strncpy(temp,OUTPUT_FILE,MAX_LINE);
  _snprintf(OUTPUT_FILENAME,MAX_LINE,"E:\Andrew\Projects\IrregularCodewordAttempt\%s%d.txt
      ↪ ",temp,LDPC_SIZE);

#ifdef ALTERA
  _snprintf(KERNEL_FILENAME,MAX_LINE,"E:\\Andrew\\Projects\\IrregularCodewordAttempt\\
      ↪ kernels%d.aocx",LDPC_SIZE);
#else
  _snprintf(KERNEL_FILENAME,MAX_LINE,"E:\\Andrew\\Projects\\IrregularCodewordAttempt\\
      ↪ kernels%dtrygmcommunicationgpu.cl",LDPC_SIZE);
#endif
}

void printResultsToFile(double overall, double variable, double check, double misc) {
  // Now let's open the file
  FILE * results = fopen(OUTPUT_FILENAME,"w"); CheckPointer(results);

  // Now let's write data to the file
  fprintf(results,"%f␣%f␣%f␣%f\n",overall,variable,check,misc);

  // Now we close the file
  fclose(results);

}





// Check if all of the check nodes are satisfied
bool checkIfSatisfied(LLR * parity)
{
#if MANUAL_VECTOR_WIDTH == 1
  // All we have to do use iterate over all and see if there is anything non zero
  for(int i=0; i<N_CHECK_NODES; i++) {
    if(parity[i]){
      printf("NOT␣SATISFIED\n");
      return false;
    }
  }
  printf("SATISFIED!\n");
  return true;
#else
  for(int j=0; j<MANUAL_VECTOR_WIDTH; j++){
    for(int i=0; i<N_CHECK_NODES; i++) {
      if(parity[i].s[j]){
        printf("NOT␣SATISFIED\n");
      }
    }
    printf("SATISFIED!\n");
  }

  return true;
#endif

}

#pragma region Codeword Operations
// Get the codeword in BPSK
float* getCodeword(int size)
{
  float *codeword = (float*)malloc(sizeof(float)*size);
  int * input = (int*) malloc(sizeof(int)*size);
  for(int i=0; i<size; i++)
    input[i] = 0;

  //input[0] = 1;

  // Convert it to BPSK
  // 0 -> 1, 1 -> -1
  // Map binary 0 to 1
  // May binary 1 to -1
  for(int i=0; i<size; i++)
  {
    if(input[i])
      codeword[i] = -1;
    else
      codeword[i] = 1;
  }

  return codeword;
}


// Introduce gaussian noise to the input codeword based on the given SNR
void introduceNoise(float* codeword, int SNR, int n_vars)
{
  double sigma = pow(10,(double)-SNR/20);

  for(int i=0; i<n_vars; i++)
  {
    codeword[i] += generate_random_gaussian()*sigma;
    //printf("Codeword[%d] = %f\n",i,codeword[i]);
  }
}

void printCodeword(const int * codeword, const int n_vars) {
  for(int i=0; i<n_vars; i++) {
    printf("codeword[%d]␣=␣%d\n",i,codeword[i]);
  }

}

/* This function will take in the given float version of the codeword
and convert it to an integer representation LLR.  Please note that
it converts the LLR into a 2's complement number first
*/
LLR_BASE_TYPE * convertToLLR(const float *codeword, const int n_vars) {
  LLR_BASE_TYPE * output = (LLR_BASE_TYPE*) malloc(sizeof(LLR_BASE_TYPE)*n_vars);

  // Iterate over the entire codeword
  for(int i=0; i<n_vars; i++) {
    float word = abs(codeword[i]);
    output[i] = 0;


    // Now let's iterate over the possible bins and see where it fits
    float division_point = ((float)MAX_LLR)/SATURATION_POINT ;
    //printf("division_point = %f\n",division_point);
    for(int j=1; j<=(MAX_LLR); j++) {

      if(word <  j / division_point) {
        output[i] = j;

        break;
      }

    }
    //printf("codeword[%d] = %f output[%d] = %u     sign = %d\n",i,codeword[i],i,output[i
        ↪ ],(output[i]&SIGN_BIT)>>15);

    // Once we have the positioning we just need to make sure it's still within the valid
        ↪ LLR
    if(word >=  MAX_LLR / division_point) {
      output[i] = MAX_LLR;

    }

    // Now set the sign bit
    if(codeword[i] < 0) {
      output[i] *= -1;

    }

  }
  return output;
}
#pragma endregion


// This function will read the parity check matrix A-list file and get the following
    ↪ information:
// - number of variable nodes, number of check nodes
// - an array of variablenode structures
// - an array of checknode structures
// PLEASE NOTE ALL VARIABLES PASSED INTO THIS FUNCTION ARE PASSED BY REFERENCE AND WILL BE
    ↪ MODIFIED
// Please pass this function empty variables to store the result
void readParityFile(VariableNode *& v_nodes, CheckNode *& c_nodes,int &n_vars,int &
    ↪ n_checks)
{
  char line[MAX_LINE];

  // Open the file
  FILE * parity_matrix = fopen(INPUT_FILENAME,"r");
  CheckPointer(parity_matrix);

  // Now let's start reading
  // The first two values in the text file will be the number of variable nodes and the
      ↪ number of check nodes
  if(fscanf(parity_matrix,"%d␣%d",&n_vars,&n_checks)!=2)
    ExitOnErrorWMsg("Error␣with␣parity_matrix␣A-list␣file␣format,");

  // Now that we know the number of variable nodes and checks we can allocate the amounts
  v_nodes = (VariableNode*)malloc(sizeof(VariableNode)*n_vars);
  c_nodes = (CheckNode*)malloc(sizeof(CheckNode)*n_checks);

  // Throw away the next line.  This line tells us the approximate number of connections
      ↪ for each node, but we want the more accurate ones
  int temp;
  if(fscanf(parity_matrix,"%d␣%d",&temp,&temp)!=2)
    ExitOnErrorWMsg("Missing␣second␣line␣in␣A-list␣file.");

  // Now let's get to the good stuff
  // According to the A-list format the next line of text will be the list of check nodes
      ↪ for each variable node in order
  // The next line will be a list of check nodes for each variable node
  // Therefore we need to iterate n_vars times to get all of the connections
  for(int i=0; i<n_vars; i++)
  {
    int num_per_variable;
    if(fscanf(parity_matrix,"%d",&num_per_variable)!=1)
      ExitOnErrorWMsg("Error␣with␣parity_matrix␣A-list␣file␣format.");

    // Store the number of check nodes for each variable node and the number of inputs
    v_nodes[i].n_checknodes = num_per_variable;
    v_nodes[i].n_inputs = num_per_variable+1; // Add one here as there is a spot for the
        ↪ initial channel measurement

    // Let's also store the index
    v_nodes[i].index = i;
  }

  // The next line will be a list of variable nodes for each check node
  // Therefore we need to iterate n_checks times to get all of the connections
  for(int i=0; i<n_checks; i++)
  {
    int num_per_check;
    if(fscanf(parity_matrix,"%d",&num_per_check)!=1)
      ExitOnErrorWMsg("Error␣with␣parity_matrix␣A-list␣file␣format.");

    // Store the number of inputs to the check node
    c_nodes[i].n_inputs = num_per_check; // Note there is not an extra here as there is no
        ↪  channel measurement

    // Also store the index
    c_nodes[i].index = i;

    // Set the satisifed bit to false
    //c_nodes[i].satisfied = false;
  }

  // Next we get the indexes for variable nodes (edges)
  // This is the most important section where we determine which nodes get connected to
      ↪ each other
  // First up are the variable node connections.  So we need to iterate over all the
      ↪ variable nodes
  for(int i=0; i<n_vars; i++)
  {
    int position;

    // Now we can use the number of checknodes we just received from the previous lines
    // We need to store which checknodes this variable node is connected to
    int j;
    for(j=0; j<v_nodes[i].n_checknodes; j++)
    {
      if(fscanf(parity_matrix,"%d",&position)!=1)
        ExitOnErrorWMsg("Error␣with␣parity_matrix␣A-list␣file␣format.");

      // Now store the position
      v_nodes[i].checknode_indexes[j] = position-1; // Note we need to subtract one here
          ↪ to start the position at 0

    }
    // Get the rest of it up to the max
    for(j = v_nodes[i].n_checknodes; j < N_VARIABLE_INPUTS; j++) {
      // So for the rest of the connections, we actually want to assign one of the garbage
          ↪  locations
      int randomCheck = rand() % n_checks;

      // Now we have a random check node
      v_nodes[i].checknode_indexes[j] = randomCheck;


    }

  }


  // Same thing, but now for the check nodes.
  // Next we get the indexes for the check nodes (edges)
  for(int i=0; i<n_checks; i++)
  {
    int position;
    int j;
    for(j=0; j<c_nodes[i].n_inputs; j++)
    {
      if(fscanf(parity_matrix,"%d",&position)!=1)
        ExitOnErrorWMsg("Error␣with␣parity_matrix␣A-list␣file␣format.");

      // Now store the position
      c_nodes[i].variablenode_indexes[j] = position-1; // Note we need to subtract one
          ↪ here to start the position at 0
    }

    // Get the rest of it up to the max
    for(j = c_nodes[i].n_inputs; j < N_CHECK_INPUTS; j++) {
      // So for the rest of the connections, we actually want to assign one of the garbage
          ↪  locations
      int randomCheck = rand() % n_vars;

      // Now we have a random check node
      c_nodes[i].variablenode_indexes[j] = randomCheck;


    }
  }

  // That concludes the reading of the parity matrix file
  // All data is now stored within the v_nodes and c_nodes arrays
  fclose(parity_matrix);
  printf("Parity␣matrix␣file␣%s␣read␣successfully\n",INPUT_FILENAME);
  return;
}


//Initialize the nodes for the first run
// This is where we'll need to load up that there are 4 of the vector chars
void initializeNodes(VariableNode *& v_nodes, CheckNode *& c_nodes, const LLR * codeword,
    ↪ const int n_vars, const int n_checks)
{

  // First we load up the channel measurement into all of the variable nodes
  // Iterate over each variable node
  for(int i=0; i<n_vars; i++)
  {
    short channel_index = v_nodes[i].n_inputs-1;
    // For each of the variable nodes, we want to only load up the very last spot that
        ↪ deals with the channel input
    // All of the other inputs should be 0.  This will allows us to start the execution
        ↪ with the variable nodes instead of the check nodes


    for(int j=0; j<v_nodes[i].n_inputs; j++)
    {

      // We need to initialize all 4 of the chars in the vector
      // Note that we use codeword.x(y) etc however they are all equal to begin but it is
          ↪ possible for these to be different
#if MANUAL_VECTOR_WIDTH == 1
      v_nodes[i].inputs[j] = codeword[i];
#else
      for(int t=0; t<MANUAL_VECTOR_WIDTH; t++) {
        v_nodes[i].inputs[j].s[t] = codeword[i].s[t];
      }
#endif
    }
  }


  // Load up the inputs for the check nodes for the first iteration
  // This means we are now passing the specific channel measurements for each connection
  for(int i=0; i<n_checks; i++)
  {
    // For each one of the inputs in this check node, we need to load up the information
        ↪ from the variable node
    for(int j=0; j<c_nodes[i].n_inputs; j++)
    {
      int variable_index = c_nodes[i].variablenode_indexes[j];
      int index = v_nodes[variable_index].n_inputs-1; // All of the spots hold the same
          ↪ value currently

      // We have to convert this value into sign and mag
      LLR sign_and_mag_ver = v_nodes[variable_index].inputs[index];

      // Now we need to do the four comparisons individually
#if MANUAL_VECTOR_WIDTH == 1
      if(sign_and_mag_ver < 0) {
        sign_and_mag_ver *= -1;
        sign_and_mag_ver |= SIGN_BIT;
      }
#else
      for(int t=0; t<MANUAL_VECTOR_WIDTH; t++) {
        if(sign_and_mag_ver.s[t] < 0) {
        sign_and_mag_ver.s[t] *= -1;
        sign_and_mag_ver.s[t] |= SIGN_BIT;
        }
      }
#endif



      // Let's store the value in our input
      c_nodes[i].inputs[j] = sign_and_mag_ver;
    }
  }
}

#pragma region Error Checking Functions
// This function will check the given input pointer for validaty and fail exit the program
    ↪  if necessary
void CheckPointerFunc(void * pointer,char const* const func, const int line)
{
  if(pointer == NULL)
  {
    fprintf(stderr,"Error␣with␣pointer␣in␣%s␣line␣%d\n",func,line);
    exit(SYSTEM_ERROR);
  }
}

void ExitOnErrorFunc(char const* msg, char const* func, const int line)
{
  fprintf(stderr,"%s␣%s␣line␣%d\n",msg,func,line);
  exit(SYSTEM_ERROR);
}
#pragma endregion

#pragma region Random Number Code
double generate_random_gaussian()
{
#define PI 3.14159265358979323846
  double U1 = generate_random_number();//(double)rand()/((doule)RAND_MAX+1);
  double U2 = generate_random_number();//(double)rand()/((double)RAND_MAX+1);

  double Z0 = sqrt(-2*log(U1))*cos(2*PI*U2);
  return Z0;
}



int * generate_VNtoCN_index_table(VariableNode * v_nodes,CheckNode * c_nodes) {
  int * table = (int*)_aligned_malloc(sizeof(int)*N_VARIABLE_NODES*N_VARIABLE_INPUTS,
      ↪ AOCL_ALIGNMENT);
  if(table == NULL)
    printf("err␣=␣%d\n",errno);

  // Iterate over the list of variable nodes
  for(int i=0; i<N_VARIABLE_NODES; i++) {

    int j;
    // For each variable node we need to find the appropriate connections for each input
    for(j=0; j<v_nodes[i].n_checknodes; j++) {
      int cnIndex = v_nodes[i].checknode_indexes[j];


      int cnInputIndex;
      // Now let's look inside that check node to find our appropriate location
      for(int n=0; n<N_CHECK_INPUTS; n++) {
        int tempIndex = c_nodes[cnIndex].variablenode_indexes[n];

        // If we found our variable node, make the table
        if(tempIndex == i) {
          cnInputIndex = n;
          break;
        }
      }
      // Now we can create our table entry
      table[i*N_VARIABLE_INPUTS + j] = (cnIndex*(N_CHECK_INPUTS+1)) + cnInputIndex;




    }

    for(j = v_nodes[i].n_checknodes; j < N_VARIABLE_INPUTS; j++) {
      // For the rest of them we want to assign the random check node at the end
      int cnIndex = v_nodes[i].checknode_indexes[j];

      // Alright so we just need to assign the last position
      table[i*N_VARIABLE_INPUTS + j] = (cnIndex * (N_CHECK_INPUTS+1)) + N_CHECK_INPUTS;
    }







  }
  return table;
}

int * generate_CNtoVN_index_table(VariableNode * v_nodes,CheckNode * c_nodes) {
  int * table = (int*)_aligned_malloc(sizeof(int)*N_CHECK_NODES*N_CHECK_INPUTS,
     ↪ AOCL_ALIGNMENT);
  if(table == NULL)
    printf("err␣=␣%d\n",errno);
  // Iterate over the list of variable nodes
  for(int i=0; i<N_CHECK_NODES; i++) {

    int j;
    // For each variable node we need to find the appropriate connections for each input
    for(j=0; j<c_nodes[i].n_inputs; j++) {
      int vnIndex = c_nodes[i].variablenode_indexes[j];

      int vnInputIndex = -100;
      // Now let's look inside that check node to find our appropriate location
      for(int n=0; n<N_VARIABLE_INPUTS; n++) {
        int tempIndex = v_nodes[vnIndex].checknode_indexes[n];

        // If we found our variable node, make the table
        if(tempIndex == i) {
          vnInputIndex = n;
          break;
        }
      }

      int foundIndex = (vnIndex * (N_VARIABLE_INPUTS+1)) + vnInputIndex;

      // Populate the table
      table[i*N_CHECK_INPUTS + j] = foundIndex;

    }

    // Alright for the rest all we need to do is give it the garbage address
    for(j = c_nodes[i].n_inputs; j < N_CHECK_INPUTS; j++) {
      int vnIndex = c_nodes[i].variablenode_indexes[j];


      table[i*N_CHECK_INPUTS + j] = (vnIndex * (N_VARIABLE_INPUTS+1)) + N_VARIABLE_INPUTS;



    }


  }

  printf("one␣in␣question␣=␣%d\n",table[1*N_CHECK_INPUTS + 15]);

  return table;
}

double generate_random_number()
{
#define C1 4294967294
#define C2 4294967288
#define C3 4294967280
#define MAX 4294967295

  srand(45);
  static unsigned int s1 = rand();
  static unsigned int s2 = rand();
  static unsigned int s3 = rand();

  // Here's the first part
  // Now let's crank the tausworthe
  unsigned int xor1 = s1 << 13;
  // The first set of and/xor
  unsigned int left_temp = xor1 ^ s1;
  unsigned int right_temp = C1 & s1;

  // Shifts
  left_temp = left_temp >> 19;
  right_temp = right_temp << 12;

  s1 = left_temp ^ right_temp;

  // Second part
  xor1 = s2<<2;
  left_temp = xor1 ^ s2;
  right_temp = C2 & s2;

  left_temp = left_temp >> 25;
  right_temp = right_temp << 4;

  s2 = left_temp ^ right_temp;

  // Third part
  xor1 = s3 << 3;
  left_temp = xor1 ^ s3;
  right_temp = C3 & s3;

  left_temp = xor1 ^ s3;
  right_temp = C3 & s3;

  left_temp = left_temp >> 11;
  right_temp = right_temp << 17;

  s3 = left_temp ^ right_temp;

  // Now the return
  unsigned int output = s1 ^ s2 ^ s3;

  // Now just convert it into a double
  double last_value = (double)output /MAX;

  //cout<<"last value" << last_value<<endl;
  return last_value;

}
#pragma endregion

void defineGlobalDefinitions() {

  vector_max_llr = (LLR*) _aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  vector_min_neg_llr = (LLR*)_aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  vector_sign_bit = (LLR*)_aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  allones = (LLR*)_aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  ones = (LLR*)_aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  zeros = (LLR*)_aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  vector_too_neg_llr = (LLR*)_aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  total_vector_max_llr = (TOTAL_TYPE*)_aligned_malloc(sizeof(TOTAL_TYPE),AOCL_ALIGNMENT);
  total_vector_min_neg_llr = (TOTAL_TYPE*)_aligned_malloc(sizeof(TOTAL_TYPE),
      ↪ AOCL_ALIGNMENT);



#if MANUAL_VECTOR_WIDTH==1
  vector_max_llr = MAX_LLR;
  vector_min_neg_llr = MAX_NEG_LLR;
  vector_sign_bit = SIGN_BIT;
  allones = (LLR)(0xFF);
  ones = (LLR)(1);
  zeros = (LLR)(0);
  vector_too_neg_llr = MAX_NEG_LLR - ones;
  total_vector_max_llr = MAX_LLR;
  total_vector_min_neg_llr = MAX_NEG_LLR;
#else
  for(int i=0; i<MANUAL_VECTOR_WIDTH; i++)
  {
    (*vector_max_llr).s[i] = MAX_LLR;
    (*vector_min_neg_llr).s[i] = MAX_NEG_LLR;
    (*vector_sign_bit).s[i] = SIGN_BIT;
    (*allones).s[i] = (0xFF);
    (*ones).s[i] = (1);
    (*zeros).s[i] = (0);
    (*vector_too_neg_llr).s[i] = MAX_NEG_LLR - 1;
    (*total_vector_max_llr).s[i] = MAX_LLR;
    (*total_vector_min_neg_llr).s[i] = MAX_NEG_LLR;
  }
#endif
}

void parseProgramInputs(int n_args,char* args[]) {
  for(int i=0; i<n_args-1; i++) {
    if(strncmp(args[i],"-nVars",MAX_LINE)==0) {
      // If we found the n_vars just set the variable
      int new_value = atoi(args[i+1]);
      if(new_value <= 0) {
        ExitOnErrorWMsg("Not a valid LDPC size input\n");
      }
      N_VARIABLE_NODES = atoi(args[i+1]);
      LDPC_SIZE = N_VARIABLE_NODES;
    }
    else if (strncmp(args[i], "-nCheck",MAX_LINE)==0) {
      int new_value = atoi(args[i+1]);
      if(new_value <=0 ) {
        ExitOnErrorWMsg("Not a valid LDPC size input\n");
      }
      N_CHECK_NODES = atoi(args[i+1]);
    }
  }

}
```

Appendix L

Irregular Design F Host Code (C)

```
// This is the start of the Channel LDPC implementation
#include <iostream>
#include <fstream>
#include <ctime>
#include <math.h>
#include <malloc.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "E:\Andrew\Projects\common\ax_common.h"
#include <Windows.h>



//#define ALTERA

#define MANUAL_VECTOR_WIDTH 16

int N_VARIABLE_NODES = 1120;
int N_CHECK_NODES =280;
#define MAX_VARIABLE_EDGES 5
#define MAX_CHECK_EDGES 16


#define PROJECT_TITLE "Irregular Global Memory LDPC"
#define PROJECT_DESCRIPTION "This host code supports the multi kernel implementation of
     ↪ the LDPC decoder using the char16 data type. Within the char16 data type, each of
     ↪ the chars will contain a different codeword to decode"


// These options apply to the code being used as well as the number of iterations to do
int LDPC_SIZE = N_VARIABLE_NODES; // Note this is overwritten when a value is passed in
     ↪ via command line

#define FILENAME "matrix_"
#define OUTPUT_FILE "results_"
#define N_ITERATIONS 32

#define N_VARIABLE_INPUTS (MAX_VARIABLE_EDGES)
#define N_CHECK_INPUTS (MAX_CHECK_EDGES)


#define N_VARIABLE_INPUTS_W_CHANNEL (N_VARIABLE_INPUTS+1)


#define MAX_LINE (1024*3)

static char * INPUT_FILENAME = (char*) malloc(sizeof(char)*MAX_LINE);
static char * OUTPUT_FILENAME = (char*) malloc(sizeof(char)*MAX_LINE);

#if MANUAL_VECTOR_WIDTH==2
#define LLR_TYPE cl_char2
#define TOTAL_TYPE cl_short2
#define CONVERT_TO_SHORT(a) convert_short2(a)
#define CONVERT_TO_CHAR(a) convert_char2(a)
#elif MANUAL_VECTOR_WIDTH==4
#define LLR_TYPE cl_char4
#define TOTAL_TYPE cl_short4
#define CONVERT_TO_SHORT(a) convert_short4(a)
#define CONVERT_TO_CHAR(a) convert_char4(a)
#elif MANUAL_VECTOR_WIDTH==8
#define LLR_TYPE cl_char8
```

```
#define TOTAL_TYPE cl_short8
#define CONVERT_TO_SHORT(a) convert_short8(a)
#define CONVERT_TO_CHAR(a) convert_char8(a)
#elif MANUAL_VECTOR_WIDTH==16
#define LLR_TYPE cl_char16
#define TOTAL_TYPE cl_short16
#define CONVERT_TO_SHORT(a) convert_short16(a)
#define CONVERT_TO_CHAR(a) convert_char16(a)
#else
#define LLR_TYPE cl_char
#define TOTAL_TYPE cl_short
#define CONVERT_TO_SHORT(a) convert_short(a)
#define CONVERT_TO_CHAR(a) convert_char(a)
#endif


// Integer options
#define LLR_BASE_TYPE char
#define N_BITS_LLR (sizeof(LLR_BASE_TYPE)*8) // 8 bits per byte
#define MAX_LLR ((1 << (N_BITS_LLR-1))-1) // 127
#define SIGN_BIT (MAX_LLR+1)
#define MAX_NEG_LLR (0-MAX_LLR)
typedef LLR_TYPE LLR;

LLR *vector_max_llr;
LLR *vector_min_neg_llr;
LLR *vector_n_bits_llr;
LLR *vector_sign_bit;
LLR *vector_too_neg_llr;
LLR *allones;
LLR *ones;
LLR *zeros;
TOTAL_TYPE *total_vector_max_llr;
TOTAL_TYPE *total_vector_min_neg_llr;


#define BINS_PER_ONE_STEP 4
#define SATURATION_POINT int(MAX_LLR/BINS_PER_ONE_STEP) //31



//#define PRINT_DEBUG
//#define QUICK_DEBUG

#ifdef PRINT_DEBUG
#undef QUICK_DEBUG
#endif

#define DEBUG_FILENAME "output_debug_integer.csv"
#define CHECKNODE_DEBUG_FILENAME "checknode_debug.csv"
#define VARIABLENODE_DEBUG_FILENAME "variablenode_debug.csv"




#pragma region OpenCL Specific Declarations
static char * KERNEL_FILENAME  = (char*) malloc(sizeof(char)*MAX_LINE);
static char const* COMPILER_OPTIONS = "-w";
#pragma endregion

#define DEBUG 1
#define AOCL_ALIGNMENT 64





#pragma region Error Checking Macros
#define SYSTEM_ERROR -1
#define CheckPointer(pointer) CheckPointerFunc(pointer,__FUNCTION__, __LINE__)
#define ExitOnErrorWMsg(msg) ExitOnErrorFunc(msg,__FUNCTION__,__LINE__)
#define ExitOnError() ExitOnErrorWMsg("Error! Please refer to")
#pragma endregion

#pragma region Structure Defintions
// This structure holds all data required for the variable nodes.
// Note that the last input will be the channel input for each variable node
// the n_inputs will always be n_checknodes+1
typedef struct VariableNodes
{
  LLR inputs[MAX_VARIABLE_EDGES+1];
  cl_short checknode_indexes[MAX_VARIABLE_EDGES];
  cl_short index;
  cl_short n_inputs;
  cl_short n_checknodes;
}VariableNode;

// This structure holds all data required for the check nodes
typedef struct CheckNodes
{
  LLR inputs[MAX_CHECK_EDGES];
  cl_short variablenode_indexes[MAX_CHECK_EDGES];
  cl_short index;
  cl_short n_inputs;
}CheckNode;
#pragma endregion

#pragma region Function Declaractions
static void CheckPointerFunc(void * pointer,char const* const func,const int line);
static void ExitOnErrorFunc(char const* msg, char const* func, const int line);
static void readParityFile(VariableNode*&, CheckNode*&, int&, int&);
static bool checkIfSatisfied(LLR * parity);
static void updateParity(CheckNode *& c_nodes, const int n_checks);
static LLR * convertCodewordToBinary(VariableNode*&,const int n_vars);
static float* getCodeword(int size);
static LLR_BASE_TYPE * convertToLLR(const float * codeword, const int n_vars);
static void printCodeword(const int * codeword, const int n_vars);
static void printResults(LLR * results);
static void printResultsToFile(double overall, double variable, double check, double misc)
    ↪ ;
static void setFileNames();

double generate_random_gaussian();
double generate_random_number();
void introduceNoise(float*,int,int);

int * generate_VNtoCN_index_table(VariableNode* ,CheckNode* );
int * generate_CNtoVN_index_table(VariableNode* ,CheckNode* );



static void startOpenCL(cl_context,cl_device_id device, cl_program program);
static void startSimulationOpenCL(cl_context, cl_device_id,cl_program,VariableNode*&,
    ↪ CheckNode*&,const int, const int);
static void ldpcDecodeOpenCL(cl_context, cl_command_queue,cl_command_queue,cl_kernel,
    ↪ cl_kernel,VariableNode*&,CheckNode*&,const int, const int,float*);
void initializeNodes(VariableNode*&, CheckNode*&,const LLR*,const int, const int);
void defineGlobalDefinitions();

void parseProgramInputs(int,char*[]);
#pragma endregion






int main(int argc, char * argv[])
{
  printf("%s\n",PROJECT_TITLE);
  printf("%s\n",PROJECT_DESCRIPTION);

  parseProgramInputs(argc,argv);

  // Let's do a quick check on the number
  if(LDPC_SIZE <=0) {
    printf("Codelength␣(number␣of␣variable␣nodes)␣is␣not␣valid\n");
    ExitOnError();
  }

  // Set the file names for the input and output
  setFileNames();


  // Set up the program
  cl_context context;
  cl_device_id device;
  cl_program program;
  cl_device_type type = CL_DEVICE_TYPE_ALL;

  printf("\nOpenCL␣API\n\n");
  printf("Max␣LLR␣=␣%d\n",MAX_LLR);


  // Let's do this
  AxCreateContext(type,AX_PLATFORM_STRING,&context,&device);
#ifdef ALTERA
  AxBuildProgramAltera(&program,device,context,KERNEL_FILENAME,COMPILER_OPTIONS);
#else
  AxBuildProgram(&program,device,context,KERNEL_FILENAME,COMPILER_OPTIONS);
#endif

  size_t max_value;

  AxCheckError(clGetDeviceInfo(device,CL_DEVICE_MAX_WORK_GROUP_SIZE,sizeof(max_value),&
      ↪ max_value,NULL));

  printf("Buffer␣=␣%d\n",max_value);

  // Now let's start the rest of the program
  startOpenCL(context,device,program);

  clReleaseProgram(program);
  clReleaseContext(context);
```

```
}


// Start the main chunk of the host code
void startOpenCL(cl_context context, cl_device_id device, cl_program program)
{

  defineGlobalDefinitions();
  // Declare the variables
  // Declare an array to hold the checknodes
  CheckNode * c_nodes=NULL;

  // Declare an array to hold the variablenodes
  VariableNode * v_nodes=NULL;

  // The number of check nodes and variable nodes
  int n_checks=0;
  int n_vars=0;

  // Let's read the parity file to get the nodes
  readParityFile(v_nodes,c_nodes,n_vars,n_checks);
  printf("Testing␣file␣for␣accuracy....");
  for(int i=0; i<n_vars; i++) {
    // Check that every fanout isn't greater than the max
    if(v_nodes[i].n_inputs > N_VARIABLE_INPUTS_W_CHANNEL)
      ExitOnErrorWMsg("Software␣defined␣N_VARIABLE_INPUTS␣too␣small");

  }
  for(int i=0; i<n_checks; i++) {

    if(c_nodes[i].n_inputs > N_CHECK_INPUTS)
      ExitOnErrorWMsg("Software␣defined␣N_CHECK_INPUTS␣too␣small");
  }
  printf("Successful\n");

  // Let's just do a few quick checks to make sure that the read worked properly
  // We should check the c_nodes and v_nodes pointers
  // We also need to check that the number of variable nodes and check nodes are valid
  CheckPointer(c_nodes);CheckPointer(v_nodes);
  if(n_vars <= 0 || n_checks <= 0)
    ExitOnErrorWMsg("Number␣of␣variables␣and␣check␣nodes␣not␣valid.");

  printf("\n**********************************************\n");
  printf("Variable␣Nodes␣%d,␣Check␣Nodes␣%d\n",n_vars,n_checks);
  printf("%d␣Log␣Likelihood␣Ratio␣Bits\n",N_BITS_LLR);
  printf("Max␣Variable␣Edges␣%d,␣Max␣Check␣Edges␣%d\n",MAX_VARIABLE_EDGES,MAX_CHECK_EDGES)
      ↪ ;

  // Now we are ready to start our simulation
  startSimulationOpenCL(context,device,program,v_nodes,c_nodes,n_vars,n_checks);

}

void startSimulationOpenCL(cl_context context, cl_device_id device,cl_program program,
    ↪ VariableNode*& v_nodes,CheckNode*& c_nodes, const int n_vars, const int n_checks)
{

  // Firstly, we need to get our codeword to decode
  float * codeword = getCodeword(n_vars);


  cl_int err;
  // Let's create our openCL stuff

  /*cl_command_queue testQueue = clCreateCommandQueue(context,device,
      ↪ CL_QUEUE_PROFILING_ENABLE,&err);AxCheckError(err);

  cl_kernel kernel = clCreateKernel(program, "hello",&err);AxCheckError(err);
  size_t global = 10;
  size_t local = 10;

  char string[MAX_LINE*4];
  int test[2];

  cl_mem string_d = clCreateBuffer(context,CL_MEM_COPY_HOST_PTR | CL_MEM_READ_WRITE,
      ↪ MAX_LINE*4,string,&err);
  AxCheckError(err);
  cl_mem test_d = clCreateBuffer(context,CL_MEM_READ_WRITE |CL_MEM_COPY_HOST_PTR, 2, test,
      ↪  &err);
  AxCheckError(err);

  AxCheckError(clSetKernelArg(kernel,0,sizeof(string_d),&string_d));
  AxCheckError(clSetKernelArg(kernel,1,sizeof(test_d),&test_d));

  AxCheckError(clEnqueueNDRangeKernel(testQueue,kernel,1,NULL,&global,&local,0,NULL,NULL))
      ↪ ;

  AxCheckError(clEnqueueReadBuffer(testQueue,test_d,CL_TRUE,0,2,&test,0,NULL,NULL));
  AxCheckError(clEnqueueReadBuffer(testQueue,string_d,CL_TRUE,0,MAX_LINE*4,&string,0,NULL,
      ↪ NULL));*/
  //printf("String %s\n",string);
  // Now we need to create our command queues
  // We will create one command queue for each kernel
  cl_command_queue VariableQueue = clCreateCommandQueue(context,device,
       ↪ CL_QUEUE_PROFILING_ENABLE,&err);AxCheckError(err);
  cl_command_queue CheckQueue = clCreateCommandQueue(context,device,
       ↪ CL_QUEUE_PROFILING_ENABLE,&err);AxCheckError(err);



  // Now we need to create the kernels themselves.
  cl_kernel VariableKernel = clCreateKernel(program,"VariableNode",&err);AxCheckError(err)
       ↪ ;
  cl_kernel CheckKernel = clCreateKernel(program,"CheckNode",&err);AxCheckError(err);

  // Now let's simply start the LDPC
  ldpcDecodeOpenCL(context, VariableQueue,CheckQueue,VariableKernel,CheckKernel,v_nodes,
       ↪ c_nodes,n_vars,n_checks,codeword);
}

void copyCodewordFloat(float *& source, float *& dest, int n_vars) {

  for(int i=0; i<n_vars; i++) {
    dest[i] = source[i];
  }
}

void copyCodewordLLRBase(LLR_BASE_TYPE *& source, LLR_BASE_TYPE *& dest, int n_vars) {
  for(int i=0; i<n_vars; i++) {
    dest[i] = source[i];
  }
}

void ldpcDecodeOpenCL(cl_context       context,
             cl_command_queue     VariableQueue,
             cl_command_queue     CheckQueue,
             cl_kernel        VariableKernel,
             cl_kernel        CheckKernel,
             VariableNode*&     v_nodes,
             CheckNode*&      c_nodes,
             const int        n_vars,
             const int        n_checks,
             float*         codeword)
{
  cl_int err;


  // Note this is where I would initiate an LLR for fixed integer
  size_t channelSize = sizeof(LLR) * N_VARIABLE_NODES*2;
  LLR * codeword_LLR = (LLR*) _aligned_malloc(channelSize,AOCL_ALIGNMENT);


  float * codeword0 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword1 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword2 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword3 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword4 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword5 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword6 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword7 = (float*) malloc(sizeof(float) * n_vars);

  float * codeword8 = (float*) malloc(sizeof(float) * n_vars);
  float * codeword9 = (float*) malloc(sizeof(float) * n_vars);
  float * codeworda = (float*) malloc(sizeof(float) * n_vars);
  float * codewordb = (float*) malloc(sizeof(float) * n_vars);
  float * codewordc = (float*) malloc(sizeof(float) * n_vars);
  float * codewordd = (float*) malloc(sizeof(float) * n_vars);
  float * codeworde = (float*) malloc(sizeof(float) * n_vars);
  float * codewordf = (float*) malloc(sizeof(float) * n_vars);


  copyCodewordFloat(codeword,codeword0,n_vars);
  copyCodewordFloat(codeword,codeword1,n_vars);
  copyCodewordFloat(codeword,codeword2,n_vars);
  copyCodewordFloat(codeword,codeword3,n_vars);
  copyCodewordFloat(codeword,codeword4,n_vars);
  copyCodewordFloat(codeword,codeword5,n_vars);
  copyCodewordFloat(codeword,codeword6,n_vars);
  copyCodewordFloat(codeword,codeword7,n_vars);

  copyCodewordFloat(codeword,codeword8,n_vars);
  copyCodewordFloat(codeword,codeword9,n_vars);
  copyCodewordFloat(codeword,codeworda,n_vars);
  copyCodewordFloat(codeword,codewordb,n_vars);
  copyCodewordFloat(codeword,codewordc,n_vars);
  copyCodewordFloat(codeword,codewordd,n_vars);
  copyCodewordFloat(codeword,codeworde,n_vars);
  copyCodewordFloat(codeword,codewordf,n_vars);
  // Now we need to introduce some noise into our codeword
  // Note that this function changes our codeword input variable
  // But let's add noise individually to each one
  introduceNoise(codeword0,1000,n_vars);
  introduceNoise(codeword1,-20,n_vars);
  introduceNoise(codeword2,-20,n_vars);
  introduceNoise(codeword3,-12,n_vars);
  introduceNoise(codeword4,-10,n_vars);
  introduceNoise(codeword5,-20,n_vars);
  introduceNoise(codeword6,-20,n_vars);
  introduceNoise(codeword7,-12,n_vars);

  introduceNoise(codeword8,-10,n_vars);
  introduceNoise(codeword9,-20,n_vars);
  introduceNoise(codeworda,-20,n_vars);
  introduceNoise(codewordb,-12,n_vars);
  introduceNoise(codewordc,10,n_vars);
  introduceNoise(codewordd,20,n_vars);
  introduceNoise(codeworde,-20,n_vars);
  introduceNoise(codewordf,-12,n_vars);

  /*for(int i=0; i<n_vars; i++) {
    printf("codeword[%d] = %f\n",i,codeword0[i]);
  }*/

  // Now I need to add in the step to conver the number to an integer
  char * llr0 = convertToLLR(codeword0,n_vars);
  char * llr1 = convertToLLR(codeword1,n_vars);
  char * llr2 = convertToLLR(codeword2,n_vars);
  char * llr3 = convertToLLR(codeword3,n_vars);
  char * llr4 = convertToLLR(codeword4,n_vars);
  char * llr5 = convertToLLR(codeword5,n_vars);
  char * llr6 = convertToLLR(codeword6,n_vars);
  char * llr7 = convertToLLR(codeword7,n_vars);

  char * llr8 = convertToLLR(codeword8,n_vars);
  char * llr9 = convertToLLR(codeword9,n_vars);
  char * llra = convertToLLR(codeworda,n_vars);
  char * llrb = convertToLLR(codewordb,n_vars);
  char * llrc = convertToLLR(codewordc,n_vars);
  char * llrd = convertToLLR(codewordd,n_vars);
  char * llre = convertToLLR(codeworde,n_vars);
  char * llrf = convertToLLR(codewordf,n_vars);


  for(int i=0; i<n_vars; i++) {
#if MANUAL_VECTOR_WIDTH == 1
    codeword_LLR[i]= llr0[i];
#else

    for(int j=0; j <MANUAL_VECTOR_WIDTH; j++) {
      codeword_LLR[i].s[j] = llr0[i];
    }


#endif


  }

  int k=0;
  for(int i=n_vars; i<n_vars*2;i++) {

    for(int j=0; j<MANUAL_VECTOR_WIDTH; j++) {
      codeword_LLR[i].s[j] = llr0[k];
    }
    k++;
  }

  /*for(int i=0; i<n_vars; i++) {
#if MANUAL_VECTOR_WIDTH == 1
    printf("codeword[%d] = %d\n",i,codeword_LLR[i]);//.x,codeword_LLR[i].y,codeword_LLR[i
        ↪ ].z,codeword_LLR[i].w);
#else
    printf("codeword[%d] = %d\n",i,codeword_LLR[i].s0);//codeword_LLR[i].y,codeword_LLR[i
        ↪ ].z,codeword_LLR[i].w);
#endif
  }*/

  // Let's start initializing the inputs
  // For the codewords starting at the VN we just need to create a buffer with the
      ↪ measurements
```

```
// For the second version we need to start by doing the first VN portion
//initializeNodes(v_nodes,c_nodes,codeword_LLR,n_vars,n_checks);

// Let's get the connection tables
size_t VNtoCNSize = sizeof(int)*N_VARIABLE_NODES*N_VARIABLE_INPUTS;
size_t CNtoVNSize = sizeof(int)*N_CHECK_NODES*N_CHECK_INPUTS;
printf("Generating␣VntoCN␣lookup␣table....");

int * VNtoCNLookup_h = generate_VNtoCN_index_table(v_nodes,c_nodes);
printf("Successful\n");


printf("Generating␣CntoVN␣lookup␣table....");
int * CNtoVNLookup_h = generate_CNtoVN_index_table(v_nodes,c_nodes);

printf("Successful\n");




cl_mem VNtoCNLookup = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    ↪ VNtoCNSize,VNtoCNLookup_h,&err);AxCheckError(err);
cl_mem CNtoVNLookup = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    ↪ CNtoVNSize,CNtoVNLookup_h,&err);AxCheckError(err);




// Make the constant memory info

cl_mem vector_max_llr_d = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
    ↪ sizeof(LLR),vector_max_llr,&err);AxCheckError(err);
cl_mem vector_sign_bit_d = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
    ↪ sizeof(LLR),vector_sign_bit,&err);AxCheckError(err);
cl_mem allones_d = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,sizeof(
    ↪ LLR),allones,&err);AxCheckError(err);
cl_mem ones_d = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,sizeof(LLR)
    ↪ ,ones,&err);AxCheckError(err);
cl_mem zeros_d = clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,sizeof(LLR
    ↪ ),zeros,&err);AxCheckError(err);
cl_mem total_vector_max_llr_d = clCreateBuffer(context,CL_MEM_READ_ONLY|
    ↪ CL_MEM_COPY_HOST_PTR,sizeof(TOTAL_TYPE),total_vector_max_llr,&err);AxCheckError(
    ↪ err);
cl_mem total_vector_min_neg_llr_d = clCreateBuffer(context,CL_MEM_READ_ONLY|
    ↪ CL_MEM_COPY_HOST_PTR,sizeof(TOTAL_TYPE),total_vector_min_neg_llr,&err);
    ↪ AxCheckError(err);

size_t vnResultSize = sizeof(LLR) * N_VARIABLE_NODES*2;
size_t cnResultSize = sizeof(LLR) * N_CHECK_NODES*2;

LLR * vnResult = (LLR*) _aligned_malloc(vnResultSize,AOCL_ALIGNMENT);
LLR * cnResult = (LLR*) _aligned_malloc(cnResultSize,AOCL_ALIGNMENT);

cl_mem vnResult_d = clCreateBuffer(context,CL_MEM_READ_WRITE,vnResultSize,NULL,&err);
    ↪ AxCheckError(err);
cl_mem cnResult_d = clCreateBuffer(context,CL_MEM_READ_WRITE,cnResultSize,NULL,&err);
    ↪ AxCheckError(err);


cl_mem initialChannelMeasurements = clCreateBuffer(context,CL_MEM_READ_WRITE |
    ↪ CL_MEM_COPY_HOST_PTR, channelSize,codeword_LLR,&err);AxCheckError(err);

// We now need to create the buffers for each node in global memory (for data transfer
    ↪ between the nodes)
size_t vnInputsGMSize = sizeof(LLR)*N_VARIABLE_NODES*(N_VARIABLE_INPUTS+1)*2;
size_t cnInputsGMSize = sizeof(LLR)*N_CHECK_NODES*(N_CHECK_INPUTS+1)*2;

// We need to initialize them to 0 which means we need to create and send that data
LLR * initializationVNBuffer = (LLR*) _aligned_malloc(vnInputsGMSize,AOCL_ALIGNMENT);
LLR * initializationCNBuffer = (LLR*) _aligned_malloc(cnInputsGMSize,AOCL_ALIGNMENT);

CheckPointer(initializationVNBuffer);
CheckPointer(initializationCNBuffer);


printf("Initializing␣VN␣Buffers....");
for(int i=0; i < n_vars*2; i++) {

  // For each one let's look at the inputs
  int j;
  for(j = 0; j< (N_VARIABLE_INPUTS + 1); j++) {
    for(int k=0; k<MANUAL_VECTOR_WIDTH; k++) {
      initializationVNBuffer[i*(N_VARIABLE_INPUTS+1) + j].s[k]= (LLR_BASE_TYPE) 0;
    }
  }
}


printf("Successful\n");
printf("Initializing␣CN␣Buffers....");
for(int i=0; i< n_checks * 2; i++) {
  int c = i % n_checks;
```

```
    int j;
    for(j = 0; j < c_nodes[c].n_inputs; j++) {
      for(int k=0; k<MANUAL_VECTOR_WIDTH; k++) {
        initializationCNBuffer[i*(N_CHECK_INPUTS +1) + j].s[k]= (LLR_BASE_TYPE) 0;
      }
    }

    for(j = c_nodes[c].n_inputs; j < (N_CHECK_INPUTS + 1); j++) {
      for(int k=0; k<MANUAL_VECTOR_WIDTH; k++) {
        initializationCNBuffer[i*(N_CHECK_INPUTS +1) + j].s[k]= (LLR_BASE_TYPE) MAX_LLR;
      }
    }

  }

  printf("Successful\n");

  // Start with the Variable Node buffer
  cl_mem vnInputsBuffer = clCreateBuffer(context,CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR
       ↪ ,vnInputsGMSize,initializationVNBuffer,&err);AxCheckError(err);
  cl_mem cnInputsBuffer = clCreateBuffer(context,CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR,
       ↪  cnInputsGMSize, initializationCNBuffer, &err); AxCheckError(err);

  // Make the sync buffers
  char * VNSync_h = (char*) _aligned_malloc(1,AOCL_ALIGNMENT);
  char * CNSync_h = (char*) _aligned_malloc(1,AOCL_ALIGNMENT);

  (*VNSync_h) = -1;
  (*CNSync_h) = -1;

  cl_mem VNSync = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, 1,
       ↪ VNSync_h,&err);AxCheckError(err);
  cl_mem CNSync = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, 1,
       ↪ CNSync_h,&err);AxCheckError(err);

  /*printf("Host pointers\n");
  printf("VNtoCNLookup %lx\n",VNtoCNLookup_h);
  printf("CNtoVNLookup %lx\n",CNtoVNLookup_h);
  printf("codeword_LLR %lx\n",codeword_LLR);
  printf("initializationVNBuffer %lx\n",initializationVNBuffer);
  printf("initiailzationCNBuffer %lx\n",initializationCNBuffer);

  printf("VNSync %lx\n",VNSync_h);
  printf("CNSync %lx\n",CNSync_h);
  printf("vector_max_llr %lx\n",vector_max_llr);
  printf("vector_sign_bit %lx\n",vector_sign_bit);
  printf("allones %lx\n",allones);
  printf("ones %lx\n",ones);
  printf("total_vector_max_llr %lx\n",total_vector_max_llr);
  printf("total_vector_min_neg_llr %lx\n",total_vector_min_neg_llr);
  printf("VNResult %lx\n",vnResult);
  printf("CNResult %lx\n",cnResult);*/




  // Let's set up the kernels
  // Starting with the Variable Node
  int vnodearg = 0;
  int cnodearg = 0;
  AxCheckError(clSetKernelArg(VariableKernel,vnodearg++,sizeof(VNtoCNLookup),&VNtoCNLookup
       ↪ ));
  AxCheckError(clSetKernelArg(VariableKernel,vnodearg++,sizeof(initialChannelMeasurements)
       ↪ ,&initialChannelMeasurements));
  AxCheckError(clSetKernelArg(VariableKernel,vnodearg++,sizeof(vnInputsBuffer),&
       ↪ vnInputsBuffer));
  AxCheckError(clSetKernelArg(VariableKernel,vnodearg++,sizeof(cnInputsBuffer),&
       ↪ cnInputsBuffer));
  AxCheckError(clSetKernelArg(VariableKernel,vnodearg++,sizeof(vnResult_d),&vnResult_d));
  AxCheckError(clSetKernelArg(VariableKernel,vnodearg++,sizeof(vector_max_llr_d),&
       ↪ vector_max_llr_d));
  AxCheckError(clSetKernelArg(VariableKernel,vnodearg++,sizeof(vector_sign_bit_d),&
       ↪ vector_sign_bit_d));
  AxCheckError(clSetKernelArg(VariableKernel,vnodearg++,sizeof(allones_d),&allones_d));
  AxCheckError(clSetKernelArg(VariableKernel,vnodearg++,sizeof(ones_d),&ones_d));
  AxCheckError(clSetKernelArg(VariableKernel,vnodearg++,sizeof(zeros_d),&zeros_d));
  AxCheckError(clSetKernelArg(VariableKernel,vnodearg++,sizeof(total_vector_max_llr_d),&
       ↪ total_vector_max_llr_d));
  AxCheckError(clSetKernelArg(VariableKernel,vnodearg++,sizeof(total_vector_min_neg_llr_d)
       ↪ ,&total_vector_min_neg_llr_d));

  // Check Node
  AxCheckError(clSetKernelArg(CheckKernel,cnodearg++,sizeof(CNtoVNLookup),&CNtoVNLookup));
  AxCheckError(clSetKernelArg(CheckKernel,cnodearg++,sizeof(vector_max_llr_d),&
       ↪ vector_max_llr_d));
  AxCheckError(clSetKernelArg(CheckKernel,cnodearg++,sizeof(vector_sign_bit_d),&
       ↪ vector_sign_bit_d));
  AxCheckError(clSetKernelArg(CheckKernel,cnodearg++,sizeof(allones_d),&allones_d));
  AxCheckError(clSetKernelArg(CheckKernel,cnodearg++,sizeof(ones_d),&ones_d));
  AxCheckError(clSetKernelArg(CheckKernel,cnodearg++,sizeof(zeros_d),&zeros_d));
  AxCheckError(clSetKernelArg(CheckKernel,cnodearg++,sizeof(cnInputsBuffer),&
      ↪ cnInputsBuffer));
  AxCheckError(clSetKernelArg(CheckKernel,cnodearg++,sizeof(vnInputsBuffer),&
      ↪ vnInputsBuffer));
  AxCheckError(clSetKernelArg(CheckKernel,cnodearg++,sizeof(cnResult_d),&cnResult_d));


  // Now let's run the kernels
  size_t global_size_vn= N_VARIABLE_NODES;
#ifdef ALTERA
  size_t local_size_vn = N_VARIABLE_NODES;
#else
  size_t local_size_vn=  NULL;
#endif
  // Now we start the Check Node Kernel
  size_t global_size_cn = N_CHECK_NODES;
#ifdef ALTERA
  size_t local_size_cn = N_CHECK_NODES;
#else
  size_t local_size_cn = NULL;
#endif
  cl_event kernelTimeVN;
  cl_event kernelTimeCN;

  cl_ulong time_start,time_end;
  double total_time;

  unsigned _int64 freq;
  QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
  double timerFrequency = (1.0/freq);


  unsigned __int64 startTime;
  QueryPerformanceCounter((LARGE_INTEGER *)&startTime);

  char offsetIndexVN;
  char offsetIndexCN;


  // Now let's start the iterations
  // In the first two iterations, we need to run just the variable node

  for(char iter = 0; iter < (N_ITERATIONS*2) + 2; iter++) {

    offsetIndexVN = (iter %2);
    offsetIndexCN = ((iter+1) %2);
    // For the first and last iteration, it is simply the VN that runs
    if(iter == 0  || iter == (N_ITERATIONS*2) + 1) {
      // Let's set the iter argument
      AxCheckError(clSetKernelArg(VariableKernel,vnodearg,sizeof(iter),&iter));
      AxCheckError(clSetKernelArg(VariableKernel,vnodearg+1,sizeof(offsetIndexVN),&
          ↪ offsetIndexVN));

      // Now let's start the kernel
      //printf("Starting VN Kernel iteration %d\n",iter);
      AxCheckError(clEnqueueNDRangeKernel(VariableQueue,VariableKernel,1,NULL,&
          ↪ global_size_vn,&local_size_vn,0,NULL,&kernelTimeVN));

      // Wait for the kernel to finish
      clFinish(VariableQueue);
    }
    else {
      // For the other iterations we run both

      // Set the kernel arguments
      AxCheckError(clSetKernelArg(VariableKernel,vnodearg,sizeof(iter),&iter));
      AxCheckError(clSetKernelArg(VariableKernel,vnodearg+1,sizeof(offsetIndexVN),&
          ↪ offsetIndexVN));


      AxCheckError(clSetKernelArg(CheckKernel,cnodearg,sizeof(iter),&iter));
      AxCheckError(clSetKernelArg(CheckKernel,cnodearg+1,sizeof(offsetIndexCN),&
          ↪ offsetIndexCN));

      // Now let's start the kernel
      //printf("Starting VN Kernel iteration %d\n",iter);
      AxCheckError(clEnqueueNDRangeKernel(VariableQueue,VariableKernel,1,NULL,&
          ↪ global_size_vn,&local_size_vn,0,NULL,&kernelTimeVN));


      //printf("Starting CN Kernel iteration %d\n",iter);
      AxCheckError(clEnqueueNDRangeKernel(CheckQueue,CheckKernel,1,NULL,&global_size_cn,&
          ↪ local_size_cn,0,NULL,&kernelTimeCN));


      // Now before going on, we need to wait for both kernels to complete
      clFinish(VariableQueue);
      clFinish(CheckQueue);

    }
  }



  unsigned __int64 endTime;
  QueryPerformanceCounter((LARGE_INTEGER *)&endTime);
  double timeDifferenceInMilliseconds = ((endTime-startTime) * timerFrequency)*1000;

  printf("Timemilliseconds␣=␣%f\n",timeDifferenceInMilliseconds);
  printf("frequency␣=␣%f\n",timerFrequency);
  printf("LArger␣start␣=␣%llu\n",startTime);
  printf("Larger␣end␣␣%llu\n",endTime);

  printf("Finished\n");


  /*
  cl_ulong time_start,time_end;
  double total_time;

  AxCheckError(clGetEventProfilingInfo(kernelTime,CL_PROFILING_COMMAND_START,sizeof(
      ↪ time_start),&time_start,NULL));
  AxCheckError(clGetEventProfilingInfo(kernelTime,CL_PROFILING_COMMAND_END,sizeof(time_end
      ↪ ),&time_end,NULL));

  total_time = time_end-time_start;
  // Now we can just get the data
  AxCheckError(clEnqueueReadBuffer(LDPCQueue,vnResult_d,CL_TRUE,0,vnResultSize,vnResult,0,
      ↪ NULL,NULL));
  AxCheckError(clEnqueueReadBuffer(LDPCQueue,cnResult_d,CL_TRUE,0,cnResultSize,cnResult,0,
      ↪ NULL,NULL));
  */
  // Let's check the accuracy
  // Ok let's print it out
  //checkIfSatisfied(cnResult);
  //printResults(vnResult);
  double milisecondTime = timeDifferenceInMilliseconds;//total_time/1000000.0;

  printf("The␣kernel␣execution␣time␣was␣%f␣ms\n",milisecondTime);
  double throughput = N_VARIABLE_NODES*MANUAL_VECTOR_WIDTH/((milisecondTime)/1000.0);
  printf("The␣throughput␣was␣%f␣bps\n",throughput);

  FILE * results = fopen("results_kernel.csv","w");

  fprintf(results,"%f,%f\n",milisecondTime,throughput);
  fclose(results);



}

void printResults(LLR * results) {

  // Need to get a value in binary first

#if MANUAL_VECTOR_WIDTH == 1
  printf("Vector␣0:␣");
  for(int i=0; i<N_VARIABLE_NODES; i++) {

    if(results[i] > 0)
      printf("0");
    else
      printf("1");
  }
  printf("\n");
#else
  for(int j=0; j<MANUAL_VECTOR_WIDTH; j++) {

    printf("Vector␣%d:␣",j);
    for(int i=0; i<N_VARIABLE_NODES; i++) {


      if(results[i].s[j] > 0)
        printf("0");
      else
        printf("1");
    }
    printf("\n");

  }
#endif
}

void setFileNames() {
  char temp[MAX_LINE];
  strncpy(temp,FILENAME,MAX_LINE);
  _snprintf(INPUT_FILENAME,MAX_LINE,"E:\\Andrew\\Projects\\%s%d.txt",temp,LDPC_SIZE);

  strncpy(temp,OUTPUT_FILE,MAX_LINE);
  _snprintf(OUTPUT_FILENAME,MAX_LINE,"E:\\Andrew\\Projects\\IrregularCodewordAttempt\\
      ↪ NoIterationTry\\results%dgpu.csv",temp,LDPC_SIZE);

#ifdef ALTERA
  _snprintf(KERNEL_FILENAME,MAX_LINE,"E:\\Andrew\\Projects\\IrregularCodewordAttempt\\
      ↪ NoIterationTry\\kernels%dnoiteration.aocx",LDPC_SIZE);
#else
  _snprintf(KERNEL_FILENAME,MAX_LINE,"E:\\Andrew\\Projects\\IrregularCodewordAttempt\\
      ↪ NoIterationTry\\kernels%dnoiterationgpu.cl",LDPC_SIZE);
#endif
}

void printResultsToFile(double overall, double variable, double check, double misc) {
  // Now let's open the file
  FILE * results = fopen(OUTPUT_FILENAME,"w"); CheckPointer(results);

  // Now let's write data to the file
  fprintf(results,"%f␣%f␣%f␣%f\n",overall,variable,check,misc);

  // Now we close the file
  fclose(results);

}





// Check if all of the check nodes are satisfied
bool checkIfSatisfied(LLR * parity)
{
#if MANUAL_VECTOR_WIDTH == 1
  // All we have to do use iterate over all and see if there is anything non zero
  for(int i=0; i<N_CHECK_NODES; i++) {
    if(parity[i]){
      printf("NOT␣SATISFIED\n");
      return false;
    }
  }
  printf("SATISFIED!\n");
  return true;
#else
  for(int j=0; j<MANUAL_VECTOR_WIDTH; j++){
    for(int i=0; i<N_CHECK_NODES; i++) {
      if(parity[i].s[j]){
        printf("NOT␣SATISFIED\n");
      }
    }
    printf("SATISFIED!\n");
  }

  return true;
#endif

}

#pragma region Codeword Operations
// Get the codeword in BPSK
float* getCodeword(int size)
{
  float *codeword = (float*)malloc(sizeof(float)*size);
  int * input = (int*) malloc(sizeof(int)*size);
  for(int i=0; i<size; i++)
    input[i] = 0;

  //input[0] = 1;

  // Convert it to BPSK
  // 0 -> 1, 1 -> -1
  // Map binary 0 to 1
  // May binary 1 to -1
  for(int i=0; i<size; i++)
  {
    if(input[i])
      codeword[i] = -1;
    else
      codeword[i] = 1;
  }

  return codeword;
}


// Introduce gaussian noise to the input codeword based on the given SNR
void introduceNoise(float* codeword, int SNR, int n_vars)
{
  double sigma = pow(10,(double)-SNR/20);

  for(int i=0; i<n_vars; i++)
  {
    codeword[i] += generate_random_gaussian()*sigma;
    //printf("Codeword[%d] = %f\n",i,codeword[i]);
  }
}

void printCodeword(const int * codeword, const int n_vars) {
  for(int i=0; i<n_vars; i++) {
    printf("codeword[%d]␣=␣%d\n",i,codeword[i]);
  }

}

/* This function will take in the given float version of the codeword
and convert it to an integer representation LLR.  Please note that
it converts the LLR into a 2's complement number first
*/
LLR_BASE_TYPE * convertToLLR(const float *codeword, const int n_vars) {
  LLR_BASE_TYPE * output = (LLR_BASE_TYPE*) malloc(sizeof(LLR_BASE_TYPE)*n_vars);
  float division_point = ((float)MAX_LLR)/SATURATION_POINT ;

  // Iterate over the entire codeword
  for(int i=0; i<n_vars; i++) {
    float word = abs(codeword[i]);
    output[i] = 0;


    // Now let's iterate over the possible bins and see where it fits
    float division_point = ((float)MAX_LLR)/SATURATION_POINT ;
    for(int j=1; j<=(MAX_LLR); j++) {

      if(word <  j / division_point) {
        output[i] = j;

        break;
      }

    }


    // Once we have the positioning we just need to make sure it's still within the valid
        ↪ LLR
    if(word >=  MAX_LLR / division_point) {
      output[i] = MAX_LLR;

    }

    // Now set the sign bit
    if(codeword[i] < 0) {
      output[i] *= -1;

    }

  }
  return output;
}
#pragma endregion


// This function will read the parity check matrix A-list file and get the following
     ↪ information:
// - number of variable nodes, number of check nodes
// - an array of variablenode structures
// - an array of checknode structures
// PLEASE NOTE ALL VARIABLES PASSED INTO THIS FUNCTION ARE PASSED BY REFERENCE AND WILL BE
     ↪  MODIFIED
// Please pass this function empty variables to store the result
void readParityFile(VariableNode *& v_nodes, CheckNode *& c_nodes,int &n_vars,int &
     ↪ n_checks)
{
  char line[MAX_LINE];

  // Open the file
  FILE * parity_matrix = fopen(INPUT_FILENAME,"r");
  CheckPointer(parity_matrix);

  // Now let's start reading
  // The first two values in the text file will be the number of variable nodes and the
       ↪ number of check nodes
  if(fscanf(parity_matrix,"%d␣%d",&n_vars,&n_checks)!=2)
    ExitOnErrorWMsg("Error␣with␣parity_matrix␣A-list␣file␣format,");

  // Now that we know the number of variable nodes and checks we can allocate the amounts
  v_nodes = (VariableNode*)malloc(sizeof(VariableNode)*n_vars);
  c_nodes = (CheckNode*)malloc(sizeof(CheckNode)*n_checks);

  // Throw away the next line.  This line tells us the approximate number of connections
       ↪ for each node, but we want the more accurate ones
  int temp;
  if(fscanf(parity_matrix,"%d␣%d",&temp,&temp)!=2)
    ExitOnErrorWMsg("Missing␣second␣line␣in␣A-list␣file.");

  // Now let's get to the good stuff
  // According to the A-list format the next line of text will be the list of check nodes
       ↪ for each variable node in order
```

```
// The next line will be a list of check nodes for each variable node
// Therefore we need to iterate n_vars times to get all of the connections
for(int i=0; i<n_vars; i++)
{
  int num_per_variable;
  if(fscanf(parity_matrix,"%d",&num_per_variable)!=1)
    ExitOnErrorWMsg("Error␣with␣parity_matrix␣A-list␣file␣format.");

  // Store the number of check nodes for each variable node and the number of inputs
  v_nodes[i].n_checknodes = num_per_variable;
  v_nodes[i].n_inputs = num_per_variable+1; // Add one here as there is a spot for the
      ↪ initial channel measurement

  // Let's also store the index
  v_nodes[i].index = i;
}

// The next line will be a list of variable nodes for each check node
// Therefore we need to iterate n_checks times to get all of the connections
for(int i=0; i<n_checks; i++)
{
  int num_per_check;
  if(fscanf(parity_matrix,"%d",&num_per_check)!=1)
    ExitOnErrorWMsg("Error␣with␣parity_matrix␣A-list␣file␣format.");

  // Store the number of inputs to the check node
  c_nodes[i].n_inputs = num_per_check; // Note there is not an extra here as there is no
      ↪  channel measurement

  // Also store the index
  c_nodes[i].index = i;

  // Set the satisifed bit to false
  //c_nodes[i].satisfied = false;
}

// Next we get the indexes for variable nodes (edges)
// This is the most important section where we determine which nodes get connected to
    ↪ each other
// First up are the variable node connections.  So we need to iterate over all the
    ↪ variable nodes
for(int i=0; i<n_vars; i++)
{
  int position;

  // Now we can use the number of checknodes we just received from the previous lines
  // We need to store which checknodes this variable node is connected to
  int j;
  for(j=0; j<v_nodes[i].n_checknodes; j++)
  {
    if(fscanf(parity_matrix,"%d",&position)!=1)
      ExitOnErrorWMsg("Error␣with␣parity_matrix␣A-list␣file␣format.");

    // Now store the position
    v_nodes[i].checknode_indexes[j] = position-1; // Note we need to subtract one here
        ↪ to start the position at 0

  }

  // Get the rest of it up to the max
  for(j = v_nodes[i].n_checknodes; j < N_VARIABLE_INPUTS; j++) {
    // So for the rest of the connections, we actually want to assign one of the garbage
        ↪  locations
    int randomCheck = rand() % n_checks;

    // Now we have a random check node
    v_nodes[i].checknode_indexes[j] = randomCheck;


  }

}


// Same thing, but now for the check nodes.
// Next we get the indexes for the check nodes (edges)
for(int i=0; i<n_checks; i++)
{
  int position;
  int j;
  for(j=0; j<c_nodes[i].n_inputs; j++)
  {
    if(fscanf(parity_matrix,"%d",&position)!=1)
      ExitOnErrorWMsg("Error␣with␣parity_matrix␣A-list␣file␣format.");

    // Now store the position
    c_nodes[i].variablenode_indexes[j] = position-1; // Note we need to subtract one
        ↪ here to start the position at 0
  }

  // Get the rest of it up to the max
  for(j = c_nodes[i].n_inputs; j < N_CHECK_INPUTS; j++) {
      // So for the rest of the connections, we actually want to assign one of the garbage
          ↪  locations
      int randomCheck = rand() % n_vars;

      // Now we have a random check node
      c_nodes[i].variablenode_indexes[j] = randomCheck;


    }
  }

  // That concludes the reading of the parity matrix file
  // All data is now stored within the v_nodes and c_nodes arrays
  fclose(parity_matrix);
  printf("Parity␣matrix␣file␣%s␣read␣successfully\n",INPUT_FILENAME);
  return;
}


//Initialize the nodes for the first run
// This is where we'll need to load up that there are 4 of the vector chars
void initializeNodes(VariableNode *& v_nodes, CheckNode *& c_nodes, const LLR * codeword,
    ↪ const int n_vars, const int n_checks)
{

  // First we load up the channel measurement into all of the variable nodes
  // Iterate over each variable node
  for(int i=0; i<n_vars; i++)
  {
    short channel_index = v_nodes[i].n_inputs-1;
    // For each of the variable nodes, we want to only load up the very last spot that
        ↪ deals with the channel input
    // All of the other inputs should be 0.  This will allows us to start the execution
        ↪ with the variable nodes instead of the check nodes


    for(int j=0; j<v_nodes[i].n_inputs; j++)
    {

      // We need to initialize all 4 of the chars in the vector
      // Note that we use codeword.x(y) etc however they are all equal to begin but it is
          ↪ possible for these to be different
#if MANUAL_VECTOR_WIDTH == 1
      v_nodes[i].inputs[j] = codeword[i];
#else
      for(int t=0; t<MANUAL_VECTOR_WIDTH; t++) {
        v_nodes[i].inputs[j].s[t] = codeword[i].s[t];
      }
#endif
    }
  }


  // Load up the inputs for the check nodes for the first iteration
  // This means we are now passing the specific channel measurements for each connection
  for(int i=0; i<n_checks; i++)
  {

    // For each one of the inputs in this check node, we need to load up the information
        ↪ from the variable node
    for(int j=0; j<c_nodes[i].n_inputs; j++)
    {
      int variable_index = c_nodes[i].variablenode_indexes[j];
      int index = v_nodes[variable_index].n_inputs-1; // All of the spots hold the same
          ↪ value currently

      // We have to convert this value into sign and mag
      LLR sign_and_mag_ver = v_nodes[variable_index].inputs[index];

      // Now we need to do the four comparisons individually
#if MANUAL_VECTOR_WIDTH == 1
      if(sign_and_mag_ver < 0) {
        sign_and_mag_ver *= -1;
        sign_and_mag_ver |= SIGN_BIT;
      }
#else
      for(int t=0; t<MANUAL_VECTOR_WIDTH; t++) {
        if(sign_and_mag_ver.s[t] < 0) {
        sign_and_mag_ver.s[t] *= -1;
        sign_and_mag_ver.s[t] |= SIGN_BIT;
        }
      }
#endif



      // Let's store the value in our input
      c_nodes[i].inputs[j] = sign_and_mag_ver;
    }
  }
}
```

```
#pragma region Error Checking Functions
// This function will check the given input pointer for validaty and fail exit the program
    ↪  if necessary
void CheckPointerFunc(void * pointer,char const* const func, const int line)
{
  if(pointer == NULL)
  {
    fprintf(stderr,"Error␣with␣pointer␣in␣%s␣line␣%d\n",func,line);
    exit(SYSTEM_ERROR);
  }
}

void ExitOnErrorFunc(char const* msg, char const* func, const int line)
{
  fprintf(stderr,"%s␣%s␣line␣%d\n",msg,func,line);
  exit(SYSTEM_ERROR);
}
#pragma endregion

#pragma region Random Number Code
double generate_random_gaussian()
{
#define PI 3.14159265358979323846
  double U1 = generate_random_number();//(double)rand()/((doule)RAND_MAX+1);
  double U2 = generate_random_number();//(double)rand()/((doule)RAND_MAX+1);

  double Z0 = sqrt(-2*log(U1))*cos(2*PI*U2);
  return Z0;
}



int * generate_VNtoCN_index_table(VariableNode * v_nodes,CheckNode * c_nodes) {
  int * table = (int*)_aligned_malloc(sizeof(int)*N_VARIABLE_NODES*N_VARIABLE_INPUTS,
      ↪ AOCL_ALIGNMENT);
  if(table == NULL)
    printf("err␣=␣%d\n",errno);

  // Iterate over the list of variable nodes
  for(int i=0; i<N_VARIABLE_NODES; i++) {

    int j;
    // For each variable node we need to find the appropriate connections for each input
    for(j=0; j<v_nodes[i].n_checknodes; j++) {
      int cnIndex = v_nodes[i].checknode_indexes[j];


      int cnInputIndex;
      // Now let's look inside that check node to find our appropriate location
      for(int n=0; n<N_CHECK_INPUTS; n++) {
        int tempIndex = c_nodes[cnIndex].variablenode_indexes[n];

        // If we found our variable node, make the table
        if(tempIndex == i) {
          cnInputIndex = n;
          break;
        }
      }

      // Now we can create our table entry
      table[i*N_VARIABLE_INPUTS + j] = (cnIndex*(N_CHECK_INPUTS+1)) + cnInputIndex;




    }

    for(j = v_nodes[i].n_checknodes; j < N_VARIABLE_INPUTS; j++) {
      // For the rest of them we want to assign the random check node at the end
      int cnIndex = v_nodes[i].checknode_indexes[j];

      // Alright so we just need to assign the last position
      table[i*N_VARIABLE_INPUTS + j] = (cnIndex * (N_CHECK_INPUTS+1)) + N_CHECK_INPUTS;
    }







  }
  return table;
}

int * generate_CNtoVN_index_table(VariableNode * v_nodes,CheckNode * c_nodes) {
  int * table = (int*)_aligned_malloc(sizeof(int)*N_CHECK_NODES*N_CHECK_INPUTS,
      ↪ AOCL_ALIGNMENT);
  if(table == NULL)
    printf("err␣=␣%d\n",errno);
  // Iterate over the list of variable nodes
  for(int i=0; i<N_CHECK_NODES; i++) {
    int j;
    // For each variable node we need to find the appropriate connections for each input
    for(j=0; j<c_nodes[i].n_inputs; j++) {
      int vnIndex = c_nodes[i].variablenode_indexes[j];

      int vnInputIndex = -100;
      // Now let's look inside that check node to find our appropriate location
      for(int n=0; n<N_VARIABLE_INPUTS; n++) {
        int tempIndex = v_nodes[vnIndex].checknode_indexes[n];

        // If we found our variable node, make the table
        if(tempIndex == i) {
          vnInputIndex = n;
          break;
        }
      }

      int foundIndex = (vnIndex * (N_VARIABLE_INPUTS+1)) + vnInputIndex;

      // Populate the table
      table[i*N_CHECK_INPUTS + j] = foundIndex;

    }

    // Alright for the rest all we need to do is give it the garbage address
    for(j = c_nodes[i].n_inputs; j < N_CHECK_INPUTS; j++) {
      int vnIndex = c_nodes[i].variablenode_indexes[j];


      table[i*N_CHECK_INPUTS + j] = (vnIndex * (N_VARIABLE_INPUTS+1)) + N_VARIABLE_INPUTS;



    }


  }

  printf("one␣in␣question␣=␣%d\n",table[1*N_CHECK_INPUTS + 15]);

  return table;
}

double generate_random_number()
{
#define C1 4294967294
#define C2 4294967288
#define C3 4294967280
#define MAX 4294967295

  srand(45);
  static unsigned int s1 = rand();
  static unsigned int s2 = rand();
  static unsigned int s3 = rand();

  // Here's the first part
  // Now let's crank the tausworthe
  unsigned int xor1 = s1 << 13;

  // The first set of and/xor
  unsigned int left_temp = xor1 ^ s1;
  unsigned int right_temp = C1 & s1;

  // Shifts
  left_temp = left_temp >> 19;
  right_temp = right_temp << 12;

  s1 = left_temp ^ right_temp;

  // Second part
  xor1 = s2<<2;
  left_temp = xor1 ^ s2;
  right_temp = C2 & s2;

  left_temp = left_temp >> 25;
  right_temp = right_temp << 4;

  s2 = left_temp ^ right_temp;

  // Third part
  xor1 = s3 << 3;
  left_temp = xor1 ^ s3;
  right_temp = C3 & s3;

  left_temp = xor1 ^ s3;
  right_temp = C3 & s3;

  left_temp = left_temp >> 11;
  right_temp = right_temp << 17;

  s3 = left_temp ^ right_temp;
  // Now the return
  unsigned int output = s1 ^ s2 ^ s3;

  // Now just convert it into a double
  double last_value = (double)output /MAX;

  //cout<<"last value" << last_value<<endl;
  return last_value;

}
#pragma endregion

void defineGlobalDefinitions() {

  vector_max_llr = (LLR*) _aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  vector_min_neg_llr = (LLR*)_aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  vector_sign_bit = (LLR*)_aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  allones = (LLR*)_aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  ones = (LLR*)_aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  zeros = (LLR*)_aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  vector_too_neg_llr = (LLR*)_aligned_malloc(sizeof(LLR),AOCL_ALIGNMENT);
  total_vector_max_llr = (TOTAL_TYPE*)_aligned_malloc(sizeof(TOTAL_TYPE),AOCL_ALIGNMENT);
  total_vector_min_neg_llr = (TOTAL_TYPE*)_aligned_malloc(sizeof(TOTAL_TYPE),
      ↪ AOCL_ALIGNMENT);



#if MANUAL_VECTOR_WIDTH==1
  vector_max_llr = MAX_LLR;
  vector_min_neg_llr = MAX_NEG_LLR;
  vector_sign_bit = SIGN_BIT;
  allones = (LLR)(0xFF);
  ones = (LLR)(1);
  zeros = (LLR)(0);
  vector_too_neg_llr = MAX_NEG_LLR - ones;
  total_vector_max_llr = MAX_LLR;
  total_vector_min_neg_llr = MAX_NEG_LLR;
#else
  for(int i=0; i<MANUAL_VECTOR_WIDTH; i++)
  {
    (*vector_max_llr).s[i] = MAX_LLR;
    (*vector_min_neg_llr).s[i] = MAX_NEG_LLR;
    (*vector_sign_bit).s[i] = SIGN_BIT;
    (*allones).s[i] = (0xFF);
    (*ones).s[i] = (1);
    (*zeros).s[i] = (0);
    (*vector_too_neg_llr).s[i] = MAX_NEG_LLR - 1;
    (*total_vector_max_llr).s[i] = MAX_LLR;
    (*total_vector_min_neg_llr).s[i] = MAX_NEG_LLR;
  }
#endif
}

void parseProgramInputs(int n_args,char* args[]) {
  for(int i=0; i<n_args-1; i++) {
    if(strncmp(args[i],"-nVars",MAX_LINE)==0) {
      // If we found the n_vars just set the variable

      int new_value = atoi(args[i+1]);
      if(new_value <= 0) {
        ExitOnErrorWMsg("Not␣a␣valid␣LDPC␣size␣input\n");
      }
      N_VARIABLE_NODES = atoi(args[i+1]);
      LDPC_SIZE = N_VARIABLE_NODES;
    }
    else if (strncmp(args[i], "-nCheck",MAX_LINE)==0) {
      int new_value = atoi(args[i+1]);
      if(new_value <=0 ) {
        ExitOnErrorWMsg("Not␣a␣valid␣LDPC␣size␣input\n");
      }
      N_CHECK_NODES = atoi(args[i+1]);
    }
  }

}
```

Appendix M

Single Work-Item (Design D) Irregular Length-1120 Kernel Source Code (OpenCL)

```
#ifndef N_VARIABLE_NODES
#define N_VARIABLE_NODES 1120
#endif

#ifndef N_CHECK_NODES
#define N_CHECK_NODES 280
#endif

#ifndef SIMD
#define SIMD 1
#endif

// Based on the Tanner graph, how many connections are there
#define MAX_VARIABLE_EDGES 5
#define MAX_CHECK_EDGES 16


// Let's define the usable number of inputs
#define N_VARIABLE_INPUTS (MAX_VARIABLE_EDGES)
#define N_CHECK_INPUTS (MAX_CHECK_EDGES)

#define N_VARIABLE_INPUTS_W_CHANNEL (N_VARIABLE_INPUTS + 1)

#define N_ITERATIONS (32)



// Set the vector width and the appropriate LLR types
#define MANUAL_VECTOR_WIDTH (16)

#if MANUAL_VECTOR_WIDTH==2
#define LLR_TYPE char2
#define TOTAL_TYPE short2
#define CONVERT_TO_SHORT(a) convert_short2(a)
#define CONVERT_TO_CHAR(a) convert_char2(a)
#elif MANUAL_VECTOR_WIDTH==4
#define LLR_TYPE char4
#define TOTAL_TYPE short4
#define CONVERT_TO_SHORT(a) convert_short4(a)
#define CONVERT_TO_CHAR(a) convert_char4(a)
#elif MANUAL_VECTOR_WIDTH==8
#define LLR_TYPE char8
#define TOTAL_TYPE short8
#define CONVERT_TO_SHORT(a) convert_short8(a)
#define CONVERT_TO_CHAR(a) convert_char8(a)
#elif MANUAL_VECTOR_WIDTH==16
#define LLR_TYPE char16
#define TOTAL_TYPE short16
#define CONVERT_TO_SHORT(a) convert_short16(a)
#define CONVERT_TO_CHAR(a) convert_char16(a)
#else
#define LLR_TYPE char
#define TOTAL_TYPE short
#define CONVERT_TO_SHORT(a) convert_short(a)
#define CONVERT_TO_CHAR(a) convert_char(a)
#endif
```

```
typedef LLR_TYPE LLR;


// Now for the channels
#pragma OPENCL EXTENSION cl_altera_channels : enable
channel char vnStart __attribute__((depth(N_ITERATIONS)));
channel char cnStart __attribute__((depth(N_ITERATIONS)));
#define SENDDATA (1)

//Variable Node Kernel
__kernel
__attribute__((reqd_work_group_size(N_VARIABLE_NODES,1,1)))
__attribute__((num_compute_units(1)))
__attribute__((num_simd_work_items(SIMD)))
void VariableNode(

// Kernel parameteres
// As in the notes page, there is a table that determines the write location in the CNs
__constant short * restrict vntocnIndex,

// Now that we have the connection information let's get the inputs
// Remember there is one memory location for the Variable Node inputs with two slots for
    ↪ pipelining
// Let's start with the channel measurements
__global LLR * restrict channelMeasurements,

// Now let's get the inputs
__global LLR * restrict vnInputsGM,

// Now let's get the output pointer to the CNs
__global LLR * restrict vnOutputsGM,

// And of course finally, we need to get the result
__global TOTAL_TYPE * restrict vnResults,


// We need a few constants where it's easier to compute them in the host
__constant LLR * restrict vector_max_llr,
__constant LLR * restrict vector_sign_bit,
__constant LLR * restrict allones,
__constant LLR * restrict ones,
__constant LLR * restrict zeros,
__constant TOTAL_TYPE * restrict total_vector_max_llr,
__constant TOTAL_TYPE * restrict total_vector_min_neg_llr
)
{

  // First we need to get our global IDsa
  int id = get_global_id(0);



  // We also need a short version of the LLR values to total
  TOTAL_TYPE vnInputsShort[N_VARIABLE_INPUTS_W_CHANNEL];

  // Let's make an array for the offsets
  int vnMemoryOffsets[2];
  vnMemoryOffsets[0] = 0;
  vnMemoryOffsets[1] = N_VARIABLE_NODES * (N_VARIABLE_INPUTS+1);

  int cnMemoryOffsets[2];
  cnMemoryOffsets[0] = 0;
  cnMemoryOffsets[1] = N_CHECK_NODES * (N_CHECK_INPUTS+1);

  // Let's make an array index lookup for the input
  int vntocnNodeIndex = id * (N_VARIABLE_INPUTS);
  int vnGlobalNodeIndex = id * (N_VARIABLE_INPUTS+ 1);

  // Some other variables
  LLR vnCurrent;
  TOTAL_TYPE vnTempCurrent;
  LLR vnConditional;
  LLR vnSignB;
  LLR vnMag;

  // Now let's store the pointer
  char offsetIndex = 0;

  //printf("VN,%d,prepped\n",id);

  // Let's start the process
  // We need to step through the iterations (aka start the loop)
  // Since both VN and CN will be running concurrently on the same data, we will need
      ↪ N_ITERATIONS*2+1
  for(char iter = (char)(0); iter < ((N_ITERATIONS * 2) + 2); iter++) {

    // Let's just do some printing
    //printf("VN,%d,%d\n",id,iter);
    // Alright so each iteration we need to do the following:
    // Each VN:
    // 1) Wait for the OK from the CNs to start on the data
    // 2) Load up the inputs from DRAM into private memory
    // 3) Do the VN
    // 4) Output the data to the appropriate buffer
    // 5) Send the ok to the CN to start execution
    // 6) Swap buffers
    // Repeat


    // Let's wait for the CNs to tell us it's ok to start execution
    // Note that we can throw away the data as this is a blocking call
    if(id == 0 && iter > 1)
      read_channel_altera(vnStart);

    barrier(CLK_GLOBAL_MEM_FENCE);


    // Now that we've got the OK. Let's start execution
    // First let's load up the inputs and convert them to shorts so that we can get the
        ↪ appropriate totals without saturation
    #pragma unroll
    for(char input = 0; input < N_VARIABLE_INPUTS; input++) {
      vnInputsShort[input] = CONVERT_TO_SHORT(vnInputsGM[vnMemoryOffsets[offsetIndex] +
          ↪ vnGlobalNodeIndex + input]);
    }

    // Now let's put in the channel measurement for our node
    vnInputsShort[N_VARIABLE_INPUTS] = CONVERT_TO_SHORT(channelMeasurements[
        ↪ vnMemoryOffsets[offsetIndex] + vntocnNodeIndex]);

    // Now we need to add it all together

    TOTAL_TYPE total = (TOTAL_TYPE)(0);
    #pragma unroll
    for(char input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {
      total += vnInputsShort[input];
    }

    // Now that we have the total we can do the rest
    for(char input = 0; input < N_VARIABLE_INPUTS; input++) {

      // Subtract the current from the total to get the output
      // We simply subtract the large total and then saturate it down to the appropriate
          ↪ size
      vnTempCurrent = total - vnInputsShort[input];

      // Now saturate
      vnTempCurrent =   ((vnTempCurrent >= (*total_vector_max_llr)) ?
                (*total_vector_max_llr) :
                ((vnTempCurrent <= (*total_vector_min_neg_llr)) ?
                (*total_vector_min_neg_llr) :
                vnTempCurrent));
      // Now we reverse the procedure and store the data back into the LLR type
      vnCurrent = CONVERT_TO_CHAR(vnTempCurrent);

      // Ok now we can finally send the data
      // First we check if it's negative
      // Except we need to convert it to sign and magnitude
      vnConditional = ((vnCurrent & (*vector_sign_bit)) == (*vector_sign_bit)) ? (*allones
          ↪ ) : (*zeros);

      // If it's negative we need to xor it and add one and mask out the sign bit
      vnMag = ((vnConditional & ((vnCurrent ^ (*allones)) + (*ones))) | ((vnConditional ^
          ↪ (*allones)) & vnCurrent)) & (*vector_max_llr);

      // Now get the sign bit
      vnSignB = (vnConditional & (*vector_sign_bit));

      // Now we just need to find the location in the array to place the data
      short cnIndex = vntocnIndex[vntocnNodeIndex + input] + cnMemoryOffsets[offsetIndex];

      // Now we can just output the value
      // Since we're writing to the CNs we need to output to the write memory location
      vnOutputsGM[cnIndex] = vnMag | vnSignB;

    }

    if((iter+1)==N_ITERATIONS*2+1) {
      TOTAL_TYPE send_total = 0;
      for(short input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {
        send_total += vnInputsShort[input];
      }

      vnResults[vnMemoryOffsets[offsetIndex] + id] = send_total;
    }

    barrier(CLK_GLOBAL_MEM_FENCE);

    //printf("VN,%d,%d,beforechannelwrite\n",id,iter);
    // Finally, we need to tell the check node it can operate on the data
    if(id == 0)
      write_channel_altera(cnStart, SENDDATA);
    // Now let's switch the index over
    // To do this we can simply XOR it with the value of 1
    offsetIndex ^= 1;




  }

  TOTAL_TYPE send_total = 0;
  for(short input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {
    send_total += vnInputsShort[input];
  }

  vnResults[vnMemoryOffsets[offsetIndex] + id] = send_total;


}


__kernel
__attribute__((reqd_work_group_size(N_CHECK_NODES,1,1)))
__attribute__((num_compute_units(1)))
__attribute__((num_simd_work_items(SIMD)))
void CheckNode(

// Let's start with the table for the interleaver connections just like with the VN
__constant short * restrict cntovnIndex,

// Now we can do the other constants
__constant LLR * restrict vector_max_llr,
__constant LLR * restrict vector_sign_bit,
__constant LLR * restrict allones,
__constant LLR * restrict ones,
__constant LLR * restrict zeros,

// Now we can focus on the inputs
__global LLR * restrict cnInputsGM,

// Now the outputs
__global LLR * restrict cnOutputsGM,

// Now a place for the results
__global LLR * restrict cnResults

) {



  int id = get_global_id(0);

  // First we need to store a place in private memory for operation
  LLR cnInputs[N_CHECK_INPUTS];

  // And also a spot for the magnitudes
  LLR cnInputsMag[N_CHECK_INPUTS];

  // Let's compute the base index
  int cnInputsNodeIndex = id * N_CHECK_INPUTS;
  int cnInputsGlobalIndex = id * (N_CHECK_INPUTS+1);

  // Check Node Private Variables
  LLR cnFirstMin;
  LLR cnSecondMin;
  LLR cnParity;


  // Ok. Now we need to do some similar work to the VNs.
  // Let's make an array for the offsets
  int vnMemoryOffsets[2];
  vnMemoryOffsets[0] = 0;
  vnMemoryOffsets[1] = N_VARIABLE_NODES * (N_VARIABLE_INPUTS+1);

  int cnMemoryOffsets[2];
  cnMemoryOffsets[0] = 0;
  cnMemoryOffsets[1] = N_CHECK_NODES * (N_CHECK_INPUTS+1);

  // And let's store an offset
  char offsetIndex = 0; // Note that we start on the same data as VN but we have to wait
      ↪ for the ramp up of the VN to complete the first iterations before we can execute

  //printf("CN,%d,prepped\n",id);

  // Alright let's start the iterations
  for(char iter = 0; iter < (N_ITERATIONS * 2); iter++) {

    //printf("CN,%d,%d\n",id,iter);
    // For each iteration we need to:
    // 1) Wait for the ok from the VNs to start executing on the information
    // 2) Load in the new inputs from GM
    // 3) Do the Cn
    // 4) Send the data to the VN via GM
    // 5) Tell the VN we have finished

    // So let's wait for the Ok to start
    if(id == 0)
      read_channel_altera(cnStart);

    barrier(CLK_GLOBAL_MEM_FENCE | CLK_CHANNEL_MEM_FENCE);

    // Now we can begin our execution
    cnFirstMin = (*vector_max_llr);
    cnSecondMin = (*vector_max_llr);

    // First let's do the minimum calculations

    // But first, we need to make sure that we are taking the absolute value as we don't
        ↪ want to include that in the min calculation
    // Therefore, we have to mask off the sign bits
    #pragma unroll
    for(short input = 0; input < N_CHECK_INPUTS; input++) {
      cnInputs[input] = cnInputsGM[cnMemoryOffsets[offsetIndex] + cnInputsGlobalIndex +
          ↪ input];
    }

    // Now that we have the normal inputs, we need to get the magnitudes
    #pragma unroll
    for(short input = 0; input < N_CHECK_INPUTS; input++) {
      cnInputsMag[input] = cnInputs[input] & (*vector_max_llr);
    }

    // Ok now that we have the magnitudes, let's get to work by getting the overall parity
        ↪  first
    cnParity = (*zeros);
    #pragma unroll
    for(short input = 0; input < N_CHECK_INPUTS; input++) {
      cnParity ^= cnInputs[input] & (*vector_sign_bit);
    }

    // Now cnParity holds the overall parity of the check node

    // We're going to try something a little different here
    // We're going to implement the tree using min on the magnitudes

    // This is a little trickier as we need to find the losers a lot
    // But let's start it out.


    /************************************* ROUND 1 *************************************/
    LLR comp_0_1  = (cnInputsMag[0] < cnInputsMag[1]) ? (*allones) : (*zeros);
    LLR comp_2_3  = (cnInputsMag[2] < cnInputsMag[3]) ? (*allones) : (*zeros);
    LLR comp_4_5  = (cnInputsMag[4] < cnInputsMag[5]) ? (*allones) : (*zeros);
    LLR comp_6_7  = (cnInputsMag[6] < cnInputsMag[7]) ? (*allones) : (*zeros);
    LLR comp_8_9  = (cnInputsMag[8] < cnInputsMag[9]) ? (*allones) : (*zeros);
    LLR comp_10_11  = (cnInputsMag[10] < cnInputsMag[11]) ? (*allones) : (*zeros);
    LLR comp_12_13  = (cnInputsMag[12] < cnInputsMag[13]) ? (*allones) : (*zeros);
    LLR comp_14_15  = (cnInputsMag[14] < cnInputsMag[15]) ? (*allones) : (*zeros);



    // First Minimum
    LLR winner_0_to_1 = (cnInputsMag[0] & comp_0_1) | (cnInputsMag[1] & (comp_0_1 ^ (*
        ↪ allones)));
    // Second Minimum
    LLR loser_0 = (cnInputsMag[1] & comp_0_1) | (cnInputsMag[0] & (comp_0_1 ^ (*allones)))
        ↪ ;


    // First Minimum
    LLR winner_2_to_3 = (cnInputsMag[2] & comp_2_3) | (cnInputsMag[3] & (comp_2_3 ^ (*
        ↪ allones)));
    // Second Minimum
    LLR loser_1 = (cnInputsMag[3] & comp_2_3) | (cnInputsMag[2] & (comp_2_3 ^ (*allones)))
        ↪ ;


    // First Minimum
    LLR winner_4_to_5 = (cnInputsMag[4] & comp_4_5) | (cnInputsMag[5] & (comp_4_5 ^ (*
        ↪ allones)));
    // Second Minimum
    LLR loser_2 = (cnInputsMag[5] & comp_4_5) | (cnInputsMag[4] & (comp_4_5 ^ (*allones)))
        ↪ ;


    // First Minimum
    LLR winner_6_to_7 = (cnInputsMag[6] & comp_6_7) | (cnInputsMag[7] & (comp_6_7 ^ (*
        ↪ allones)));
    // Second Minimum
    LLR loser_3 = (cnInputsMag[7] & comp_6_7) | (cnInputsMag[6] & (comp_6_7 ^ (*allones)))
        ↪ ;


    // First Minimum
```

```
LLR winner_8_to_9 = (cnInputsMag[8] & comp_8_9) | (cnInputsMag[9] & (comp_8_9 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_4 = (cnInputsMag[9] & comp_8_9) | (cnInputsMag[8] & (comp_8_9 ^ (*allones)))
    ↪ ;


// First Minimum
LLR winner_10_to_11 = (cnInputsMag[10] & comp_10_11) | (cnInputsMag[11] & (comp_10_11
    ↪ ^ (*allones)));
// Second Minimum
LLR loser_5 = (cnInputsMag[11] & comp_10_11) | (cnInputsMag[10] & (comp_10_11 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_12_to_13 = (cnInputsMag[12] & comp_12_13) | (cnInputsMag[13] & (comp_12_13
    ↪ ^ (*allones)));
// Second Minimum
LLR loser_6 = (cnInputsMag[13] & comp_12_13) | (cnInputsMag[12] & (comp_12_13 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_14_to_15 = (cnInputsMag[14] & comp_14_15) | (cnInputsMag[15] & (comp_14_15
    ↪ ^ (*allones)));
// Second Minimum
LLR loser_7 = (cnInputsMag[15] & comp_14_15) | (cnInputsMag[14] & (comp_14_15 ^ (*
    ↪ allones)));


/*----------------------------------- END ROUND 1 -----------------------------------
    ↪ */




/*********************************** ROUND 2 ***********************************/
LLR comp_0_to_3 = (winner_0_to_1 < winner_2_to_3) ? (*allones) : (*zeros);
LLR comp_4_to_7 = (winner_4_to_5 < winner_6_to_7) ? (*allones) : (*zeros);
LLR comp_8_to_11  = (winner_8_to_9 < winner_10_to_11) ? (*allones) : (*zeros);
LLR comp_12_to_15 = (winner_12_to_13 < winner_14_to_15) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_3 = (winner_0_to_1 & comp_0_to_3) | (winner_2_to_3 & (comp_0_to_3 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_8 = (winner_2_to_3 & comp_0_to_3) | (winner_0_to_1 & (comp_0_to_3 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_4_to_7 = (winner_4_to_5 & comp_4_to_7) | (winner_6_to_7 & (comp_4_to_7 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_9 = (winner_6_to_7 & comp_4_to_7) | (winner_4_to_5 & (comp_4_to_7 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_8_to_11  = (winner_8_to_9 & comp_8_to_11) | (winner_10_to_11 & (
    ↪ comp_8_to_11 ^ (*allones)));
// Second Minimum
LLR loser_10  = (winner_10_to_11 & comp_8_to_11) | (winner_8_to_9 & (comp_8_to_11 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_12_to_15 = (winner_12_to_13 & comp_12_to_15) | (winner_14_to_15 & (
    ↪ comp_12_to_15 ^ (*allones)));
// Second Minimum
LLR loser_11  = (winner_14_to_15 & comp_12_to_15) | (winner_12_to_13 & (comp_12_to_15
    ↪ ^ (*allones)));


/*----------------------------------- END ROUND 2 -----------------------------------
    ↪ */




/*********************************** ROUND 3 ***********************************/
LLR comp_0_to_7 = (winner_0_to_3 < winner_4_to_7) ? (*allones) : (*zeros);
LLR comp_8_to_15  = (winner_8_to_11 < winner_12_to_15) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_7 = (winner_0_to_3 & comp_0_to_7) | (winner_4_to_7 & (comp_0_to_7 ^ (*
    ↪ allones)));
```

```
// Second Minimum
LLR loser_12  = (winner_4_to_7 & comp_0_to_7) | (winner_0_to_3 & (comp_0_to_7 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_8_to_15  = (winner_8_to_11 & comp_8_to_15) | (winner_12_to_15 & (
    ↪ comp_8_to_15 ^ (*allones)));
// Second Minimum
LLR loser_13  = (winner_12_to_15 & comp_8_to_15) | (winner_8_to_11 & (comp_8_to_15 ^
    ↪ (*allones)));


/*------------------------------------ END ROUND 3 -----------------------------------
    ↪ */




/************************************ ROUND 4 ************************************/
LLR comp_0_to_15  = (winner_0_to_7 < winner_8_to_15) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_15  = (winner_0_to_7 & comp_0_to_15) | (winner_8_to_15 & (comp_0_to_15
    ↪  ^ (*allones)));
// Second Minimum
LLR loser_14  = (winner_8_to_15 & comp_0_to_15) | (winner_0_to_7 & (comp_0_to_15 ^ (*
    ↪ allones)));


/*------------------------------------ END ROUND 4 -----------------------------------
    ↪ */




cnFirstMin = winner_0_to_15;

/************************************ ROUND 1 ************************************/
LLR second_comp_0_to_1  = (loser_0 < loser_1) ? (*allones) : (*zeros);
LLR second_comp_2_to_3  = (loser_2 < loser_3) ? (*allones) : (*zeros);
LLR second_comp_4_to_5  = (loser_4 < loser_5) ? (*allones) : (*zeros);
LLR second_comp_6_to_7  = (loser_6 < loser_7) ? (*allones) : (*zeros);
LLR second_comp_8_to_9  = (loser_8 < loser_9) ? (*allones) : (*zeros);
LLR second_comp_10_to_11  = (loser_10 < loser_11) ? (*allones) : (*zeros);
LLR second_comp_12_to_13  = (loser_12 < loser_13) ? (*allones) : (*zeros);



LLR second_winner_0_to_1  = (loser_0 & second_comp_0_to_1) | (loser_1 & (
    ↪ second_comp_0_to_1 ^ (*allones)));
LLR second_winner_2_to_3  = (loser_2 & second_comp_2_to_3) | (loser_3 & (
    ↪ second_comp_2_to_3 ^ (*allones)));
LLR second_winner_4_to_5  = (loser_4 & second_comp_4_to_5) | (loser_5 & (
    ↪ second_comp_4_to_5 ^ (*allones)));
LLR second_winner_6_to_7  = (loser_6 & second_comp_6_to_7) | (loser_7 & (
    ↪ second_comp_6_to_7 ^ (*allones)));
LLR second_winner_8_to_9  = (loser_8 & second_comp_8_to_9) | (loser_9 & (
    ↪ second_comp_8_to_9 ^ (*allones)));
LLR second_winner_10_to_11  = (loser_10 & second_comp_10_to_11) | (loser_11 & (
    ↪ second_comp_10_to_11 ^ (*allones)));
LLR second_winner_12_to_13  = (loser_12 & second_comp_12_to_13) | (loser_13 & (
    ↪ second_comp_12_to_13 ^ (*allones)));
/*------------------------------------ END ROUND 1 -----------------------------------
    ↪ */




/************************************ ROUND 2 ************************************/
LLR second_comp_0_to_3  = (second_winner_0_to_1 < second_winner_2_to_3) ? (*allones) :
    ↪  (*zeros);
LLR second_comp_4_to_7  = (second_winner_4_to_5 < second_winner_6_to_7) ? (*allones) :
    ↪  (*zeros);
LLR second_comp_8_to_11 = (second_winner_8_to_9 < second_winner_10_to_11) ? (*allones)
    ↪  : (*zeros);
LLR second_comp_12_to_14  = (second_winner_12_to_13 < loser_14) ? (*allones) : (*zeros
    ↪ );

LLR second_winner_0_to_3 = (second_winner_0_to_1 & second_comp_0_to_3) | (
    ↪ second_winner_2_to_3 & (second_comp_0_to_3 ^ (*allones)));
LLR second_winner_4_to_7 = (second_winner_4_to_5 & second_comp_4_to_7) | (
    ↪ second_winner_6_to_7 & (second_comp_4_to_7 ^ (*allones)));
LLR second_winner_8_to_11 = (second_winner_8_to_9 & second_comp_8_to_11) | (
    ↪ second_winner_10_to_11 & (second_comp_8_to_11 ^ (*allones)));
LLR second_winner_12_to_14 = (second_winner_12_to_13 & second_comp_12_to_14) | (
    ↪ loser_14 & (second_comp_12_to_14 ^ (*allones)));
/*------------------------------------ END ROUND 2 -----------------------------------
    ↪ */
    /************************************** ROUND 3 *************************************/
    LLR second_comp_0_to_7 = (second_winner_0_to_3 < second_winner_4_to_7) ? (*allones) :
        ↪ (*zeros);
    LLR second_comp_8_to_14 = (second_winner_8_to_11 < second_winner_12_to_14) ? (*allones
        ↪ ) : (*zeros);

    LLR second_winner_0_to_7 = (second_winner_0_to_3 & second_comp_0_to_7) | (
        ↪ second_winner_4_to_7 & (second_comp_0_to_7 ^ (*allones)));
    LLR second_winner_8_to_14 = (second_winner_8_to_11 & second_comp_8_to_14) | (
        ↪ second_winner_12_to_14 & (second_comp_8_to_14 ^ (*allones)));
    /*----------------------------------- END ROUND 3 -----------------------------------
        ↪ */

    /************************************** ROUND 4 *************************************/
    LLR second_comp_0_to_14 = (second_winner_0_to_7 < second_comp_8_to_14) ? (*allones) :
        ↪ (*zeros);

    LLR second_winner_0_to_14 = (second_winner_0_to_7 & second_comp_0_to_14) | (
        ↪ second_comp_8_to_14 & (second_comp_0_to_14 ^ (*allones)));
    /*----------------------------------- END ROUND 4 -----------------------------------
        ↪ */

    cnSecondMin = second_winner_0_to_14;

    // Now that we have the minimums and the parity, we should convert the first and
        ↪ second minimums to 2's complement assuming negative
    LLR first_min_neg = (((cnFirstMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));

    LLR second_min_neg = (((cnSecondMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));



    // Now let's start the check nodes
    // We only need to execute one check node for each calculation as there are
        ↪ N_CHECK_NODE number of work-items
    //#pragma unroll
    for(char input = 0; input < N_CHECK_INPUTS; input++) {



      // For each of these we need to determine if it was the first minimum
      // We just need a conditional that has either 1 or 0 depending if it's min or second
          ↪  min
      // And then we can make the expression LLR to_vnode = cnFirstMin*condition1 +
          ↪ cnSecondMin*inv(condition1)
      LLR min_conditional = ((cnInputs[input] & (*vector_max_llr)) == cnFirstMin) ? (*
          ↪ allones) : (*zeros);

      // the LLR conditional variable will now have a vector of either 0 or 1.
      // Now we can just use the expression

      // Now we can move on to calculating the output
      LLR sign_b = cnParity;

      // The sign bit is assigned to complete the parity compared to the other inputs
      sign_b ^= cnInputs[input] & (*vector_sign_bit);

      // Now we just send the data in 2's complement which was already calculated
      LLR sign_conditional = (sign_b == (*vector_sign_bit)) ? (*allones) : (*zeros);

      // If it's negative, give it the negative value, if not then the positive
      // Start with cnSecondMin, second_min_neg, cnFirstMin, first_min_neg
      LLR to_vnode =  ((min_conditional & sign_conditional & second_min_neg)) |
            ((min_conditional & (sign_conditional ^ (*allones)) & cnSecondMin)) |
            ((min_conditional ^ (*allones)) & sign_conditional & first_min_neg) |
            ((min_conditional ^ (*allones)) & (sign_conditional ^ (*allones)) & cnFirstMin
                ↪ );



      // Now we just need to find the variable node index
      short vnIndex = cntovnIndex[cnInputsNodeIndex + input] + cnMemoryOffsets[offsetIndex
          ↪ ];

      cnOutputsGM[vnIndex] = to_vnode;

      ////printf("Iteration,%d,CNode,%d,output,%d,%d,vnIndex,%d\n",iter,id,input,to_vnode,
          ↪ vnIndex);

    }
    barrier(CLK_GLOBAL_MEM_FENCE | CLK_CHANNEL_MEM_FENCE);
    offsetIndex ^= 1;
    if(id == 0)
      write_channel_altera(vnStart,SENDDATA);




  }

}
```

Appendix N

Single Work-Item (Design D) Irregular Length-5940 Kernel Source Code (OpenCL)

```
#ifndef N_VARIABLE_NODES
#define N_VARIABLE_NODES 5940
#endif

#ifndef N_CHECK_NODES
#define N_CHECK_NODES 900
#endif

#ifndef SIMD
#define SIMD 1
#endif

// Based on the Tanner graph, how many connections are there
#define MAX_VARIABLE_EDGES 5
#define MAX_CHECK_EDGES 27


// Let's define the usable number of inputs
#define N_VARIABLE_INPUTS (MAX_VARIABLE_EDGES)
#define N_CHECK_INPUTS (MAX_CHECK_EDGES)

#define N_VARIABLE_INPUTS_W_CHANNEL (N_VARIABLE_INPUTS + 1)

#define N_ITERATIONS (32)



// Set the vector width and the appropriate LLR types
#define MANUAL_VECTOR_WIDTH (16)

#if MANUAL_VECTOR_WIDTH==2
#define LLR_TYPE char2
#define TOTAL_TYPE short2
#define CONVERT_TO_SHORT(a) convert_short2(a)
#define CONVERT_TO_CHAR(a) convert_char2(a)
#elif MANUAL_VECTOR_WIDTH==4
#define LLR_TYPE char4
#define TOTAL_TYPE short4
#define CONVERT_TO_SHORT(a) convert_short4(a)
#define CONVERT_TO_CHAR(a) convert_char4(a)
#elif MANUAL_VECTOR_WIDTH==8
#define LLR_TYPE char8
#define TOTAL_TYPE short8
#define CONVERT_TO_SHORT(a) convert_short8(a)
#define CONVERT_TO_CHAR(a) convert_char8(a)
#elif MANUAL_VECTOR_WIDTH==16
#define LLR_TYPE char16
#define TOTAL_TYPE short16
#define CONVERT_TO_SHORT(a) convert_short16(a)
#define CONVERT_TO_CHAR(a) convert_char16(a)
#else
#define LLR_TYPE char
#define TOTAL_TYPE short
#define CONVERT_TO_SHORT(a) convert_short(a)
#define CONVERT_TO_CHAR(a) convert_char(a)
#endif
```

```
typedef LLR_TYPE LLR;


// Now for the channels
#pragma OPENCL EXTENSION cl_altera_channels : enable
channel char vnStart __attribute__((depth(N_ITERATIONS)));
channel char cnStart __attribute__((depth(N_ITERATIONS)));
#define SENDDATA (1)

//Variable Node Kernel
__kernel
__attribute__((reqd_work_group_size(N_VARIABLE_NODES,1,1)))
__attribute__((num_compute_units(1)))
__attribute__((num_simd_work_items(SIMD)))
void VariableNode(

// Kernel parameteres
// As in the notes page, there is a table that determines the write location in the CNs
__constant int * restrict vntocnIndex,

// Now that we have the connection information let's get the inputs
// Remember there is one memory location for the Variable Node inputs with two slots for
    ↪ pipelining
// Let's start with the channel measurements
__global LLR * restrict channelMeasurements,

// Now let's get the inputs
__global LLR * restrict vnInputsGM,

// Now let's get the output pointer to the CNs
__global LLR * restrict vnOutputsGM,

// And of course finally, we need to get the result
__global TOTAL_TYPE * restrict vnResults,


// We need a few constants where it's easier to compute them in the host
__constant LLR * restrict vector_max_llr,
__constant LLR * restrict vector_sign_bit,
__constant LLR * restrict allones,
__constant LLR * restrict ones,
__constant LLR * restrict zeros,
__constant TOTAL_TYPE * restrict total_vector_max_llr,
__constant TOTAL_TYPE * restrict total_vector_min_neg_llr
)
{

  // First we need to get our global IDsa
  int id = get_global_id(0);
  TOTAL_TYPE vnInputsShort[N_VARIABLE_INPUTS_W_CHANNEL];



  // Let's make an array for the offsets
  int vnMemoryOffsets[2];
  vnMemoryOffsets[0] = 0;
  vnMemoryOffsets[1] = N_VARIABLE_NODES * (N_VARIABLE_INPUTS+1);

  int vnChannelOffsets[2];
  vnChannelOffsets[0] = 0;
  vnChannelOffsets[1] = N_VARIABLE_NODES;

  int cnMemoryOffsets[2];
  cnMemoryOffsets[0] = 0;
  cnMemoryOffsets[1] = N_CHECK_NODES * (N_CHECK_INPUTS+1);

  // Let's make an array index lookup for the input


  // Some other variables


  // Now let's store the pointer
  char temp;
  char offsetIndex = 0;

  // Let's make an array index lookup for the input
  int vntocnNodeIndex = id * (N_VARIABLE_INPUTS);
  int vnGlobalNodeIndex = id * (N_VARIABLE_INPUTS+ 1);

  // Some other variables
  LLR vnCurrent;
  TOTAL_TYPE vnTempCurrent;
  LLR vnConditional;
  LLR vnSignB;
  LLR vnMag;

  //printf("VN,%d,prepped\n",id);

  // Let's start the process
  // We need to step through the iterations (aka start the loop)
  // Since both VN and CN will be running concurrently on the same data, we will need
       ↪ N_ITERATIONS*2+1
  for(char iter = 0; iter < ((N_ITERATIONS * 2) + 2); iter++) {

    //printf("VN Starting Iteration %d\n",iter);

    if(id == 0 && iter > 1)
      temp = read_channel_altera(vnStart);

    barrier(CLK_GLOBAL_MEM_FENCE);
    //printf("After channel read\n");



    // Let's just do some printing
    //printf("VN,%d,%d\n",id,iter);
    // Alright so each iteration we need to do the following:
    // Each VN:
    // 1) Wait for the OK from the CNs to start on the data
    // 2) Load up the inputs from DRAM into private memory
    // 3) Do the VN
    // 4) Output the data to the appropriate buffer
    // 5) Send the ok to the CN to start execution
    // 6) Swap buffers
    // Repeat







    //printf("VN,%d,%d,afterbarrier\n",id,iter);

    // Now that we've got the OK. Let's start execution
    // First let's load up the inputs and convert them to shorts so that we can get the
        ↪ appropriate totals without saturation
    #pragma unroll
    for(char input = 0; input < N_VARIABLE_INPUTS; input++) {
      vnInputsShort[input] = CONVERT_TO_SHORT(vnInputsGM[vnMemoryOffsets[offsetIndex] +
          ↪ vnGlobalNodeIndex + input]);
    }

    // Now let's put in the channel measurement for our node
    vnInputsShort[N_VARIABLE_INPUTS] = CONVERT_TO_SHORT(channelMeasurements[
        ↪ vnChannelOffsets[offsetIndex] + id]);

    // Now we need to add it all together

    TOTAL_TYPE total = (TOTAL_TYPE)(0);
    #pragma unroll
    for(char input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {
      total += vnInputsShort[input];
    }


    /*for(int i=0; i<N_VARIABLE_INPUTS_W_CHANNEL; i++) {
      printf("Iteration,%d,VNode,%d,input,%d,%d,%d,address,%d\n",iter,id,i,total.s0,
          ↪ vnInputsShort[i].s0,vnMemoryOffsets[offsetIndex] + vnGlobalNodeIndex + i);
    }*/

    // Now that we have the total we can do the rest
    for(char input = 0; input < N_VARIABLE_INPUTS; input++) {

      // Subtract the current from the total to get the output
      // We simply subtract the large total and then saturate it down to the appropriate
          ↪ size
      vnTempCurrent = total - vnInputsShort[input];

      // Now saturate
      vnTempCurrent =   ((vnTempCurrent >= (*total_vector_max_llr)) ?
                (*total_vector_max_llr) :
                ((vnTempCurrent <= (*total_vector_min_neg_llr)) ?
                (*total_vector_min_neg_llr) :
                vnTempCurrent));
      // Now we reverse the procedure and store the data back into the LLR type
      vnCurrent = CONVERT_TO_CHAR(vnTempCurrent);

      // Ok now we can finally send the data
      // First we check if it's negative
      // Except we need to convert it to sign and magnitude
      vnConditional = ((vnCurrent & (*vector_sign_bit)) == (*vector_sign_bit)) ? (*allones
          ↪ ) : (*zeros);

      // If it's negative we need to xor it and add one and mask out the sign bit
      vnMag = ((vnConditional & ((vnCurrent ^ (*allones)) + (*ones))) | ((vnConditional ^
          ↪ (*allones)) & vnCurrent)) & (*vector_max_llr);

      // Now get the sign bit
      vnSignB = (vnConditional & (*vector_sign_bit));

      // Now we just need to find the location in the array to place the data
      int cnIndex = vntocnIndex[input] + cnMemoryOffsets[offsetIndex];
```

```
      // Now we can just output the value
      // Since we're writing to the CNs we need to output to the write memory location
      vnOutputsGM[cnIndex] = vnMag | vnSignB;

      //printf("Iteration,%d,VNode,%d,output,%d,%d,cnIndex,%d,vntocnIndex,%d,offset,%d\n",
          ↪ iter,id,input,vnMag.s0 | vnSignB.s0,cnIndex,vntocnIndex[vntocnNodeIndex +
          ↪ input],cnMemoryOffsets[offsetIndex]);

    }

    if((iter+1)==N_ITERATIONS*2+1 || (iter + 1) == N_ITERATIONS*2 + 2) {
      TOTAL_TYPE send_total = 0;
      for(short input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {
        send_total += vnInputsShort[input];
      }

      vnResults[vnMemoryOffsets[offsetIndex] + id] = send_total;
    }



    //printf("VN,%d,%d,beforechannelwrite\n",id,iter);
    // Finally, we need to tell the check node it can operate on the data
    barrier(CLK_GLOBAL_MEM_FENCE);
    if(id==0)
      write_channel_altera(cnStart, SENDDATA);

    //barrier(CLK_GLOBAL_MEM_FENCE | CLK_CHANNEL_MEM_FENCE);

    // Now let's switch the index over
    // To do this we can simply XOR it with the value of 1
    offsetIndex ^= 1;




  }



}


__kernel
__attribute__((reqd_work_group_size(N_CHECK_NODES,1,1)))
__attribute__((num_compute_units(1)))
__attribute__((num_simd_work_items(SIMD)))
void CheckNode(

// Let's start with the table for the interleaver connections just like with the VN
__constant int * restrict cntovnIndex,

// Now we can do the other constants
__constant LLR * restrict vector_max_llr,
__constant LLR * restrict vector_sign_bit,
__constant LLR * restrict allones,
__constant LLR * restrict ones,
__constant LLR * restrict zeros,

// Now we can focus on the inputs
__global LLR * restrict cnInputsGM,

// Now the outputs
__global LLR * restrict cnOutputsGM,

// Now a place for the results
__global LLR * restrict cnResults

) {



  int id = get_global_id(0);
  //printf("CN,%d,started\n",id);

  // First we need to store a place in private memory for operation
  LLR cnInputs[N_CHECK_INPUTS];

  // And also a spot for the magnitudes
  LLR cnInputsMag[N_CHECK_INPUTS];

  // Ok. Now we need to do some similar work to the VNs.
  // Let's make an array for the offsets
  int vnMemoryOffsets[2];
  vnMemoryOffsets[0] = 0;
  vnMemoryOffsets[1] = N_VARIABLE_NODES * (N_VARIABLE_INPUTS+1);

  int cnMemoryOffsets[2];
  cnMemoryOffsets[0] = 0;
  cnMemoryOffsets[1] = N_CHECK_NODES * (N_CHECK_INPUTS+1);
```

```
// Check Node Private Variables
LLR cnFirstMin;
LLR cnSecondMin;
LLR cnParity;

// And let's store an offset
char offsetIndex = 0; // Note that we start on the same data as VN but we have to wait
    ↪ for the ramp up of the VN to complete the first iterations before we can execute

//printf("CN,%d,prepped\n",id);
char temp;

int cnInputsNodeIndex = id * N_CHECK_INPUTS;
int cnInputsGlobalIndex = id * (N_CHECK_INPUTS+1);

// Alright let's start the iterations
for(char iter = 0; iter < (N_ITERATIONS * 2); iter++) {

  if(id == 0)
    temp = read_channel_altera(cnStart);

  barrier(CLK_GLOBAL_MEM_FENCE);


  //printf("CN,%d,%d\n",id,iter);
  // For each iteration we need to:
  // 1) Wait for the ok from the VNs to start executing on the information
  // 2) Load in the new inputs from GM
  // 3) Do the Cn
  // 4) Send the data to the VN via GM
  // 5) Tell the VN we have finished


  // Now we can begin our execution
  cnFirstMin = (*vector_max_llr);
  cnSecondMin = (*vector_max_llr);

  // First let's do the minimum calculations

  // But first, we need to make sure that we are taking the absolute value as we don't
      ↪ want to include that in the min calculation
  // Therefore, we have to mask off the sign bits
  #pragma unroll
  for(short input = 0; input < N_CHECK_INPUTS; input++) {
    cnInputs[input] = cnInputsGM[cnMemoryOffsets[offsetIndex] + cnInputsGlobalIndex +
        ↪ input];
  }

  // Now that we have the normal inputs, we need to get the magnitudes
  #pragma unroll
  for(short input = 0; input < N_CHECK_INPUTS; input++) {
    cnInputsMag[input] = cnInputs[input] & (*vector_max_llr);
  }

  // Ok now that we have the magnitudes, let's get to work by getting the overall parity
      ↪  first
  cnParity = (*zeros);
  #pragma unroll
  for(short input = 0; input < N_CHECK_INPUTS; input++) {
    cnParity ^= cnInputs[input] & (*vector_sign_bit);
  }

  // Now cnParity holds the overall parity of the check node

  // We're going to try something a little different here
  // We're going to implement the tree using min on the magnitudes

  // This is a little trickier as we need to find the losers a lot
  // But let's start it out.


  /************************************ ROUND 1 ************************************/
  LLR comp_0_to_1 = (cnInputsMag[0] < cnInputsMag[1]) ? (*allones) : (*zeros);
  LLR comp_2_to_3 = (cnInputsMag[2] < cnInputsMag[3]) ? (*allones) : (*zeros);
  LLR comp_4_to_5 = (cnInputsMag[4] < cnInputsMag[5]) ? (*allones) : (*zeros);
  LLR comp_6_to_7 = (cnInputsMag[6] < cnInputsMag[7]) ? (*allones) : (*zeros);
  LLR comp_8_to_9 = (cnInputsMag[8] < cnInputsMag[9]) ? (*allones) : (*zeros);
  LLR comp_10_to_11 = (cnInputsMag[10] < cnInputsMag[11]) ? (*allones) : (*zeros);
  LLR comp_12_to_13 = (cnInputsMag[12] < cnInputsMag[13]) ? (*allones) : (*zeros);
  LLR comp_14_to_15 = (cnInputsMag[14] < cnInputsMag[15]) ? (*allones) : (*zeros);
  LLR comp_16_to_17 = (cnInputsMag[16] < cnInputsMag[17]) ? (*allones) : (*zeros);
  LLR comp_18_to_19 = (cnInputsMag[18] < cnInputsMag[19]) ? (*allones) : (*zeros);
  LLR comp_20_to_21 = (cnInputsMag[20] < cnInputsMag[21]) ? (*allones) : (*zeros);
  LLR comp_22_to_23 = (cnInputsMag[22] < cnInputsMag[23]) ? (*allones) : (*zeros);
  LLR comp_24_to_25 = (cnInputsMag[24] < cnInputsMag[25]) ? (*allones) : (*zeros);



  // First Minimum
  LLR winner_0_to_1 = (cnInputsMag[0] & comp_0_to_1) | (cnInputsMag[1] & (comp_0_to_1 ^
      ↪ (*allones)));
  // Second Minimum
```

```
LLR loser_0 = (cnInputsMag[1] & comp_0_to_1) | (cnInputsMag[0] & (comp_0_to_1 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_2_to_3 = (cnInputsMag[2] & comp_2_to_3) | (cnInputsMag[3] & (comp_2_to_3 ^
    ↪ (*allones)));
// Second Minimum
LLR loser_1 = (cnInputsMag[3] & comp_2_to_3) | (cnInputsMag[2] & (comp_2_to_3 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_4_to_5 = (cnInputsMag[4] & comp_4_to_5) | (cnInputsMag[5] & (comp_4_to_5 ^
    ↪ (*allones)));
// Second Minimum
LLR loser_2 = (cnInputsMag[5] & comp_4_to_5) | (cnInputsMag[4] & (comp_4_to_5 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_6_to_7 = (cnInputsMag[6] & comp_6_to_7) | (cnInputsMag[7] & (comp_6_to_7 ^
    ↪ (*allones)));
// Second Minimum
LLR loser_3 = (cnInputsMag[7] & comp_6_to_7) | (cnInputsMag[6] & (comp_6_to_7 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_8_to_9 = (cnInputsMag[8] & comp_8_to_9) | (cnInputsMag[9] & (comp_8_to_9 ^
    ↪ (*allones)));
// Second Minimum
LLR loser_4 = (cnInputsMag[9] & comp_8_to_9) | (cnInputsMag[8] & (comp_8_to_9 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_10_to_11 = (cnInputsMag[10] & comp_10_to_11) | (cnInputsMag[11] & (
    ↪ comp_10_to_11 ^ (*allones)));
// Second Minimum
LLR loser_5 = (cnInputsMag[11] & comp_10_to_11) | (cnInputsMag[10] & (comp_10_to_11 ^
    ↪ (*allones)));


// First Minimum
LLR winner_12_to_13 = (cnInputsMag[12] & comp_12_to_13) | (cnInputsMag[13] & (
    ↪ comp_12_to_13 ^ (*allones)));
// Second Minimum
LLR loser_6 = (cnInputsMag[13] & comp_12_to_13) | (cnInputsMag[12] & (comp_12_to_13 ^
    ↪ (*allones)));


// First Minimum
LLR winner_14_to_15 = (cnInputsMag[14] & comp_14_to_15) | (cnInputsMag[15] & (
    ↪ comp_14_to_15 ^ (*allones)));
// Second Minimum
LLR loser_7 = (cnInputsMag[15] & comp_14_to_15) | (cnInputsMag[14] & (comp_14_to_15 ^
    ↪ (*allones)));


// First Minimum
LLR winner_16_to_17 = (cnInputsMag[16] & comp_16_to_17) | (cnInputsMag[17] & (
    ↪ comp_16_to_17 ^ (*allones)));
// Second Minimum
LLR loser_8 = (cnInputsMag[17] & comp_16_to_17) | (cnInputsMag[16] & (comp_16_to_17 ^
    ↪ (*allones)));


// First Minimum
LLR winner_18_to_19 = (cnInputsMag[18] & comp_18_to_19) | (cnInputsMag[19] & (
    ↪ comp_18_to_19 ^ (*allones)));
// Second Minimum
LLR loser_9 = (cnInputsMag[19] & comp_18_to_19) | (cnInputsMag[18] & (comp_18_to_19 ^
    ↪ (*allones)));


// First Minimum
LLR winner_20_to_21 = (cnInputsMag[20] & comp_20_to_21) | (cnInputsMag[21] & (
    ↪ comp_20_to_21 ^ (*allones)));
// Second Minimum
LLR loser_10  = (cnInputsMag[21] & comp_20_to_21) | (cnInputsMag[20] & (comp_20_to_21
    ↪ ^ (*allones)));


// First Minimum
LLR winner_22_to_23 = (cnInputsMag[22] & comp_22_to_23) | (cnInputsMag[23] & (
    ↪ comp_22_to_23 ^ (*allones)));
// Second Minimum
LLR loser_11  = (cnInputsMag[23] & comp_22_to_23) | (cnInputsMag[22] & (comp_22_to_23
    ↪ ^ (*allones)));
```

```
// First Minimum
LLR winner_24_to_25 = (cnInputsMag[24] & comp_24_to_25) | (cnInputsMag[25] & (
     ↪ comp_24_to_25 ^ (*allones)));
// Second Minimum
LLR loser_12  = (cnInputsMag[25] & comp_24_to_25) | (cnInputsMag[24] & (comp_24_to_25
     ↪ ^ (*allones)));


/*----------------------------------- END ROUND 1 -----------------------------------
     ↪ */




/************************************ ROUND 2 ************************************/
LLR comp_0_to_3 = (winner_0_to_1 < winner_2_to_3) ? (*allones) : (*zeros);
LLR comp_4_to_7 = (winner_4_to_5 < winner_6_to_7) ? (*allones) : (*zeros);
LLR comp_8_to_11  = (winner_8_to_9 < winner_10_to_11) ? (*allones) : (*zeros);
LLR comp_12_to_15 = (winner_12_to_13 < winner_14_to_15) ? (*allones) : (*zeros);
LLR comp_16_to_19 = (winner_16_to_17 < winner_18_to_19) ? (*allones) : (*zeros);
LLR comp_20_to_23 = (winner_20_to_21 < winner_22_to_23) ? (*allones) : (*zeros);
LLR comp_24_to_26 = (winner_24_to_25 < cnInputsMag[26]) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_3 = (winner_0_to_1 & comp_0_to_3) | (winner_2_to_3 & (comp_0_to_3 ^ (*
     ↪ allones)));
// Second Minimum
LLR loser_13  = (winner_2_to_3 & comp_0_to_3) | (winner_0_to_1 & (comp_0_to_3 ^ (*
     ↪ allones)));


// First Minimum
LLR winner_4_to_7 = (winner_4_to_5 & comp_4_to_7) | (winner_6_to_7 & (comp_4_to_7 ^ (*
     ↪ allones)));
// Second Minimum
LLR loser_14  = (winner_6_to_7 & comp_4_to_7) | (winner_4_to_5 & (comp_4_to_7 ^ (*
     ↪ allones)));


// First Minimum
LLR winner_8_to_11  = (winner_8_to_9 & comp_8_to_11) | (winner_10_to_11 & (
     ↪ comp_8_to_11 ^ (*allones)));
// Second Minimum
LLR loser_15  = (winner_10_to_11 & comp_8_to_11) | (winner_8_to_9 & (comp_8_to_11 ^ (*
     ↪ allones)));


// First Minimum
LLR winner_12_to_15 = (winner_12_to_13 & comp_12_to_15) | (winner_14_to_15 & (
     ↪ comp_12_to_15 ^ (*allones)));
// Second Minimum
LLR loser_16  = (winner_14_to_15 & comp_12_to_15) | (winner_12_to_13 & (comp_12_to_15
     ↪ ^ (*allones)));


// First Minimum
LLR winner_16_to_19 = (winner_16_to_17 & comp_16_to_19) | (winner_18_to_19 & (
     ↪ comp_16_to_19 ^ (*allones)));
// Second Minimum
LLR loser_17  = (winner_18_to_19 & comp_16_to_19) | (winner_16_to_17 & (comp_16_to_19
     ↪ ^ (*allones)));


// First Minimum
LLR winner_20_to_23 = (winner_20_to_21 & comp_20_to_23) | (winner_22_to_23 & (
     ↪ comp_20_to_23 ^ (*allones)));
// Second Minimum
LLR loser_18  = (winner_22_to_23 & comp_20_to_23) | (winner_20_to_21 & (comp_20_to_23
     ↪ ^ (*allones)));


// First Minimum
LLR winner_24_to_26 = (winner_24_to_25 & comp_24_to_26) | (cnInputsMag[26] & (
     ↪ comp_24_to_26 ^ (*allones)));
// Second Minimum
LLR loser_19  = (cnInputsMag[26] & comp_24_to_26) | (winner_24_to_25 & (comp_24_to_26
     ↪ ^ (*allones)));


/*----------------------------------- END ROUND 2 -----------------------------------
     ↪ */




/************************************ ROUND 3 ************************************/
LLR comp_0_to_7 = (winner_0_to_3 < winner_4_to_7) ? (*allones) : (*zeros);
LLR comp_8_to_15  = (winner_8_to_11 < winner_12_to_15) ? (*allones) : (*zeros);
LLR comp_16_to_23 = (winner_16_to_19 < winner_20_to_23) ? (*allones) : (*zeros);
```

```
// First Minimum
LLR winner_0_to_7 = (winner_0_to_3 & comp_0_to_7) | (winner_4_to_7 & (comp_0_to_7 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_20  = (winner_4_to_7 & comp_0_to_7) | (winner_0_to_3 & (comp_0_to_7 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_8_to_15  = (winner_8_to_11 & comp_8_to_15) | (winner_12_to_15 & (
    ↪ comp_8_to_15 ^ (*allones)));
// Second Minimum
LLR loser_21  = (winner_12_to_15 & comp_8_to_15) | (winner_8_to_11 & (comp_8_to_15 ^
    ↪ (*allones)));


// First Minimum
LLR winner_16_to_23 = (winner_16_to_19 & comp_16_to_23) | (winner_20_to_23 & (
    ↪ comp_16_to_23 ^ (*allones)));
// Second Minimum
LLR loser_22  = (winner_20_to_23 & comp_16_to_23) | (winner_16_to_19 & (comp_16_to_23
    ↪ ^ (*allones)));


/*----------------------------------- END ROUND 3 -----------------------------------
    ↪ */




/*************************************** ROUND 4 **************************************/
LLR comp_0_to_15  = (winner_0_to_7 < winner_8_to_15) ? (*allones) : (*zeros);
LLR comp_16_to_26 = (winner_16_to_23 < winner_24_to_26) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_15  = (winner_0_to_7 & comp_0_to_15) | (winner_8_to_15 & (comp_0_to_15
    ↪  ^ (*allones)));
// Second Minimum
LLR loser_23  = (winner_8_to_15 & comp_0_to_15) | (winner_0_to_7 & (comp_0_to_15 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_16_to_26 = (winner_16_to_23 & comp_16_to_26) | (winner_24_to_26 & (
    ↪ comp_16_to_26 ^ (*allones)));
// Second Minimum
LLR loser_24  = (winner_24_to_26 & comp_16_to_26) | (winner_16_to_23 & (comp_16_to_26
    ↪ ^ (*allones)));


/*----------------------------------- END ROUND 4 -----------------------------------
    ↪ */




/*************************************** ROUND 5 **************************************/
LLR comp_0_to_26  = (winner_0_to_15 < winner_16_to_26) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_26  = (winner_0_to_15 & comp_0_to_26) | (winner_16_to_26 & (
    ↪ comp_0_to_26 ^ (*allones)));
// Second Minimum
LLR loser_25  = (winner_16_to_26 & comp_0_to_26) | (winner_0_to_15 & (comp_0_to_26 ^
    ↪ (*allones)));


/*----------------------------------- END ROUND 5 -----------------------------------
    ↪ */




cnFirstMin = winner_0_to_26;





/******************************** SECOND MINIMUM **********************************/


/*************************************** ROUND 1 **************************************/
LLR second_comp_0_to_1  = (loser_0  < loser_1) ? (*allones) : (*zeros);
LLR second_comp_2_to_3  = (loser_2  < loser_3) ? (*allones) : (*zeros);
LLR second_comp_4_to_5  = (loser_4  < loser_5) ? (*allones) : (*zeros);
LLR second_comp_6_to_7  = (loser_6  < loser_7) ? (*allones) : (*zeros);
LLR second_comp_8_to_9  = (loser_8  < loser_9) ? (*allones) : (*zeros);
LLR second_comp_10_to_11  = (loser_10  < loser_11) ? (*allones) : (*zeros);
LLR second_comp_12_to_13  = (loser_12  < loser_13) ? (*allones) : (*zeros);
LLR second_comp_14_to_15  = (loser_14  < loser_15) ? (*allones) : (*zeros);
LLR second_comp_16_to_17  = (loser_16  < loser_17) ? (*allones) : (*zeros);
LLR second_comp_18_to_19  = (loser_18  < loser_19) ? (*allones) : (*zeros);
LLR second_comp_20_to_21  = (loser_20  < loser_21) ? (*allones) : (*zeros);
LLR second_comp_22_to_23  = (loser_22  < loser_23) ? (*allones) : (*zeros);
LLR second_comp_24_to_25  = (loser_24  < loser_25) ? (*allones) : (*zeros);



LLR second_winner_0_to_1  = (loser_0 & second_comp_0_to_1) | (loser_1 & (
    ↪ second_comp_0_to_1 ^ (*allones)));
LLR second_winner_2_to_3  = (loser_2 & second_comp_2_to_3) | (loser_3 & (
    ↪ second_comp_2_to_3 ^ (*allones)));
LLR second_winner_4_to_5  = (loser_4 & second_comp_4_to_5) | (loser_5 & (
    ↪ second_comp_4_to_5 ^ (*allones)));
LLR second_winner_6_to_7  = (loser_6 & second_comp_6_to_7) | (loser_7 & (
    ↪ second_comp_6_to_7 ^ (*allones)));
LLR second_winner_8_to_9  = (loser_8 & second_comp_8_to_9) | (loser_9 & (
    ↪ second_comp_8_to_9 ^ (*allones)));
LLR second_winner_10_to_11  = (loser_10 & second_comp_10_to_11) | (loser_11 & (
    ↪ second_comp_10_to_11 ^ (*allones)));
LLR second_winner_12_to_13  = (loser_12 & second_comp_12_to_13) | (loser_13 & (
    ↪ second_comp_12_to_13 ^ (*allones)));
LLR second_winner_14_to_15  = (loser_14 & second_comp_14_to_15) | (loser_15 & (
    ↪ second_comp_14_to_15 ^ (*allones)));
LLR second_winner_16_to_17  = (loser_16 & second_comp_16_to_17) | (loser_17 & (
    ↪ second_comp_16_to_17 ^ (*allones)));
LLR second_winner_18_to_19  = (loser_18 & second_comp_18_to_19) | (loser_19 & (
    ↪ second_comp_18_to_19 ^ (*allones)));
LLR second_winner_20_to_21  = (loser_20 & second_comp_20_to_21) | (loser_21 & (
    ↪ second_comp_20_to_21 ^ (*allones)));
LLR second_winner_22_to_23  = (loser_22 & second_comp_22_to_23) | (loser_23 & (
    ↪ second_comp_22_to_23 ^ (*allones)));
LLR second_winner_24_to_25  = (loser_24 & second_comp_24_to_25) | (loser_25 & (
    ↪ second_comp_24_to_25 ^ (*allones)));
/*---------------------------------- END ROUND 1 ----------------------------------
    ↪ */




/*********************************** ROUND 2 ***********************************/
LLR second_comp_0_to_3  = (second_winner_0_to_1 < second_winner_2_to_3) ? (*allones) :
    ↪  (*zeros);
LLR second_comp_4_to_7  = (second_winner_4_to_5 < second_winner_6_to_7) ? (*allones) :
    ↪  (*zeros);
LLR second_comp_8_to_11 = (second_winner_8_to_9 < second_winner_10_to_11) ? (*allones)
    ↪  : (*zeros);
LLR second_comp_12_to_15  = (second_winner_12_to_13 < second_winner_14_to_15) ? (*
    ↪ allones) : (*zeros);
LLR second_comp_16_to_19  = (second_winner_16_to_17 < second_winner_18_to_19) ? (*
    ↪ allones) : (*zeros);
LLR second_comp_20_to_23  = (second_winner_20_to_21 < second_winner_22_to_23) ? (*
    ↪ allones) : (*zeros);



LLR second_winner_0_to_3  = (second_winner_0_to_1 & second_comp_0_to_3) | (
    ↪ second_winner_2_to_3 & (second_comp_0_to_3 ^ (*allones)));
LLR second_winner_4_to_7  = (second_winner_4_to_5 & second_comp_4_to_7) | (
    ↪ second_winner_6_to_7 & (second_comp_4_to_7 ^ (*allones)));
LLR second_winner_8_to_11 = (second_winner_8_to_9 & second_comp_8_to_11) | (
    ↪ second_winner_10_to_11 & (second_comp_8_to_11 ^ (*allones)));
LLR second_winner_12_to_15  = (second_winner_12_to_13 & second_comp_12_to_15) | (
    ↪ second_winner_14_to_15 & (second_comp_12_to_15 ^ (*allones)));
LLR second_winner_16_to_19  = (second_winner_16_to_17 & second_comp_16_to_19) | (
    ↪ second_winner_18_to_19 & (second_comp_16_to_19 ^ (*allones)));
LLR second_winner_20_to_23  = (second_winner_20_to_21 & second_comp_20_to_23) | (
    ↪ second_winner_22_to_23 & (second_comp_20_to_23 ^ (*allones)));
/*---------------------------------- END ROUND 2 ----------------------------------
    ↪ */




/*********************************** ROUND 3 ***********************************/
LLR second_comp_0_to_7  = (second_winner_0_to_3 < second_winner_4_to_7) ? (*allones) :
    ↪  (*zeros);
LLR second_comp_8_to_15 = (second_winner_8_to_11 < second_winner_12_to_15) ? (*allones
    ↪ ) : (*zeros);
LLR second_comp_16_to_23  = (second_winner_16_to_19 < second_winner_20_to_23) ? (*
    ↪ allones) : (*zeros);



LLR second_winner_0_to_7  = (second_winner_0_to_3 & second_comp_0_to_7) | (
    ↪ second_winner_4_to_7 & (second_comp_0_to_7 ^ (*allones)));
```

```
LLR second_winner_8_to_15 = (second_winner_8_to_11 & second_comp_8_to_15) | (
    ↪ second_winner_12_to_15 & (second_comp_8_to_15 ^ (*allones)));
LLR second_winner_16_to_23  = (second_winner_16_to_19 & second_comp_16_to_23) | (
    ↪ second_winner_20_to_23 & (second_comp_16_to_23 ^ (*allones)));
/*---------------------------------- END ROUND 3 ----------------------------------
    ↪ */




/*********************************** ROUND 4 ***********************************/
LLR second_comp_0_to_15 = (second_winner_0_to_7 < second_winner_8_to_15) ? (*allones)
    ↪ : (*zeros);
LLR second_comp_16_to_25  = (second_winner_16_to_23 < second_winner_24_to_25) ? (*
    ↪ allones) : (*zeros);



LLR second_winner_0_to_15 = (second_winner_0_to_7 & second_comp_0_to_15) | (
    ↪ second_winner_8_to_15 & (second_comp_0_to_15 ^ (*allones)));
LLR second_winner_16_to_25  = (second_winner_16_to_23 & second_comp_16_to_25) | (
    ↪ second_winner_24_to_25 & (second_comp_16_to_25 ^ (*allones)));
/*---------------------------------- END ROUND 4 ----------------------------------
    ↪ */




/*********************************** ROUND 5 ***********************************/
LLR second_comp_0_to_25 = (second_winner_0_to_15 < second_winner_16_to_25) ? (*allones
    ↪ ) : (*zeros);



LLR second_winner_0_to_25 = (second_winner_0_to_15 & second_comp_0_to_25) | (
    ↪ second_winner_16_to_25 & (second_comp_0_to_25 ^ (*allones)));
/*---------------------------------- END ROUND 5 ----------------------------------
    ↪ */




cnSecondMin = second_winner_0_to_25;




// Now that we have the minimums and the parity, we should convert the first and
    ↪ second minimums to 2's complement assuming negative
LLR first_min_neg = (((cnFirstMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));

LLR second_min_neg = (((cnSecondMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));

/*for(int i=0; i<N_CHECK_INPUTS; i++) {
  printf("Iteration,%d,CNode,%d,input,%d,%d,%d,min,%d,second_min,%d,address,%d\n",iter
      ↪ ,id,i,cnInputs[i].s0 & (*vector_sign_bit).s0,cnInputsMag[i].s0,cnFirstMin.s0
      ↪ ,cnSecondMin.s0,cnMemoryOffsets[offsetIndex] + cnInputsGlobalIndex + i);
}*/




// Now let's start the check nodes
// We only need to execute one check node for each calculation as there are
    ↪ N_CHECK_NODE number of work-items
//#pragma unroll
for(char input = 0; input < N_CHECK_INPUTS; input++) {



  // For each of these we need to determine if it was the first minimum
  // We just need a conditional that has either 1 or 0 depending if it's min or second
      ↪ min
  // And then we can make the expression LLR to_vnode = cnFirstMin*condition1 +
      ↪ cnSecondMin*inv(condition1)
  LLR min_conditional = ((cnInputs[input] & (*vector_max_llr)) == cnFirstMin) ? (*
      ↪ allones) : (*zeros);

  // the LLR conditional variable will now have a vector of either 0 or 1.
  // Now we can just use the expression

  // Now we can move on to calculating the output
  LLR sign_b = cnParity;

  // The sign bit is assigned to complete the parity compared to the other inputs
  sign_b ^= cnInputs[input] & (*vector_sign_bit);

  // Now we just send the data in 2's complement which was already calculated
  LLR sign_conditional = (sign_b == (*vector_sign_bit)) ? (*allones) : (*zeros);

  // If it's negative, give it the negative value, if not then the positive
  // Start with cnSecondMin, second_min_neg, cnFirstMin, first_min_neg
```

```
      LLR to_vnode =  ((min_conditional & sign_conditional & second_min_neg)) |
            ((min_conditional & (sign_conditional ^ (*allones)) & cnSecondMin)) |
            ((min_conditional ^ (*allones)) & sign_conditional & first_min_neg) |
            ((min_conditional ^ (*allones)) & (sign_conditional ^ (*allones)) & cnFirstMin
                ↪ );



      // Now we just need to find the variable node index
      int vnIndex = cntovnIndex[cnInputsNodeIndex + input] + vnMemoryOffsets[offsetIndex];

      cnOutputsGM[vnIndex] = to_vnode;

      //printf("Iteration,%d,CNode,%d,output,%d,%d,vnIndex,%d,\n",iter,id,input,to_vnode.
          ↪ s0,vnIndex);

    }






    barrier(CLK_GLOBAL_MEM_FENCE);
    offsetIndex ^= 1;
    if(id == 0)
      write_channel_altera(vnStart,SENDDATA);




  }

}
```

Appendix O

Single Work-Item (Design D) Irregular Length-16200 Kernel Source Code (OpenCL)

```
#ifndef N_VARIABLE_NODES
#define N_VARIABLE_NODES 16200
#endif

#ifndef N_CHECK_NODES
#define N_CHECK_NODES 1800
#endif

#ifndef SIMD
#define SIMD 1
#endif

// Based on the Tanner graph, how many connections are there
#define MAX_VARIABLE_EDGES 5
#define MAX_CHECK_EDGES 34


// Let's define the usable number of inputs
#define N_VARIABLE_INPUTS (MAX_VARIABLE_EDGES)
#define N_CHECK_INPUTS (MAX_CHECK_EDGES)

#define N_VARIABLE_INPUTS_W_CHANNEL (N_VARIABLE_INPUTS + 1)

#define N_ITERATIONS (32)



// Set the vector width and the appropriate LLR types
#define MANUAL_VECTOR_WIDTH (16)

#if MANUAL_VECTOR_WIDTH==2
#define LLR_TYPE char2
#define TOTAL_TYPE short2
#define CONVERT_TO_SHORT(a) convert_short2(a)
#define CONVERT_TO_CHAR(a) convert_char2(a)
#elif MANUAL_VECTOR_WIDTH==4
#define LLR_TYPE char4
#define TOTAL_TYPE short4
#define CONVERT_TO_SHORT(a) convert_short4(a)
#define CONVERT_TO_CHAR(a) convert_char4(a)
#elif MANUAL_VECTOR_WIDTH==8
#define LLR_TYPE char8
#define TOTAL_TYPE short8
#define CONVERT_TO_SHORT(a) convert_short8(a)
#define CONVERT_TO_CHAR(a) convert_char8(a)
#elif MANUAL_VECTOR_WIDTH==16
#define LLR_TYPE char16
#define TOTAL_TYPE short16
#define CONVERT_TO_SHORT(a) convert_short16(a)
#define CONVERT_TO_CHAR(a) convert_char16(a)
#else
#define LLR_TYPE char
#define TOTAL_TYPE short
#define CONVERT_TO_SHORT(a) convert_short(a)
#define CONVERT_TO_CHAR(a) convert_char(a)
#endif
```

```
typedef LLR_TYPE LLR;


// Now for the channels
#pragma OPENCL EXTENSION cl_altera_channels : enable
channel char vnStart __attribute__((depth(N_ITERATIONS)));
channel char cnStart __attribute__((depth(N_ITERATIONS)));
#define SENDDATA (1)

//Variable Node Kernel
__kernel
__attribute__((reqd_work_group_size(N_VARIABLE_NODES,1,1)))
__attribute__((num_compute_units(1)))
__attribute__((num_simd_work_items(SIMD)))
void VariableNode(

// Kernel parameteres
// As in the notes page, there is a table that determines the write location in the CNs
__constant int * restrict vntocnIndex,

// Now that we have the connection information let's get the inputs
// Remember there is one memory location for the Variable Node inputs with two slots for
    ↪ pipelining
// Let's start with the channel measurements
__global LLR * restrict channelMeasurements,

// Now let's get the inputs
__global LLR * restrict vnInputsGM,

// Now let's get the output pointer to the CNs
__global LLR * restrict vnOutputsGM,

// And of course finally, we need to get the result
__global TOTAL_TYPE * restrict vnResults,


// We need a few constants where it's easier to compute them in the host
__constant LLR * restrict vector_max_llr,
__constant LLR * restrict vector_sign_bit,
__constant LLR * restrict allones,
__constant LLR * restrict ones,
__constant LLR * restrict zeros,
__constant TOTAL_TYPE * restrict total_vector_max_llr,
__constant TOTAL_TYPE * restrict total_vector_min_neg_llr
)
{

  // First we need to get our global IDsa
  int id = get_global_id(0);
  printf("VN,%d,Started\n",id);
  TOTAL_TYPE vnInputsShort[N_VARIABLE_INPUTS_W_CHANNEL];



  // Let's make an array for the offsets
  int vnMemoryOffsets[2];
  vnMemoryOffsets[0] = 0;
  vnMemoryOffsets[1] = N_VARIABLE_NODES * (N_VARIABLE_INPUTS+1);

  int vnChannelOffsets[2];
  vnChannelOffsets[0] = 0;
  vnChannelOffsets[1] = N_VARIABLE_NODES;

  int cnMemoryOffsets[2];
  cnMemoryOffsets[0] = 0;
  cnMemoryOffsets[1] = N_CHECK_NODES * (N_CHECK_INPUTS+1);

  // Let's make an array index lookup for the input


  // Some other variables


  // Now let's store the pointer
  char temp;
  char offsetIndex = 0;

  // Let's make an array index lookup for the input
  int vntocnNodeIndex = id * (N_VARIABLE_INPUTS);
  int vnGlobalNodeIndex = id * (N_VARIABLE_INPUTS+ 1);

  // Some other variables
  LLR vnCurrent;
  TOTAL_TYPE vnTempCurrent;
  LLR vnConditional;
  LLR vnSignB;
  LLR vnMag;

  //printf("VN,%d,prepped\n",id);

  // Let's start the process
  // We need to step through the iterations (aka start the loop)
  // Since both VN and CN will be running concurrently on the same data, we will need
       ↪ N_ITERATIONS*2+1
  for(char iter = 0; iter < ((N_ITERATIONS * 2) + 2); iter++) {

    //printf("VN Starting Iteration %d\n",iter);

    if(id == 0 && iter > 1)
      temp = read_channel_altera(vnStart);

    barrier(CLK_GLOBAL_MEM_FENCE);
    //printf("After channel read\n");



    // Let's just do some printing
    printf("VN,%d,%d\n",id,iter);
    // Alright so each iteration we need to do the following:
    // Each VN:
    // 1) Wait for the OK from the CNs to start on the data
    // 2) Load up the inputs from DRAM into private memory
    // 3) Do the VN
    // 4) Output the data to the appropriate buffer
    // 5) Send the ok to the CN to start execution
    // 6) Swap buffers
    // Repeat






    //printf("VN,%d,%d,afterbarrier\n",id,iter);

    // Now that we've got the OK. Let's start execution
    // First let's load up the inputs and convert them to shorts so that we can get the
        ↪ appropriate totals without saturation
    #pragma unroll
    for(char input = 0; input < N_VARIABLE_INPUTS; input++) {
      vnInputsShort[input] = CONVERT_TO_SHORT(vnInputsGM[vnMemoryOffsets[offsetIndex] +
          ↪ vnGlobalNodeIndex + input]);
    }

    // Now let's put in the channel measurement for our node
    vnInputsShort[N_VARIABLE_INPUTS] = CONVERT_TO_SHORT(channelMeasurements[
        ↪ vnChannelOffsets[offsetIndex] + id]);

    // Now we need to add it all together

    TOTAL_TYPE total = (TOTAL_TYPE)(0);
    #pragma unroll
    for(char input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {
      total += vnInputsShort[input];
    }


    /*for(int i=0; i<N_VARIABLE_INPUTS_W_CHANNEL; i++) {
      printf("Iteration,%d,VNode,%d,input,%d,%d,%d,address,%d\n",iter,id,i,total.s0,
          ↪ vnInputsShort[i].s0,vnMemoryOffsets[offsetIndex] + vnGlobalNodeIndex + i);
    }*/

    // Now that we have the total we can do the rest
    for(char input = 0; input < N_VARIABLE_INPUTS; input++) {

      // Subtract the current from the total to get the output
      // We simply subtract the large total and then saturate it down to the appropriate
          ↪ size
      vnTempCurrent = total - vnInputsShort[input];

      // Now saturate
      vnTempCurrent =   ((vnTempCurrent >= (*total_vector_max_llr)) ?
                  (*total_vector_max_llr) :
                  ((vnTempCurrent <= (*total_vector_min_neg_llr)) ?
                  (*total_vector_min_neg_llr) :
                  vnTempCurrent));
      // Now we reverse the procedure and store the data back into the LLR type
      vnCurrent = CONVERT_TO_CHAR(vnTempCurrent);

      // Ok now we can finally send the data
      // First we check if it's negative
      // Except we need to convert it to sign and magnitude
      vnConditional = ((vnCurrent & (*vector_sign_bit)) == (*vector_sign_bit)) ? (*allones
          ↪ ) : (*zeros);

      // If it's negative we need to xor it and add one and mask out the sign bit
      vnMag = ((vnConditional & ((vnCurrent ^ (*allones)) + (*ones))) | ((vnConditional ^
          ↪ (*allones)) & vnCurrent)) & (*vector_max_llr);

      // Now get the sign bit
      vnSignB = (vnConditional & (*vector_sign_bit));

      // Now we just need to find the location in the array to place the data
      int cnIndex = vntocnIndex[input] + cnMemoryOffsets[offsetIndex];
```

```
      // Now we can just output the value
      // Since we're writing to the CNs we need to output to the write memory location
      vnOutputsGM[cnIndex] = vnMag | vnSignB;

      //printf("Iteration,%d,VNode,%d,output,%d,%d,cnIndex,%d,vntocnIndex,%d,offset,%d\n",
          ↪ iter,id,input,vnMag.s0 | vnSignB.s0,cnIndex,vntocnIndex[vntocnNodeIndex +
          ↪ input],cnMemoryOffsets[offsetIndex]);

    }

    if((iter+1)==N_ITERATIONS*2+1 || (iter + 1) == N_ITERATIONS*2 + 2) {
      TOTAL_TYPE send_total = 0;
      for(short input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {
        send_total += vnInputsShort[input];
      }

      vnResults[vnMemoryOffsets[offsetIndex] + id] = send_total;
    }



    //printf("VN,%d,%d,beforechannelwrite\n",id,iter);
    // Finally, we need to tell the check node it can operate on the data
    barrier(CLK_GLOBAL_MEM_FENCE);
    if(id==0)
      write_channel_altera(cnStart, SENDDATA);

    //barrier(CLK_GLOBAL_MEM_FENCE | CLK_CHANNEL_MEM_FENCE);

    // Now let's switch the index over
    // To do this we can simply XOR it with the value of 1
    offsetIndex ^= 1;




  }



}


__kernel
__attribute__((reqd_work_group_size(N_CHECK_NODES,1,1)))
__attribute__((num_compute_units(1)))
__attribute__((num_simd_work_items(SIMD)))
void CheckNode(

// Let's start with the table for the interleaver connections just like with the VN
__constant int * restrict cntovnIndex,

// Now we can do the other constants
__constant LLR * restrict vector_max_llr,
__constant LLR * restrict vector_sign_bit,
__constant LLR * restrict allones,
__constant LLR * restrict ones,
__constant LLR * restrict zeros,

// Now we can focus on the inputs
__global LLR * restrict cnInputsGM,

// Now the outputs
__global LLR * restrict cnOutputsGM,

// Now a place for the results
__global LLR * restrict cnResults

) {



  int id = get_global_id(0);
  printf("CN,%d,started\n",id);

  // First we need to store a place in private memory for operation
  LLR cnInputs[N_CHECK_INPUTS];

  // And also a spot for the magnitudes
  LLR cnInputsMag[N_CHECK_INPUTS];

  // Ok. Now we need to do some similar work to the VNs.
  // Let's make an array for the offsets
  int vnMemoryOffsets[2];
  vnMemoryOffsets[0] = 0;
  vnMemoryOffsets[1] = N_VARIABLE_NODES * (N_VARIABLE_INPUTS+1);

  int cnMemoryOffsets[2];
  cnMemoryOffsets[0] = 0;
  cnMemoryOffsets[1] = N_CHECK_NODES * (N_CHECK_INPUTS+1);
  // Check Node Private Variables
  LLR cnFirstMin;
  LLR cnSecondMin;
  LLR cnParity;

  // And let's store an offset
  char offsetIndex = 0; // Note that we start on the same data as VN but we have to wait
      ↪ for the ramp up of the VN to complete the first iterations before we can execute

  printf("CN,%d,prepped\n",id);
  char temp;

  int cnInputsNodeIndex = id * N_CHECK_INPUTS;
  int cnInputsGlobalIndex = id * (N_CHECK_INPUTS+1);

  // Alright let's start the iterations
  for(char iter = 0; iter < (N_ITERATIONS * 2); iter++) {

    if(id == 0)
      temp = read_channel_altera(cnStart);

    barrier(CLK_GLOBAL_MEM_FENCE);


    printf("CN,%d,%d\n",id,iter);
    // For each iteration we need to:
    // 1) Wait for the ok from the VNs to start executing on the information
    // 2) Load in the new inputs from GM
    // 3) Do the Cn
    // 4) Send the data to the VN via GM
    // 5) Tell the VN we have finished


    // Now we can begin our execution
    cnFirstMin = (*vector_max_llr);
    cnSecondMin = (*vector_max_llr);

    // First let's do the minimum calculations

    // But first, we need to make sure that we are taking the absolute value as we don't
        ↪ want to include that in the min calculation
    // Therefore, we have to mask off the sign bits
    #pragma unroll
    for(short input = 0; input < N_CHECK_INPUTS; input++) {
      cnInputs[input] = cnInputsGM[cnMemoryOffsets[offsetIndex] + cnInputsGlobalIndex +
          ↪ input];
    }

    // Now that we have the normal inputs, we need to get the magnitudes
    #pragma unroll
    for(short input = 0; input < N_CHECK_INPUTS; input++) {
      cnInputsMag[input] = cnInputs[input] & (*vector_max_llr);
    }

    // Ok now that we have the magnitudes, let's get to work by getting the overall parity
        ↪  first
    cnParity = (*zeros);
    #pragma unroll
    for(short input = 0; input < N_CHECK_INPUTS; input++) {
      cnParity ^= cnInputs[input] & (*vector_sign_bit);
    }

    // Now cnParity holds the overall parity of the check node

    // We're going to try something a little different here
    // We're going to implement the tree using min on the magnitudes

    // This is a little trickier as we need to find the losers a lot
    // But let's start it out.


    /************************************* ROUND 1 ************************************/
    /************************************* ROUND 1 ************************************/
LLR comp_0_to_1 = (cnInputsMag[0] < cnInputsMag[1]) ? (*allones) : (*zeros);
LLR comp_2_to_3 = (cnInputsMag[2] < cnInputsMag[3]) ? (*allones) : (*zeros);
LLR comp_4_to_5 = (cnInputsMag[4] < cnInputsMag[5]) ? (*allones) : (*zeros);
LLR comp_6_to_7 = (cnInputsMag[6] < cnInputsMag[7]) ? (*allones) : (*zeros);
LLR comp_8_to_9 = (cnInputsMag[8] < cnInputsMag[9]) ? (*allones) : (*zeros);
LLR comp_10_to_11 = (cnInputsMag[10] < cnInputsMag[11]) ? (*allones) : (*zeros);
LLR comp_12_to_13 = (cnInputsMag[12] < cnInputsMag[13]) ? (*allones) : (*zeros);
LLR comp_14_to_15 = (cnInputsMag[14] < cnInputsMag[15]) ? (*allones) : (*zeros);
LLR comp_16_to_17 = (cnInputsMag[16] < cnInputsMag[17]) ? (*allones) : (*zeros);
LLR comp_18_to_19 = (cnInputsMag[18] < cnInputsMag[19]) ? (*allones) : (*zeros);
LLR comp_20_to_21 = (cnInputsMag[20] < cnInputsMag[21]) ? (*allones) : (*zeros);
LLR comp_22_to_23 = (cnInputsMag[22] < cnInputsMag[23]) ? (*allones) : (*zeros);
LLR comp_24_to_25 = (cnInputsMag[24] < cnInputsMag[25]) ? (*allones) : (*zeros);
LLR comp_26_to_27 = (cnInputsMag[26] < cnInputsMag[27]) ? (*allones) : (*zeros);
LLR comp_28_to_29 = (cnInputsMag[28] < cnInputsMag[29]) ? (*allones) : (*zeros);
LLR comp_30_to_31 = (cnInputsMag[30] < cnInputsMag[31]) ? (*allones) : (*zeros);
LLR comp_32_to_33 = (cnInputsMag[32] < cnInputsMag[33]) ? (*allones) : (*zeros);
```

```
// First Minimum
LLR winner_0_to_1 = (cnInputsMag[0] & comp_0_to_1) | (cnInputsMag[1] & (comp_0_to_1 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_0 = (cnInputsMag[1] & comp_0_to_1) | (cnInputsMag[0] & (comp_0_to_1 ^ (*allones)
    ↪ ));


// First Minimum
LLR winner_2_to_3 = (cnInputsMag[2] & comp_2_to_3) | (cnInputsMag[3] & (comp_2_to_3 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_1 = (cnInputsMag[3] & comp_2_to_3) | (cnInputsMag[2] & (comp_2_to_3 ^ (*allones)
    ↪ ));


// First Minimum
LLR winner_4_to_5 = (cnInputsMag[4] & comp_4_to_5) | (cnInputsMag[5] & (comp_4_to_5 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_2 = (cnInputsMag[5] & comp_4_to_5) | (cnInputsMag[4] & (comp_4_to_5 ^ (*allones)
    ↪ ));


// First Minimum
LLR winner_6_to_7 = (cnInputsMag[6] & comp_6_to_7) | (cnInputsMag[7] & (comp_6_to_7 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_3 = (cnInputsMag[7] & comp_6_to_7) | (cnInputsMag[6] & (comp_6_to_7 ^ (*allones)
    ↪ ));


// First Minimum
LLR winner_8_to_9 = (cnInputsMag[8] & comp_8_to_9) | (cnInputsMag[9] & (comp_8_to_9 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_4 = (cnInputsMag[9] & comp_8_to_9) | (cnInputsMag[8] & (comp_8_to_9 ^ (*allones)
    ↪ ));


// First Minimum
LLR winner_10_to_11 = (cnInputsMag[10] & comp_10_to_11) | (cnInputsMag[11] & (
    ↪ comp_10_to_11 ^ (*allones)));
// Second Minimum
LLR loser_5 = (cnInputsMag[11] & comp_10_to_11) | (cnInputsMag[10] & (comp_10_to_11 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_12_to_13 = (cnInputsMag[12] & comp_12_to_13) | (cnInputsMag[13] & (
    ↪ comp_12_to_13 ^ (*allones)));
// Second Minimum
LLR loser_6 = (cnInputsMag[13] & comp_12_to_13) | (cnInputsMag[12] & (comp_12_to_13 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_14_to_15 = (cnInputsMag[14] & comp_14_to_15) | (cnInputsMag[15] & (
    ↪ comp_14_to_15 ^ (*allones)));
// Second Minimum
LLR loser_7 = (cnInputsMag[15] & comp_14_to_15) | (cnInputsMag[14] & (comp_14_to_15 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_16_to_17 = (cnInputsMag[16] & comp_16_to_17) | (cnInputsMag[17] & (
    ↪ comp_16_to_17 ^ (*allones)));
// Second Minimum
LLR loser_8 = (cnInputsMag[17] & comp_16_to_17) | (cnInputsMag[16] & (comp_16_to_17 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_18_to_19 = (cnInputsMag[18] & comp_18_to_19) | (cnInputsMag[19] & (
    ↪ comp_18_to_19 ^ (*allones)));
// Second Minimum
LLR loser_9 = (cnInputsMag[19] & comp_18_to_19) | (cnInputsMag[18] & (comp_18_to_19 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_20_to_21 = (cnInputsMag[20] & comp_20_to_21) | (cnInputsMag[21] & (
    ↪ comp_20_to_21 ^ (*allones)));
// Second Minimum
LLR loser_10  = (cnInputsMag[21] & comp_20_to_21) | (cnInputsMag[20] & (comp_20_to_21 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_22_to_23 = (cnInputsMag[22] & comp_22_to_23) | (cnInputsMag[23] & (
    ↪ comp_22_to_23 ^ (*allones)));
```

```
// Second Minimum
LLR loser_11  = (cnInputsMag[23] & comp_22_to_23) | (cnInputsMag[22] & (comp_22_to_23 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_24_to_25 = (cnInputsMag[24] & comp_24_to_25) | (cnInputsMag[25] & (
    ↪ comp_24_to_25 ^ (*allones)));
// Second Minimum
LLR loser_12  = (cnInputsMag[25] & comp_24_to_25) | (cnInputsMag[24] & (comp_24_to_25 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_26_to_27 = (cnInputsMag[26] & comp_26_to_27) | (cnInputsMag[27] & (
    ↪ comp_26_to_27 ^ (*allones)));
// Second Minimum
LLR loser_13  = (cnInputsMag[27] & comp_26_to_27) | (cnInputsMag[26] & (comp_26_to_27 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_28_to_29 = (cnInputsMag[28] & comp_28_to_29) | (cnInputsMag[29] & (
    ↪ comp_28_to_29 ^ (*allones)));
// Second Minimum
LLR loser_14  = (cnInputsMag[29] & comp_28_to_29) | (cnInputsMag[28] & (comp_28_to_29 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_30_to_31 = (cnInputsMag[30] & comp_30_to_31) | (cnInputsMag[31] & (
    ↪ comp_30_to_31 ^ (*allones)));
// Second Minimum
LLR loser_15  = (cnInputsMag[31] & comp_30_to_31) | (cnInputsMag[30] & (comp_30_to_31 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_32_to_33 = (cnInputsMag[32] & comp_32_to_33) | (cnInputsMag[33] & (
    ↪ comp_32_to_33 ^ (*allones)));
// Second Minimum
LLR loser_16  = (cnInputsMag[33] & comp_32_to_33) | (cnInputsMag[32] & (comp_32_to_33 ^ (*
    ↪ allones)));


/*---------------------------------- END ROUND 1 ----------------------------------*/




/*********************************** ROUND 2 ***********************************/
LLR comp_0_to_3 = (winner_0_to_1 < winner_2_to_3) ? (*allones) : (*zeros);
LLR comp_4_to_7 = (winner_4_to_5 < winner_6_to_7) ? (*allones) : (*zeros);
LLR comp_8_to_11  = (winner_8_to_9 < winner_10_to_11) ? (*allones) : (*zeros);
LLR comp_12_to_15 = (winner_12_to_13 < winner_14_to_15) ? (*allones) : (*zeros);
LLR comp_16_to_19 = (winner_16_to_17 < winner_18_to_19) ? (*allones) : (*zeros);
LLR comp_20_to_23 = (winner_20_to_21 < winner_22_to_23) ? (*allones) : (*zeros);
LLR comp_24_to_27 = (winner_24_to_25 < winner_26_to_27) ? (*allones) : (*zeros);
LLR comp_28_to_31 = (winner_28_to_29 < winner_30_to_31) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_3 = (winner_0_to_1 & comp_0_to_3) | (winner_2_to_3 & (comp_0_to_3 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_17  = (winner_2_to_3 & comp_0_to_3) | (winner_0_to_1 & (comp_0_to_3 ^ (*allones)
    ↪ ));


// First Minimum
LLR winner_4_to_7 = (winner_4_to_5 & comp_4_to_7) | (winner_6_to_7 & (comp_4_to_7 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_18  = (winner_6_to_7 & comp_4_to_7) | (winner_4_to_5 & (comp_4_to_7 ^ (*allones)
    ↪ ));


// First Minimum
LLR winner_8_to_11  = (winner_8_to_9 & comp_8_to_11) | (winner_10_to_11 & (comp_8_to_11 ^
    ↪ (*allones)));
// Second Minimum
LLR loser_19  = (winner_10_to_11 & comp_8_to_11) | (winner_8_to_9 & (comp_8_to_11 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_12_to_15 = (winner_12_to_13 & comp_12_to_15) | (winner_14_to_15 & (
    ↪ comp_12_to_15 ^ (*allones)));
// Second Minimum
LLR loser_20  = (winner_14_to_15 & comp_12_to_15) | (winner_12_to_13 & (comp_12_to_15 ^ (*
    ↪ allones)));
```

```
// First Minimum
LLR winner_16_to_19 = (winner_16_to_17 & comp_16_to_19) | (winner_18_to_19 & (
    ↪ comp_16_to_19 ^ (*allones)));
// Second Minimum
LLR loser_21  = (winner_18_to_19 & comp_16_to_19) | (winner_16_to_17 & (comp_16_to_19 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_20_to_23 = (winner_20_to_21 & comp_20_to_23) | (winner_22_to_23 & (
    ↪ comp_20_to_23 ^ (*allones)));
// Second Minimum
LLR loser_22  = (winner_22_to_23 & comp_20_to_23) | (winner_20_to_21 & (comp_20_to_23 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_24_to_27 = (winner_24_to_25 & comp_24_to_27) | (winner_26_to_27 & (
    ↪ comp_24_to_27 ^ (*allones)));
// Second Minimum
LLR loser_23  = (winner_26_to_27 & comp_24_to_27) | (winner_24_to_25 & (comp_24_to_27 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_28_to_31 = (winner_28_to_29 & comp_28_to_31) | (winner_30_to_31 & (
    ↪ comp_28_to_31 ^ (*allones)));
// Second Minimum
LLR loser_24  = (winner_30_to_31 & comp_28_to_31) | (winner_28_to_29 & (comp_28_to_31 ^ (*
    ↪ allones)));


/*---------------------------------- END ROUND 2 ----------------------------------*/




/*********************************** ROUND 3 ***********************************/
LLR comp_0_to_7 = (winner_0_to_3 < winner_4_to_7) ? (*allones) : (*zeros);
LLR comp_8_to_15  = (winner_8_to_11 < winner_12_to_15) ? (*allones) : (*zeros);
LLR comp_16_to_23 = (winner_16_to_19 < winner_20_to_23) ? (*allones) : (*zeros);
LLR comp_24_to_31 = (winner_24_to_27 < winner_28_to_31) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_7 = (winner_0_to_3 & comp_0_to_7) | (winner_4_to_7 & (comp_0_to_7 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_25  = (winner_4_to_7 & comp_0_to_7) | (winner_0_to_3 & (comp_0_to_7 ^ (*allones)
    ↪ ));


// First Minimum
LLR winner_8_to_15  = (winner_8_to_11 & comp_8_to_15) | (winner_12_to_15 & (comp_8_to_15 ^
    ↪  (*allones)));
// Second Minimum
LLR loser_26  = (winner_12_to_15 & comp_8_to_15) | (winner_8_to_11 & (comp_8_to_15 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_16_to_23 = (winner_16_to_19 & comp_16_to_23) | (winner_20_to_23 & (
    ↪ comp_16_to_23 ^ (*allones)));
// Second Minimum
LLR loser_27  = (winner_20_to_23 & comp_16_to_23) | (winner_16_to_19 & (comp_16_to_23 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_24_to_31 = (winner_24_to_27 & comp_24_to_31) | (winner_28_to_31 & (
    ↪ comp_24_to_31 ^ (*allones)));
// Second Minimum
LLR loser_28  = (winner_28_to_31 & comp_24_to_31) | (winner_24_to_27 & (comp_24_to_31 ^ (*
    ↪ allones)));


/*---------------------------------- END ROUND 3 ----------------------------------*/




/*********************************** ROUND 4 ***********************************/
LLR comp_0_to_15  = (winner_0_to_7 < winner_8_to_15) ? (*allones) : (*zeros);
LLR comp_16_to_31 = (winner_16_to_23 < winner_24_to_31) ? (*allones) : (*zeros);



// First Minimum
```

```
LLR winner_0_to_15  = (winner_0_to_7 & comp_0_to_15) | (winner_8_to_15 & (comp_0_to_15 ^
    ↪ (*allones)));
// Second Minimum
LLR loser_29  = (winner_8_to_15 & comp_0_to_15) | (winner_0_to_7 & (comp_0_to_15 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_16_to_31 = (winner_16_to_23 & comp_16_to_31) | (winner_24_to_31 & (
    ↪ comp_16_to_31 ^ (*allones)));
// Second Minimum
LLR loser_30  = (winner_24_to_31 & comp_16_to_31) | (winner_16_to_23 & (comp_16_to_31 ^ (*
    ↪ allones)));


/*---------------------------------- END ROUND 4 ----------------------------------*/




/*********************************** ROUND 5 ***********************************/
LLR comp_0_to_31  = (winner_0_to_15 < winner_16_to_31) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_31  = (winner_0_to_15 & comp_0_to_31) | (winner_16_to_31 & (comp_0_to_31 ^
    ↪  (*allones)));
// Second Minimum
LLR loser_31  = (winner_16_to_31 & comp_0_to_31) | (winner_0_to_15 & (comp_0_to_31 ^ (*
    ↪ allones)));


/*---------------------------------- END ROUND 5 ----------------------------------*/




/*********************************** ROUND 6 ***********************************/
LLR comp_0_to_33  = (winner_0_to_31 < winner_32_to_33) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_33  = (winner_0_to_31 & comp_0_to_33) | (winner_32_to_33 & (comp_0_to_33 ^
    ↪  (*allones)));
// Second Minimum
LLR loser_32  = (winner_32_to_33 & comp_0_to_33) | (winner_0_to_31 & (comp_0_to_33 ^ (*
    ↪ allones)));


/*---------------------------------- END ROUND 6 ----------------------------------*/




cnFirstMin = winner_0_to_33;





/****************************** SECOND MINIMUM ********************************/


/*********************************** ROUND 1 ***********************************/
LLR second_comp_0_to_1  = (loser_0  < loser_1) ? (*allones) : (*zeros);
LLR second_comp_2_to_3  = (loser_2  < loser_3) ? (*allones) : (*zeros);
LLR second_comp_4_to_5  = (loser_4  < loser_5) ? (*allones) : (*zeros);
LLR second_comp_6_to_7  = (loser_6  < loser_7) ? (*allones) : (*zeros);
LLR second_comp_8_to_9  = (loser_8  < loser_9) ? (*allones) : (*zeros);
LLR second_comp_10_to_11  = (loser_10  < loser_11) ? (*allones) : (*zeros);
LLR second_comp_12_to_13  = (loser_12  < loser_13) ? (*allones) : (*zeros);
LLR second_comp_14_to_15  = (loser_14  < loser_15) ? (*allones) : (*zeros);
LLR second_comp_16_to_17  = (loser_16  < loser_17) ? (*allones) : (*zeros);
LLR second_comp_18_to_19  = (loser_18  < loser_19) ? (*allones) : (*zeros);
LLR second_comp_20_to_21  = (loser_20  < loser_21) ? (*allones) : (*zeros);
LLR second_comp_22_to_23  = (loser_22  < loser_23) ? (*allones) : (*zeros);
LLR second_comp_24_to_25  = (loser_24  < loser_25) ? (*allones) : (*zeros);
LLR second_comp_26_to_27  = (loser_26  < loser_27) ? (*allones) : (*zeros);
LLR second_comp_28_to_29  = (loser_28  < loser_29) ? (*allones) : (*zeros);
LLR second_comp_30_to_31  = (loser_30  < loser_31) ? (*allones) : (*zeros);



LLR second_winner_0_to_1  = (loser_0 & second_comp_0_to_1) | (loser_1 & (
    ↪ second_comp_0_to_1 ^ (*allones)));
LLR second_winner_2_to_3  = (loser_2 & second_comp_2_to_3) | (loser_3 & (
    ↪ second_comp_2_to_3 ^ (*allones)));
LLR second_winner_4_to_5  = (loser_4 & second_comp_4_to_5) | (loser_5 & (
    ↪ second_comp_4_to_5 ^ (*allones)));
```

```
LLR second_winner_6_to_7  = (loser_6 & second_comp_6_to_7) | (loser_7 & (
     ↪ second_comp_6_to_7 ^ (*allones)));
LLR second_winner_8_to_9  = (loser_8 & second_comp_8_to_9) | (loser_9 & (
     ↪ second_comp_8_to_9 ^ (*allones)));
LLR second_winner_10_to_11  = (loser_10 & second_comp_10_to_11) | (loser_11 & (
     ↪ second_comp_10_to_11 ^ (*allones)));
LLR second_winner_12_to_13  = (loser_12 & second_comp_12_to_13) | (loser_13 & (
     ↪ second_comp_12_to_13 ^ (*allones)));
LLR second_winner_14_to_15  = (loser_14 & second_comp_14_to_15) | (loser_15 & (
     ↪ second_comp_14_to_15 ^ (*allones)));
LLR second_winner_16_to_17  = (loser_16 & second_comp_16_to_17) | (loser_17 & (
     ↪ second_comp_16_to_17 ^ (*allones)));
LLR second_winner_18_to_19  = (loser_18 & second_comp_18_to_19) | (loser_19 & (
     ↪ second_comp_18_to_19 ^ (*allones)));
LLR second_winner_20_to_21  = (loser_20 & second_comp_20_to_21) | (loser_21 & (
     ↪ second_comp_20_to_21 ^ (*allones)));
LLR second_winner_22_to_23  = (loser_22 & second_comp_22_to_23) | (loser_23 & (
     ↪ second_comp_22_to_23 ^ (*allones)));
LLR second_winner_24_to_25  = (loser_24 & second_comp_24_to_25) | (loser_25 & (
     ↪ second_comp_24_to_25 ^ (*allones)));
LLR second_winner_26_to_27  = (loser_26 & second_comp_26_to_27) | (loser_27 & (
     ↪ second_comp_26_to_27 ^ (*allones)));
LLR second_winner_28_to_29  = (loser_28 & second_comp_28_to_29) | (loser_29 & (
     ↪ second_comp_28_to_29 ^ (*allones)));
LLR second_winner_30_to_31  = (loser_30 & second_comp_30_to_31) | (loser_31 & (
     ↪ second_comp_30_to_31 ^ (*allones)));
/*----------------------------------- END ROUND 1 -----------------------------------*/




/************************************* ROUND 2 ************************************/
LLR second_comp_0_to_3  = (second_winner_0_to_1 < second_winner_2_to_3) ? (*allones) : (*
     ↪ zeros);
LLR second_comp_4_to_7  = (second_winner_4_to_5 < second_winner_6_to_7) ? (*allones) : (*
     ↪ zeros);
LLR second_comp_8_to_11 = (second_winner_8_to_9 < second_winner_10_to_11) ? (*allones) :
     ↪ (*zeros);
LLR second_comp_12_to_15  = (second_winner_12_to_13 < second_winner_14_to_15) ? (*allones)
     ↪  : (*zeros);
LLR second_comp_16_to_19  = (second_winner_16_to_17 < second_winner_18_to_19) ? (*allones)
     ↪  : (*zeros);
LLR second_comp_20_to_23  = (second_winner_20_to_21 < second_winner_22_to_23) ? (*allones)
     ↪  : (*zeros);
LLR second_comp_24_to_27  = (second_winner_24_to_25 < second_winner_26_to_27) ? (*allones)
     ↪  : (*zeros);
LLR second_comp_28_to_31  = (second_winner_28_to_29 < second_winner_30_to_31) ? (*allones)
     ↪  : (*zeros);



LLR second_winner_0_to_3  = (second_winner_0_to_1 & second_comp_0_to_3) | (
     ↪ second_winner_2_to_3 & (second_comp_0_to_3 ^ (*allones)));
LLR second_winner_4_to_7  = (second_winner_4_to_5 & second_comp_4_to_7) | (
     ↪ second_winner_6_to_7 & (second_comp_4_to_7 ^ (*allones)));
LLR second_winner_8_to_11 = (second_winner_8_to_9 & second_comp_8_to_11) | (
     ↪ second_winner_10_to_11 & (second_comp_8_to_11 ^ (*allones)));
LLR second_winner_12_to_15  = (second_winner_12_to_13 & second_comp_12_to_15) | (
     ↪ second_winner_14_to_15 & (second_comp_12_to_15 ^ (*allones)));
LLR second_winner_16_to_19  = (second_winner_16_to_17 & second_comp_16_to_19) | (
     ↪ second_winner_18_to_19 & (second_comp_16_to_19 ^ (*allones)));
LLR second_winner_20_to_23  = (second_winner_20_to_21 & second_comp_20_to_23) | (
     ↪ second_winner_22_to_23 & (second_comp_20_to_23 ^ (*allones)));
LLR second_winner_24_to_27  = (second_winner_24_to_25 & second_comp_24_to_27) | (
     ↪ second_winner_26_to_27 & (second_comp_24_to_27 ^ (*allones)));
LLR second_winner_28_to_31  = (second_winner_28_to_29 & second_comp_28_to_31) | (
     ↪ second_winner_30_to_31 & (second_comp_28_to_31 ^ (*allones)));
/*----------------------------------- END ROUND 2 -----------------------------------*/




/************************************* ROUND 3 ************************************/
LLR second_comp_0_to_7  = (second_winner_0_to_3 < second_winner_4_to_7) ? (*allones) : (*
     ↪ zeros);
LLR second_comp_8_to_15 = (second_winner_8_to_11 < second_winner_12_to_15) ? (*allones) :
     ↪ (*zeros);
LLR second_comp_16_to_23  = (second_winner_16_to_19 < second_winner_20_to_23) ? (*allones)
     ↪  : (*zeros);
LLR second_comp_24_to_31  = (second_winner_24_to_27 < second_winner_28_to_31) ? (*allones)
     ↪  : (*zeros);



LLR second_winner_0_to_7  = (second_winner_0_to_3 & second_comp_0_to_7) | (
     ↪ second_winner_4_to_7 & (second_comp_0_to_7 ^ (*allones)));
LLR second_winner_8_to_15 = (second_winner_8_to_11 & second_comp_8_to_15) | (
     ↪ second_winner_12_to_15 & (second_comp_8_to_15 ^ (*allones)));
LLR second_winner_16_to_23  = (second_winner_16_to_19 & second_comp_16_to_23) | (
     ↪ second_winner_20_to_23 & (second_comp_16_to_23 ^ (*allones)));
LLR second_winner_24_to_31  = (second_winner_24_to_27 & second_comp_24_to_31) | (
     ↪ second_winner_28_to_31 & (second_comp_24_to_31 ^ (*allones)));
```

```
/*---------------------------------- END ROUND 3 ----------------------------------*/




/*********************************** ROUND 4 ***********************************/
LLR second_comp_0_to_15 = (second_winner_0_to_7 < second_winner_8_to_15) ? (*allones) : (*
    ↪ zeros);
LLR second_comp_16_to_31  = (second_winner_16_to_23 < second_winner_24_to_31) ? (*allones)
    ↪  : (*zeros);



LLR second_winner_0_to_15 = (second_winner_0_to_7 & second_comp_0_to_15) | (
    ↪ second_winner_8_to_15 & (second_comp_0_to_15 ^ (*allones)));
LLR second_winner_16_to_31  = (second_winner_16_to_23 & second_comp_16_to_31) | (
    ↪ second_winner_24_to_31 & (second_comp_16_to_31 ^ (*allones)));
/*---------------------------------- END ROUND 4 ----------------------------------*/




/*********************************** ROUND 5 ***********************************/
LLR second_comp_0_to_31 = (second_winner_0_to_15 < second_winner_16_to_31) ? (*allones) :
    ↪ (*zeros);



LLR second_winner_0_to_31 = (second_winner_0_to_15 & second_comp_0_to_31) | (
    ↪ second_winner_16_to_31 & (second_comp_0_to_31 ^ (*allones)));
/*---------------------------------- END ROUND 5 ----------------------------------*/




/*********************************** ROUND 6 ***********************************/
LLR second_comp_0_to_32 = (second_winner_0_to_31 < loser_32) ? (*allones) : (*zeros);



LLR second_winner_0_to_32 = (second_winner_0_to_31 & second_comp_0_to_32) | (loser_32 & (
    ↪ second_comp_0_to_32 ^ (*allones)));
/*---------------------------------- END ROUND 6 ----------------------------------*/




cnSecondMin = second_winner_0_to_32;






    // Now that we have the minimums and the parity, we should convert the first and
        ↪ second minimums to 2's complement assuming negative
    LLR first_min_neg = (((cnFirstMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));

    LLR second_min_neg = (((cnSecondMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));

    /*for(int i=0; i<N_CHECK_INPUTS; i++) {
      printf("Iteration,%d,CNode,%d,input,%d,%d,%d,min,%d,second_min,%d,address,%d\n",iter
        ↪ ,id,i,cnInputs[i].s0 & (*vector_sign_bit).s0,cnInputsMag[i].s0,cnFirstMin.s0
        ↪ ,cnSecondMin.s0,cnMemoryOffsets[offsetIndex] + cnInputsGlobalIndex + i);
    }*/





    // Now let's start the check nodes
    // We only need to execute one check node for each calculation as there are
        ↪ N_CHECK_NODE number of work-items
    //#pragma unroll
    for(char input = 0; input < N_CHECK_INPUTS; input++) {



      // For each of these we need to determine if it was the first minimum
      // We just need a conditional that has either 1 or 0 depending if it's min or second
        ↪  min
      // And then we can make the expression LLR to_vnode = cnFirstMin*condition1 +
        ↪ cnSecondMin*inv(condition1)
      LLR min_conditional = ((cnInputs[input] & (*vector_max_llr)) == cnFirstMin) ? (*
        ↪ allones) : (*zeros);

      // the LLR conditional variable will now have a vector of either 0 or 1.
      // Now we can just use the expression

      // Now we can move on to calculating the output
      LLR sign_b = cnParity;

      // The sign bit is assigned to complete the parity compared to the other inputs
      sign_b ^= cnInputs[input] & (*vector_sign_bit);

      // Now we just send the data in 2's complement which was already calculated
      LLR sign_conditional = (sign_b == (*vector_sign_bit)) ? (*allones) : (*zeros);

      // If it's negative, give it the negative value, if not then the positive
      // Start with cnSecondMin, second_min_neg, cnFirstMin, first_min_neg
      LLR to_vnode =  ((min_conditional & sign_conditional & second_min_neg)) |
            ((min_conditional & (sign_conditional ^ (*allones)) & cnSecondMin)) |
            ((min_conditional ^ (*allones)) & sign_conditional & first_min_neg) |
            ((min_conditional ^ (*allones)) & (sign_conditional ^ (*allones)) & cnFirstMin
                ↪ );



      // Now we just need to find the variable node index
      int vnIndex = cntovnIndex[cnInputsNodeIndex + input] + vnMemoryOffsets[offsetIndex];

      cnOutputsGM[vnIndex] = to_vnode;

      //printf("Iteration,%d,CNode,%d,output,%d,%d,vnIndex,%d,\n",iter,id,input,to_vnode.
          ↪ s0,vnIndex);

    }






    barrier(CLK_GLOBAL_MEM_FENCE);
    offsetIndex ^= 1;
    if(id == 0)
      write_channel_altera(vnStart,SENDDATA);



  }

}
```

Appendix P

An NDRange Decoder Implementation (Design E) Irregular Length-1120 Kernel Source Code (OpenCL)

```
#ifndef N_VARIABLE_NODES
#define N_VARIABLE_NODES 1120
#endif

#ifndef N_CHECK_NODES
#define N_CHECK_NODES 280
#endif

#ifndef SIMD
#define SIMD 1
#endif

// Based on the Tanner graph, how many connections are there
#define MAX_VARIABLE_EDGES 5
#define MAX_CHECK_EDGES 16


// Let's define the usable number of inputs
#define N_VARIABLE_INPUTS (MAX_VARIABLE_EDGES)
#define N_CHECK_INPUTS (MAX_CHECK_EDGES)

#define N_VARIABLE_INPUTS_W_CHANNEL (N_VARIABLE_INPUTS + 1)

#define N_ITERATIONS (32)



// Set the vector width and the appropriate LLR types
#define MANUAL_VECTOR_WIDTH (16)

#if MANUAL_VECTOR_WIDTH==2
#define LLR_TYPE char2
#define TOTAL_TYPE short2
#define CONVERT_TO_SHORT(a) convert_short2(a)
#define CONVERT_TO_CHAR(a) convert_char2(a)
#elif MANUAL_VECTOR_WIDTH==4
#define LLR_TYPE char4
#define TOTAL_TYPE short4
#define CONVERT_TO_SHORT(a) convert_short4(a)
#define CONVERT_TO_CHAR(a) convert_char4(a)
#elif MANUAL_VECTOR_WIDTH==8
#define LLR_TYPE char8
#define TOTAL_TYPE short8
#define CONVERT_TO_SHORT(a) convert_short8(a)
#define CONVERT_TO_CHAR(a) convert_char8(a)
#elif MANUAL_VECTOR_WIDTH==16
#define LLR_TYPE char16
#define TOTAL_TYPE short16
#define CONVERT_TO_SHORT(a) convert_short16(a)
#define CONVERT_TO_CHAR(a) convert_char16(a)
#else
#define LLR_TYPE char
#define TOTAL_TYPE short
```

```
#define CONVERT_TO_SHORT(a) convert_short(a)
#define CONVERT_TO_CHAR(a) convert_char(a)
#endif

typedef LLR_TYPE LLR;


// Now for the channels
#pragma OPENCL EXTENSION cl_altera_channels : enable
channel char vnStart __attribute__((depth(N_ITERATIONS)));
channel char cnStart __attribute__((depth(N_ITERATIONS)));
#define SENDDATA (1)

//Variable Node Kernel
__kernel
__attribute__((reqd_work_group_size(N_VARIABLE_NODES,1,1)))
__attribute__((num_compute_units(1)))
__attribute__((num_simd_work_items(SIMD)))
void VariableNode(

// Kernel parameteres
// As in the notes page, there is a table that determines the write location in the CNs
__constant short * restrict vntocnIndex,

// Now that we have the connection information let's get the inputs
// Remember there is one memory location for the Variable Node inputs with two slots for
    ↪ pipelining
// Let's start with the channel measurements
__global LLR * restrict channelMeasurements,

// Now let's get the inputs
__global LLR * restrict vnInputsGM,

// Now let's get the output pointer to the CNs
__global LLR * restrict vnOutputsGM,

// And of course finally, we need to get the result
__global TOTAL_TYPE * restrict vnResults,


// We need a few constants where it's easier to compute them in the host
__constant LLR * restrict vector_max_llr,
__constant LLR * restrict vector_sign_bit,
__constant LLR * restrict allones,
__constant LLR * restrict ones,
__constant LLR * restrict zeros,
__constant TOTAL_TYPE * restrict total_vector_max_llr,
__constant TOTAL_TYPE * restrict total_vector_min_neg_llr
)
{

  // First we need to get our global IDsa
  int id = get_global_id(0);



  // We also need a short version of the LLR values to total
  TOTAL_TYPE vnInputsShort[N_VARIABLE_INPUTS_W_CHANNEL];

  // Let's make an array for the offsets
  int vnMemoryOffsets[2];
  vnMemoryOffsets[0] = 0;
  vnMemoryOffsets[1] = N_VARIABLE_NODES * (N_VARIABLE_INPUTS+1);

  int cnMemoryOffsets[2];
  cnMemoryOffsets[0] = 0;
  cnMemoryOffsets[1] = N_CHECK_NODES * (N_CHECK_INPUTS+1);

  // Let's make an array index lookup for the input
  int vntocnNodeIndex = id * (N_VARIABLE_INPUTS);
  int vnGlobalNodeIndex = id * (N_VARIABLE_INPUTS+ 1);

  // Some other variables
  LLR vnCurrent;
  TOTAL_TYPE vnTempCurrent;
  LLR vnConditional;
  LLR vnSignB;
  LLR vnMag;

  // Now let's store the pointer
  char offsetIndex = 0;

  //printf("VN,%d,prepped\n",id);

  // Let's start the process
  // We need to step through the iterations (aka start the loop)
  // Since both VN and CN will be running concurrently on the same data, we will need
      ↪ N_ITERATIONS*2+1
  for(char iter = (char)(0); iter < ((N_ITERATIONS * 2) + 2); iter++) {

    // Let's just do some printing
    //printf("VN,%d,%d\n",id,iter);
    // Alright so each iteration we need to do the following:
    // Each VN:
    // 1) Wait for the OK from the CNs to start on the data
    // 2) Load up the inputs from DRAM into private memory
    // 3) Do the VN
    // 4) Output the data to the appropriate buffer
    // 5) Send the ok to the CN to start execution
    // 6) Swap buffers
    // Repeat


    // Let's wait for the CNs to tell us it's ok to start execution
    // Note that we can throw away the data as this is a blocking call
    if(id == 0 && iter > 1)
      read_channel_altera(vnStart);

    barrier(CLK_GLOBAL_MEM_FENCE);


    // Now that we've got the OK. Let's start execution
    // First let's load up the inputs and convert them to shorts so that we can get the
        ↪ appropriate totals without saturation
    #pragma unroll
    for(char input = 0; input < N_VARIABLE_INPUTS; input++) {
      vnInputsShort[input] = CONVERT_TO_SHORT(vnInputsGM[vnMemoryOffsets[offsetIndex] +
          ↪ vnGlobalNodeIndex + input]);
    }

    // Now let's put in the channel measurement for our node
    vnInputsShort[N_VARIABLE_INPUTS] = CONVERT_TO_SHORT(channelMeasurements[
        ↪ vnMemoryOffsets[offsetIndex] + vntocnNodeIndex]);

    // Now we need to add it all together

    TOTAL_TYPE total = (TOTAL_TYPE)(0);
    #pragma unroll
    for(char input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {
      total += vnInputsShort[input];
    }

    // Now that we have the total we can do the rest
    for(char input = 0; input < N_VARIABLE_INPUTS; input++) {

      // Subtract the current from the total to get the output
      // We simply subtract the large total and then saturate it down to the appropriate
          ↪ size
      vnTempCurrent = total - vnInputsShort[input];

      // Now saturate
      vnTempCurrent =   ((vnTempCurrent >= (*total_vector_max_llr)) ?
                (*total_vector_max_llr) :
                ((vnTempCurrent <= (*total_vector_min_neg_llr)) ?
                (*total_vector_min_neg_llr) :
                 vnTempCurrent));
      // Now we reverse the procedure and store the data back into the LLR type
      vnCurrent = CONVERT_TO_CHAR(vnTempCurrent);

      // Ok now we can finally send the data
      // First we check if it's negative
      // Except we need to convert it to sign and magnitude
      vnConditional = ((vnCurrent & (*vector_sign_bit)) == (*vector_sign_bit)) ? (*allones
          ↪ ) : (*zeros);

      // If it's negative we need to xor it and add one and mask out the sign bit
      vnMag = ((vnConditional & ((vnCurrent ^ (*allones)) + (*ones))) | ((vnConditional ^
          ↪ (*allones)) & vnCurrent)) & (*vector_max_llr);

      // Now get the sign bit
      vnSignB = (vnConditional & (*vector_sign_bit));

      // Now we just need to find the location in the array to place the data
      short cnIndex = vntocnIndex[vntocnNodeIndex + input] + cnMemoryOffsets[offsetIndex];

      // Now we can just output the value
      // Since we're writing to the CNs we need to output to the write memory location
      vnOutputsGM[cnIndex] = vnMag | vnSignB;

    }

    if((iter+1)==N_ITERATIONS*2+1) {
      TOTAL_TYPE send_total = 0;
      for(short input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {
        send_total += vnInputsShort[input];
      }

      vnResults[vnMemoryOffsets[offsetIndex] + id] = send_total;
    }

    barrier(CLK_GLOBAL_MEM_FENCE);

    //printf("VN,%d,%d,beforechannelwrite\n",id,iter);
    // Finally, we need to tell the check node it can operate on the data
```

```
    if(id == 0)
      write_channel_altera(cnStart, SENDDATA);


    // Now let's switch the index over
    // To do this we can simply XOR it with the value of 1
    offsetIndex ^= 1;




  }

  TOTAL_TYPE send_total = 0;
  for(short input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {
    send_total += vnInputsShort[input];
  }

  vnResults[vnMemoryOffsets[offsetIndex] + id] = send_total;


}


__kernel
__attribute__((reqd_work_group_size(N_CHECK_NODES,1,1)))
__attribute__((num_compute_units(1)))
__attribute__((num_simd_work_items(SIMD)))
void CheckNode(

// Let's start with the table for the interleaver connections just like with the VN
__constant short * restrict cntovnIndex,

// Now we can do the other constants
__constant LLR * restrict vector_max_llr,
__constant LLR * restrict vector_sign_bit,
__constant LLR * restrict allones,
__constant LLR * restrict ones,
__constant LLR * restrict zeros,

// Now we can focus on the inputs
__global LLR * restrict cnInputsGM,

// Now the outputs
__global LLR * restrict cnOutputsGM,

// Now a place for the results
__global LLR * restrict cnResults

) {



  int id = get_global_id(0);

  // First we need to store a place in private memory for operation
  LLR cnInputs[N_CHECK_INPUTS];

  // And also a spot for the magnitudes
  LLR cnInputsMag[N_CHECK_INPUTS];

  // Let's compute the base index
  int cnInputsNodeIndex = id * N_CHECK_INPUTS;
  int cnInputsGlobalIndex = id * (N_CHECK_INPUTS+1);

  // Check Node Private Variables
  LLR cnFirstMin;
  LLR cnSecondMin;
  LLR cnParity;


  // Ok. Now we need to do some similar work to the VNs.
  // Let's make an array for the offsets
  int vnMemoryOffsets[2];
  vnMemoryOffsets[0] = 0;
  vnMemoryOffsets[1] = N_VARIABLE_NODES * (N_VARIABLE_INPUTS+1);

  int cnMemoryOffsets[2];
  cnMemoryOffsets[0] = 0;
  cnMemoryOffsets[1] = N_CHECK_NODES * (N_CHECK_INPUTS+1);

  // And let's store an offset
  char offsetIndex = 0; // Note that we start on the same data as VN but we have to wait
      ↪ for the ramp up of the VN to complete the first iterations before we can execute

  //printf("CN,%d,prepped\n",id);

  // Alright let's start the iterations
  for(char iter = 0; iter < (N_ITERATIONS * 2); iter++) {

    //printf("CN,%d,%d\n",id,iter);
    // For each iteration we need to:
```

```
// 1) Wait for the ok from the VNs to start executing on the information
// 2) Load in the new inputs from GM
// 3) Do the Cn
// 4) Send the data to the VN via GM
// 5) Tell the VN we have finished

// So let's wait for the Ok to start
if(id == 0)
  read_channel_altera(cnStart);

barrier(CLK_GLOBAL_MEM_FENCE | CLK_CHANNEL_MEM_FENCE);

// Now we can begin our execution
cnFirstMin = (*vector_max_llr);
cnSecondMin = (*vector_max_llr);

// First let's do the minimum calculations

// But first, we need to make sure that we are taking the absolute value as we don't
    ↪ want to include that in the min calculation
// Therefore, we have to mask off the sign bits
#pragma unroll
for(short input = 0; input < N_CHECK_INPUTS; input++) {
  cnInputs[input] = cnInputsGM[cnMemoryOffsets[offsetIndex] + cnInputsGlobalIndex +
      ↪ input];
}

// Now that we have the normal inputs, we need to get the magnitudes
#pragma unroll
for(short input = 0; input < N_CHECK_INPUTS; input++) {
  cnInputsMag[input] = cnInputs[input] & (*vector_max_llr);
}

// Ok now that we have the magnitudes, let's get to work by getting the overall parity
    ↪  first
cnParity = (*zeros);
#pragma unroll
for(short input = 0; input < N_CHECK_INPUTS; input++) {
  cnParity ^= cnInputs[input] & (*vector_sign_bit);
}

// Now cnParity holds the overall parity of the check node

// We're going to try something a little different here
// We're going to implement the tree using min on the magnitudes

// This is a little trickier as we need to find the losers a lot
// But let's start it out.


/************************************ ROUND 1 ************************************/
LLR comp_0_1  = (cnInputsMag[0] < cnInputsMag[1]) ? (*allones) : (*zeros);
LLR comp_2_3  = (cnInputsMag[2] < cnInputsMag[3]) ? (*allones) : (*zeros);
LLR comp_4_5  = (cnInputsMag[4] < cnInputsMag[5]) ? (*allones) : (*zeros);
LLR comp_6_7  = (cnInputsMag[6] < cnInputsMag[7]) ? (*allones) : (*zeros);
LLR comp_8_9  = (cnInputsMag[8] < cnInputsMag[9]) ? (*allones) : (*zeros);
LLR comp_10_11  = (cnInputsMag[10] < cnInputsMag[11]) ? (*allones) : (*zeros);
LLR comp_12_13  = (cnInputsMag[12] < cnInputsMag[13]) ? (*allones) : (*zeros);
LLR comp_14_15  = (cnInputsMag[14] < cnInputsMag[15]) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_1 = (cnInputsMag[0] & comp_0_1) | (cnInputsMag[1] & (comp_0_1 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_0 = (cnInputsMag[1] & comp_0_1) | (cnInputsMag[0] & (comp_0_1 ^ (*allones)))
    ↪ ;


// First Minimum
LLR winner_2_to_3 = (cnInputsMag[2] & comp_2_3) | (cnInputsMag[3] & (comp_2_3 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_1 = (cnInputsMag[3] & comp_2_3) | (cnInputsMag[2] & (comp_2_3 ^ (*allones)))
    ↪ ;


// First Minimum
LLR winner_4_to_5 = (cnInputsMag[4] & comp_4_5) | (cnInputsMag[5] & (comp_4_5 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_2 = (cnInputsMag[5] & comp_4_5) | (cnInputsMag[4] & (comp_4_5 ^ (*allones)))
    ↪ ;


// First Minimum
LLR winner_6_to_7 = (cnInputsMag[6] & comp_6_7) | (cnInputsMag[7] & (comp_6_7 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_3 = (cnInputsMag[7] & comp_6_7) | (cnInputsMag[6] & (comp_6_7 ^ (*allones)))
    ↪ ;
```

```
// First Minimum
LLR winner_8_to_9 = (cnInputsMag[8] & comp_8_9) | (cnInputsMag[9] & (comp_8_9 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_4 = (cnInputsMag[9] & comp_8_9) | (cnInputsMag[8] & (comp_8_9 ^ (*allones)))
    ↪ ;


// First Minimum
LLR winner_10_to_11 = (cnInputsMag[10] & comp_10_11) | (cnInputsMag[11] & (comp_10_11
    ↪ ^ (*allones)));
// Second Minimum
LLR loser_5 = (cnInputsMag[11] & comp_10_11) | (cnInputsMag[10] & (comp_10_11 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_12_to_13 = (cnInputsMag[12] & comp_12_13) | (cnInputsMag[13] & (comp_12_13
    ↪ ^ (*allones)));
// Second Minimum
LLR loser_6 = (cnInputsMag[13] & comp_12_13) | (cnInputsMag[12] & (comp_12_13 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_14_to_15 = (cnInputsMag[14] & comp_14_15) | (cnInputsMag[15] & (comp_14_15
    ↪ ^ (*allones)));
// Second Minimum
LLR loser_7 = (cnInputsMag[15] & comp_14_15) | (cnInputsMag[14] & (comp_14_15 ^ (*
    ↪ allones)));


/*----------------------------------- END ROUND 1 -----------------------------------
    ↪ */




/************************************* ROUND 2 *************************************/
LLR comp_0_to_3 = (winner_0_to_1 < winner_2_to_3) ? (*allones) : (*zeros);
LLR comp_4_to_7 = (winner_4_to_5 < winner_6_to_7) ? (*allones) : (*zeros);
LLR comp_8_to_11  = (winner_8_to_9 < winner_10_to_11) ? (*allones) : (*zeros);
LLR comp_12_to_15 = (winner_12_to_13 < winner_14_to_15) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_3 = (winner_0_to_1 & comp_0_to_3) | (winner_2_to_3 & (comp_0_to_3 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_8 = (winner_2_to_3 & comp_0_to_3) | (winner_0_to_1 & (comp_0_to_3 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_4_to_7 = (winner_4_to_5 & comp_4_to_7) | (winner_6_to_7 & (comp_4_to_7 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_9 = (winner_6_to_7 & comp_4_to_7) | (winner_4_to_5 & (comp_4_to_7 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_8_to_11  = (winner_8_to_9 & comp_8_to_11) | (winner_10_to_11 & (
    ↪ comp_8_to_11 ^ (*allones)));
// Second Minimum
LLR loser_10  = (winner_10_to_11 & comp_8_to_11) | (winner_8_to_9 & (comp_8_to_11 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_12_to_15 = (winner_12_to_13 & comp_12_to_15) | (winner_14_to_15 & (
    ↪ comp_12_to_15 ^ (*allones)));
// Second Minimum
LLR loser_11  = (winner_14_to_15 & comp_12_to_15) | (winner_12_to_13 & (comp_12_to_15
    ↪ ^ (*allones)));


/*----------------------------------- END ROUND 2 -----------------------------------
    ↪ */




/************************************* ROUND 3 *************************************/
LLR comp_0_to_7 = (winner_0_to_3 < winner_4_to_7) ? (*allones) : (*zeros);
LLR comp_8_to_15  = (winner_8_to_11 < winner_12_to_15) ? (*allones) : (*zeros);
```

```
// First Minimum
LLR winner_0_to_7 = (winner_0_to_3 & comp_0_to_7) | (winner_4_to_7 & (comp_0_to_7 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_12  = (winner_4_to_7 & comp_0_to_7) | (winner_0_to_3 & (comp_0_to_7 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_8_to_15  = (winner_8_to_11 & comp_8_to_15) | (winner_12_to_15 & (
    ↪ comp_8_to_15 ^ (*allones)));
// Second Minimum
LLR loser_13  = (winner_12_to_15 & comp_8_to_15) | (winner_8_to_11 & (comp_8_to_15 ^
    ↪ (*allones)));


/*----------------------------------- END ROUND 3 -----------------------------------
    ↪ */




/*************************************** ROUND 4 ***************************************/
LLR comp_0_to_15  = (winner_0_to_7 < winner_8_to_15) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_15  = (winner_0_to_7 & comp_0_to_15) | (winner_8_to_15 & (comp_0_to_15
    ↪  ^ (*allones)));
// Second Minimum
LLR loser_14  = (winner_8_to_15 & comp_0_to_15) | (winner_0_to_7 & (comp_0_to_15 ^ (*
    ↪ allones)));


/*----------------------------------- END ROUND 4 -----------------------------------
    ↪ */




cnFirstMin = winner_0_to_15;

/*************************************** ROUND 1 ***************************************/
LLR second_comp_0_to_1  = (loser_0 < loser_1) ? (*allones) : (*zeros);
LLR second_comp_2_to_3  = (loser_2 < loser_3) ? (*allones) : (*zeros);
LLR second_comp_4_to_5  = (loser_4 < loser_5) ? (*allones) : (*zeros);
LLR second_comp_6_to_7  = (loser_6 < loser_7) ? (*allones) : (*zeros);
LLR second_comp_8_to_9  = (loser_8 < loser_9) ? (*allones) : (*zeros);
LLR second_comp_10_to_11  = (loser_10 < loser_11) ? (*allones) : (*zeros);
LLR second_comp_12_to_13  = (loser_12 < loser_13) ? (*allones) : (*zeros);



LLR second_winner_0_to_1  = (loser_0 & second_comp_0_to_1) | (loser_1 & (
    ↪ second_comp_0_to_1 ^ (*allones)));
LLR second_winner_2_to_3  = (loser_2 & second_comp_2_to_3) | (loser_3 & (
    ↪ second_comp_2_to_3 ^ (*allones)));
LLR second_winner_4_to_5  = (loser_4 & second_comp_4_to_5) | (loser_5 & (
    ↪ second_comp_4_to_5 ^ (*allones)));
LLR second_winner_6_to_7  = (loser_6 & second_comp_6_to_7) | (loser_7 & (
    ↪ second_comp_6_to_7 ^ (*allones)));
LLR second_winner_8_to_9  = (loser_8 & second_comp_8_to_9) | (loser_9 & (
    ↪ second_comp_8_to_9 ^ (*allones)));
LLR second_winner_10_to_11  = (loser_10 & second_comp_10_to_11) | (loser_11 & (
    ↪ second_comp_10_to_11 ^ (*allones)));
LLR second_winner_12_to_13  = (loser_12 & second_comp_12_to_13) | (loser_13 & (
    ↪ second_comp_12_to_13 ^ (*allones)));
/*----------------------------------- END ROUND 1 -----------------------------------
    ↪ */




/*************************************** ROUND 2 ***************************************/
LLR second_comp_0_to_3  = (second_winner_0_to_1 < second_winner_2_to_3) ? (*allones) :
    ↪  (*zeros);
LLR second_comp_4_to_7  = (second_winner_4_to_5 < second_winner_6_to_7) ? (*allones) :
    ↪  (*zeros);
LLR second_comp_8_to_11 = (second_winner_8_to_9 < second_winner_10_to_11) ? (*allones)
    ↪  : (*zeros);
LLR second_comp_12_to_14  = (second_winner_12_to_13 < loser_14) ? (*allones) : (*zeros
    ↪ );

LLR second_winner_0_to_3 = (second_winner_0_to_1 & second_comp_0_to_3) | (
    ↪ second_winner_2_to_3 & (second_comp_0_to_3 ^ (*allones)));
LLR second_winner_4_to_7 = (second_winner_4_to_5 & second_comp_4_to_7) | (
    ↪ second_winner_6_to_7 & (second_comp_4_to_7 ^ (*allones)));
LLR second_winner_8_to_11 = (second_winner_8_to_9 & second_comp_8_to_11) | (
    ↪ second_winner_10_to_11 & (second_comp_8_to_11 ^ (*allones)));
LLR second_winner_12_to_14 = (second_winner_12_to_13 & second_comp_12_to_14) | (
    ↪ loser_14 & (second_comp_12_to_14 ^ (*allones)));
```

```
/*------------------------------------ END ROUND 2 -----------------------------------
    ↪ */


/************************************ ROUND 3 ************************************/
LLR second_comp_0_to_7 = (second_winner_0_to_3 < second_winner_4_to_7) ? (*allones) :
    ↪ (*zeros);
LLR second_comp_8_to_14 = (second_winner_8_to_11 < second_winner_12_to_14) ? (*allones
    ↪ ) : (*zeros);

LLR second_winner_0_to_7 = (second_winner_0_to_3 & second_comp_0_to_7) | (
    ↪ second_winner_4_to_7 & (second_comp_0_to_7 ^ (*allones)));
LLR second_winner_8_to_14 = (second_winner_8_to_11 & second_comp_8_to_14) | (
    ↪ second_winner_12_to_14 & (second_comp_8_to_14 ^ (*allones)));
/*------------------------------------ END ROUND 3 -----------------------------------
    ↪ */

/************************************ ROUND 4 ************************************/
LLR second_comp_0_to_14 = (second_winner_0_to_7 < second_comp_8_to_14) ? (*allones) :
    ↪ (*zeros);

LLR second_winner_0_to_14 = (second_winner_0_to_7 & second_comp_0_to_14) | (
    ↪ second_comp_8_to_14 & (second_comp_0_to_14 ^ (*allones)));
/*------------------------------------ END ROUND 4 -----------------------------------
    ↪ */

cnSecondMin = second_winner_0_to_14;

// Now that we have the minimums and the parity, we should convert the first and
    ↪ second minimums to 2's complement assuming negative
LLR first_min_neg = (((cnFirstMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));

LLR second_min_neg = (((cnSecondMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));



// Now let's start the check nodes
// We only need to execute one check node for each calculation as there are
    ↪ N_CHECK_NODE number of work-items
//#pragma unroll
for(char input = 0; input < N_CHECK_INPUTS; input++) {



  // For each of these we need to determine if it was the first minimum
  // We just need a conditional that has either 1 or 0 depending if it's min or second
      ↪ min
  // And then we can make the expression LLR to_vnode = cnFirstMin*condition1 +
      ↪ cnSecondMin*inv(condition1)
  LLR min_conditional = ((cnInputs[input] & (*vector_max_llr)) == cnFirstMin) ? (*
      ↪ allones) : (*zeros);

  // the LLR conditional variable will now have a vector of either 0 or 1.
  // Now we can just use the expression

  // Now we can move on to calculating the output
  LLR sign_b = cnParity;

  // The sign bit is assigned to complete the parity compared to the other inputs
  sign_b ^= cnInputs[input] & (*vector_sign_bit);

  // Now we just send the data in 2's complement which was already calculated
  LLR sign_conditional = (sign_b == (*vector_sign_bit)) ? (*allones) : (*zeros);

  // If it's negative, give it the negative value, if not then the positive
  // Start with cnSecondMin, second_min_neg, cnFirstMin, first_min_neg
  LLR to_vnode =  ((min_conditional & sign_conditional & second_min_neg)) |
        ((min_conditional & (sign_conditional ^ (*allones)) & cnSecondMin)) |
        ((min_conditional ^ (*allones)) & sign_conditional & first_min_neg) |
        ((min_conditional ^ (*allones)) & (sign_conditional ^ (*allones)) & cnFirstMin
            ↪ );



  // Now we just need to find the variable node index
  short vnIndex = cntovnIndex[cnInputsNodeIndex + input] + cnMemoryOffsets[offsetIndex
      ↪ ];

  cnOutputsGM[vnIndex] = to_vnode;

  ////printf("Iteration,%d,CNode,%d,output,%d,%d,vnIndex,%d\n",iter,id,input,to_vnode,
      ↪ vnIndex);

}
barrier(CLK_GLOBAL_MEM_FENCE | CLK_CHANNEL_MEM_FENCE);
offsetIndex ^= 1;
if(id == 0)
  write_channel_altera(vnStart,SENDDATA);
```

```
    }
}
```

Appendix Q

An NDRange Decoder Implementation (Design E) Irregular Length-5940 Kernel Source Code (OpenCL)

```
#ifndef N_VARIABLE_NODES
#define N_VARIABLE_NODES 5940
#endif

#ifndef N_CHECK_NODES
#define N_CHECK_NODES 900
#endif

#ifndef SIMD
#define SIMD 1
#endif

// Based on the Tanner graph, how many connections are there
#define MAX_VARIABLE_EDGES 5
#define MAX_CHECK_EDGES 27


// Let's define the usable number of inputs
#define N_VARIABLE_INPUTS (MAX_VARIABLE_EDGES)
#define N_CHECK_INPUTS (MAX_CHECK_EDGES)

#define N_VARIABLE_INPUTS_W_CHANNEL (N_VARIABLE_INPUTS + 1)

#define N_ITERATIONS (32)



// Set the vector width and the appropriate LLR types
#define MANUAL_VECTOR_WIDTH (16)

#if MANUAL_VECTOR_WIDTH==2
#define LLR_TYPE char2
#define TOTAL_TYPE short2
#define CONVERT_TO_SHORT(a) convert_short2(a)
#define CONVERT_TO_CHAR(a) convert_char2(a)
#elif MANUAL_VECTOR_WIDTH==4
#define LLR_TYPE char4
#define TOTAL_TYPE short4
#define CONVERT_TO_SHORT(a) convert_short4(a)
#define CONVERT_TO_CHAR(a) convert_char4(a)
#elif MANUAL_VECTOR_WIDTH==8
#define LLR_TYPE char8
#define TOTAL_TYPE short8
#define CONVERT_TO_SHORT(a) convert_short8(a)
#define CONVERT_TO_CHAR(a) convert_char8(a)
#elif MANUAL_VECTOR_WIDTH==16
#define LLR_TYPE char16
#define TOTAL_TYPE short16
#define CONVERT_TO_SHORT(a) convert_short16(a)
#define CONVERT_TO_CHAR(a) convert_char16(a)
#else
#define LLR_TYPE char
#define TOTAL_TYPE short
```

```
#define CONVERT_TO_SHORT(a) convert_short(a)
#define CONVERT_TO_CHAR(a) convert_char(a)
#endif

typedef LLR_TYPE LLR;


// Now for the channels
#pragma OPENCL EXTENSION cl_altera_channels : enable
channel char vnStart __attribute__((depth(N_ITERATIONS)));
channel char cnStart __attribute__((depth(N_ITERATIONS)));
#define SENDDATA (1)

//Variable Node Kernel
__kernel
__attribute__((reqd_work_group_size(N_VARIABLE_NODES,1,1)))
__attribute__((num_compute_units(1)))
__attribute__((num_simd_work_items(SIMD)))
void VariableNode(

// Kernel parameteres
// As in the notes page, there is a table that determines the write location in the CNs
__constant int * restrict vntocnIndex,

// Now that we have the connection information let's get the inputs
// Remember there is one memory location for the Variable Node inputs with two slots for
    ↪ pipelining
// Let's start with the channel measurements
__global LLR * restrict channelMeasurements,

// Now let's get the inputs
__global LLR * restrict vnInputsGM,

// Now let's get the output pointer to the CNs
__global LLR * restrict vnOutputsGM,

// And of course finally, we need to get the result
__global TOTAL_TYPE * restrict vnResults,


// We need a few constants where it's easier to compute them in the host
__constant LLR * restrict vector_max_llr,
__constant LLR * restrict vector_sign_bit,
__constant LLR * restrict allones,
__constant LLR * restrict ones,
__constant LLR * restrict zeros,
__constant TOTAL_TYPE * restrict total_vector_max_llr,
__constant TOTAL_TYPE * restrict total_vector_min_neg_llr
)
{

  // First we need to get our global IDsa
  int id = get_global_id(0);
  TOTAL_TYPE vnInputsShort[N_VARIABLE_INPUTS_W_CHANNEL];



  // Let's make an array for the offsets
  int vnMemoryOffsets[2];
  vnMemoryOffsets[0] = 0;
  vnMemoryOffsets[1] = N_VARIABLE_NODES * (N_VARIABLE_INPUTS+1);

  int vnChannelOffsets[2];
  vnChannelOffsets[0] = 0;
  vnChannelOffsets[1] = N_VARIABLE_NODES;

  int cnMemoryOffsets[2];
  cnMemoryOffsets[0] = 0;
  cnMemoryOffsets[1] = N_CHECK_NODES * (N_CHECK_INPUTS+1);

  // Let's make an array index lookup for the input


  // Some other variables


  // Now let's store the pointer
  char temp;
  char offsetIndex = 0;

  // Let's make an array index lookup for the input
  int vntocnNodeIndex = id * (N_VARIABLE_INPUTS);
  int vnGlobalNodeIndex = id * (N_VARIABLE_INPUTS+ 1);

  // Some other variables
  LLR vnCurrent;
  TOTAL_TYPE vnTempCurrent;
  LLR vnConditional;
  LLR vnSignB;
  LLR vnMag;

  //printf("VN,%d,prepped\n",id);
  // Let's start the process
  // We need to step through the iterations (aka start the loop)
  // Since both VN and CN will be running concurrently on the same data, we will need
       ↪ N_ITERATIONS*2+1
  for(char iter = 0; iter < ((N_ITERATIONS * 2) + 2); iter++) {

    //printf("VN Starting Iteration %d\n",iter);

    if(id == 0 && iter > 1)
      temp = read_channel_altera(vnStart);

    barrier(CLK_GLOBAL_MEM_FENCE);
    //printf("After channel read\n");



    // Let's just do some printing
    //printf("VN,%d,%d\n",id,iter);
    // Alright so each iteration we need to do the following:
    // Each VN:
    // 1) Wait for the OK from the CNs to start on the data
    // 2) Load up the inputs from DRAM into private memory
    // 3) Do the VN
    // 4) Output the data to the appropriate buffer
    // 5) Send the ok to the CN to start execution
    // 6) Swap buffers
    // Repeat






    //printf("VN,%d,%d,afterbarrier\n",id,iter);

    // Now that we've got the OK. Let's start execution
    // First let's load up the inputs and convert them to shorts so that we can get the
        ↪ appropriate totals without saturation
    #pragma unroll
    for(char input = 0; input < N_VARIABLE_INPUTS; input++) {
      vnInputsShort[input] = CONVERT_TO_SHORT(vnInputsGM[vnMemoryOffsets[offsetIndex] +
          ↪ vnGlobalNodeIndex + input]);
    }

    // Now let's put in the channel measurement for our node
    vnInputsShort[N_VARIABLE_INPUTS] = CONVERT_TO_SHORT(channelMeasurements[
        ↪ vnChannelOffsets[offsetIndex] + id]);

    // Now we need to add it all together

    TOTAL_TYPE total = (TOTAL_TYPE)(0);
    #pragma unroll
    for(char input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {
      total += vnInputsShort[input];
    }


    /*for(int i=0; i<N_VARIABLE_INPUTS_W_CHANNEL; i++) {
      printf("Iteration,%d,VNode,%d,input,%d,%d,%d,address,%d\n",iter,id,i,total.s0,
          ↪ vnInputsShort[i].s0,vnMemoryOffsets[offsetIndex] + vnGlobalNodeIndex + i);
    }*/

    // Now that we have the total we can do the rest
    for(char input = 0; input < N_VARIABLE_INPUTS; input++) {

      // Subtract the current from the total to get the output
      // We simply subtract the large total and then saturate it down to the appropriate
          ↪ size
      vnTempCurrent = total - vnInputsShort[input];

      // Now saturate
      vnTempCurrent =   ((vnTempCurrent >= (*total_vector_max_llr)) ?
                (*total_vector_max_llr) :
                ((vnTempCurrent <= (*total_vector_min_neg_llr)) ?
                (*total_vector_min_neg_llr) :
                vnTempCurrent));
      // Now we reverse the procedure and store the data back into the LLR type
      vnCurrent = CONVERT_TO_CHAR(vnTempCurrent);

      // Ok now we can finally send the data
      // First we check if it's negative
      // Except we need to convert it to sign and magnitude
      vnConditional = ((vnCurrent & (*vector_sign_bit)) == (*vector_sign_bit)) ? (*allones
          ↪ ) : (*zeros);

      // If it's negative we need to xor it and add one and mask out the sign bit
      vnMag = ((vnConditional & ((vnCurrent ^ (*allones)) + (*ones))) | ((vnConditional ^
          ↪ (*allones)) & vnCurrent)) & (*vector_max_llr);

      // Now get the sign bit
      vnSignB = (vnConditional & (*vector_sign_bit));
```

```
      // Now we just need to find the location in the array to place the data
      int cnIndex = vntocnIndex[input] + cnMemoryOffsets[offsetIndex];

      // Now we can just output the value
      // Since we're writing to the CNs we need to output to the write memory location
      vnOutputsGM[cnIndex] = vnMag | vnSignB;

      //printf("Iteration,%d,VNode,%d,output,%d,%d,cnIndex,%d,vntocnIndex,%d,offset,%d\n",
          ↪ iter,id,input,vnMag.s0 | vnSignB.s0,cnIndex,vntocnIndex[vntocnNodeIndex +
          ↪ input],cnMemoryOffsets[offsetIndex]);

    }

    if((iter+1)==N_ITERATIONS*2+1 || (iter + 1) == N_ITERATIONS*2 + 2) {
      TOTAL_TYPE send_total = 0;
      for(short input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {
        send_total += vnInputsShort[input];
      }

      vnResults[vnMemoryOffsets[offsetIndex] + id] = send_total;
    }



    //printf("VN,%d,%d,beforechannelwrite\n",id,iter);
    // Finally, we need to tell the check node it can operate on the data
    barrier(CLK_GLOBAL_MEM_FENCE);
    if(id==0)
      write_channel_altera(cnStart, SENDDATA);

    //barrier(CLK_GLOBAL_MEM_FENCE | CLK_CHANNEL_MEM_FENCE);

    // Now let's switch the index over
    // To do this we can simply XOR it with the value of 1
    offsetIndex ^= 1;




  }



}


__kernel
__attribute__((reqd_work_group_size(N_CHECK_NODES,1,1)))
__attribute__((num_compute_units(1)))
__attribute__((num_simd_work_items(SIMD)))
void CheckNode(

// Let's start with the table for the interleaver connections just like with the VN
__constant int * restrict cntovnIndex,

// Now we can do the other constants
__constant LLR * restrict vector_max_llr,
__constant LLR * restrict vector_sign_bit,
__constant LLR * restrict allones,
__constant LLR * restrict ones,
__constant LLR * restrict zeros,

// Now we can focus on the inputs
__global LLR * restrict cnInputsGM,

// Now the outputs
__global LLR * restrict cnOutputsGM,

// Now a place for the results
__global LLR * restrict cnResults

) {



  int id = get_global_id(0);
  //printf("CN,%d,started\n",id);

  // First we need to store a place in private memory for operation
  LLR cnInputs[N_CHECK_INPUTS];

  // And also a spot for the magnitudes
  LLR cnInputsMag[N_CHECK_INPUTS];

  // Ok. Now we need to do some similar work to the VNs.
  // Let's make an array for the offsets
  int vnMemoryOffsets[2];
  vnMemoryOffsets[0] = 0;
  vnMemoryOffsets[1] = N_VARIABLE_NODES * (N_VARIABLE_INPUTS+1);

  int cnMemoryOffsets[2];
  cnMemoryOffsets[0] = 0;
  cnMemoryOffsets[1] = N_CHECK_NODES * (N_CHECK_INPUTS+1);

  // Check Node Private Variables
  LLR cnFirstMin;
  LLR cnSecondMin;
  LLR cnParity;

  // And let's store an offset
  char offsetIndex = 0; // Note that we start on the same data as VN but we have to wait
      ↪ for the ramp up of the VN to complete the first iterations before we can execute

  //printf("CN,%d,prepped\n",id);
  char temp;

  int cnInputsNodeIndex = id * N_CHECK_INPUTS;
  int cnInputsGlobalIndex = id * (N_CHECK_INPUTS+1);

  // Alright let's start the iterations
  for(char iter = 0; iter < (N_ITERATIONS * 2); iter++) {

    if(id == 0)
      temp = read_channel_altera(cnStart);

    barrier(CLK_GLOBAL_MEM_FENCE);


    //printf("CN,%d,%d\n",id,iter);
    // For each iteration we need to:
    // 1) Wait for the ok from the VNs to start executing on the information
    // 2) Load in the new inputs from GM
    // 3) Do the Cn
    // 4) Send the data to the VN via GM
    // 5) Tell the VN we have finished


    // Now we can begin our execution
    cnFirstMin = (*vector_max_llr);
    cnSecondMin = (*vector_max_llr);

    // First let's do the minimum calculations

    // But first, we need to make sure that we are taking the absolute value as we don't
        ↪ want to include that in the min calculation
    // Therefore, we have to mask off the sign bits
    #pragma unroll
    for(short input = 0; input < N_CHECK_INPUTS; input++) {
      cnInputs[input] = cnInputsGM[cnMemoryOffsets[offsetIndex] + cnInputsGlobalIndex +
          ↪ input];
    }

    // Now that we have the normal inputs, we need to get the magnitudes
    #pragma unroll
    for(short input = 0; input < N_CHECK_INPUTS; input++) {
      cnInputsMag[input] = cnInputs[input] & (*vector_max_llr);
    }

    // Ok now that we have the magnitudes, let's get to work by getting the overall parity
        ↪  first
    cnParity = (*zeros);
    #pragma unroll
    for(short input = 0; input < N_CHECK_INPUTS; input++) {
      cnParity ^= cnInputs[input] & (*vector_sign_bit);
    }

    // Now cnParity holds the overall parity of the check node

    // We're going to try something a little different here
    // We're going to implement the tree using min on the magnitudes

    // This is a little trickier as we need to find the losers a lot
    // But let's start it out.


    /************************************ ROUND 1 ************************************/
    LLR comp_0_to_1 = (cnInputsMag[0] < cnInputsMag[1]) ? (*allones) : (*zeros);
    LLR comp_2_to_3 = (cnInputsMag[2] < cnInputsMag[3]) ? (*allones) : (*zeros);
    LLR comp_4_to_5 = (cnInputsMag[4] < cnInputsMag[5]) ? (*allones) : (*zeros);
    LLR comp_6_to_7 = (cnInputsMag[6] < cnInputsMag[7]) ? (*allones) : (*zeros);
    LLR comp_8_to_9 = (cnInputsMag[8] < cnInputsMag[9]) ? (*allones) : (*zeros);
    LLR comp_10_to_11 = (cnInputsMag[10] < cnInputsMag[11]) ? (*allones) : (*zeros);
    LLR comp_12_to_13 = (cnInputsMag[12] < cnInputsMag[13]) ? (*allones) : (*zeros);
    LLR comp_14_to_15 = (cnInputsMag[14] < cnInputsMag[15]) ? (*allones) : (*zeros);
    LLR comp_16_to_17 = (cnInputsMag[16] < cnInputsMag[17]) ? (*allones) : (*zeros);
    LLR comp_18_to_19 = (cnInputsMag[18] < cnInputsMag[19]) ? (*allones) : (*zeros);
    LLR comp_20_to_21 = (cnInputsMag[20] < cnInputsMag[21]) ? (*allones) : (*zeros);
    LLR comp_22_to_23 = (cnInputsMag[22] < cnInputsMag[23]) ? (*allones) : (*zeros);
    LLR comp_24_to_25 = (cnInputsMag[24] < cnInputsMag[25]) ? (*allones) : (*zeros);



    // First Minimum
```

```
LLR winner_0_to_1 = (cnInputsMag[0] & comp_0_to_1) | (cnInputsMag[1] & (comp_0_to_1 ^
    ↪ (*allones)));
// Second Minimum
LLR loser_0 = (cnInputsMag[1] & comp_0_to_1) | (cnInputsMag[0] & (comp_0_to_1 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_2_to_3 = (cnInputsMag[2] & comp_2_to_3) | (cnInputsMag[3] & (comp_2_to_3 ^
    ↪ (*allones)));
// Second Minimum
LLR loser_1 = (cnInputsMag[3] & comp_2_to_3) | (cnInputsMag[2] & (comp_2_to_3 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_4_to_5 = (cnInputsMag[4] & comp_4_to_5) | (cnInputsMag[5] & (comp_4_to_5 ^
    ↪ (*allones)));
// Second Minimum
LLR loser_2 = (cnInputsMag[5] & comp_4_to_5) | (cnInputsMag[4] & (comp_4_to_5 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_6_to_7 = (cnInputsMag[6] & comp_6_to_7) | (cnInputsMag[7] & (comp_6_to_7 ^
    ↪ (*allones)));
// Second Minimum
LLR loser_3 = (cnInputsMag[7] & comp_6_to_7) | (cnInputsMag[6] & (comp_6_to_7 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_8_to_9 = (cnInputsMag[8] & comp_8_to_9) | (cnInputsMag[9] & (comp_8_to_9 ^
    ↪ (*allones)));
// Second Minimum
LLR loser_4 = (cnInputsMag[9] & comp_8_to_9) | (cnInputsMag[8] & (comp_8_to_9 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_10_to_11 = (cnInputsMag[10] & comp_10_to_11) | (cnInputsMag[11] & (
    ↪ comp_10_to_11 ^ (*allones)));
// Second Minimum
LLR loser_5 = (cnInputsMag[11] & comp_10_to_11) | (cnInputsMag[10] & (comp_10_to_11 ^
    ↪ (*allones)));


// First Minimum
LLR winner_12_to_13 = (cnInputsMag[12] & comp_12_to_13) | (cnInputsMag[13] & (
    ↪ comp_12_to_13 ^ (*allones)));
// Second Minimum
LLR loser_6 = (cnInputsMag[13] & comp_12_to_13) | (cnInputsMag[12] & (comp_12_to_13 ^
    ↪ (*allones)));


// First Minimum
LLR winner_14_to_15 = (cnInputsMag[14] & comp_14_to_15) | (cnInputsMag[15] & (
    ↪ comp_14_to_15 ^ (*allones)));
// Second Minimum
LLR loser_7 = (cnInputsMag[15] & comp_14_to_15) | (cnInputsMag[14] & (comp_14_to_15 ^
    ↪ (*allones)));


// First Minimum
LLR winner_16_to_17 = (cnInputsMag[16] & comp_16_to_17) | (cnInputsMag[17] & (
    ↪ comp_16_to_17 ^ (*allones)));
// Second Minimum
LLR loser_8 = (cnInputsMag[17] & comp_16_to_17) | (cnInputsMag[16] & (comp_16_to_17 ^
    ↪ (*allones)));


// First Minimum
LLR winner_18_to_19 = (cnInputsMag[18] & comp_18_to_19) | (cnInputsMag[19] & (
    ↪ comp_18_to_19 ^ (*allones)));
// Second Minimum
LLR loser_9 = (cnInputsMag[19] & comp_18_to_19) | (cnInputsMag[18] & (comp_18_to_19 ^
    ↪ (*allones)));


// First Minimum
LLR winner_20_to_21 = (cnInputsMag[20] & comp_20_to_21) | (cnInputsMag[21] & (
    ↪ comp_20_to_21 ^ (*allones)));
// Second Minimum
LLR loser_10  = (cnInputsMag[21] & comp_20_to_21) | (cnInputsMag[20] & (comp_20_to_21
    ↪ ^ (*allones)));


// First Minimum
LLR winner_22_to_23 = (cnInputsMag[22] & comp_22_to_23) | (cnInputsMag[23] & (
    ↪ comp_22_to_23 ^ (*allones)));
// Second Minimum
```

```
LLR loser_11  = (cnInputsMag[23] & comp_22_to_23) | (cnInputsMag[22] & (comp_22_to_23
    ↪ ^ (*allones)));


// First Minimum
LLR winner_24_to_25 = (cnInputsMag[24] & comp_24_to_25) | (cnInputsMag[25] & (
    ↪ comp_24_to_25 ^ (*allones)));
// Second Minimum
LLR loser_12  = (cnInputsMag[25] & comp_24_to_25) | (cnInputsMag[24] & (comp_24_to_25
    ↪ ^ (*allones)));


/*------------------------------------ END ROUND 1 -----------------------------------
    ↪ */




/************************************ ROUND 2 ************************************/
LLR comp_0_to_3 = (winner_0_to_1 < winner_2_to_3) ? (*allones) : (*zeros);
LLR comp_4_to_7 = (winner_4_to_5 < winner_6_to_7) ? (*allones) : (*zeros);
LLR comp_8_to_11  = (winner_8_to_9 < winner_10_to_11) ? (*allones) : (*zeros);
LLR comp_12_to_15 = (winner_12_to_13 < winner_14_to_15) ? (*allones) : (*zeros);
LLR comp_16_to_19 = (winner_16_to_17 < winner_18_to_19) ? (*allones) : (*zeros);
LLR comp_20_to_23 = (winner_20_to_21 < winner_22_to_23) ? (*allones) : (*zeros);
LLR comp_24_to_26 = (winner_24_to_25 < cnInputsMag[26]) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_3 = (winner_0_to_1 & comp_0_to_3) | (winner_2_to_3 & (comp_0_to_3 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_13  = (winner_2_to_3 & comp_0_to_3) | (winner_0_to_1 & (comp_0_to_3 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_4_to_7 = (winner_4_to_5 & comp_4_to_7) | (winner_6_to_7 & (comp_4_to_7 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_14  = (winner_6_to_7 & comp_4_to_7) | (winner_4_to_5 & (comp_4_to_7 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_8_to_11  = (winner_8_to_9 & comp_8_to_11) | (winner_10_to_11 & (
    ↪ comp_8_to_11 ^ (*allones)));
// Second Minimum
LLR loser_15  = (winner_10_to_11 & comp_8_to_11) | (winner_8_to_9 & (comp_8_to_11 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_12_to_15 = (winner_12_to_13 & comp_12_to_15) | (winner_14_to_15 & (
    ↪ comp_12_to_15 ^ (*allones)));
// Second Minimum
LLR loser_16  = (winner_14_to_15 & comp_12_to_15) | (winner_12_to_13 & (comp_12_to_15
    ↪ ^ (*allones)));


// First Minimum
LLR winner_16_to_19 = (winner_16_to_17 & comp_16_to_19) | (winner_18_to_19 & (
    ↪ comp_16_to_19 ^ (*allones)));
// Second Minimum
LLR loser_17  = (winner_18_to_19 & comp_16_to_19) | (winner_16_to_17 & (comp_16_to_19
    ↪ ^ (*allones)));


// First Minimum
LLR winner_20_to_23 = (winner_20_to_21 & comp_20_to_23) | (winner_22_to_23 & (
    ↪ comp_20_to_23 ^ (*allones)));
// Second Minimum
LLR loser_18  = (winner_22_to_23 & comp_20_to_23) | (winner_20_to_21 & (comp_20_to_23
    ↪ ^ (*allones)));


// First Minimum
LLR winner_24_to_26 = (winner_24_to_25 & comp_24_to_26) | (cnInputsMag[26] & (
    ↪ comp_24_to_26 ^ (*allones)));
// Second Minimum
LLR loser_19  = (cnInputsMag[26] & comp_24_to_26) | (winner_24_to_25 & (comp_24_to_26
    ↪ ^ (*allones)));


/*------------------------------------ END ROUND 2 -----------------------------------
    ↪ */




/************************************ ROUND 3 ************************************/
```

```
LLR comp_0_to_7 = (winner_0_to_3 < winner_4_to_7) ? (*allones) : (*zeros);
LLR comp_8_to_15  = (winner_8_to_11 < winner_12_to_15) ? (*allones) : (*zeros);
LLR comp_16_to_23 = (winner_16_to_19 < winner_20_to_23) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_7 = (winner_0_to_3 & comp_0_to_7) | (winner_4_to_7 & (comp_0_to_7 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_20  = (winner_4_to_7 & comp_0_to_7) | (winner_0_to_3 & (comp_0_to_7 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_8_to_15  = (winner_8_to_11 & comp_8_to_15) | (winner_12_to_15 & (
    ↪ comp_8_to_15 ^ (*allones)));
// Second Minimum
LLR loser_21  = (winner_12_to_15 & comp_8_to_15) | (winner_8_to_11 & (comp_8_to_15 ^
    ↪ (*allones)));


// First Minimum
LLR winner_16_to_23 = (winner_16_to_19 & comp_16_to_23) | (winner_20_to_23 & (
    ↪ comp_16_to_23 ^ (*allones)));
// Second Minimum
LLR loser_22  = (winner_20_to_23 & comp_16_to_23) | (winner_16_to_19 & (comp_16_to_23
    ↪ ^ (*allones)));


/*------------------------------------ END ROUND 3 -----------------------------------
    ↪ */




/************************************ ROUND 4 ************************************/
LLR comp_0_to_15  = (winner_0_to_7 < winner_8_to_15) ? (*allones) : (*zeros);
LLR comp_16_to_26 = (winner_16_to_23 < winner_24_to_26) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_15  = (winner_0_to_7 & comp_0_to_15) | (winner_8_to_15 & (comp_0_to_15
    ↪  ^ (*allones)));
// Second Minimum
LLR loser_23  = (winner_8_to_15 & comp_0_to_15) | (winner_0_to_7 & (comp_0_to_15 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_16_to_26 = (winner_16_to_23 & comp_16_to_26) | (winner_24_to_26 & (
    ↪ comp_16_to_26 ^ (*allones)));
// Second Minimum
LLR loser_24  = (winner_24_to_26 & comp_16_to_26) | (winner_16_to_23 & (comp_16_to_26
    ↪ ^ (*allones)));


/*------------------------------------ END ROUND 4 -----------------------------------
    ↪ */




/************************************ ROUND 5 ************************************/
LLR comp_0_to_26  = (winner_0_to_15 < winner_16_to_26) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_26  = (winner_0_to_15 & comp_0_to_26) | (winner_16_to_26 & (
    ↪ comp_0_to_26 ^ (*allones)));
// Second Minimum
LLR loser_25  = (winner_16_to_26 & comp_0_to_26) | (winner_0_to_15 & (comp_0_to_26 ^
    ↪ (*allones)));


/*------------------------------------ END ROUND 5 -----------------------------------
    ↪ */




cnFirstMin = winner_0_to_26;





/******************************** SECOND MINIMUM **********************************/
```

```
/************************************ ROUND 1 ************************************/
LLR second_comp_0_to_1  = (loser_0  < loser_1) ? (*allones) : (*zeros);
LLR second_comp_2_to_3  = (loser_2  < loser_3) ? (*allones) : (*zeros);
LLR second_comp_4_to_5  = (loser_4  < loser_5) ? (*allones) : (*zeros);
LLR second_comp_6_to_7  = (loser_6  < loser_7) ? (*allones) : (*zeros);
LLR second_comp_8_to_9  = (loser_8  < loser_9) ? (*allones) : (*zeros);
LLR second_comp_10_to_11  = (loser_10  < loser_11) ? (*allones) : (*zeros);
LLR second_comp_12_to_13  = (loser_12  < loser_13) ? (*allones) : (*zeros);
LLR second_comp_14_to_15  = (loser_14  < loser_15) ? (*allones) : (*zeros);
LLR second_comp_16_to_17  = (loser_16  < loser_17) ? (*allones) : (*zeros);
LLR second_comp_18_to_19  = (loser_18  < loser_19) ? (*allones) : (*zeros);
LLR second_comp_20_to_21  = (loser_20  < loser_21) ? (*allones) : (*zeros);
LLR second_comp_22_to_23  = (loser_22  < loser_23) ? (*allones) : (*zeros);
LLR second_comp_24_to_25  = (loser_24  < loser_25) ? (*allones) : (*zeros);



LLR second_winner_0_to_1  = (loser_0 & second_comp_0_to_1) | (loser_1 & (
    ↪ second_comp_0_to_1 ^ (*allones)));
LLR second_winner_2_to_3  = (loser_2 & second_comp_2_to_3) | (loser_3 & (
    ↪ second_comp_2_to_3 ^ (*allones)));
LLR second_winner_4_to_5  = (loser_4 & second_comp_4_to_5) | (loser_5 & (
    ↪ second_comp_4_to_5 ^ (*allones)));
LLR second_winner_6_to_7  = (loser_6 & second_comp_6_to_7) | (loser_7 & (
    ↪ second_comp_6_to_7 ^ (*allones)));
LLR second_winner_8_to_9  = (loser_8 & second_comp_8_to_9) | (loser_9 & (
    ↪ second_comp_8_to_9 ^ (*allones)));
LLR second_winner_10_to_11  = (loser_10 & second_comp_10_to_11) | (loser_11 & (
    ↪ second_comp_10_to_11 ^ (*allones)));
LLR second_winner_12_to_13  = (loser_12 & second_comp_12_to_13) | (loser_13 & (
    ↪ second_comp_12_to_13 ^ (*allones)));
LLR second_winner_14_to_15  = (loser_14 & second_comp_14_to_15) | (loser_15 & (
    ↪ second_comp_14_to_15 ^ (*allones)));
LLR second_winner_16_to_17  = (loser_16 & second_comp_16_to_17) | (loser_17 & (
    ↪ second_comp_16_to_17 ^ (*allones)));
LLR second_winner_18_to_19  = (loser_18 & second_comp_18_to_19) | (loser_19 & (
    ↪ second_comp_18_to_19 ^ (*allones)));
LLR second_winner_20_to_21  = (loser_20 & second_comp_20_to_21) | (loser_21 & (
    ↪ second_comp_20_to_21 ^ (*allones)));
LLR second_winner_22_to_23  = (loser_22 & second_comp_22_to_23) | (loser_23 & (
    ↪ second_comp_22_to_23 ^ (*allones)));
LLR second_winner_24_to_25  = (loser_24 & second_comp_24_to_25) | (loser_25 & (
    ↪ second_comp_24_to_25 ^ (*allones)));
/*---------------------------------- END ROUND 1 ----------------------------------
    ↪ */




/************************************ ROUND 2 ************************************/
LLR second_comp_0_to_3  = (second_winner_0_to_1 < second_winner_2_to_3) ? (*allones) :
    ↪  (*zeros);
LLR second_comp_4_to_7  = (second_winner_4_to_5 < second_winner_6_to_7) ? (*allones) :
    ↪  (*zeros);
LLR second_comp_8_to_11 = (second_winner_8_to_9 < second_winner_10_to_11) ? (*allones)
    ↪  : (*zeros);
LLR second_comp_12_to_15  = (second_winner_12_to_13 < second_winner_14_to_15) ? (*
    ↪ allones) : (*zeros);
LLR second_comp_16_to_19  = (second_winner_16_to_17 < second_winner_18_to_19) ? (*
    ↪ allones) : (*zeros);
LLR second_comp_20_to_23  = (second_winner_20_to_21 < second_winner_22_to_23) ? (*
    ↪ allones) : (*zeros);



LLR second_winner_0_to_3  = (second_winner_0_to_1 & second_comp_0_to_3) | (
    ↪ second_winner_2_to_3 & (second_comp_0_to_3 ^ (*allones)));
LLR second_winner_4_to_7  = (second_winner_4_to_5 & second_comp_4_to_7) | (
    ↪ second_winner_6_to_7 & (second_comp_4_to_7 ^ (*allones)));
LLR second_winner_8_to_11 = (second_winner_8_to_9 & second_comp_8_to_11) | (
    ↪ second_winner_10_to_11 & (second_comp_8_to_11 ^ (*allones)));
LLR second_winner_12_to_15  = (second_winner_12_to_13 & second_comp_12_to_15) | (
    ↪ second_winner_14_to_15 & (second_comp_12_to_15 ^ (*allones)));
LLR second_winner_16_to_19  = (second_winner_16_to_17 & second_comp_16_to_19) | (
    ↪ second_winner_18_to_19 & (second_comp_16_to_19 ^ (*allones)));
LLR second_winner_20_to_23  = (second_winner_20_to_21 & second_comp_20_to_23) | (
    ↪ second_winner_22_to_23 & (second_comp_20_to_23 ^ (*allones)));
/*---------------------------------- END ROUND 2 ----------------------------------
    ↪ */




/************************************ ROUND 3 ************************************/
LLR second_comp_0_to_7  = (second_winner_0_to_3 < second_winner_4_to_7) ? (*allones) :
    ↪  (*zeros);
LLR second_comp_8_to_15 = (second_winner_8_to_11 < second_winner_12_to_15) ? (*allones
    ↪ ) : (*zeros);
LLR second_comp_16_to_23  = (second_winner_16_to_19 < second_winner_20_to_23) ? (*
    ↪ allones) : (*zeros);
```

```
LLR second_winner_0_to_7  = (second_winner_0_to_3 & second_comp_0_to_7) | (
    ↪ second_winner_4_to_7 & (second_comp_0_to_7 ^ (*allones)));
LLR second_winner_8_to_15 = (second_winner_8_to_11 & second_comp_8_to_15) | (
    ↪ second_winner_12_to_15 & (second_comp_8_to_15 ^ (*allones)));
LLR second_winner_16_to_23  = (second_winner_16_to_19 & second_comp_16_to_23) | (
    ↪ second_winner_20_to_23 & (second_comp_16_to_23 ^ (*allones)));
/*----------------------------------- END ROUND 3 -----------------------------------
    ↪ */




/************************************* ROUND 4 *************************************/
LLR second_comp_0_to_15 = (second_winner_0_to_7 < second_winner_8_to_15) ? (*allones)
    ↪ : (*zeros);
LLR second_comp_16_to_25  = (second_winner_16_to_23 < second_winner_24_to_25) ? (*
    ↪ allones) : (*zeros);



LLR second_winner_0_to_15 = (second_winner_0_to_7 & second_comp_0_to_15) | (
    ↪ second_winner_8_to_15 & (second_comp_0_to_15 ^ (*allones)));
LLR second_winner_16_to_25  = (second_winner_16_to_23 & second_comp_16_to_25) | (
    ↪ second_winner_24_to_25 & (second_comp_16_to_25 ^ (*allones)));
/*----------------------------------- END ROUND 4 -----------------------------------
    ↪ */




/************************************* ROUND 5 *************************************/
LLR second_comp_0_to_25 = (second_winner_0_to_15 < second_winner_16_to_25) ? (*allones
    ↪ ) : (*zeros);



LLR second_winner_0_to_25 = (second_winner_0_to_15 & second_comp_0_to_25) | (
    ↪ second_winner_16_to_25 & (second_comp_0_to_25 ^ (*allones)));
/*----------------------------------- END ROUND 5 -----------------------------------
    ↪ */




cnSecondMin = second_winner_0_to_25;





// Now that we have the minimums and the parity, we should convert the first and
    ↪ second minimums to 2's complement assuming negative
LLR first_min_neg = (((cnFirstMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));

LLR second_min_neg = (((cnSecondMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));

/*for(int i=0; i<N_CHECK_INPUTS; i++) {
  printf("Iteration,%d,CNode,%d,input,%d,%d,%d,min,%d,second_min,%d,address,%d\n",iter
      ↪ ,id,i,cnInputs[i].s0 & (*vector_sign_bit).s0,cnInputsMag[i].s0,cnFirstMin.s0
      ↪ ,cnSecondMin.s0,cnMemoryOffsets[offsetIndex] + cnInputsGlobalIndex + i);
}*/





// Now let's start the check nodes
// We only need to execute one check node for each calculation as there are
    ↪ N_CHECK_NODE number of work-items
//#pragma unroll
for(char input = 0; input < N_CHECK_INPUTS; input++) {



  // For each of these we need to determine if it was the first minimum
  // We just need a conditional that has either 1 or 0 depending if it's min or second
      ↪  min
  // And then we can make the expression LLR to_vnode = cnFirstMin*condition1 +
      ↪ cnSecondMin*inv(condition1)
  LLR min_conditional = ((cnInputs[input] & (*vector_max_llr)) == cnFirstMin) ? (*
      ↪ allones) : (*zeros);

  // the LLR conditional variable will now have a vector of either 0 or 1.
  // Now we can just use the expression

  // Now we can move on to calculating the output
  LLR sign_b = cnParity;

  // The sign bit is assigned to complete the parity compared to the other inputs
  sign_b ^= cnInputs[input] & (*vector_sign_bit);

  // Now we just send the data in 2's complement which was already calculated
  LLR sign_conditional = (sign_b == (*vector_sign_bit)) ? (*allones) : (*zeros);
```

```
      // If it's negative, give it the negative value, if not then the positive
      // Start with cnSecondMin, second_min_neg, cnFirstMin, first_min_neg
      LLR to_vnode =  ((min_conditional & sign_conditional & second_min_neg)) |
            ((min_conditional & (sign_conditional ^ (*allones)) & cnSecondMin)) |
            ((min_conditional ^ (*allones)) & sign_conditional & first_min_neg) |
            ((min_conditional ^ (*allones)) & (sign_conditional ^ (*allones)) & cnFirstMin
                ↪ );



      // Now we just need to find the variable node index
      int vnIndex = cntovnIndex[cnInputsNodeIndex + input] + vnMemoryOffsets[offsetIndex];

      cnOutputsGM[vnIndex] = to_vnode;

      //printf("Iteration,%d,CNode,%d,output,%d,%d,vnIndex,%d,\n",iter,id,input,to_vnode.
           ↪ s0,vnIndex);

    }






    barrier(CLK_GLOBAL_MEM_FENCE);
    offsetIndex ^= 1;
    if(id == 0)
      write_channel_altera(vnStart,SENDDATA);




  }

}
```

Appendix R

An NDRange Decoder Implementation (Design E) Irregular Length-16200 Kernel Source Code (OpenCL)

```
#ifndef N_VARIABLE_NODES
#define N_VARIABLE_NODES 16200
#endif

#ifndef N_CHECK_NODES
#define N_CHECK_NODES 1800
#endif

#ifndef SIMD
#define SIMD 1
#endif

// Based on the Tanner graph, how many connections are there
#define MAX_VARIABLE_EDGES 5
#define MAX_CHECK_EDGES 34


// Let's define the usable number of inputs
#define N_VARIABLE_INPUTS (MAX_VARIABLE_EDGES)
#define N_CHECK_INPUTS (MAX_CHECK_EDGES)

#define N_VARIABLE_INPUTS_W_CHANNEL (N_VARIABLE_INPUTS + 1)

#define N_ITERATIONS (32)



// Set the vector width and the appropriate LLR types
#define MANUAL_VECTOR_WIDTH (16)

#if MANUAL_VECTOR_WIDTH==2
#define LLR_TYPE char2
#define TOTAL_TYPE short2
#define CONVERT_TO_SHORT(a) convert_short2(a)
#define CONVERT_TO_CHAR(a) convert_char2(a)
#elif MANUAL_VECTOR_WIDTH==4
#define LLR_TYPE char4
#define TOTAL_TYPE short4
#define CONVERT_TO_SHORT(a) convert_short4(a)
#define CONVERT_TO_CHAR(a) convert_char4(a)
#elif MANUAL_VECTOR_WIDTH==8
#define LLR_TYPE char8
#define TOTAL_TYPE short8
#define CONVERT_TO_SHORT(a) convert_short8(a)
#define CONVERT_TO_CHAR(a) convert_char8(a)
#elif MANUAL_VECTOR_WIDTH==16
#define LLR_TYPE char16
#define TOTAL_TYPE short16
#define CONVERT_TO_SHORT(a) convert_short16(a)
#define CONVERT_TO_CHAR(a) convert_char16(a)
#else
#define LLR_TYPE char
#define TOTAL_TYPE short
```

```
#define CONVERT_TO_SHORT(a) convert_short(a)
#define CONVERT_TO_CHAR(a) convert_char(a)
#endif

typedef LLR_TYPE LLR;


// Now for the channels
#pragma OPENCL EXTENSION cl_altera_channels : enable
channel char vnStart __attribute__((depth(N_ITERATIONS)));
channel char cnStart __attribute__((depth(N_ITERATIONS)));
#define SENDDATA (1)

//Variable Node Kernel
__kernel
__attribute__((reqd_work_group_size(N_VARIABLE_NODES,1,1)))
__attribute__((num_compute_units(1)))
__attribute__((num_simd_work_items(SIMD)))
void VariableNode(

// Kernel parameteres
// As in the notes page, there is a table that determines the write location in the CNs
__constant int * restrict vntocnIndex,

// Now that we have the connection information let's get the inputs
// Remember there is one memory location for the Variable Node inputs with two slots for
    ↪ pipelining
// Let's start with the channel measurements
__global LLR * restrict channelMeasurements,

// Now let's get the inputs
__global LLR * restrict vnInputsGM,

// Now let's get the output pointer to the CNs
__global LLR * restrict vnOutputsGM,

// And of course finally, we need to get the result
__global TOTAL_TYPE * restrict vnResults,


// We need a few constants where it's easier to compute them in the host
__constant LLR * restrict vector_max_llr,
__constant LLR * restrict vector_sign_bit,
__constant LLR * restrict allones,
__constant LLR * restrict ones,
__constant LLR * restrict zeros,
__constant TOTAL_TYPE * restrict total_vector_max_llr,
__constant TOTAL_TYPE * restrict total_vector_min_neg_llr
)
{

  // First we need to get our global IDsa
  int id = get_global_id(0);
  printf("VN,%d,Started\n",id);
  TOTAL_TYPE vnInputsShort[N_VARIABLE_INPUTS_W_CHANNEL];



  // Let's make an array for the offsets
  int vnMemoryOffsets[2];
  vnMemoryOffsets[0] = 0;
  vnMemoryOffsets[1] = N_VARIABLE_NODES * (N_VARIABLE_INPUTS+1);

  int vnChannelOffsets[2];
  vnChannelOffsets[0] = 0;
  vnChannelOffsets[1] = N_VARIABLE_NODES;

  int cnMemoryOffsets[2];
  cnMemoryOffsets[0] = 0;
  cnMemoryOffsets[1] = N_CHECK_NODES * (N_CHECK_INPUTS+1);

  // Let's make an array index lookup for the input


  // Some other variables


  // Now let's store the pointer
  char temp;
  char offsetIndex = 0;

  // Let's make an array index lookup for the input
  int vntocnNodeIndex = id * (N_VARIABLE_INPUTS);
  int vnGlobalNodeIndex = id * (N_VARIABLE_INPUTS+ 1);

  // Some other variables
  LLR vnCurrent;
  TOTAL_TYPE vnTempCurrent;
  LLR vnConditional;
  LLR vnSignB;
  LLR vnMag;
  //printf("VN,%d,prepped\n",id);

  // Let's start the process
  // We need to step through the iterations (aka start the loop)
  // Since both VN and CN will be running concurrently on the same data, we will need
       ↪ N_ITERATIONS*2+1
  for(char iter = 0; iter < ((N_ITERATIONS * 2) + 2); iter++) {

    //printf("VN Starting Iteration %d\n",iter);

    if(id == 0 && iter > 1)
      temp = read_channel_altera(vnStart);

    barrier(CLK_GLOBAL_MEM_FENCE);
    //printf("After channel read\n");



    // Let's just do some printing
    printf("VN,%d,%d\n",id,iter);
    // Alright so each iteration we need to do the following:
    // Each VN:
    // 1) Wait for the OK from the CNs to start on the data
    // 2) Load up the inputs from DRAM into private memory
    // 3) Do the VN
    // 4) Output the data to the appropriate buffer
    // 5) Send the ok to the CN to start execution
    // 6) Swap buffers
    // Repeat






    //printf("VN,%d,%d,afterbarrier\n",id,iter);

    // Now that we've got the OK. Let's start execution
    // First let's load up the inputs and convert them to shorts so that we can get the
        ↪ appropriate totals without saturation
    #pragma unroll
    for(char input = 0; input < N_VARIABLE_INPUTS; input++) {
      vnInputsShort[input] = CONVERT_TO_SHORT(vnInputsGM[vnMemoryOffsets[offsetIndex] +
          ↪ vnGlobalNodeIndex + input]);
    }

    // Now let's put in the channel measurement for our node
    vnInputsShort[N_VARIABLE_INPUTS] = CONVERT_TO_SHORT(channelMeasurements[
        ↪ vnChannelOffsets[offsetIndex] + id]);

    // Now we need to add it all together

    TOTAL_TYPE total = (TOTAL_TYPE)(0);
    #pragma unroll
    for(char input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {
      total += vnInputsShort[input];
    }


    /*for(int i=0; i<N_VARIABLE_INPUTS_W_CHANNEL; i++) {
      printf("Iteration,%d,VNode,%d,input,%d,%d,%d,address,%d\n",iter,id,i,total.s0,
          ↪ vnInputsShort[i].s0,vnMemoryOffsets[offsetIndex] + vnGlobalNodeIndex + i);
    }*/

    // Now that we have the total we can do the rest
    for(char input = 0; input < N_VARIABLE_INPUTS; input++) {

      // Subtract the current from the total to get the output
      // We simply subtract the large total and then saturate it down to the appropriate
          ↪ size
      vnTempCurrent = total - vnInputsShort[input];

      // Now saturate
      vnTempCurrent =   ((vnTempCurrent >= (*total_vector_max_llr)) ?
                (*total_vector_max_llr) :
                ((vnTempCurrent <= (*total_vector_min_neg_llr)) ?
                (*total_vector_min_neg_llr) :
                vnTempCurrent));
      // Now we reverse the procedure and store the data back into the LLR type
      vnCurrent = CONVERT_TO_CHAR(vnTempCurrent);

      // Ok now we can finally send the data
      // First we check if it's negative
      // Except we need to convert it to sign and magnitude
      vnConditional = ((vnCurrent & (*vector_sign_bit)) == (*vector_sign_bit)) ? (*allones
          ↪ ) : (*zeros);

      // If it's negative we need to xor it and add one and mask out the sign bit
      vnMag = ((vnConditional & ((vnCurrent ^ (*allones)) + (*ones))) | ((vnConditional ^
          ↪ (*allones)) & vnCurrent)) & (*vector_max_llr);

      // Now get the sign bit
```

```
      vnSignB = (vnConditional & (*vector_sign_bit));

      // Now we just need to find the location in the array to place the data
      int cnIndex = vntocnIndex[input] + cnMemoryOffsets[offsetIndex];

      // Now we can just output the value
      // Since we're writing to the CNs we need to output to the write memory location
      vnOutputsGM[cnIndex] = vnMag | vnSignB;

      //printf("Iteration,%d,VNode,%d,output,%d,%d,cnIndex,%d,vntocnIndex,%d,offset,%d\n",
          ↪ iter,id,input,vnMag.s0 | vnSignB.s0,cnIndex,vntocnIndex[vntocnNodeIndex +
          ↪ input],cnMemoryOffsets[offsetIndex]);

    }

    if((iter+1)==N_ITERATIONS*2+1 || (iter + 1) == N_ITERATIONS*2 + 2) {
      TOTAL_TYPE send_total = 0;
      for(short input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {
        send_total += vnInputsShort[input];
      }

      vnResults[vnMemoryOffsets[offsetIndex] + id] = send_total;
    }



    //printf("VN,%d,%d,beforechannelwrite\n",id,iter);
    // Finally, we need to tell the check node it can operate on the data
    barrier(CLK_GLOBAL_MEM_FENCE);
    if(id==0)
      write_channel_altera(cnStart, SENDDATA);

    //barrier(CLK_GLOBAL_MEM_FENCE | CLK_CHANNEL_MEM_FENCE);

    // Now let's switch the index over
    // To do this we can simply XOR it with the value of 1
    offsetIndex ^= 1;




  }



}


__kernel
__attribute__((reqd_work_group_size(N_CHECK_NODES,1,1)))
__attribute__((num_compute_units(1)))
__attribute__((num_simd_work_items(SIMD)))
void CheckNode(

// Let's start with the table for the interleaver connections just like with the VN
__constant int * restrict cntovnIndex,

// Now we can do the other constants
__constant LLR * restrict vector_max_llr,
__constant LLR * restrict vector_sign_bit,
__constant LLR * restrict allones,
__constant LLR * restrict ones,
__constant LLR * restrict zeros,

// Now we can focus on the inputs
__global LLR * restrict cnInputsGM,

// Now the outputs
__global LLR * restrict cnOutputsGM,

// Now a place for the results
__global LLR * restrict cnResults

) {



  int id = get_global_id(0);
  printf("CN,%d,started\n",id);

  // First we need to store a place in private memory for operation
  LLR cnInputs[N_CHECK_INPUTS];

  // And also a spot for the magnitudes
  LLR cnInputsMag[N_CHECK_INPUTS];

  // Ok. Now we need to do some similar work to the VNs.
  // Let's make an array for the offsets
  int vnMemoryOffsets[2];
  vnMemoryOffsets[0] = 0;
  vnMemoryOffsets[1] = N_VARIABLE_NODES * (N_VARIABLE_INPUTS+1);
  int cnMemoryOffsets[2];
  cnMemoryOffsets[0] = 0;
  cnMemoryOffsets[1] = N_CHECK_NODES * (N_CHECK_INPUTS+1);

  // Check Node Private Variables
  LLR cnFirstMin;
  LLR cnSecondMin;
  LLR cnParity;

  // And let's store an offset
  char offsetIndex = 0; // Note that we start on the same data as VN but we have to wait
      ↪ for the ramp up of the VN to complete the first iterations before we can execute

  printf("CN,%d,prepped\n",id);
  char temp;

  int cnInputsNodeIndex = id * N_CHECK_INPUTS;
  int cnInputsGlobalIndex = id * (N_CHECK_INPUTS+1);

  // Alright let's start the iterations
  for(char iter = 0; iter < (N_ITERATIONS * 2); iter++) {

    if(id == 0)
      temp = read_channel_altera(cnStart);

    barrier(CLK_GLOBAL_MEM_FENCE);


    printf("CN,%d,%d\n",id,iter);
    // For each iteration we need to:
    // 1) Wait for the ok from the VNs to start executing on the information
    // 2) Load in the new inputs from GM
    // 3) Do the Cn
    // 4) Send the data to the VN via GM
    // 5) Tell the VN we have finished


    // Now we can begin our execution
    cnFirstMin = (*vector_max_llr);
    cnSecondMin = (*vector_max_llr);

    // First let's do the minimum calculations

    // But first, we need to make sure that we are taking the absolute value as we don't
        ↪ want to include that in the min calculation
    // Therefore, we have to mask off the sign bits
    #pragma unroll
    for(short input = 0; input < N_CHECK_INPUTS; input++) {
      cnInputs[input] = cnInputsGM[cnMemoryOffsets[offsetIndex] + cnInputsGlobalIndex +
          ↪ input];
    }

    // Now that we have the normal inputs, we need to get the magnitudes
    #pragma unroll
    for(short input = 0; input < N_CHECK_INPUTS; input++) {
      cnInputsMag[input] = cnInputs[input] & (*vector_max_llr);
    }

    // Ok now that we have the magnitudes, let's get to work by getting the overall parity
        ↪  first
    cnParity = (*zeros);
    #pragma unroll
    for(short input = 0; input < N_CHECK_INPUTS; input++) {
      cnParity ^= cnInputs[input] & (*vector_sign_bit);
    }

    // Now cnParity holds the overall parity of the check node

    // We're going to try something a little different here
    // We're going to implement the tree using min on the magnitudes

    // This is a little trickier as we need to find the losers a lot
    // But let's start it out.


    /************************************** ROUND 1 *************************************/
    /************************************** ROUND 1 *************************************/
LLR comp_0_to_1 = (cnInputsMag[0] < cnInputsMag[1]) ? (*allones) : (*zeros);
LLR comp_2_to_3 = (cnInputsMag[2] < cnInputsMag[3]) ? (*allones) : (*zeros);
LLR comp_4_to_5 = (cnInputsMag[4] < cnInputsMag[5]) ? (*allones) : (*zeros);
LLR comp_6_to_7 = (cnInputsMag[6] < cnInputsMag[7]) ? (*allones) : (*zeros);
LLR comp_8_to_9 = (cnInputsMag[8] < cnInputsMag[9]) ? (*allones) : (*zeros);
LLR comp_10_to_11 = (cnInputsMag[10] < cnInputsMag[11]) ? (*allones) : (*zeros);
LLR comp_12_to_13 = (cnInputsMag[12] < cnInputsMag[13]) ? (*allones) : (*zeros);
LLR comp_14_to_15 = (cnInputsMag[14] < cnInputsMag[15]) ? (*allones) : (*zeros);
LLR comp_16_to_17 = (cnInputsMag[16] < cnInputsMag[17]) ? (*allones) : (*zeros);
LLR comp_18_to_19 = (cnInputsMag[18] < cnInputsMag[19]) ? (*allones) : (*zeros);
LLR comp_20_to_21 = (cnInputsMag[20] < cnInputsMag[21]) ? (*allones) : (*zeros);
LLR comp_22_to_23 = (cnInputsMag[22] < cnInputsMag[23]) ? (*allones) : (*zeros);
LLR comp_24_to_25 = (cnInputsMag[24] < cnInputsMag[25]) ? (*allones) : (*zeros);
LLR comp_26_to_27 = (cnInputsMag[26] < cnInputsMag[27]) ? (*allones) : (*zeros);
LLR comp_28_to_29 = (cnInputsMag[28] < cnInputsMag[29]) ? (*allones) : (*zeros);
LLR comp_30_to_31 = (cnInputsMag[30] < cnInputsMag[31]) ? (*allones) : (*zeros);
LLR comp_32_to_33 = (cnInputsMag[32] < cnInputsMag[33]) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_1 = (cnInputsMag[0] & comp_0_to_1) | (cnInputsMag[1] & (comp_0_to_1 ^ (*
     ↪ allones)));
// Second Minimum
LLR loser_0 = (cnInputsMag[1] & comp_0_to_1) | (cnInputsMag[0] & (comp_0_to_1 ^ (*allones)
     ↪ ));


// First Minimum
LLR winner_2_to_3 = (cnInputsMag[2] & comp_2_to_3) | (cnInputsMag[3] & (comp_2_to_3 ^ (*
     ↪ allones)));
// Second Minimum
LLR loser_1 = (cnInputsMag[3] & comp_2_to_3) | (cnInputsMag[2] & (comp_2_to_3 ^ (*allones)
     ↪ ));


// First Minimum
LLR winner_4_to_5 = (cnInputsMag[4] & comp_4_to_5) | (cnInputsMag[5] & (comp_4_to_5 ^ (*
     ↪ allones)));
// Second Minimum
LLR loser_2 = (cnInputsMag[5] & comp_4_to_5) | (cnInputsMag[4] & (comp_4_to_5 ^ (*allones)
     ↪ ));


// First Minimum
LLR winner_6_to_7 = (cnInputsMag[6] & comp_6_to_7) | (cnInputsMag[7] & (comp_6_to_7 ^ (*
     ↪ allones)));
// Second Minimum
LLR loser_3 = (cnInputsMag[7] & comp_6_to_7) | (cnInputsMag[6] & (comp_6_to_7 ^ (*allones)
     ↪ ));


// First Minimum
LLR winner_8_to_9 = (cnInputsMag[8] & comp_8_to_9) | (cnInputsMag[9] & (comp_8_to_9 ^ (*
     ↪ allones)));
// Second Minimum
LLR loser_4 = (cnInputsMag[9] & comp_8_to_9) | (cnInputsMag[8] & (comp_8_to_9 ^ (*allones)
     ↪ ));


// First Minimum
LLR winner_10_to_11 = (cnInputsMag[10] & comp_10_to_11) | (cnInputsMag[11] & (
     ↪ comp_10_to_11 ^ (*allones)));
// Second Minimum
LLR loser_5 = (cnInputsMag[11] & comp_10_to_11) | (cnInputsMag[10] & (comp_10_to_11 ^ (*
     ↪ allones)));


// First Minimum
LLR winner_12_to_13 = (cnInputsMag[12] & comp_12_to_13) | (cnInputsMag[13] & (
     ↪ comp_12_to_13 ^ (*allones)));
// Second Minimum
LLR loser_6 = (cnInputsMag[13] & comp_12_to_13) | (cnInputsMag[12] & (comp_12_to_13 ^ (*
     ↪ allones)));


// First Minimum
LLR winner_14_to_15 = (cnInputsMag[14] & comp_14_to_15) | (cnInputsMag[15] & (
     ↪ comp_14_to_15 ^ (*allones)));
// Second Minimum
LLR loser_7 = (cnInputsMag[15] & comp_14_to_15) | (cnInputsMag[14] & (comp_14_to_15 ^ (*
     ↪ allones)));


// First Minimum
LLR winner_16_to_17 = (cnInputsMag[16] & comp_16_to_17) | (cnInputsMag[17] & (
     ↪ comp_16_to_17 ^ (*allones)));
// Second Minimum
LLR loser_8 = (cnInputsMag[17] & comp_16_to_17) | (cnInputsMag[16] & (comp_16_to_17 ^ (*
     ↪ allones)));


// First Minimum
LLR winner_18_to_19 = (cnInputsMag[18] & comp_18_to_19) | (cnInputsMag[19] & (
     ↪ comp_18_to_19 ^ (*allones)));
// Second Minimum
LLR loser_9 = (cnInputsMag[19] & comp_18_to_19) | (cnInputsMag[18] & (comp_18_to_19 ^ (*
     ↪ allones)));


// First Minimum
LLR winner_20_to_21 = (cnInputsMag[20] & comp_20_to_21) | (cnInputsMag[21] & (
     ↪ comp_20_to_21 ^ (*allones)));
// Second Minimum
LLR loser_10  = (cnInputsMag[21] & comp_20_to_21) | (cnInputsMag[20] & (comp_20_to_21 ^ (*
     ↪ allones)));
```

```
// First Minimum
LLR winner_22_to_23 = (cnInputsMag[22] & comp_22_to_23) | (cnInputsMag[23] & (
    ↪ comp_22_to_23 ^ (*allones)));
// Second Minimum
LLR loser_11  = (cnInputsMag[23] & comp_22_to_23) | (cnInputsMag[22] & (comp_22_to_23 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_24_to_25 = (cnInputsMag[24] & comp_24_to_25) | (cnInputsMag[25] & (
    ↪ comp_24_to_25 ^ (*allones)));
// Second Minimum
LLR loser_12  = (cnInputsMag[25] & comp_24_to_25) | (cnInputsMag[24] & (comp_24_to_25 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_26_to_27 = (cnInputsMag[26] & comp_26_to_27) | (cnInputsMag[27] & (
    ↪ comp_26_to_27 ^ (*allones)));
// Second Minimum
LLR loser_13  = (cnInputsMag[27] & comp_26_to_27) | (cnInputsMag[26] & (comp_26_to_27 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_28_to_29 = (cnInputsMag[28] & comp_28_to_29) | (cnInputsMag[29] & (
    ↪ comp_28_to_29 ^ (*allones)));
// Second Minimum
LLR loser_14  = (cnInputsMag[29] & comp_28_to_29) | (cnInputsMag[28] & (comp_28_to_29 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_30_to_31 = (cnInputsMag[30] & comp_30_to_31) | (cnInputsMag[31] & (
    ↪ comp_30_to_31 ^ (*allones)));
// Second Minimum
LLR loser_15  = (cnInputsMag[31] & comp_30_to_31) | (cnInputsMag[30] & (comp_30_to_31 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_32_to_33 = (cnInputsMag[32] & comp_32_to_33) | (cnInputsMag[33] & (
    ↪ comp_32_to_33 ^ (*allones)));
// Second Minimum
LLR loser_16  = (cnInputsMag[33] & comp_32_to_33) | (cnInputsMag[32] & (comp_32_to_33 ^ (*
    ↪ allones)));


/*---------------------------------- END ROUND 1 ----------------------------------*/




/************************************ ROUND 2 ************************************/
LLR comp_0_to_3 = (winner_0_to_1 < winner_2_to_3) ? (*allones) : (*zeros);
LLR comp_4_to_7 = (winner_4_to_5 < winner_6_to_7) ? (*allones) : (*zeros);
LLR comp_8_to_11  = (winner_8_to_9 < winner_10_to_11) ? (*allones) : (*zeros);
LLR comp_12_to_15 = (winner_12_to_13 < winner_14_to_15) ? (*allones) : (*zeros);
LLR comp_16_to_19 = (winner_16_to_17 < winner_18_to_19) ? (*allones) : (*zeros);
LLR comp_20_to_23 = (winner_20_to_21 < winner_22_to_23) ? (*allones) : (*zeros);
LLR comp_24_to_27 = (winner_24_to_25 < winner_26_to_27) ? (*allones) : (*zeros);
LLR comp_28_to_31 = (winner_28_to_29 < winner_30_to_31) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_3 = (winner_0_to_1 & comp_0_to_3) | (winner_2_to_3 & (comp_0_to_3 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_17  = (winner_2_to_3 & comp_0_to_3) | (winner_0_to_1 & (comp_0_to_3 ^ (*allones)
    ↪ ));


// First Minimum
LLR winner_4_to_7 = (winner_4_to_5 & comp_4_to_7) | (winner_6_to_7 & (comp_4_to_7 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_18  = (winner_6_to_7 & comp_4_to_7) | (winner_4_to_5 & (comp_4_to_7 ^ (*allones)
    ↪ ));


// First Minimum
LLR winner_8_to_11  = (winner_8_to_9 & comp_8_to_11) | (winner_10_to_11 & (comp_8_to_11 ^
    ↪ (*allones)));
// Second Minimum
LLR loser_19  = (winner_10_to_11 & comp_8_to_11) | (winner_8_to_9 & (comp_8_to_11 ^ (*
    ↪ allones)));


// First Minimum
```

```
LLR winner_12_to_15 = (winner_12_to_13 & comp_12_to_15) | (winner_14_to_15 & (
    ↪ comp_12_to_15 ^ (*allones)));
// Second Minimum
LLR loser_20  = (winner_14_to_15 & comp_12_to_15) | (winner_12_to_13 & (comp_12_to_15 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_16_to_19 = (winner_16_to_17 & comp_16_to_19) | (winner_18_to_19 & (
    ↪ comp_16_to_19 ^ (*allones)));
// Second Minimum
LLR loser_21  = (winner_18_to_19 & comp_16_to_19) | (winner_16_to_17 & (comp_16_to_19 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_20_to_23 = (winner_20_to_21 & comp_20_to_23) | (winner_22_to_23 & (
    ↪ comp_20_to_23 ^ (*allones)));
// Second Minimum
LLR loser_22  = (winner_22_to_23 & comp_20_to_23) | (winner_20_to_21 & (comp_20_to_23 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_24_to_27 = (winner_24_to_25 & comp_24_to_27) | (winner_26_to_27 & (
    ↪ comp_24_to_27 ^ (*allones)));
// Second Minimum
LLR loser_23  = (winner_26_to_27 & comp_24_to_27) | (winner_24_to_25 & (comp_24_to_27 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_28_to_31 = (winner_28_to_29 & comp_28_to_31) | (winner_30_to_31 & (
    ↪ comp_28_to_31 ^ (*allones)));
// Second Minimum
LLR loser_24  = (winner_30_to_31 & comp_28_to_31) | (winner_28_to_29 & (comp_28_to_31 ^ (*
    ↪ allones)));


/*---------------------------------- END ROUND 2 ----------------------------------*/




/************************************ ROUND 3 ************************************/
LLR comp_0_to_7 = (winner_0_to_3 < winner_4_to_7) ? (*allones) : (*zeros);
LLR comp_8_to_15  = (winner_8_to_11 < winner_12_to_15) ? (*allones) : (*zeros);
LLR comp_16_to_23 = (winner_16_to_19 < winner_20_to_23) ? (*allones) : (*zeros);
LLR comp_24_to_31 = (winner_24_to_27 < winner_28_to_31) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_7 = (winner_0_to_3 & comp_0_to_7) | (winner_4_to_7 & (comp_0_to_7 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_25  = (winner_4_to_7 & comp_0_to_7) | (winner_0_to_3 & (comp_0_to_7 ^ (*allones)
    ↪ ));


// First Minimum
LLR winner_8_to_15  = (winner_8_to_11 & comp_8_to_15) | (winner_12_to_15 & (comp_8_to_15 ^
    ↪  (*allones)));
// Second Minimum
LLR loser_26  = (winner_12_to_15 & comp_8_to_15) | (winner_8_to_11 & (comp_8_to_15 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_16_to_23 = (winner_16_to_19 & comp_16_to_23) | (winner_20_to_23 & (
    ↪ comp_16_to_23 ^ (*allones)));
// Second Minimum
LLR loser_27  = (winner_20_to_23 & comp_16_to_23) | (winner_16_to_19 & (comp_16_to_23 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_24_to_31 = (winner_24_to_27 & comp_24_to_31) | (winner_28_to_31 & (
    ↪ comp_24_to_31 ^ (*allones)));
// Second Minimum
LLR loser_28  = (winner_28_to_31 & comp_24_to_31) | (winner_24_to_27 & (comp_24_to_31 ^ (*
    ↪ allones)));


/*---------------------------------- END ROUND 3 ----------------------------------*/




/************************************ ROUND 4 ************************************/
LLR comp_0_to_15  = (winner_0_to_7 < winner_8_to_15) ? (*allones) : (*zeros);
LLR comp_16_to_31 = (winner_16_to_23 < winner_24_to_31) ? (*allones) : (*zeros);
```

```
// First Minimum
LLR winner_0_to_15  = (winner_0_to_7 & comp_0_to_15) | (winner_8_to_15 & (comp_0_to_15 ^
    ↪ (*allones)));
// Second Minimum
LLR loser_29  = (winner_8_to_15 & comp_0_to_15) | (winner_0_to_7 & (comp_0_to_15 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_16_to_31 = (winner_16_to_23 & comp_16_to_31) | (winner_24_to_31 & (
    ↪ comp_16_to_31 ^ (*allones)));
// Second Minimum
LLR loser_30  = (winner_24_to_31 & comp_16_to_31) | (winner_16_to_23 & (comp_16_to_31 ^ (*
    ↪ allones)));


/*----------------------------------- END ROUND 4 -----------------------------------*/




/************************************ ROUND 5 ************************************/
LLR comp_0_to_31  = (winner_0_to_15 < winner_16_to_31) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_31  = (winner_0_to_15 & comp_0_to_31) | (winner_16_to_31 & (comp_0_to_31 ^
    ↪  (*allones)));
// Second Minimum
LLR loser_31  = (winner_16_to_31 & comp_0_to_31) | (winner_0_to_15 & (comp_0_to_31 ^ (*
    ↪ allones)));


/*----------------------------------- END ROUND 5 -----------------------------------*/




/************************************ ROUND 6 ************************************/
LLR comp_0_to_33  = (winner_0_to_31 < winner_32_to_33) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_33  = (winner_0_to_31 & comp_0_to_33) | (winner_32_to_33 & (comp_0_to_33 ^
    ↪  (*allones)));
// Second Minimum
LLR loser_32  = (winner_32_to_33 & comp_0_to_33) | (winner_0_to_31 & (comp_0_to_33 ^ (*
    ↪ allones)));


/*----------------------------------- END ROUND 6 -----------------------------------*/




cnFirstMin = winner_0_to_33;





/****************************** SECOND MINIMUM ******************************/


/************************************ ROUND 1 ************************************/
LLR second_comp_0_to_1  = (loser_0  < loser_1) ? (*allones) : (*zeros);
LLR second_comp_2_to_3  = (loser_2  < loser_3) ? (*allones) : (*zeros);
LLR second_comp_4_to_5  = (loser_4  < loser_5) ? (*allones) : (*zeros);
LLR second_comp_6_to_7  = (loser_6  < loser_7) ? (*allones) : (*zeros);
LLR second_comp_8_to_9  = (loser_8  < loser_9) ? (*allones) : (*zeros);
LLR second_comp_10_to_11  = (loser_10  < loser_11) ? (*allones) : (*zeros);
LLR second_comp_12_to_13  = (loser_12  < loser_13) ? (*allones) : (*zeros);
LLR second_comp_14_to_15  = (loser_14  < loser_15) ? (*allones) : (*zeros);
LLR second_comp_16_to_17  = (loser_16  < loser_17) ? (*allones) : (*zeros);
LLR second_comp_18_to_19  = (loser_18  < loser_19) ? (*allones) : (*zeros);
LLR second_comp_20_to_21  = (loser_20  < loser_21) ? (*allones) : (*zeros);
LLR second_comp_22_to_23  = (loser_22  < loser_23) ? (*allones) : (*zeros);
LLR second_comp_24_to_25  = (loser_24  < loser_25) ? (*allones) : (*zeros);
LLR second_comp_26_to_27  = (loser_26  < loser_27) ? (*allones) : (*zeros);
LLR second_comp_28_to_29  = (loser_28  < loser_29) ? (*allones) : (*zeros);
LLR second_comp_30_to_31  = (loser_30  < loser_31) ? (*allones) : (*zeros);



LLR second_winner_0_to_1  = (loser_0 & second_comp_0_to_1) | (loser_1 & (
    ↪ second_comp_0_to_1 ^ (*allones)));
```

```
LLR second_winner_2_to_3  = (loser_2 & second_comp_2_to_3) | (loser_3 & (
    ↪ second_comp_2_to_3 ^ (*allones)));
LLR second_winner_4_to_5  = (loser_4 & second_comp_4_to_5) | (loser_5 & (
    ↪ second_comp_4_to_5 ^ (*allones)));
LLR second_winner_6_to_7  = (loser_6 & second_comp_6_to_7) | (loser_7 & (
    ↪ second_comp_6_to_7 ^ (*allones)));
LLR second_winner_8_to_9  = (loser_8 & second_comp_8_to_9) | (loser_9 & (
    ↪ second_comp_8_to_9 ^ (*allones)));
LLR second_winner_10_to_11  = (loser_10 & second_comp_10_to_11) | (loser_11 & (
    ↪ second_comp_10_to_11 ^ (*allones)));
LLR second_winner_12_to_13  = (loser_12 & second_comp_12_to_13) | (loser_13 & (
    ↪ second_comp_12_to_13 ^ (*allones)));
LLR second_winner_14_to_15  = (loser_14 & second_comp_14_to_15) | (loser_15 & (
    ↪ second_comp_14_to_15 ^ (*allones)));
LLR second_winner_16_to_17  = (loser_16 & second_comp_16_to_17) | (loser_17 & (
    ↪ second_comp_16_to_17 ^ (*allones)));
LLR second_winner_18_to_19  = (loser_18 & second_comp_18_to_19) | (loser_19 & (
    ↪ second_comp_18_to_19 ^ (*allones)));
LLR second_winner_20_to_21  = (loser_20 & second_comp_20_to_21) | (loser_21 & (
    ↪ second_comp_20_to_21 ^ (*allones)));
LLR second_winner_22_to_23  = (loser_22 & second_comp_22_to_23) | (loser_23 & (
    ↪ second_comp_22_to_23 ^ (*allones)));
LLR second_winner_24_to_25  = (loser_24 & second_comp_24_to_25) | (loser_25 & (
    ↪ second_comp_24_to_25 ^ (*allones)));
LLR second_winner_26_to_27  = (loser_26 & second_comp_26_to_27) | (loser_27 & (
    ↪ second_comp_26_to_27 ^ (*allones)));
LLR second_winner_28_to_29  = (loser_28 & second_comp_28_to_29) | (loser_29 & (
    ↪ second_comp_28_to_29 ^ (*allones)));
LLR second_winner_30_to_31  = (loser_30 & second_comp_30_to_31) | (loser_31 & (
    ↪ second_comp_30_to_31 ^ (*allones)));
/*---------------------------------- END ROUND 1 ----------------------------------*/



/************************************* ROUND 2 *************************************/
LLR second_comp_0_to_3  = (second_winner_0_to_1 < second_winner_2_to_3) ? (*allones) : (*
    ↪ zeros);
LLR second_comp_4_to_7  = (second_winner_4_to_5 < second_winner_6_to_7) ? (*allones) : (*
    ↪ zeros);
LLR second_comp_8_to_11 = (second_winner_8_to_9 < second_winner_10_to_11) ? (*allones) :
    ↪ (*zeros);
LLR second_comp_12_to_15  = (second_winner_12_to_13 < second_winner_14_to_15) ? (*allones)
    ↪  : (*zeros);
LLR second_comp_16_to_19  = (second_winner_16_to_17 < second_winner_18_to_19) ? (*allones)
    ↪  : (*zeros);
LLR second_comp_20_to_23  = (second_winner_20_to_21 < second_winner_22_to_23) ? (*allones)
    ↪  : (*zeros);
LLR second_comp_24_to_27  = (second_winner_24_to_25 < second_winner_26_to_27) ? (*allones)
    ↪  : (*zeros);
LLR second_comp_28_to_31  = (second_winner_28_to_29 < second_winner_30_to_31) ? (*allones)
    ↪  : (*zeros);



LLR second_winner_0_to_3  = (second_winner_0_to_1 & second_comp_0_to_3) | (
    ↪ second_winner_2_to_3 & (second_comp_0_to_3 ^ (*allones)));
LLR second_winner_4_to_7  = (second_winner_4_to_5 & second_comp_4_to_7) | (
    ↪ second_winner_6_to_7 & (second_comp_4_to_7 ^ (*allones)));
LLR second_winner_8_to_11 = (second_winner_8_to_9 & second_comp_8_to_11) | (
    ↪ second_winner_10_to_11 & (second_comp_8_to_11 ^ (*allones)));
LLR second_winner_12_to_15  = (second_winner_12_to_13 & second_comp_12_to_15) | (
    ↪ second_winner_14_to_15 & (second_comp_12_to_15 ^ (*allones)));
LLR second_winner_16_to_19  = (second_winner_16_to_17 & second_comp_16_to_19) | (
    ↪ second_winner_18_to_19 & (second_comp_16_to_19 ^ (*allones)));
LLR second_winner_20_to_23  = (second_winner_20_to_21 & second_comp_20_to_23) | (
    ↪ second_winner_22_to_23 & (second_comp_20_to_23 ^ (*allones)));
LLR second_winner_24_to_27  = (second_winner_24_to_25 & second_comp_24_to_27) | (
    ↪ second_winner_26_to_27 & (second_comp_24_to_27 ^ (*allones)));
LLR second_winner_28_to_31  = (second_winner_28_to_29 & second_comp_28_to_31) | (
    ↪ second_winner_30_to_31 & (second_comp_28_to_31 ^ (*allones)));
/*---------------------------------- END ROUND 2 ----------------------------------*/



/************************************* ROUND 3 *************************************/
LLR second_comp_0_to_7  = (second_winner_0_to_3 < second_winner_4_to_7) ? (*allones) : (*
    ↪ zeros);
LLR second_comp_8_to_15 = (second_winner_8_to_11 < second_winner_12_to_15) ? (*allones) :
    ↪ (*zeros);
LLR second_comp_16_to_23  = (second_winner_16_to_19 < second_winner_20_to_23) ? (*allones)
    ↪  : (*zeros);
LLR second_comp_24_to_31  = (second_winner_24_to_27 < second_winner_28_to_31) ? (*allones)
    ↪  : (*zeros);



LLR second_winner_0_to_7  = (second_winner_0_to_3 & second_comp_0_to_7) | (
    ↪ second_winner_4_to_7 & (second_comp_0_to_7 ^ (*allones)));
LLR second_winner_8_to_15 = (second_winner_8_to_11 & second_comp_8_to_15) | (
    ↪ second_winner_12_to_15 & (second_comp_8_to_15 ^ (*allones)));
```

```
LLR second_winner_16_to_23  = (second_winner_16_to_19 & second_comp_16_to_23) | (
     ↪ second_winner_20_to_23 & (second_comp_16_to_23 ^ (*allones)));
LLR second_winner_24_to_31  = (second_winner_24_to_27 & second_comp_24_to_31) | (
     ↪ second_winner_28_to_31 & (second_comp_24_to_31 ^ (*allones)));
/*----------------------------------- END ROUND 3 -----------------------------------*/



/************************************ ROUND 4 ************************************/
LLR second_comp_0_to_15 = (second_winner_0_to_7 < second_winner_8_to_15) ? (*allones) : (*
     ↪ zeros);
LLR second_comp_16_to_31  = (second_winner_16_to_23 < second_winner_24_to_31) ? (*allones)
     ↪  : (*zeros);



LLR second_winner_0_to_15 = (second_winner_0_to_7 & second_comp_0_to_15) | (
     ↪ second_winner_8_to_15 & (second_comp_0_to_15 ^ (*allones)));
LLR second_winner_16_to_31  = (second_winner_16_to_23 & second_comp_16_to_31) | (
     ↪ second_winner_24_to_31 & (second_comp_16_to_31 ^ (*allones)));
/*----------------------------------- END ROUND 4 -----------------------------------*/



/************************************ ROUND 5 ************************************/
LLR second_comp_0_to_31 = (second_winner_0_to_15 < second_winner_16_to_31) ? (*allones) :
     ↪ (*zeros);



LLR second_winner_0_to_31 = (second_winner_0_to_15 & second_comp_0_to_31) | (
     ↪ second_winner_16_to_31 & (second_comp_0_to_31 ^ (*allones)));
/*----------------------------------- END ROUND 5 -----------------------------------*/



/************************************ ROUND 6 ************************************/
LLR second_comp_0_to_32 = (second_winner_0_to_31 < loser_32) ? (*allones) : (*zeros);



LLR second_winner_0_to_32 = (second_winner_0_to_31 & second_comp_0_to_32) | (loser_32 & (
     ↪ second_comp_0_to_32 ^ (*allones)));
/*----------------------------------- END ROUND 6 -----------------------------------*/



cnSecondMin = second_winner_0_to_32;





    // Now that we have the minimums and the parity, we should convert the first and
         ↪ second minimums to 2's complement assuming negative
    LLR first_min_neg = (((cnFirstMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));

    LLR second_min_neg = (((cnSecondMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));

    /*for(int i=0; i<N_CHECK_INPUTS; i++) {
      printf("Iteration,%d,CNode,%d,input,%d,%d,%d,min,%d,second_min,%d,address,%d\n",iter
           ↪ ,id,i,cnInputs[i].s0 & (*vector_sign_bit).s0,cnInputsMag[i].s0,cnFirstMin.s0
           ↪ ,cnSecondMin.s0,cnMemoryOffsets[offsetIndex] + cnInputsGlobalIndex + i);
    }*/




    // Now let's start the check nodes
    // We only need to execute one check node for each calculation as there are
         ↪ N_CHECK_NODE number of work-items
    //#pragma unroll
    for(char input = 0; input < N_CHECK_INPUTS; input++) {



      // For each of these we need to determine if it was the first minimum
      // We just need a conditional that has either 1 or 0 depending if it's min or second
           ↪  min
      // And then we can make the expression LLR to_vnode = cnFirstMin*condition1 +
           ↪ cnSecondMin*inv(condition1)
      LLR min_conditional = ((cnInputs[input] & (*vector_max_llr)) == cnFirstMin) ? (*
           ↪ allones) : (*zeros);

      // the LLR conditional variable will now have a vector of either 0 or 1.
      // Now we can just use the expression
      // Now we can move on to calculating the output
      LLR sign_b = cnParity;

      // The sign bit is assigned to complete the parity compared to the other inputs
      sign_b ^= cnInputs[input] & (*vector_sign_bit);

      // Now we just send the data in 2's complement which was already calculated
      LLR sign_conditional = (sign_b == (*vector_sign_bit)) ? (*allones) : (*zeros);

      // If it's negative, give it the negative value, if not then the positive
      // Start with cnSecondMin, second_min_neg, cnFirstMin, first_min_neg
      LLR to_vnode =  ((min_conditional & sign_conditional & second_min_neg)) |
            ((min_conditional & (sign_conditional ^ (*allones)) & cnSecondMin)) |
            ((min_conditional ^ (*allones)) & sign_conditional & first_min_neg) |
            ((min_conditional ^ (*allones)) & (sign_conditional ^ (*allones)) & cnFirstMin
                ↪ );



      // Now we just need to find the variable node index
      int vnIndex = cntovnIndex[cnInputsNodeIndex + input] + vnMemoryOffsets[offsetIndex];

      cnOutputsGM[vnIndex] = to_vnode;

      //printf("Iteration,%d,CNode,%d,output,%d,%d,vnIndex,%d,\n",iter,id,input,to_vnode.
          ↪ s0,vnIndex);

    }






    barrier(CLK_GLOBAL_MEM_FENCE);
    offsetIndex ^= 1;
    if(id == 0)
      write_channel_altera(vnStart,SENDDATA);




  }

}
```

Appendix S

Design F Length-1120 OpenCL Source Code

```
#ifndef N_VARIABLE_NODES
#define N_VARIABLE_NODES 1120
#endif

#ifndef N_CHECK_NODES
#define N_CHECK_NODES 280
#endif

#ifndef SIMD
#define SIMD 1
#endif

// Based on the Tanner graph, how many connections are there
#define MAX_VARIABLE_EDGES 5
#define MAX_CHECK_EDGES 16


// Let's define the usable number of inputs
#define N_VARIABLE_INPUTS (MAX_VARIABLE_EDGES)
#define N_CHECK_INPUTS (MAX_CHECK_EDGES)

#define N_VARIABLE_INPUTS_W_CHANNEL (N_VARIABLE_INPUTS + 1)

#define N_ITERATIONS (3)



// Set the vector width and the appropriate LLR types
#define MANUAL_VECTOR_WIDTH (16)

#if MANUAL_VECTOR_WIDTH==2
#define LLR_TYPE char2
#define TOTAL_TYPE short2
#define CONVERT_TO_SHORT(a) convert_short2(a)
#define CONVERT_TO_CHAR(a) convert_char2(a)
#elif MANUAL_VECTOR_WIDTH==4
#define LLR_TYPE char4
#define TOTAL_TYPE short4
#define CONVERT_TO_SHORT(a) convert_short4(a)
#define CONVERT_TO_CHAR(a) convert_char4(a)
#elif MANUAL_VECTOR_WIDTH==8
#define LLR_TYPE char8
#define TOTAL_TYPE short8
#define CONVERT_TO_SHORT(a) convert_short8(a)
#define CONVERT_TO_CHAR(a) convert_char8(a)
#elif MANUAL_VECTOR_WIDTH==16
#define LLR_TYPE char16
#define TOTAL_TYPE short16
#define CONVERT_TO_SHORT(a) convert_short16(a)
#define CONVERT_TO_CHAR(a) convert_char16(a)
#else
#define LLR_TYPE char
#define TOTAL_TYPE short
#define CONVERT_TO_SHORT(a) convert_short(a)
#define CONVERT_TO_CHAR(a) convert_char(a)
#endif

typedef LLR_TYPE LLR;


// Now for the channels
#pragma OPENCL EXTENSION cl_altera_channels : enable
```

```
channel char vnStart __attribute__((depth(N_VARIABLE_NODES*2)));
channel char cnStart __attribute__((depth(N_VARIABLE_NODES*2)));
#define SENDDATA (1)

//Variable Node Kernel
__kernel
__attribute__((reqd_work_group_size(N_VARIABLE_NODES,1,1)))
__attribute__((num_compute_units(1)))
__attribute__((num_simd_work_items(SIMD)))
void VariableNode(

// Kernel parameteres
// As in the notes page, there is a table that determines the write location in the CNs
__constant short * restrict vntocnIndex,

// Now that we have the connection information let's get the inputs
// Remember there is one memory location for the Variable Node inputs with two slots for
    ↪ pipelining
// Let's start with the channel measurements
__global LLR * restrict channelMeasurements,

// Now let's get the inputs
__global LLR * restrict vnInputsGM,

// Now let's get the output pointer to the CNs
__global LLR * restrict vnOutputsGM,

// And of course finally, we need to get the result
__global TOTAL_TYPE * restrict vnResults,


// We need a few constants where it's easier to compute them in the host
__constant LLR * restrict vector_max_llr,
__constant LLR * restrict vector_sign_bit,
__constant LLR * restrict allones,
__constant LLR * restrict ones,
__constant LLR * restrict zeros,
__constant TOTAL_TYPE * restrict total_vector_max_llr,
__constant TOTAL_TYPE * restrict total_vector_min_neg_llr,

char iter,
char offsetIndex
)
{

  // First we need to get our global IDsa
  int id = get_global_id(0);



  // We also need a short version of the LLR values to total
  TOTAL_TYPE vnInputsShort[N_VARIABLE_INPUTS_W_CHANNEL];

  // Let's make an array for the offsets
  int vnMemoryOffsets[2];
  vnMemoryOffsets[0] = 0;
  vnMemoryOffsets[1] = N_VARIABLE_NODES * (N_VARIABLE_INPUTS+1);

  int cnMemoryOffsets[2];
  cnMemoryOffsets[0] = 0;
  cnMemoryOffsets[1] = N_CHECK_NODES * (N_CHECK_INPUTS+1);

  // Let's make an array index lookup for the input
  int vntocnNodeIndex = id * (N_VARIABLE_INPUTS);
  int vnGlobalNodeIndex = id * (N_VARIABLE_INPUTS+ 1);

  // Some other variables
  LLR vnCurrent;
  TOTAL_TYPE vnTempCurrent;
  LLR vnConditional;
  LLR vnSignB;
  LLR vnMag;

  char temp = 0;



  // Let's start the process
  // We need to step through the iterations (aka start the loop)
  // Since both VN and CN will be running concurrently on the same data, we will need
      ↪ N_ITERATIONS*2+1

  // Let's just do some printing
  //printf("VN,%d,%d\n",id,iter);
  // Alright so each iteration we need to do the following:
  // Each VN:
  // 1) Wait for the OK from the CNs to start on the data
  // 2) Load up the inputs from DRAM into private memory
  // 3) Do the VN
  // 4) Output the data to the appropriate buffer
  // 5) Send the ok to the CN to start execution
  // 6) Swap buffers
  // Repeat


  // Let's wait for the CNs to tell us it's ok to start execution
  // Note that we can throw away the data as this is a blocking call



  // Now that we've got the OK. Let's start execution
  // First let's load up the inputs and convert them to shorts so that we can get the
      ↪ appropriate totals without saturation
  #pragma unroll
  for(char input = 0; input < N_VARIABLE_INPUTS; input++) {
    vnInputsShort[input] = CONVERT_TO_SHORT(vnInputsGM[vnMemoryOffsets[offsetIndex] +
        ↪ vnGlobalNodeIndex + input]);
  }

  // Now let's put in the channel measurement for our node
  vnInputsShort[N_VARIABLE_INPUTS] = CONVERT_TO_SHORT(channelMeasurements[vnMemoryOffsets[
      ↪ offsetIndex] + vntocnNodeIndex]);

  // Now we need to add it all together

  TOTAL_TYPE total = (TOTAL_TYPE)(0);
  #pragma unroll
  for(char input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {
    total += vnInputsShort[input];
  }

  // Now that we have the total we can do the rest
  for(char input = 0; input < N_VARIABLE_INPUTS; input++) {

    // Subtract the current from the total to get the output
    // We simply subtract the large total and then saturate it down to the appropriate
        ↪ size
    vnTempCurrent = total - vnInputsShort[input];

    // Now saturate
    vnTempCurrent =   ((vnTempCurrent >= (*total_vector_max_llr)) ?
              (*total_vector_max_llr) :
              ((vnTempCurrent <= (*total_vector_min_neg_llr)) ?
              (*total_vector_min_neg_llr) :
              vnTempCurrent));
    // Now we reverse the procedure and store the data back into the LLR type
    vnCurrent = CONVERT_TO_CHAR(vnTempCurrent);

    // Ok now we can finally send the data
    // First we check if it's negative
    // Except we need to convert it to sign and magnitude
    vnConditional = ((vnCurrent & (*vector_sign_bit)) == (*vector_sign_bit)) ? (*allones)
        ↪ : (*zeros);

    // If it's negative we need to xor it and add one and mask out the sign bit
    vnMag = ((vnConditional & ((vnCurrent ^ (*allones)) + (*ones))) | ((vnConditional ^ (*
        ↪ allones)) & vnCurrent)) & (*vector_max_llr);

    // Now get the sign bit
    vnSignB = (vnConditional & (*vector_sign_bit));

    // Now we just need to find the location in the array to place the data
    short cnIndex = vntocnIndex[vntocnNodeIndex + input] + cnMemoryOffsets[offsetIndex];

    // Now we can just output the value
    // Since we're writing to the CNs we need to output to the write memory location
    vnOutputsGM[cnIndex] = vnMag | vnSignB;

  }

  if((iter+1)==N_ITERATIONS*2+1 || (iter+2) == N_ITERATIONS*2+2) {
    TOTAL_TYPE send_total = 0;
    for(short input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {
      send_total += vnInputsShort[input];
    }

    vnResults[vnMemoryOffsets[offsetIndex] + id] = send_total;
  }




  // Now let's switch the index over
  // To do this we can simply XOR it with the value of 1
  offsetIndex ^= 1;

  //printf("afterbarrier");

}


__kernel
__attribute__((reqd_work_group_size(N_CHECK_NODES,1,1)))
__attribute__((num_compute_units(1)))
__attribute__((num_simd_work_items(SIMD)))
void CheckNode(

// Let's start with the table for the interleaver connections just like with the VN
__constant short * restrict cntovnIndex,

// Now we can do the other constants
__constant LLR * restrict vector_max_llr,
__constant LLR * restrict vector_sign_bit,
__constant LLR * restrict allones,
__constant LLR * restrict ones,
__constant LLR * restrict zeros,

// Now we can focus on the inputs
__global LLR * restrict cnInputsGM,

// Now the outputs
__global LLR * restrict cnOutputsGM,

// Now a place for the results
__global LLR * restrict cnResults,

char iter,
char offsetIndex

) {



  int id = get_global_id(0);

  // First we need to store a place in private memory for operation
  LLR cnInputs[N_CHECK_INPUTS];

  // And also a spot for the magnitudes
  LLR cnInputsMag[N_CHECK_INPUTS];

  // Let's compute the base index
  int cnInputsNodeIndex = id * N_CHECK_INPUTS;
  int cnInputsGlobalIndex = id * (N_CHECK_INPUTS+1);

  // Check Node Private Variables
  LLR cnFirstMin;
  LLR cnSecondMin;
  LLR cnParity;

  char temp = 0;


  // Ok. Now we need to do some similar work to the VNs.
  // Let's make an array for the offsets
  int vnMemoryOffsets[2];
  vnMemoryOffsets[0] = 0;
  vnMemoryOffsets[1] = N_VARIABLE_NODES * (N_VARIABLE_INPUTS+1);

  int cnMemoryOffsets[2];
  cnMemoryOffsets[0] = 0;
  cnMemoryOffsets[1] = N_CHECK_NODES * (N_CHECK_INPUTS+1);

  // And let's store an offset


  //printf("CN,%d,prepped\n",id);

  // Alright let's start the iterations

  //printf("CN,%d,%d\n",id,iter);
  // For each iteration we need to:
  // 1) Wait for the ok from the VNs to start executing on the information
  // 2) Load in the new inputs from GM
  // 3) Do the Cn
  // 4) Send the data to the VN via GM
  // 5) Tell the VN we have finished


  // Now we can begin our execution
  cnFirstMin = (*vector_max_llr);
  cnSecondMin = (*vector_max_llr);

  // First let's do the minimum calculations

  // But first, we need to make sure that we are taking the absolute value as we don't
      ↪ want to include that in the min calculation
  // Therefore, we have to mask off the sign bits
  #pragma unroll
  for(short input = 0; input < N_CHECK_INPUTS; input++) {
    cnInputs[input] = cnInputsGM[cnMemoryOffsets[offsetIndex] + cnInputsGlobalIndex +
        ↪ input];
  }

  // Now that we have the normal inputs, we need to get the magnitudes
  #pragma unroll
```

```
for(short input = 0; input < N_CHECK_INPUTS; input++) {
  cnInputsMag[input] = cnInputs[input] & (*vector_max_llr);
}

// Ok now that we have the magnitudes, let's get to work by getting the overall parity
    ↪ first
cnParity = (*zeros);
#pragma unroll
for(short input = 0; input < N_CHECK_INPUTS; input++) {
  cnParity ^= cnInputs[input] & (*vector_sign_bit);
}

// Now cnParity holds the overall parity of the check node

// We're going to try something a little different here
// We're going to implement the tree using min on the magnitudes

// This is a little trickier as we need to find the losers a lot
// But let's start it out.


/************************************ ROUND 1 ************************************/
LLR comp_0_1  = (cnInputsMag[0] < cnInputsMag[1]) ? (*allones) : (*zeros);
LLR comp_2_3  = (cnInputsMag[2] < cnInputsMag[3]) ? (*allones) : (*zeros);
LLR comp_4_5  = (cnInputsMag[4] < cnInputsMag[5]) ? (*allones) : (*zeros);
LLR comp_6_7  = (cnInputsMag[6] < cnInputsMag[7]) ? (*allones) : (*zeros);
LLR comp_8_9  = (cnInputsMag[8] < cnInputsMag[9]) ? (*allones) : (*zeros);
LLR comp_10_11  = (cnInputsMag[10] < cnInputsMag[11]) ? (*allones) : (*zeros);
LLR comp_12_13  = (cnInputsMag[12] < cnInputsMag[13]) ? (*allones) : (*zeros);
LLR comp_14_15  = (cnInputsMag[14] < cnInputsMag[15]) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_1 = (cnInputsMag[0] & comp_0_1) | (cnInputsMag[1] & (comp_0_1 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_0 = (cnInputsMag[1] & comp_0_1) | (cnInputsMag[0] & (comp_0_1 ^ (*allones)));


// First Minimum
LLR winner_2_to_3 = (cnInputsMag[2] & comp_2_3) | (cnInputsMag[3] & (comp_2_3 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_1 = (cnInputsMag[3] & comp_2_3) | (cnInputsMag[2] & (comp_2_3 ^ (*allones)));


// First Minimum
LLR winner_4_to_5 = (cnInputsMag[4] & comp_4_5) | (cnInputsMag[5] & (comp_4_5 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_2 = (cnInputsMag[5] & comp_4_5) | (cnInputsMag[4] & (comp_4_5 ^ (*allones)));


// First Minimum
LLR winner_6_to_7 = (cnInputsMag[6] & comp_6_7) | (cnInputsMag[7] & (comp_6_7 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_3 = (cnInputsMag[7] & comp_6_7) | (cnInputsMag[6] & (comp_6_7 ^ (*allones)));


// First Minimum
LLR winner_8_to_9 = (cnInputsMag[8] & comp_8_9) | (cnInputsMag[9] & (comp_8_9 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_4 = (cnInputsMag[9] & comp_8_9) | (cnInputsMag[8] & (comp_8_9 ^ (*allones)));


// First Minimum
LLR winner_10_to_11 = (cnInputsMag[10] & comp_10_11) | (cnInputsMag[11] & (comp_10_11 ^
    ↪ (*allones)));
// Second Minimum
LLR loser_5 = (cnInputsMag[11] & comp_10_11) | (cnInputsMag[10] & (comp_10_11 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_12_to_13 = (cnInputsMag[12] & comp_12_13) | (cnInputsMag[13] & (comp_12_13 ^
    ↪ (*allones)));
// Second Minimum
LLR loser_6 = (cnInputsMag[13] & comp_12_13) | (cnInputsMag[12] & (comp_12_13 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_14_to_15 = (cnInputsMag[14] & comp_14_15) | (cnInputsMag[15] & (comp_14_15 ^
    ↪ (*allones)));
// Second Minimum
LLR loser_7 = (cnInputsMag[15] & comp_14_15) | (cnInputsMag[14] & (comp_14_15 ^ (*
    ↪ allones)));
```

```
/*------------------------------------ END ROUND 1 -----------------------------------*/



/************************************ ROUND 2 ************************************/
LLR comp_0_to_3 = (winner_0_to_1 < winner_2_to_3) ? (*allones) : (*zeros);
LLR comp_4_to_7 = (winner_4_to_5 < winner_6_to_7) ? (*allones) : (*zeros);
LLR comp_8_to_11  = (winner_8_to_9 < winner_10_to_11) ? (*allones) : (*zeros);
LLR comp_12_to_15 = (winner_12_to_13 < winner_14_to_15) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_3 = (winner_0_to_1 & comp_0_to_3) | (winner_2_to_3 & (comp_0_to_3 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_8 = (winner_2_to_3 & comp_0_to_3) | (winner_0_to_1 & (comp_0_to_3 ^ (*allones)
    ↪ ));


// First Minimum
LLR winner_4_to_7 = (winner_4_to_5 & comp_4_to_7) | (winner_6_to_7 & (comp_4_to_7 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_9 = (winner_6_to_7 & comp_4_to_7) | (winner_4_to_5 & (comp_4_to_7 ^ (*allones)
    ↪ ));


// First Minimum
LLR winner_8_to_11  = (winner_8_to_9 & comp_8_to_11) | (winner_10_to_11 & (comp_8_to_11
    ↪ ^ (*allones)));
// Second Minimum
LLR loser_10  = (winner_10_to_11 & comp_8_to_11) | (winner_8_to_9 & (comp_8_to_11 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_12_to_15 = (winner_12_to_13 & comp_12_to_15) | (winner_14_to_15 & (
    ↪ comp_12_to_15 ^ (*allones)));
// Second Minimum
LLR loser_11  = (winner_14_to_15 & comp_12_to_15) | (winner_12_to_13 & (comp_12_to_15 ^
    ↪ (*allones)));


/*------------------------------------ END ROUND 2 -----------------------------------*/



/************************************ ROUND 3 ************************************/
LLR comp_0_to_7 = (winner_0_to_3 < winner_4_to_7) ? (*allones) : (*zeros);
LLR comp_8_to_15  = (winner_8_to_11 < winner_12_to_15) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_7 = (winner_0_to_3 & comp_0_to_7) | (winner_4_to_7 & (comp_0_to_7 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_12  = (winner_4_to_7 & comp_0_to_7) | (winner_0_to_3 & (comp_0_to_7 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_8_to_15  = (winner_8_to_11 & comp_8_to_15) | (winner_12_to_15 & (comp_8_to_15
    ↪  ^ (*allones)));
// Second Minimum
LLR loser_13  = (winner_12_to_15 & comp_8_to_15) | (winner_8_to_11 & (comp_8_to_15 ^ (*
    ↪ allones)));


/*------------------------------------ END ROUND 3 -----------------------------------*/



/************************************ ROUND 4 ************************************/
LLR comp_0_to_15  = (winner_0_to_7 < winner_8_to_15) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_15  = (winner_0_to_7 & comp_0_to_15) | (winner_8_to_15 & (comp_0_to_15 ^
    ↪  (*allones)));
// Second Minimum
LLR loser_14  = (winner_8_to_15 & comp_0_to_15) | (winner_0_to_7 & (comp_0_to_15 ^ (*
    ↪ allones)));


/*------------------------------------ END ROUND 4 -----------------------------------*/
```

```
cnFirstMin = winner_0_to_15;

/************************************ ROUND 1 ************************************/
LLR second_comp_0_to_1  = (loser_0 < loser_1) ? (*allones) : (*zeros);
LLR second_comp_2_to_3  = (loser_2 < loser_3) ? (*allones) : (*zeros);
LLR second_comp_4_to_5  = (loser_4 < loser_5) ? (*allones) : (*zeros);
LLR second_comp_6_to_7  = (loser_6 < loser_7) ? (*allones) : (*zeros);
LLR second_comp_8_to_9  = (loser_8 < loser_9) ? (*allones) : (*zeros);
LLR second_comp_10_to_11  = (loser_10 < loser_11) ? (*allones) : (*zeros);
LLR second_comp_12_to_13  = (loser_12 < loser_13) ? (*allones) : (*zeros);



LLR second_winner_0_to_1  = (loser_0 & second_comp_0_to_1) | (loser_1 & (
    ↪ second_comp_0_to_1 ^ (*allones)));
LLR second_winner_2_to_3  = (loser_2 & second_comp_2_to_3) | (loser_3 & (
    ↪ second_comp_2_to_3 ^ (*allones)));
LLR second_winner_4_to_5  = (loser_4 & second_comp_4_to_5) | (loser_5 & (
    ↪ second_comp_4_to_5 ^ (*allones)));
LLR second_winner_6_to_7  = (loser_6 & second_comp_6_to_7) | (loser_7 & (
    ↪ second_comp_6_to_7 ^ (*allones)));
LLR second_winner_8_to_9  = (loser_8 & second_comp_8_to_9) | (loser_9 & (
    ↪ second_comp_8_to_9 ^ (*allones)));
LLR second_winner_10_to_11  = (loser_10 & second_comp_10_to_11) | (loser_11 & (
    ↪ second_comp_10_to_11 ^ (*allones)));
LLR second_winner_12_to_13  = (loser_12 & second_comp_12_to_13) | (loser_13 & (
    ↪ second_comp_12_to_13 ^ (*allones)));
/*---------------------------------- END ROUND 1 ----------------------------------*/




/************************************ ROUND 2 ************************************/
LLR second_comp_0_to_3  = (second_winner_0_to_1 < second_winner_2_to_3) ? (*allones) :
    ↪ (*zeros);
LLR second_comp_4_to_7  = (second_winner_4_to_5 < second_winner_6_to_7) ? (*allones) :
    ↪ (*zeros);
LLR second_comp_8_to_11 = (second_winner_8_to_9 < second_winner_10_to_11) ? (*allones) :
    ↪  (*zeros);
LLR second_comp_12_to_14  = (second_winner_12_to_13 < loser_14) ? (*allones) : (*zeros);

LLR second_winner_0_to_3 = (second_winner_0_to_1 & second_comp_0_to_3) | (
    ↪ second_winner_2_to_3 & (second_comp_0_to_3 ^ (*allones)));
LLR second_winner_4_to_7 = (second_winner_4_to_5 & second_comp_4_to_7) | (
    ↪ second_winner_6_to_7 & (second_comp_4_to_7 ^ (*allones)));
LLR second_winner_8_to_11 = (second_winner_8_to_9 & second_comp_8_to_11) | (
    ↪ second_winner_10_to_11 & (second_comp_8_to_11 ^ (*allones)));
LLR second_winner_12_to_14 = (second_winner_12_to_13 & second_comp_12_to_14) | (loser_14
    ↪  & (second_comp_12_to_14 ^ (*allones)));
/*---------------------------------- END ROUND 2 ----------------------------------*/


/************************************ ROUND 3 ************************************/
LLR second_comp_0_to_7 = (second_winner_0_to_3 < second_winner_4_to_7) ? (*allones) : (*
    ↪ zeros);
LLR second_comp_8_to_14 = (second_winner_8_to_11 < second_winner_12_to_14) ? (*allones)
    ↪ : (*zeros);

LLR second_winner_0_to_7 = (second_winner_0_to_3 & second_comp_0_to_7) | (
    ↪ second_winner_4_to_7 & (second_comp_0_to_7 ^ (*allones)));
LLR second_winner_8_to_14 = (second_winner_8_to_11 & second_comp_8_to_14) | (
    ↪ second_winner_12_to_14 & (second_comp_8_to_14 ^ (*allones)));
/*---------------------------------- END ROUND 3 ----------------------------------*/

/************************************ ROUND 4 ************************************/
LLR second_comp_0_to_14 = (second_winner_0_to_7 < second_comp_8_to_14) ? (*allones) : (*
    ↪ zeros);

LLR second_winner_0_to_14 = (second_winner_0_to_7 & second_comp_0_to_14) | (
    ↪ second_comp_8_to_14 & (second_comp_0_to_14 ^ (*allones)));
/*---------------------------------- END ROUND 4 ----------------------------------*/

cnSecondMin = second_winner_0_to_14;

// Now that we have the minimums and the parity, we should convert the first and second
    ↪ minimums to 2's complement assuming negative
LLR first_min_neg = (((cnFirstMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));

LLR second_min_neg = (((cnSecondMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));



// Now let's start the check nodes
// We only need to execute one check node for each calculation as there are N_CHECK_NODE
    ↪  number of work-items
//#pragma unroll
for(char input = 0; input < N_CHECK_INPUTS; input++) {
    // For each of these we need to determine if it was the first minimum
    // We just need a conditional that has either 1 or 0 depending if it's min or second
        ↪ min
    // And then we can make the expression LLR to_vnode = cnFirstMin*condition1 +
        ↪ cnSecondMin*inv(condition1)
    LLR min_conditional = ((cnInputs[input] & (*vector_max_llr)) == cnFirstMin) ? (*
        ↪ allones) : (*zeros);

    // the LLR conditional variable will now have a vector of either 0 or 1.
    // Now we can just use the expression

    // Now we can move on to calculating the output
    LLR sign_b = cnParity;

    // The sign bit is assigned to complete the parity compared to the other inputs
    sign_b ^= cnInputs[input] & (*vector_sign_bit);

    // Now we just send the data in 2's complement which was already calculated
    LLR sign_conditional = (sign_b == (*vector_sign_bit)) ? (*allones) : (*zeros);

    // If it's negative, give it the negative value, if not then the positive
    // Start with cnSecondMin, second_min_neg, cnFirstMin, first_min_neg
    LLR to_vnode =  ((min_conditional & sign_conditional & second_min_neg)) |
          ((min_conditional & (sign_conditional ^ (*allones)) & cnSecondMin)) |
          ((min_conditional ^ (*allones)) & sign_conditional & first_min_neg) |
          ((min_conditional ^ (*allones)) & (sign_conditional ^ (*allones)) & cnFirstMin);



    // Now we just need to find the variable node index
    short vnIndex = cntovnIndex[cnInputsNodeIndex + input] + cnMemoryOffsets[offsetIndex];

    cnOutputsGM[vnIndex] = to_vnode;

    //printf("Iteration,%d,CNode,%d,output,%d,%d,vnIndex,%d\n",iter,id,input,to_vnode,
        ↪ vnIndex);

  }



  offsetIndex ^= 1;






}
```

Appendix T

Single Work-item (Design D) Irregular Decoder for the GPU (OpenCL)

```
#ifndef N_VARIABLE_NODES
#define N_VARIABLE_NODES 1120
#endif

#ifndef N_CHECK_NODES
#define N_CHECK_NODES 280
#endif

#ifndef SIMD
#define SIMD 1
#endif

// Based on the Tanner graph, how many connections are there
#define MAX_VARIABLE_EDGES 5
#define MAX_CHECK_EDGES 16


// Let's define the usable number of inputs
#define N_VARIABLE_INPUTS (MAX_VARIABLE_EDGES)
#define N_CHECK_INPUTS (MAX_CHECK_EDGES)

#define N_VARIABLE_INPUTS_W_CHANNEL (N_VARIABLE_INPUTS + 1)

#define N_ITERATIONS (2)



// Set the vector width and the appropriate LLR types
#define MANUAL_VECTOR_WIDTH (16)

#if MANUAL_VECTOR_WIDTH==2
#define LLR_TYPE char2
#define TOTAL_TYPE short2
#define CONVERT_TO_SHORT(a) convert_short2(a)
#define CONVERT_TO_CHAR(a) convert_char2(a)
#elif MANUAL_VECTOR_WIDTH==4
#define LLR_TYPE char4
#define TOTAL_TYPE short4
#define CONVERT_TO_SHORT(a) convert_short4(a)
#define CONVERT_TO_CHAR(a) convert_char4(a)
#elif MANUAL_VECTOR_WIDTH==8
#define LLR_TYPE char8
#define TOTAL_TYPE short8
#define CONVERT_TO_SHORT(a) convert_short8(a)
#define CONVERT_TO_CHAR(a) convert_char8(a)
#elif MANUAL_VECTOR_WIDTH==16
#define LLR_TYPE char16
#define TOTAL_TYPE short16
#define CONVERT_TO_SHORT(a) convert_short16(a)
#define CONVERT_TO_CHAR(a) convert_char16(a)
#else
#define LLR_TYPE char
#define TOTAL_TYPE short
#define CONVERT_TO_SHORT(a) convert_short(a)
#define CONVERT_TO_CHAR(a) convert_char(a)
#endif
```

```
typedef LLR_TYPE LLR;


// Now for the channels
//#pragma OPENCL EXTENSION cl_altera_channels : enable
//channel char vnStart __attribute__((depth(N_ITERATIONS)));
//channel char cnStart __attribute__((depth(N_ITERATIONS)));
//#define SENDDATA (1)

//Variable Node Kernel
__kernel
//__attribute__((reqd_work_group_size(N_VARIABLE_NODES,1,1)))
//__attribute__((num_compute_units(1)))
//__attribute__((num_simd_work_items(SIMD)))
void VariableNode(

// Kernel parameteres
// As in the notes page, there is a table that determines the write location in the CNs
__constant int * restrict vntocnIndex,

// Now that we have the connection information let's get the inputs
// Remember there is one memory location for the Variable Node inputs with two slots for
    ↪ pipelining
// Let's start with the channel measurements
__global LLR * restrict channelMeasurements,

// Now let's get the inputs
__global LLR * restrict vnInputsGM,

// Now let's get the output pointer to the CNs
__global LLR * restrict vnOutputsGM,

// And of course finally, we need to get the result
__global TOTAL_TYPE * restrict vnResults,


// Let's add the synchronization global memory location
__global char * restrict VNSync,
__global char * restrict CNSync,


// We need a few constants where it's easier to compute them in the host
__constant LLR * restrict vector_max_llr,
__constant LLR * restrict vector_sign_bit,
__constant LLR * restrict allones,
__constant LLR * restrict ones,
__constant LLR * restrict zeros,
__constant TOTAL_TYPE * restrict total_vector_max_llr,
__constant TOTAL_TYPE * restrict total_vector_min_neg_llr
)
{

  // First we need to get our global IDsa
  int id = get_global_id(0);
  int lid = get_local_id(0);

  //printf("VN,%d,started\n",id);




  // We also need a short version of the LLR values to total
  TOTAL_TYPE vnInputsShort[N_VARIABLE_INPUTS_W_CHANNEL];

  // Let's make an array for the offsets
  int vnMemoryOffsets[2];
  vnMemoryOffsets[0] = 0;
  vnMemoryOffsets[1] = N_VARIABLE_NODES * (N_VARIABLE_INPUTS+1);

  int vnChannelOffsets[2];
  vnChannelOffsets[0] = 0;
  vnChannelOffsets[1] = N_VARIABLE_NODES;

  int cnMemoryOffsets[2];
  cnMemoryOffsets[0] = 0;
  cnMemoryOffsets[1] = N_CHECK_NODES * (N_CHECK_INPUTS+1);

  // Let's make an array index lookup for the input
  int vntocnNodeIndex = id * (N_VARIABLE_INPUTS);
  int vnGlobalNodeIndex = id * (N_VARIABLE_INPUTS+ 1);

  // Some other variables
  LLR vnCurrent;
  TOTAL_TYPE vnTempCurrent;
  LLR vnConditional;
  LLR vnSignB;
  LLR vnMag;
  char temp;

  // Now let's store the pointer
  char offsetIndex = 0;
  //printf("VN,%d,prepped\n",id);

  // Let's start the process
  // We need to step through the iterations (aka start the loop)
  // Since both VN and CN will be running concurrently on the same data, we will need
       ↪ N_ITERATIONS*2+1
  for(char iter = 0; iter < ((N_ITERATIONS * 2) + 2); iter++) {

    // Let's just do some printing
    printf("VN,%d,%d\n",id,iter);
    printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
        ↪ ␣␣␣␣␣␣␣\n");
    // Alright so each iteration we need to do the following:
    // Each VN:
    // 1) Wait for the OK from the CNs to start on the data
    // 2) Load up the inputs from DRAM into private memory
    // 3) Do the VN
    // 4) Output the data to the appropriate buffer
    // 5) Send the ok to the CN to start execution
    // 6) Swap buffers
    // Repeat

    if( iter > 0)
    {
      // At the second iteration or higher we will start to worry about this.
      // We need to check the VNsync variable to see if we can operate yet
      char tempValue = -1;
      // We wait for the VNSync variable to be at least or greater than our iteration
           ↪ number
      while(tempValue < iter) {
        tempValue = (*VNSync);
        //mem_fence(CLK_GLOBAL_MEM_FENCE);
      }
    }

    // Let's wait for the CNs to tell us it's ok to start execution
    // Note that we can throw away the data as this is a blocking call
    //if(id == 0 && iter > 1)
    //  temp = read_channel_altera(vnStart);

    barrier(0);


    //printf("VN,%d,%d,afterbarrier\n",id,iter);

    // Now that we've got the OK. Let's start execution
    // First let's load up the inputs and convert them to shorts so that we can get the
         ↪ appropriate totals without saturation
    #pragma unroll
    for(char input = 0; input < N_VARIABLE_INPUTS; input++) {
      vnInputsShort[input] = CONVERT_TO_SHORT(vnInputsGM[vnMemoryOffsets[offsetIndex] +
           ↪ vnGlobalNodeIndex + input]);
    }

    // Now let's put in the channel measurement for our node
    vnInputsShort[N_VARIABLE_INPUTS] = CONVERT_TO_SHORT(channelMeasurements[
         ↪ vnChannelOffsets[offsetIndex] + id]);

    // Now we need to add it all together

    TOTAL_TYPE total = (TOTAL_TYPE)(0);
    #pragma unroll
    for(char input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {
      total += vnInputsShort[input];
    }


    /*for(int i=0; i<N_VARIABLE_INPUTS_W_CHANNEL; i++) {
      printf("Iteration,%d,VNode,%d,GMInputsIndex,%d,GMInputs,%d,input,%d,total,%d,value,%
           ↪ d,address_index,%d\n",iter,id,vnMemoryOffsets[offsetIndex] +
           ↪ vnGlobalNodeIndex + i,vnInputsGM[vnMemoryOffsets[offsetIndex] +
           ↪ vnGlobalNodeIndex + i].s0,i,total.s0,vnInputsShort[i].s0,vnMemoryOffsets[
           ↪ offsetIndex] + vnGlobalNodeIndex + i);
    }*/

    // Now that we have the total we can do the rest
    for(char input = 0; input < N_VARIABLE_INPUTS; input++) {

      // Subtract the current from the total to get the output
      // We simply subtract the large total and then saturate it down to the appropriate
           ↪ size
      vnTempCurrent = total - vnInputsShort[input];

      // Now saturate
      vnTempCurrent =   ((vnTempCurrent >= (*total_vector_max_llr)) ?
                (*total_vector_max_llr) :
                ((vnTempCurrent <= (*total_vector_min_neg_llr)) ?
                (*total_vector_min_neg_llr) :
                vnTempCurrent));
      // Now we reverse the procedure and store the data back into the LLR type
      vnCurrent = CONVERT_TO_CHAR(vnTempCurrent);
```

```
      // Ok now we can finally send the data
      // First we check if it's negative
      // Except we need to convert it to sign and magnitude
      vnConditional = ((vnCurrent & (*vector_sign_bit)) == (*vector_sign_bit)) ? (*allones
          ↪ ) : (*zeros);

      // If it's negative we need to xor it and add one and mask out the sign bit
      vnMag = ((vnConditional & ((vnCurrent ^ (*allones)) + (*ones))) | ((vnConditional ^
          ↪ (*allones)) & vnCurrent)) & (*vector_max_llr);

      // Now get the sign bit
      vnSignB = (vnConditional & (*vector_sign_bit));

      // Now we just need to find the location in the array to place the data
      int cnIndex = vntocnIndex[vntocnNodeIndex + input] + cnMemoryOffsets[offsetIndex];

      // Now we can just output the value
      // Since we're writing to the CNs we need to output to the write memory location
      vnOutputsGM[cnIndex] = vnMag | vnSignB;

      //printf("Iteration,%d,VNode,%d,output,%d,sign_bit,%d,mag,%d,cnIndex,%d,vntocnIndex
          ↪ ,%d,offset,%d\n",iter,id,input,vnSignB.s0,vnMag.s0,cnIndex,vntocnIndex[
          ↪ vntocnNodeIndex + input],cnMemoryOffsets[offsetIndex]);

    }

    if((iter+1)==N_ITERATIONS*2+1) {
      TOTAL_TYPE send_total = 0;
      for(short input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {
        send_total += vnInputsShort[input];
      }

      vnResults[vnMemoryOffsets[offsetIndex] + id] = send_total;
    }

    barrier(CLK_GLOBAL_MEM_FENCE);



    // Now let's send the iteration value to the CN so it knows
    if(id ==0)
      (*CNSync) = iter;


    //mem_fence(CLK_GLOBAL_MEM_FENCE);


    //printf("VN,%d,%d,beforechannelwrite\n",id,iter);
    // Finally, we need to tell the check node it can operate on the data
    //if(id == 0)
    //   write_channel_altera(cnStart, SENDDATA);

    //barrier(CLK_GLOBAL_MEM_FENCE | CLK_CHANNEL_MEM_FENCE);

    // Now let's switch the index over
    // To do this we can simply XOR it with the value of 1
    offsetIndex ^= 1;




  }

  TOTAL_TYPE send_total = 0;
  for(short input = 0; input < N_VARIABLE_INPUTS_W_CHANNEL; input++) {
    send_total += vnInputsShort[input];
  }

  vnResults[vnMemoryOffsets[offsetIndex] + id] = send_total;


}


__kernel
// __attribute__((reqd_work_group_size(N_CHECK_NODES,1,1)))
// __attribute__((num_compute_units(1)))
// __attribute__((num_simd_work_items(SIMD)))
void CheckNode(

// Let's start with the table for the interleaver connections just like with the VN
__constant int * restrict cntovnIndex,

// Now we can do the other constants
__constant LLR * restrict vector_max_llr,
__constant LLR * restrict vector_sign_bit,
__constant LLR * restrict allones,
__constant LLR * restrict ones,
__constant LLR * restrict zeros,

// Let's add the synchronization global memory location
__global char * restrict VNSync,
__global char * restrict CNSync,

// Now we can focus on the inputs
__global LLR * restrict cnInputsGM,

// Now the outputs
__global LLR * restrict cnOutputsGM,

// Now a place for the results
__global LLR * restrict cnResults

) {



  int id = get_global_id(0);
  printf("CN,%d,started\n",id);

  // First we need to store a place in private memory for operation
  LLR cnInputs[N_CHECK_INPUTS];

  // And also a spot for the magnitudes
  LLR cnInputsMag[N_CHECK_INPUTS];

  // Let's compute the base index
  int cnInputsNodeIndex = id * N_CHECK_INPUTS;
  int cnInputsGlobalIndex = id * (N_CHECK_INPUTS+1);

  // Check Node Private Variables
  LLR cnFirstMin;
  LLR cnSecondMin;
  LLR cnParity;
  char temp;

  // Ok. Now we need to do some similar work to the VNs.
  // Let's make an array for the offsets
  int vnMemoryOffsets[2];
  vnMemoryOffsets[0] = 0;
  vnMemoryOffsets[1] = N_VARIABLE_NODES * (N_VARIABLE_INPUTS+1);

  int cnMemoryOffsets[2];
  cnMemoryOffsets[0] = 0;
  cnMemoryOffsets[1] = N_CHECK_NODES * (N_CHECK_INPUTS+1);

  // And let's store an offset
  char offsetIndex = 0; // Note that we start on the same data as VN but we have to wait
      ↪ for the ramp up of the VN to complete the first iterations before we can execute

  //printf("CN,%d,prepped\n",id);

  // Alright let's start the iterations
  for(char iter = 0; iter < (N_ITERATIONS * 2); iter++) {

    printf("CN,%d,%d\n",id,iter);
    // For each iteration we need to:
    // 1) Wait for the ok from the VNs to start executing on the information
    // 2) Load in the new inputs from GM
    // 3) Do the Cn
    // 4) Send the data to the VN via GM
    // 5) Tell the VN we have finished
    //barrier(0);

    //if(id == 0)

      // We need to check the CNsync variable to see if we can operate yet
      char tempValue = -1;
      // We wait for the CNSync variable to be at least or greater than our iteration
          ↪ number
      while(tempValue < iter) {
        tempValue = (*CNSync);
        //mem_fence(CLK_GLOBAL_MEM_FENCE);
      }

    // So let's wait for the Ok to start
    //if(id == 0)
    //  temp = read_channel_altera(cnStart);

    barrier(0);

    // Now we can begin our execution
    cnFirstMin = (*vector_max_llr);
    cnSecondMin = (*vector_max_llr);

    // First let's do the minimum calculations

    // But first, we need to make sure that we are taking the absolute value as we don't
        ↪ want to include that in the min calculation
    // Therefore, we have to mask off the sign bits
    #pragma unroll
    for(short input = 0; input < N_CHECK_INPUTS; input++) {
      cnInputs[input] = cnInputsGM[cnMemoryOffsets[offsetIndex] + cnInputsGlobalIndex +
          ↪ input];
```

```
}

// Now that we have the normal inputs, we need to get the magnitudes
#pragma unroll
for(short input = 0; input < N_CHECK_INPUTS; input++) {
  cnInputsMag[input] = cnInputs[input] & (*vector_max_llr);
}

// Ok now that we have the magnitudes, let's get to work by getting the overall parity
    ↪  first
cnParity = (*zeros);
#pragma unroll
for(short input = 0; input < N_CHECK_INPUTS; input++) {
  cnParity ^= cnInputs[input] & (*vector_sign_bit);
}

// Now cnParity holds the overall parity of the check node

// We're going to try something a little different here
// We're going to implement the tree using min on the magnitudes

// This is a little trickier as we need to find the losers a lot
// But let's start it out.


/*********************************** ROUND 1 ***********************************/
LLR comp_0_1  = (cnInputsMag[0] < cnInputsMag[1]) ? (*allones) : (*zeros);
LLR comp_2_3  = (cnInputsMag[2] < cnInputsMag[3]) ? (*allones) : (*zeros);
LLR comp_4_5  = (cnInputsMag[4] < cnInputsMag[5]) ? (*allones) : (*zeros);
LLR comp_6_7  = (cnInputsMag[6] < cnInputsMag[7]) ? (*allones) : (*zeros);
LLR comp_8_9  = (cnInputsMag[8] < cnInputsMag[9]) ? (*allones) : (*zeros);
LLR comp_10_11  = (cnInputsMag[10] < cnInputsMag[11]) ? (*allones) : (*zeros);
LLR comp_12_13  = (cnInputsMag[12] < cnInputsMag[13]) ? (*allones) : (*zeros);
LLR comp_14_15  = (cnInputsMag[14] < cnInputsMag[15]) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_1 = (cnInputsMag[0] & comp_0_1) | (cnInputsMag[1] & (comp_0_1 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_0 = (cnInputsMag[1] & comp_0_1) | (cnInputsMag[0] & (comp_0_1 ^ (*allones)))
    ↪ ;


// First Minimum
LLR winner_2_to_3 = (cnInputsMag[2] & comp_2_3) | (cnInputsMag[3] & (comp_2_3 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_1 = (cnInputsMag[3] & comp_2_3) | (cnInputsMag[2] & (comp_2_3 ^ (*allones)))
    ↪ ;


// First Minimum
LLR winner_4_to_5 = (cnInputsMag[4] & comp_4_5) | (cnInputsMag[5] & (comp_4_5 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_2 = (cnInputsMag[5] & comp_4_5) | (cnInputsMag[4] & (comp_4_5 ^ (*allones)))
    ↪ ;


// First Minimum
LLR winner_6_to_7 = (cnInputsMag[6] & comp_6_7) | (cnInputsMag[7] & (comp_6_7 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_3 = (cnInputsMag[7] & comp_6_7) | (cnInputsMag[6] & (comp_6_7 ^ (*allones)))
    ↪ ;


// First Minimum
LLR winner_8_to_9 = (cnInputsMag[8] & comp_8_9) | (cnInputsMag[9] & (comp_8_9 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_4 = (cnInputsMag[9] & comp_8_9) | (cnInputsMag[8] & (comp_8_9 ^ (*allones)))
    ↪ ;


// First Minimum
LLR winner_10_to_11 = (cnInputsMag[10] & comp_10_11) | (cnInputsMag[11] & (comp_10_11
    ↪ ^ (*allones)));
// Second Minimum
LLR loser_5 = (cnInputsMag[11] & comp_10_11) | (cnInputsMag[10] & (comp_10_11 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_12_to_13 = (cnInputsMag[12] & comp_12_13) | (cnInputsMag[13] & (comp_12_13
    ↪ ^ (*allones)));
// Second Minimum
LLR loser_6 = (cnInputsMag[13] & comp_12_13) | (cnInputsMag[12] & (comp_12_13 ^ (*
    ↪ allones)));
```

```
// First Minimum
LLR winner_14_to_15 = (cnInputsMag[14] & comp_14_15) | (cnInputsMag[15] & (comp_14_15
    ↪ ^ (*allones)));
// Second Minimum
LLR loser_7 = (cnInputsMag[15] & comp_14_15) | (cnInputsMag[14] & (comp_14_15 ^ (*
    ↪ allones)));


/*---------------------------------- END ROUND 1 ----------------------------------
    ↪ */




/************************************ ROUND 2 ************************************/
LLR comp_0_to_3 = (winner_0_to_1 < winner_2_to_3) ? (*allones) : (*zeros);
LLR comp_4_to_7 = (winner_4_to_5 < winner_6_to_7) ? (*allones) : (*zeros);
LLR comp_8_to_11  = (winner_8_to_9 < winner_10_to_11) ? (*allones) : (*zeros);
LLR comp_12_to_15 = (winner_12_to_13 < winner_14_to_15) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_3 = (winner_0_to_1 & comp_0_to_3) | (winner_2_to_3 & (comp_0_to_3 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_8 = (winner_2_to_3 & comp_0_to_3) | (winner_0_to_1 & (comp_0_to_3 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_4_to_7 = (winner_4_to_5 & comp_4_to_7) | (winner_6_to_7 & (comp_4_to_7 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_9 = (winner_6_to_7 & comp_4_to_7) | (winner_4_to_5 & (comp_4_to_7 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_8_to_11  = (winner_8_to_9 & comp_8_to_11) | (winner_10_to_11 & (
    ↪ comp_8_to_11 ^ (*allones)));
// Second Minimum
LLR loser_10  = (winner_10_to_11 & comp_8_to_11) | (winner_8_to_9 & (comp_8_to_11 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_12_to_15 = (winner_12_to_13 & comp_12_to_15) | (winner_14_to_15 & (
    ↪ comp_12_to_15 ^ (*allones)));
// Second Minimum
LLR loser_11  = (winner_14_to_15 & comp_12_to_15) | (winner_12_to_13 & (comp_12_to_15
    ↪ ^ (*allones)));


/*---------------------------------- END ROUND 2 ----------------------------------
    ↪ */




/************************************ ROUND 3 ************************************/
LLR comp_0_to_7 = (winner_0_to_3 < winner_4_to_7) ? (*allones) : (*zeros);
LLR comp_8_to_15  = (winner_8_to_11 < winner_12_to_15) ? (*allones) : (*zeros);



// First Minimum
LLR winner_0_to_7 = (winner_0_to_3 & comp_0_to_7) | (winner_4_to_7 & (comp_0_to_7 ^ (*
    ↪ allones)));
// Second Minimum
LLR loser_12  = (winner_4_to_7 & comp_0_to_7) | (winner_0_to_3 & (comp_0_to_7 ^ (*
    ↪ allones)));


// First Minimum
LLR winner_8_to_15  = (winner_8_to_11 & comp_8_to_15) | (winner_12_to_15 & (
    ↪ comp_8_to_15 ^ (*allones)));
// Second Minimum
LLR loser_13  = (winner_12_to_15 & comp_8_to_15) | (winner_8_to_11 & (comp_8_to_15 ^
    ↪ (*allones)));


/*---------------------------------- END ROUND 3 ----------------------------------
    ↪ */




/************************************ ROUND 4 ************************************/
LLR comp_0_to_15  = (winner_0_to_7 < winner_8_to_15) ? (*allones) : (*zeros);
```

```
// First Minimum
LLR winner_0_to_15  = (winner_0_to_7 & comp_0_to_15) | (winner_8_to_15 & (comp_0_to_15
    ↪  ^ (*allones)));
// Second Minimum
LLR loser_14  = (winner_8_to_15 & comp_0_to_15) | (winner_0_to_7 & (comp_0_to_15 ^ (*
    ↪ allones)));


/*----------------------------------- END ROUND 4 -----------------------------------
    ↪ */




cnFirstMin = winner_0_to_15;

/************************************ ROUND 1 ************************************/
LLR second_comp_0_to_1  = (loser_0 < loser_1) ? (*allones) : (*zeros);
LLR second_comp_2_to_3  = (loser_2 < loser_3) ? (*allones) : (*zeros);
LLR second_comp_4_to_5  = (loser_4 < loser_5) ? (*allones) : (*zeros);
LLR second_comp_6_to_7  = (loser_6 < loser_7) ? (*allones) : (*zeros);
LLR second_comp_8_to_9  = (loser_8 < loser_9) ? (*allones) : (*zeros);
LLR second_comp_10_to_11  = (loser_10 < loser_11) ? (*allones) : (*zeros);
LLR second_comp_12_to_13  = (loser_12 < loser_13) ? (*allones) : (*zeros);



LLR second_winner_0_to_1  = (loser_0 & second_comp_0_to_1) | (loser_1 & (
    ↪ second_comp_0_to_1 ^ (*allones)));
LLR second_winner_2_to_3  = (loser_2 & second_comp_2_to_3) | (loser_3 & (
    ↪ second_comp_2_to_3 ^ (*allones)));
LLR second_winner_4_to_5  = (loser_4 & second_comp_4_to_5) | (loser_5 & (
    ↪ second_comp_4_to_5 ^ (*allones)));
LLR second_winner_6_to_7  = (loser_6 & second_comp_6_to_7) | (loser_7 & (
    ↪ second_comp_6_to_7 ^ (*allones)));
LLR second_winner_8_to_9  = (loser_8 & second_comp_8_to_9) | (loser_9 & (
    ↪ second_comp_8_to_9 ^ (*allones)));
LLR second_winner_10_to_11  = (loser_10 & second_comp_10_to_11) | (loser_11 & (
    ↪ second_comp_10_to_11 ^ (*allones)));
LLR second_winner_12_to_13  = (loser_12 & second_comp_12_to_13) | (loser_13 & (
    ↪ second_comp_12_to_13 ^ (*allones)));
/*----------------------------------- END ROUND 1 -----------------------------------
    ↪ */




/************************************ ROUND 2 ************************************/
LLR second_comp_0_to_3  = (second_winner_0_to_1 < second_winner_2_to_3) ? (*allones) :
    ↪  (*zeros);
LLR second_comp_4_to_7  = (second_winner_4_to_5 < second_winner_6_to_7) ? (*allones) :
    ↪  (*zeros);
LLR second_comp_8_to_11 = (second_winner_8_to_9 < second_winner_10_to_11) ? (*allones)
    ↪  : (*zeros);
LLR second_comp_12_to_14  = (second_winner_12_to_13 < loser_14) ? (*allones) : (*zeros
    ↪ );

LLR second_winner_0_to_3 = (second_winner_0_to_1 & second_comp_0_to_3) | (
    ↪ second_winner_2_to_3 & (second_comp_0_to_3 ^ (*allones)));
LLR second_winner_4_to_7 = (second_winner_4_to_5 & second_comp_4_to_7) | (
    ↪ second_winner_6_to_7 & (second_comp_4_to_7 ^ (*allones)));
LLR second_winner_8_to_11 = (second_winner_8_to_9 & second_comp_8_to_11) | (
    ↪ second_winner_10_to_11 & (second_comp_8_to_11 ^ (*allones)));
LLR second_winner_12_to_14 = (second_winner_12_to_13 & second_comp_12_to_14) | (
    ↪ loser_14 & (second_comp_12_to_14 ^ (*allones)));
/*----------------------------------- END ROUND 2 -----------------------------------
    ↪ */


/************************************ ROUND 3 ************************************/
LLR second_comp_0_to_7 = (second_winner_0_to_3 < second_winner_4_to_7) ? (*allones) :
    ↪ (*zeros);
LLR second_comp_8_to_14 = (second_winner_8_to_11 < second_winner_12_to_14) ? (*allones
    ↪ ) : (*zeros);

LLR second_winner_0_to_7 = (second_winner_0_to_3 & second_comp_0_to_7) | (
    ↪ second_winner_4_to_7 & (second_comp_0_to_7 ^ (*allones)));
LLR second_winner_8_to_14 = (second_winner_8_to_11 & second_comp_8_to_14) | (
    ↪ second_winner_12_to_14 & (second_comp_8_to_14 ^ (*allones)));
/*----------------------------------- END ROUND 3 -----------------------------------
    ↪ */

/************************************ ROUND 4 ************************************/
LLR second_comp_0_to_14 = (second_winner_0_to_7 < second_winner_8_to_14) ? (*allones)
    ↪ : (*zeros);

LLR second_winner_0_to_14 = (second_winner_0_to_7 & second_comp_0_to_14) | (
    ↪ second_winner_8_to_14 & (second_comp_0_to_14 ^ (*allones)));
/*----------------------------------- END ROUND 4 -----------------------------------
    ↪ */
    cnSecondMin = second_winner_0_to_14;

    // Now that we have the minimums and the parity, we should convert the first and
        ↪ second minimums to 2's complement assuming negative
    LLR first_min_neg = (((cnFirstMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));

    LLR second_min_neg = (((cnSecondMin ^ (*allones)) | (*vector_sign_bit)) + (*ones));

    /*for(int i=0; i<N_CHECK_INPUTS+1; i++) {
      printf("Iteration,%d,CNode,%d,input,%d,sign_bit,%d,value,%d,min,%d,second_min,%d,
          ↪ address,%d\n",iter,id,i,cnInputs[i].s0 & (*vector_sign_bit).s0,cnInputsMag[i
          ↪ ].s0,cnFirstMin.s0,cnSecondMin.s0,cnMemoryOffsets[offsetIndex] +
          ↪ cnInputsGlobalIndex + i);
    }*/




    // Now let's start the check nodes
    // We only need to execute one check node for each calculation as there are
        ↪ N_CHECK_NODE number of work-items
    #pragma unroll
    for(char input = 0; input < N_CHECK_INPUTS; input++) {



      // For each of these we need to determine if it was the first minimum
      // We just need a conditional that has either 1 or 0 depending if it's min or second
          ↪  min
      // And then we can make the expression LLR to_vnode = cnFirstMin*condition1 +
          ↪ cnSecondMin*inv(condition1)
      LLR min_conditional = ((cnInputs[input] & (*vector_max_llr)) == cnFirstMin) ? (*
          ↪ allones) : (*zeros);

      // the LLR conditional variable will now have a vector of either 0 or 1.
      // Now we can just use the expression

      // Now we can move on to calculating the output
      LLR sign_b = cnParity;

      // The sign bit is assigned to complete the parity compared to the other inputs
      sign_b ^= cnInputs[input] & (*vector_sign_bit);

      // Now we just send the data in 2's complement which was already calculated
      LLR sign_conditional = (sign_b == (*vector_sign_bit)) ? (*allones) : (*zeros);

      // If it's negative, give it the negative value, if not then the positive
      // Start with cnSecondMin, second_min_neg, cnFirstMin, first_min_neg
      LLR to_vnode =  ((min_conditional & sign_conditional & second_min_neg)) |
            ((min_conditional & (sign_conditional ^ (*allones)) & cnSecondMin)) |
            ((min_conditional ^ (*allones)) & sign_conditional & first_min_neg) |
            ((min_conditional ^ (*allones)) & (sign_conditional ^ (*allones)) & cnFirstMin
                ↪ );



      // Now we just need to find the variable node index
      int vnIndex = cntovnIndex[cnInputsNodeIndex + input] + vnMemoryOffsets[offsetIndex];

      cnOutputsGM[vnIndex] = to_vnode;

      //printf("Iteration,%d,CNode,%d,output,%d,value,%d,vnIndex,%d\n",iter,id,input,
          ↪ to_vnode.s0,vnIndex);

    }
    barrier(CLK_GLOBAL_MEM_FENCE);
    //if(id == 0)

      // Let's send info to the VN to start execution
      // Except the VNs can be up to two iterations ahead
      (*VNSync) = (iter + 2);

    //mem_fence(CLK_GLOBAL_MEM_FENCE);

    //mem_fence(CLK_GLOBAL_MEM_FENCE | CLK_CHANNEL_MEM_FENCE);
    offsetIndex ^= 1;
    //if(id == 0)
    //  write_channel_altera(vnStart,SENDDATA);

    //barrier(CLK_GLOBAL_MEM_FENCE | CLK_CHANNEL_MEM_FENCE);


  }

}
```

Appendix U

Minimum Tournament Generator Source Code (C#)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;

namespace MinTournamentGenerator
{
    class Program
    {

        string allones = "(*allones)";
        string zeros = "(*zeros)";


        static void Main(string[] args)
        {
            Console.WriteLine("Please␣enter␣in␣the␣number␣of␣inputs␣for␣the␣tournament");
            int n_inputs = Convert.ToInt32(Console.ReadLine());
            while (n_inputs > 250 || n_inputs < 0)
            {
                // This is the error case
                Console.WriteLine("Sorry.␣It␣must␣be␣a␣number␣between␣0␣and␣250.␣Please␣
                    ↪ try␣again");
                n_inputs = Convert.ToInt16(Console.ReadLine());
            }
            int i;
            int num_rounds = Convert.ToInt32(Math.Log(n_inputs, 2));
            // Ok now that we have a valid number. Let's start creating the code

            // Wait we need a location to store the data first
            Console.WriteLine("Please␣enter␣a␣filename␣for␣the␣code␣to␣be␣written");
            Console.WriteLine("Enter␣nothing␣for␣the␣default␣output.txt");
            string filename = Console.ReadLine();

            if (filename == "")
                filename = "output.txt";


            // Let's open the output file
            StreamWriter outFile = new StreamWriter(filename);

            int loserIndex = 0;

            // Ok nowww we can start

            // First we start with the first round of the first minimum. This compares all
                ↪ adjacent entries to each other
            // Let's iterate over half the inputs
            outFile.WriteLine("/**************************************␣ROUND␣1␣
                ↪ *************************************/");
            for (i = 0; i < (n_inputs-1); i+=2)
            {
                // Ok. So in each of these we compare the neighbors and add two
                // We need to write LLR comp_i_j  = (cnInputsMag[i] < cnInputsMag[j]) ? (*
                    ↪ allones) : (*zeros); where j = i+1
                // Let's start with the beginning
                outFile.Write("LLR␣comp_" + i + "_to_" + (i + 1) + "\t=\t(cnInputsMag[" +
                    ↪ i + "]␣<␣cnInputsMag[" + (i + 1) + "])␣?␣(*allones)␣:␣(*zeros);\n"
                    ↪ );

            }

            outFile.WriteLine("\n\n");

            // Next we get the winners
            for(i=0; i < (n_inputs-1); i+=2)
            {
                outFile.WriteLine("//␣First␣Minimum");
                outFile.Write("LLR␣winner_" + i + "_to_" + (i + 1) + "\t=\t(cnInputsMag["
                    ↪ + i + "]␣&␣comp_" + i + "_to_" + (i+1) + ")␣|␣(cnInputsMag[" + (i
                    ↪ +1) + "]␣&␣(comp_" + i + "_to_" + (i+1) + "␣^␣(*allones)));\n");
                outFile.WriteLine("//␣Second␣Minimum");
                outFile.Write("LLR␣loser_" + (loserIndex++) + "\t=\t(cnInputsMag[" + (i +
                    ↪ 1) + "]␣&␣comp_" + i + "_to_" + (i + 1) + ")␣|␣(cnInputsMag[" + (i
                    ↪ ) + "]␣&␣(comp_" + i + "_to_" + (i + 1) + "␣^␣(*allones)));\n");
                outFile.WriteLine("\n");
            }
            outFile.WriteLine("/*-----------------------------------␣END␣ROUND␣1␣
                ↪ -----------------------------------*/");
            outFile.WriteLine("\n\n\n");

            // Alright so this next part is basically automated now.
            // Essentially we need to take the winners from above and advance them


            // Next round!
            outFile.WriteLine("/*************************************␣ROUND␣2␣
                ↪ *************************************/");
            // Okie so for round two what we do is take the winners

            // If we had an odd number from the last round, we need to add it to this
                ↪ round at the end


            int iter_jump = 4;


            for (i = 0; i < (n_inputs-iter_jump-1); i += iter_jump)
            {
                // Alright let's get this started
                // So basically we take the winners from the previous round, test them and
                    ↪  compare

                outFile.Write("LLR␣comp_" + i + "_to_" + (i + iter_jump - 1) + "\t=\t(
                    ↪ winner_" + i + "_to_" + (i + 1) + "␣<␣winner_" + (i + 2) + "_to_"
                    ↪ + (i + 3) + ")␣?␣(*allones)␣:␣(*zeros);\n");
            }
            if (n_inputs % 2 != 0)
            {
                // Add the last one to the list of them but how?
                outFile.Write("LLR␣comp_" + i + "_to_" + (n_inputs - 1) + "\t=\t(winner_"
                    ↪ + i + "_to_" + (i + 1) + "␣<␣cnInputsMag[" + (n_inputs - 1) +  "])
                    ↪ ␣?␣(*allones)␣:␣(*zeros);\n");

            }

            outFile.WriteLine("\n\n");


            int num_iterations;
            if (n_inputs % 2 != 0)
            {
                num_iterations = (n_inputs - 1) / 2;
            }
            else
                num_iterations = n_inputs / 2;

            // Now we write the winners
            for (i = 0; i < (n_inputs-iter_jump-1); i += iter_jump)
            {
                outFile.WriteLine("//␣First␣Minimum");
                outFile.Write("LLR␣winner_" + i + "_to_" + (i + iter_jump - 1) + "\t=\t(
                    ↪ winner_" + i + "_to_" + (i + 1) + "␣&␣comp_" + i + "_to_" + (i +
                    ↪ iter_jump - 1) + ")␣|␣(winner_" + (i + 2) + "_to_" + (i + 3) + "␣&
                    ↪ ␣(comp_" + i + "_to_" + (i +iter_jump - 1) + "␣^␣(*allones)));\n")
                    ↪ ;
                outFile.WriteLine("//␣Second␣Minimum");
                outFile.Write("LLR␣loser_" + (loserIndex++) + "\t=\t(winner_" + (i + 2) +
                    ↪ "_to_" + (i + 3) + "␣&␣comp_" + i + "_to_" + (i + iter_jump - 1) +
                    ↪  ")␣|␣(winner_" + (i) + "_to_" + (i + 1) + "␣&␣(comp_" + i + "_to_
                    ↪ " + (i + iter_jump - 1) + "␣^␣(*allones)));\n");
                outFile.WriteLine("\n");
            }
            if (n_inputs % 2 != 0)
            {
                outFile.WriteLine("//␣First␣Minimum");
                outFile.Write("LLR␣winner_" + i + "_to_" + (n_inputs - 1) + "\t=\t(winner_
                    ↪ " + i + "_to_" + (i + 1) + "␣&␣comp_" + i + "_to_" + (n_inputs -
                    ↪ 1) + ")␣|␣(cnInputsMag[" + (n_inputs - 1) + "]␣&␣(comp_" + i + "
                    ↪ _to_" + (n_inputs - 1) + "␣^␣(*allones));\n");
                outFile.WriteLine("//␣Second␣Minimum");
                outFile.Write("LLR␣loser_" + (loserIndex++) + "\t=\t(cnInputsMag[" + (
                    ↪ n_inputs − 1) + "]␣&␣comp_" + i + "_to_" + (n_inputs − 1) + ")␣|␣(
                    ↪ winner_" + (i) + "_to_" + (i + 1) + "␣&␣(comp_" + i + "_to_" + (
                    ↪ n_inputs − 1) + "␣^␣(*allones)));\n");
                outFile.WriteLine("\n");
            }
            outFile.WriteLine("/*-----------------------------------␣END␣ROUND␣2␣
                ↪ -----------------------------------*/");
            outFile.WriteLine("\n\n\n");




            int prev_iter_jump = iter_jump;
            int carry = 0;
            int iseven = 0;
            int previseven = 0;

            for (int round = 3; round <= num_rounds; round++)
            {
                outFile.WriteLine("/*************************************␣ROUND␣" + round
                    ↪ + "␣*************************************/");
                previseven = iseven;
                iseven = 0;
                prev_iter_jump = iter_jump;
                iter_jump = iter_jump * 2;
                for (i = 0; i < (n_inputs − iter_jump − 1); i += iter_jump)
                {
                    // Alright let's get this started
                    // So basically we take the winners from the previous round, test them
                        ↪  and compare
                    outFile.Write("LLR␣comp_" + i + "_to_" + (i + iter_jump − 1) + "\t=\t(
                        ↪ winner_" + i + "_to_" + (i + prev_iter_jump − 1) + "␣<␣winner_
                        ↪ " + (i + prev_iter_jump) + "_to_" + (i + prev_iter_jump +
                        ↪ prev_iter_jump − 1) + ")␣?␣(*allones)␣:␣(*zeros);\n");
                    iseven = (iseven + 1) % 2;
                }
                if (carry > 0 && previseven == 1)
                {
                    // If we had one remaining at the last iteration, add it here
                    outFile.Write("LLR␣comp_" + i + "_to_" + (n_inputs − 1) + "\t=\t(
                        ↪ winner_" + i + "_to_" + (i + prev_iter_jump − 1) + "␣<␣winner_
                        ↪ " + (i + prev_iter_jump) + "_to_" + (n_inputs − 1) + ")␣?␣(*
                        ↪ allones)␣:␣(*zeros);\n");

                }
                outFile.WriteLine("\n\n");

                // Now we write the winners
                for (i = 0; i < (n_inputs − iter_jump − 1); i += iter_jump)
                {
                    outFile.WriteLine("//␣First␣Minimum");
                    outFile.Write("LLR␣winner_" + i + "_to_" + (i + iter_jump − 1) + "\t=\
                        ↪ t(winner_" + i + "_to_" + (i + prev_iter_jump − 1) + "␣&␣comp_
                        ↪ " + i + "_to_" + (i + iter_jump − 1) + ")␣|␣(winner_" + (i +
                        ↪ prev_iter_jump) + "_to_" + (i + prev_iter_jump +
                        ↪ prev_iter_jump − 1) + "␣&␣(comp_" + i + "_to_" + (i +
                        ↪ iter_jump − 1) + "␣^␣(*allones)));\n");
                    outFile.WriteLine("//␣Second␣Minimum");
                    outFile.Write("LLR␣loser_" + (loserIndex++) + "\t=\t(winner_" + (i +
                        ↪ prev_iter_jump) + "_to_" + (i + prev_iter_jump +
                        ↪ prev_iter_jump − 1) + "␣&␣comp_" + i + "_to_" + (i + iter_jump
                        ↪  − 1) + ")␣|␣(winner_" + (i) + "_to_" + (i + prev_iter_jump −
                        ↪ 1) + "␣&␣(comp_" + i + "_to_" + (i + iter_jump − 1) + "␣^␣(*
                        ↪ allones)));\n");
                    outFile.WriteLine("\n");
                }
                if (carry > 0 && previseven == 1)
                {
                    outFile.WriteLine("//␣First␣Minimum");
                    outFile.Write("LLR␣winner_" + i + "_to_" + (n_inputs − 1) + "\t=\t(
                        ↪ winner_" + i + "_to_" + (i + prev_iter_jump − 1) + "␣&␣comp_"
                        ↪ + i + "_to_" + (n_inputs − 1) + ")␣|␣(winner_" + (i +
                        ↪ prev_iter_jump) + "_to_" + (n_inputs − 1) + "␣&␣(comp_" + i +
                        ↪ "_to_" + (n_inputs − 1) + "␣^␣(*allones)));\n");
                    outFile.WriteLine("//␣Second␣Minimum");
                    outFile.Write("LLR␣loser_" + (loserIndex++) + "\t=\t(winner_" + (i +
                        ↪ prev_iter_jump) + "_to_" + (n_inputs − 1) + "␣&␣comp_" + i + "
                        ↪ _to_" + (n_inputs − 1) + ")␣|␣(winner_" + (i) + "_to_" + (i +
                        ↪ prev_iter_jump − 1) + "␣&␣(comp_" + i + "_to_" + (n_inputs −
                        ↪ 1) + "␣^␣(*allones)));\n");
                    outFile.WriteLine("\n");
                }

                carry = n_inputs − i;

                outFile.WriteLine("/*-----------------------------------␣END␣ROUND␣" +
                    ↪ round + "␣-----------------------------------*/");
                outFile.WriteLine("\n\n\n");
            }
```

```
outFile.WriteLine("/*************************************␣ROUND␣6␣
     ↪ *************************************/");
previseven = iseven;
if (carry > 0 && previseven == 1)
{
    outFile.Write("LLR␣comp_" + 0 + "_to_" + (n_inputs - 1) + "\t=\t(winner_"
         ↪ + 0 + "_to_" + (i - 1) + "␣<␣winner_" + (i) + "_to_" + (n_inputs -
         ↪  1) + ")␣?␣(*allones)␣:␣(*zeros);\n");
    outFile.WriteLine("\n\n");
    outFile.WriteLine("//␣First␣Minimum");
    outFile.Write("LLR␣winner_" + 0 + "_to_" + (n_inputs - 1) + "\t=\t(winner_
         ↪ " + 0 + "_to_" + (i - 1) + "␣&␣comp_" + 0 + "_to_" + (n_inputs -
         ↪ 1) + ")␣|␣(winner_" + (i) + "_to_" + (n_inputs - 1) + "␣&␣(comp_"
         ↪ + 0 + "_to_" + (n_inputs - 1) + "␣^␣(*allones)));\n");
    outFile.WriteLine("//␣Second␣Minimum");
    outFile.Write("LLR␣loser_" + (loserIndex++) + "\t=\t(winner_" + (i) + "
         ↪ _to_" + (n_inputs - 1) + "␣&␣comp_" + 0 + "_to_" + (n_inputs - 1)
         ↪ + ")␣|␣(winner_" + (0) + "_to_" + (i - 1) + "␣&␣(comp_" + 0 + "
         ↪ _to_" + (n_inputs - 1) + "␣^␣(*allones)));\n");
    outFile.WriteLine("\n");
}

outFile.WriteLine("/*-----------------------------------␣END␣ROUND␣6␣
     ↪ -----------------------------------*/");
outFile.WriteLine("\n\n\n");


outFile.WriteLine("cnFirstMin␣=␣winner_" + 0 + "_to_" + (n_inputs - 1) + ";\n"
     ↪ );



outFile.WriteLine("\n\n\n");
outFile.WriteLine("/*********************************␣SECOND␣MINIMUM␣
     ↪ **********************************/\n\n");

// Alright so we have all (loserIndex) number of losers, we just need to redo
     ↪ the tournament.

// This time we don't have different naming conventions
// New size
n_inputs--;
num_rounds = Convert.ToInt32(Math.Log(n_inputs, 2));
iseven = 0;
previseven = 0;
iter_jump = 2;
// First we start with the first round of the first minimum. This compares all
     ↪  adjacent entries to each other
// Let's iterate over half the inputs
outFile.WriteLine("/*************************************␣ROUND␣1␣
     ↪ *************************************/");
for (i = 0; i < (n_inputs - 1); i += iter_jump)
{
    // Ok. So in each of these we compare the neighbors and add two
    // We need to write LLR comp_i_j  = (cnInputsMag[i] < cnInputsMag[j]) ? (*
         ↪ allones) : (*zeros); where j = i+1
    // Let's start with the beginning
    outFile.Write("LLR␣second_comp_" + i + "_to_" + (i + 1) + "\t=\t(loser_" +
         ↪  (i) + "␣␣<␣loser_" + (i+1) + ")␣?␣(*allones)␣:␣(*zeros);\n");
    iseven = (iseven + 1) % 2;
}

outFile.WriteLine("\n\n");

// Next we get the winners
for (i = 0; i < (n_inputs - 1); i += iter_jump)
{
    outFile.WriteLine("LLR␣second_winner_" + i + "_to_" + (i + 1) + "\t=\t(
         ↪ loser_" + (i) + "␣&␣second_comp_" + i + "_to_" + (i + 1) + ")␣|␣(
         ↪ loser_" + (i+1) + "␣&␣(second_comp_" + i + "_to_" + (i + 1) + "␣^␣
         ↪ (*allones)));");
}

carry = n_inputs - i;
outFile.WriteLine("/*-----------------------------------␣END␣ROUND␣1␣
     ↪ -----------------------------------*/");
outFile.WriteLine("\n\n\n");


previseven = iseven;
// Alright so this next part is basically automated now.
// Essentially we need to take the winners from above and advance them


// Next round!
outFile.WriteLine("/*************************************␣ROUND␣2␣
     ↪ *************************************/");
// Okie so for round two what we do is take the winners

// If we had an odd number from the last round, we need to add it to this
     ↪ round at the end
```

```
iter_jump = 4;


for (i = 0; i < (n_inputs - iter_jump); i += iter_jump)
{
    // Alright let's get this started
    // So basically we take the winners from the previous round, test them and
        ↪  compare

    outFile.Write("LLR␣second_comp_" + i + "_to_" + (i + iter_jump - 1) + "\t
        ↪ =\t(second_winner_" + i + "_to_" + (i + 1) + "␣<␣second_winner_" +
        ↪  (i + 2) + "_to_" + (i + 3) + ")␣?␣(*allones)␣:␣(*zeros);\n");
    iseven = (iseven + 1) % 2;
}
if (carry != 0 && previseven == 1)
{
    // Add the last one to the list of them but how?
    outFile.Write("LLR␣second_comp_" + i + "_to_" + (n_inputs - 1) + "\t=\t(
        ↪ second_winner_" + i + "_to_" + (i + 1) + "␣<␣loser_" + (n_inputs
        ↪ -1) + ")␣?␣(*allones)␣:␣(*zeros);\n");

}
carry = n_inputs - i;
outFile.WriteLine("\n\n");



// Now we write the winners
for (i = 0; i < (n_inputs - iter_jump); i += iter_jump)
{
    outFile.Write("LLR␣second_winner_" + i + "_to_" + (i + iter_jump - 1) + "\
        ↪ t=\t(second_winner_" + i + "_to_" + (i + 1) + "␣&␣second_comp_" +
        ↪ i + "_to_" + (i + iter_jump - 1) + ")␣|␣(second_winner_" + (i + 2)
        ↪  + "_to_" + (i + 3) + "␣&␣(second_comp_" + i + "_to_" + (i +
        ↪ iter_jump - 1) + "␣^␣(*allones)));\n");
}
if (carry != 0 && previseven == 1)
{
    outFile.Write("LLR␣second_winner_" + i + "_to_" + (n_inputs - 1) + "\t=\t(
        ↪ second_winner_" + i + "_to_" + (i + 1) + "␣&␣second_comp_" + i + "
        ↪ _to_" + (n_inputs - 1) + ")␣|␣(loser_" + i + "␣&␣(second_comp_" +
        ↪ i + "_to_" + (n_inputs - 1) + "␣^␣(*allones)));\n");

}
carry = n_inputs - i;

outFile.WriteLine("/*-----------------------------------␣END␣ROUND␣2␣
    ↪ ----------------------------------*/");
outFile.WriteLine("\n\n\n");


previseven = iseven;
for (int round = 3; round <= num_rounds; round++)
{
    outFile.WriteLine("/*************************************␣ROUND␣" + round
        ↪ + "␣*************************************/");
    previseven = iseven;
    iseven = 0;
    prev_iter_jump = iter_jump;
    iter_jump = iter_jump * 2;
    for (i = 0; i < (n_inputs - iter_jump); i += iter_jump)
    {
        // Alright let's get this started
        // So basically we take the winners from the previous round, test them
            ↪  and compare
        outFile.Write("LLR␣second_comp_" + i + "_to_" + (i + iter_jump - 1) +
            ↪ "\t=\t(second_winner_" + i + "_to_" + (i + prev_iter_jump - 1)
            ↪  + "␣<␣second_winner_" + (i + prev_iter_jump) + "_to_" + (i +
            ↪ prev_iter_jump + prev_iter_jump - 1) + ")␣?␣(*allones)␣:␣(*
            ↪ zeros);\n");
        iseven = (iseven + 1) % 2;
    }
    if (carry > 0 && previseven == 1)
    {
        // If we had one remaining at the last iteration, add it here
        outFile.Write("LLR␣second_comp_" + i + "_to_" + (n_inputs - 1) + "\t=\
            ↪ t(second_winner_" + i + "_to_" + (i + prev_iter_jump - 1) + "␣
            ↪ <␣second_winner_" + (i + prev_iter_jump) + "_to_" + (n_inputs
            ↪ - 1) + ")␣?␣(*allones)␣:␣(*zeros);\n");

    }
    outFile.WriteLine("\n\n");

    // Now we write the winners
    for (i = 0; i < (n_inputs - iter_jump); i += iter_jump)
    {
        outFile.Write("LLR␣second_winner_" + i + "_to_" + (i + iter_jump - 1)
            ↪ + "\t=\t(second_winner_" + i + "_to_" + (i + prev_iter_jump -
            ↪ 1) + "␣&␣second_comp_" + i + "_to_" + (i + iter_jump - 1) + ")
            ↪ ␣|␣(second_winner_" + (i + prev_iter_jump) + "_to_" + (i +
                        ↪ prev_iter_jump + prev_iter_jump - 1) + "␣&␣(second_comp_" + i
                        ↪ + "_to_" + (i + iter_jump - 1) + "␣^␣(*allones)));\n");
                }
                if (carry > 0 && previseven == 1)
                {
                    outFile.Write("LLR␣second_winner_" + i + "_to_" + (n_inputs - 1) + "\t
                        ↪ =\t(second_winner_" + i + "_to_" + (i + prev_iter_jump - 1) +
                        ↪ "␣&␣second_comp_" + i + "_to_" + (n_inputs - 1) + ")␣|␣(
                        ↪ second_winner_" + (i + prev_iter_jump) + "_to_" + (n_inputs -
                        ↪ 1) + "␣&␣(second_comp_" + i + "_to_" + (n_inputs - 1) + "␣^␣(*
                        ↪ allones)));\n");
                }

                carry = n_inputs - i;

                outFile.WriteLine("/*----------------------------------␣END␣ROUND␣" +
                    ↪ round + "␣----------------------------------*/");
                outFile.WriteLine("\n\n\n");
            }

            outFile.WriteLine("/************************************␣ROUND␣6␣
                ↪ ************************************/");
            previseven = iseven;
            if (carry > 0 && previseven == 1)
            {
                outFile.Write("LLR␣second_comp_" + 0 + "_to_" + (n_inputs - 1) + "\t=\t(
                    ↪ second_winner_" + 0 + "_to_" + (i - 1) + "␣<␣loser_" + (n_inputs
                    ↪ -1) + ")␣?␣(*allones)␣:␣(*zeros);\n");
                outFile.WriteLine("\n\n");
                outFile.Write("LLR␣second_winner_" + 0 + "_to_" + (n_inputs - 1) + "\t=\t(
                    ↪ second_winner_" + 0 + "_to_" + (i - 1) + "␣&␣second_comp_" + 0 + "
                    ↪ _to_" + (n_inputs - 1) + ")␣|␣(loser_" + (n_inputs-1) + "␣&␣(
                    ↪ second_comp_" + 0 + "_to_" + (n_inputs - 1) + "␣^␣(*allones)));\n"
                    ↪ );

            }

            outFile.WriteLine("/*----------------------------------␣END␣ROUND␣6␣
                ↪ ----------------------------------*/");
            outFile.WriteLine("\n\n\n");

            // Finally let's state the second minimum

            outFile.WriteLine("cnSecondMin␣=␣second_winner_" + 0 + "_to_" + (n_inputs - 1)
                ↪  + ";\n");




            outFile.Close();

        }
    }
}
```

Appendix V

Emulator Compilation Setup Script

```
start cmd.exe /k ""C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\bin\amd64\
    ↪ vcvars64.bat"␣&␣title␣Emulator␣Compiler"
```

Appendix W

Emulator Execution Setup Script

```
start cmd.exe /k "set CL_CONTEXT_EMULATOR_DEVICE_ALTERA=1 & title Emulator Run"
```

Appendix X

Nvidia GTX 690 Device Information Query Results

```
oclDeviceQuery.exe Starting...

OpenCL SW Info:

 CL_PLATFORM_NAME:   NVIDIA CUDA
 CL_PLATFORM_VERSION:    OpenCL 1.2 CUDA 8.0.0
 OpenCL SDK Revision:    7027912


OpenCL Device Info:

 2 devices found supporting OpenCL:

 ---------------------------------
 Device GeForce GTX 690
 ---------------------------------
  CL_DEVICE_NAME:          GeForce GTX 690
  CL_DEVICE_VENDOR:          NVIDIA Corporation
  CL_DRIVER_VERSION:          368.81
  CL_DEVICE_VERSION:          OpenCL 1.2 CUDA
  CL_DEVICE_OPENCL_C_VERSION:       OpenCL C 1.2
  CL_DEVICE_TYPE:        CL_DEVICE_TYPE_GPU
  CL_DEVICE_MAX_COMPUTE_UNITS:       8
  CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS:  3
  CL_DEVICE_MAX_WORK_ITEM_SIZES:   1024 / 1024 / 64
  CL_DEVICE_MAX_WORK_GROUP_SIZE:   1024
  CL_DEVICE_MAX_CLOCK_FREQUENCY:   1019 MHz
  CL_DEVICE_ADDRESS_BITS:     64
  CL_DEVICE_MAX_MEM_ALLOC_SIZE:    512 MByte
  CL_DEVICE_GLOBAL_MEM_SIZE:      2048 MByte
  CL_DEVICE_ERROR_CORRECTION_SUPPORT: no
  CL_DEVICE_LOCAL_MEM_TYPE:     local
  CL_DEVICE_LOCAL_MEM_SIZE:     48 KByte
  CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE: 64 KByte
  CL_DEVICE_QUEUE_PROPERTIES:     CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE
  CL_DEVICE_QUEUE_PROPERTIES:     CL_QUEUE_PROFILING_ENABLE
  CL_DEVICE_IMAGE_SUPPORT:      1
  CL_DEVICE_MAX_READ_IMAGE_ARGS:   256
  CL_DEVICE_MAX_WRITE_IMAGE_ARGS:  16
  CL_DEVICE_SINGLE_FP_CONFIG:    denorms INF-quietNaNs round-to-nearest round-to-zero round
      ↪ -to-inf fma

  CL_DEVICE_IMAGE <dim>       2D_MAX_WIDTH    16384
          2D_MAX_HEIGHT   16384
          3D_MAX_WIDTH    4096
          3D_MAX_HEIGHT   4096
          3D_MAX_DEPTH    4096

  CL_DEVICE_EXTENSIONS:       cl_khr_global_int32_base_atomics
          cl_khr_global_int32_extended_atomics
          cl_khr_local_int32_base_atomics
          cl_khr_local_int32_extended_atomics
          cl_khr_fp64
          cl_khr_byte_addressable_store
          cl_khr_icd
          cl_khr_gl_sharing
          cl_nv_compiler_options
          cl_nv_device_attribute_query
          cl_nv_pragma_unroll
          cl_nv_d3d9_sharing
          cl_nv_d3d10_sharing
          cl_khr_d3d10_sharing
```

```
        cl_nv_d3d11_sharing


 CL_DEVICE_COMPUTE_CAPABILITY_NV:   3.0
 NUMBER OF MULTIPROCESSORS:      8
 NUMBER OF CUDA CORES:       1536
 CL_DEVICE_REGISTERS_PER_BLOCK_NV: 65536
 CL_DEVICE_WARP_SIZE_NV:    32
 CL_DEVICE_GPU_OVERLAP_NV:    CL_TRUE
 CL_DEVICE_KERNEL_EXEC_TIMEOUT_NV: CL_FALSE
 CL_DEVICE_INTEGRATED_MEMORY_NV: CL_FALSE
 CL_DEVICE_PREFERRED_VECTOR_WIDTH_<t>  CHAR 1, SHORT 1, INT 1, LONG 1, FLOAT 1, DOUBLE 1


---------------------------------
Device GeForce GTX 690
---------------------------------
 CL_DEVICE_NAME:          GeForce GTX 690
 CL_DEVICE_VENDOR:          NVIDIA Corporation
 CL_DRIVER_VERSION:         368.81
 CL_DEVICE_VERSION:         OpenCL 1.2 CUDA
 CL_DEVICE_OPENCL_C_VERSION:       OpenCL C 1.2
 CL_DEVICE_TYPE:        CL_DEVICE_TYPE_GPU
 CL_DEVICE_MAX_COMPUTE_UNITS:      8
 CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS: 3
 CL_DEVICE_MAX_WORK_ITEM_SIZES:   1024 / 1024 / 64
 CL_DEVICE_MAX_WORK_GROUP_SIZE:   1024
 CL_DEVICE_MAX_CLOCK_FREQUENCY:   1019 MHz
 CL_DEVICE_ADDRESS_BITS:    64
 CL_DEVICE_MAX_MEM_ALLOC_SIZE:    512 MByte
 CL_DEVICE_GLOBAL_MEM_SIZE:      2048 MByte
 CL_DEVICE_ERROR_CORRECTION_SUPPORT: no
 CL_DEVICE_LOCAL_MEM_TYPE:     local
 CL_DEVICE_LOCAL_MEM_SIZE:    48 KByte
 CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE: 64 KByte
 CL_DEVICE_QUEUE_PROPERTIES:    CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE
 CL_DEVICE_QUEUE_PROPERTIES:    CL_QUEUE_PROFILING_ENABLE
 CL_DEVICE_IMAGE_SUPPORT:      1
 CL_DEVICE_MAX_READ_IMAGE_ARGS:   256
 CL_DEVICE_MAX_WRITE_IMAGE_ARGS: 16
 CL_DEVICE_SINGLE_FP_CONFIG:    denorms INF-quietNaNs round-to-nearest round-to-zero round
     ↪ -to-inf fma

 CL_DEVICE_IMAGE <dim>       2D_MAX_WIDTH    16384
        2D_MAX_HEIGHT  16384
        3D_MAX_WIDTH    4096
        3D_MAX_HEIGHT   4096
        3D_MAX_DEPTH    4096

 CL_DEVICE_EXTENSIONS:       cl_khr_global_int32_base_atomics
        cl_khr_global_int32_extended_atomics
        cl_khr_local_int32_base_atomics
        cl_khr_local_int32_extended_atomics
        cl_khr_fp64
        cl_khr_byte_addressable_store
        cl_khr_icd
        cl_khr_gl_sharing
        cl_nv_compiler_options
        cl_nv_device_attribute_query
        cl_nv_pragma_unroll
        cl_nv_d3d9_sharing
        cl_nv_d3d10_sharing
        cl_khr_d3d10_sharing
        cl_nv_d3d11_sharing


 CL_DEVICE_COMPUTE_CAPABILITY_NV:   3.0
 NUMBER OF MULTIPROCESSORS:      8
 NUMBER OF CUDA CORES:       1536
 CL_DEVICE_REGISTERS_PER_BLOCK_NV: 65536
 CL_DEVICE_WARP_SIZE_NV:    32
 CL_DEVICE_GPU_OVERLAP_NV:    CL_TRUE
 CL_DEVICE_KERNEL_EXEC_TIMEOUT_NV: CL_FALSE
 CL_DEVICE_INTEGRATED_MEMORY_NV: CL_FALSE
 CL_DEVICE_PREFERRED_VECTOR_WIDTH_<t>  CHAR 1, SHORT 1, INT 1, LONG 1, FLOAT 1, DOUBLE 1


 ---------------------------------
 2D Image Formats Supported (75)
 ---------------------------------
 #      Channel Order    Channel Type

 1      CL_R             CL_FLOAT
 2      CL_R             CL_HALF_FLOAT
 3      CL_R             CL_UNORM_INT8
 4      CL_R             CL_UNORM_INT16
 5      CL_R             CL_SNORM_INT16
 6      CL_R             CL_SIGNED_INT8
 7      CL_R             CL_SIGNED_INT16
 8      CL_R             CL_SIGNED_INT32
 9      CL_R             CL_UNSIGNED_INT8
 10     CL_R             CL_UNSIGNED_INT16
```

```
11      CL_R              CL_UNSIGNED_INT32
12      CL_A              CL_FLOAT
13      CL_A              CL_HALF_FLOAT
14      CL_A              CL_UNORM_INT8
15      CL_A              CL_UNORM_INT16
16      CL_A              CL_SNORM_INT16
17      CL_A              CL_SIGNED_INT8
18      CL_A              CL_SIGNED_INT16
19      CL_A              CL_SIGNED_INT32
20      CL_A              CL_UNSIGNED_INT8
21      CL_A              CL_UNSIGNED_INT16
22      CL_A              CL_UNSIGNED_INT32
23      CL_RG             CL_FLOAT
24      CL_RG             CL_HALF_FLOAT
25      CL_RG             CL_UNORM_INT8
26      CL_RG             CL_UNORM_INT16
27      CL_RG             CL_SNORM_INT16
28      CL_RG             CL_SIGNED_INT8
29      CL_RG             CL_SIGNED_INT16
30      CL_RG             CL_SIGNED_INT32
31      CL_RG             CL_UNSIGNED_INT8
32      CL_RG             CL_UNSIGNED_INT16
33      CL_RG             CL_UNSIGNED_INT32
34      CL_RA             CL_FLOAT
35      CL_RA             CL_HALF_FLOAT
36      CL_RA             CL_UNORM_INT8
37      CL_RA             CL_UNORM_INT16
38      CL_RA             CL_SNORM_INT16
39      CL_RA             CL_SIGNED_INT8
40      CL_RA             CL_SIGNED_INT16
41      CL_RA             CL_SIGNED_INT32
42      CL_RA             CL_UNSIGNED_INT8
43      CL_RA             CL_UNSIGNED_INT16
44      CL_RA             CL_UNSIGNED_INT32
45      CL_RGBA           CL_FLOAT
46      CL_RGBA           CL_HALF_FLOAT
47      CL_RGBA           CL_UNORM_INT8
48      CL_RGBA           CL_UNORM_INT16
49      CL_RGBA           CL_SNORM_INT16
50      CL_RGBA           CL_SIGNED_INT8
51      CL_RGBA           CL_SIGNED_INT16
52      CL_RGBA           CL_SIGNED_INT32
53      CL_RGBA           CL_UNSIGNED_INT8
54      CL_RGBA           CL_UNSIGNED_INT16
55      CL_RGBA           CL_UNSIGNED_INT32
56      CL_BGRA           CL_UNORM_INT8
57      CL_BGRA           CL_SIGNED_INT8
58      CL_BGRA           CL_UNSIGNED_INT8
59      CL_ARGB           CL_UNORM_INT8
60      CL_ARGB           CL_SIGNED_INT8
61      CL_ARGB           CL_UNSIGNED_INT8
62      CL_INTENSITY      CL_FLOAT
63      CL_INTENSITY      CL_HALF_FLOAT
64      CL_INTENSITY      CL_UNORM_INT8
65      CL_INTENSITY      CL_UNORM_INT16
66      CL_INTENSITY      CL_SNORM_INT16
67      CL_LUMINANCE      CL_FLOAT
68      CL_LUMINANCE      CL_HALF_FLOAT
69      CL_LUMINANCE      CL_UNORM_INT8
70      CL_LUMINANCE      CL_UNORM_INT16
71      CL_LUMINANCE      CL_SNORM_INT16
72      CL_BGRA           CL_SNORM_INT8
73      CL_BGRA           CL_SNORM_INT16
74      CL_ARGB           CL_SNORM_INT8
75      CL_ARGB           CL_SNORM_INT16

--------------------------------
3D Image Formats Supported (75)
--------------------------------
#       Channel Order     Channel Type

1       CL_R              CL_FLOAT
2       CL_R              CL_HALF_FLOAT
3       CL_R              CL_UNORM_INT8
4       CL_R              CL_UNORM_INT16
5       CL_R              CL_SNORM_INT16
6       CL_R              CL_SIGNED_INT8
7       CL_R              CL_SIGNED_INT16
8       CL_R              CL_SIGNED_INT32
9       CL_R              CL_UNSIGNED_INT8
10      CL_R              CL_UNSIGNED_INT16
11      CL_R              CL_UNSIGNED_INT32
12      CL_A              CL_FLOAT
13      CL_A              CL_HALF_FLOAT
14      CL_A              CL_UNORM_INT8
15      CL_A              CL_UNORM_INT16
16      CL_A              CL_SNORM_INT16
17      CL_A              CL_SIGNED_INT8
18      CL_A              CL_SIGNED_INT16
19      CL_A              CL_SIGNED_INT32
20      CL_A              CL_UNSIGNED_INT8
21      CL_A              CL_UNSIGNED_INT16
```

```
  22    CL_A              CL_UNSIGNED_INT32
  23    CL_RG             CL_FLOAT
  24    CL_RG             CL_HALF_FLOAT
  25    CL_RG             CL_UNORM_INT8
  26    CL_RG             CL_UNORM_INT16
  27    CL_RG             CL_SNORM_INT16
  28    CL_RG             CL_SIGNED_INT8
  29    CL_RG             CL_SIGNED_INT16
  30    CL_RG             CL_SIGNED_INT32
  31    CL_RG             CL_UNSIGNED_INT8
  32    CL_RG             CL_UNSIGNED_INT16
  33    CL_RG             CL_UNSIGNED_INT32
  34    CL_RA             CL_FLOAT
  35    CL_RA             CL_HALF_FLOAT
  36    CL_RA             CL_UNORM_INT8
  37    CL_RA             CL_UNORM_INT16
  38    CL_RA             CL_SNORM_INT16
  39    CL_RA             CL_SIGNED_INT8
  40    CL_RA             CL_SIGNED_INT16
  41    CL_RA             CL_SIGNED_INT32
  42    CL_RA             CL_UNSIGNED_INT8
  43    CL_RA             CL_UNSIGNED_INT16
  44    CL_RA             CL_UNSIGNED_INT32
  45    CL_RGBA           CL_FLOAT
  46    CL_RGBA           CL_HALF_FLOAT
  47    CL_RGBA           CL_UNORM_INT8
  48    CL_RGBA           CL_UNORM_INT16
  49    CL_RGBA           CL_SNORM_INT16
  50    CL_RGBA           CL_SIGNED_INT8
  51    CL_RGBA           CL_SIGNED_INT16
  52    CL_RGBA           CL_SIGNED_INT32
  53    CL_RGBA           CL_UNSIGNED_INT8
  54    CL_RGBA           CL_UNSIGNED_INT16
  55    CL_RGBA           CL_UNSIGNED_INT32
  56    CL_BGRA           CL_UNORM_INT8
  57    CL_BGRA           CL_SIGNED_INT8
  58    CL_BGRA           CL_UNSIGNED_INT8
  59    CL_ARGB           CL_UNORM_INT8
  60    CL_ARGB           CL_SIGNED_INT8
  61    CL_ARGB           CL_UNSIGNED_INT8
  62    CL_INTENSITY      CL_FLOAT
  63    CL_INTENSITY      CL_HALF_FLOAT
  64    CL_INTENSITY      CL_UNORM_INT8
  65    CL_INTENSITY      CL_UNORM_INT16
  66    CL_INTENSITY      CL_SNORM_INT16
  67    CL_LUMINANCE      CL_FLOAT
  68    CL_LUMINANCE      CL_HALF_FLOAT
  69    CL_LUMINANCE      CL_UNORM_INT8
  70    CL_LUMINANCE      CL_UNORM_INT16
  71    CL_LUMINANCE      CL_SNORM_INT16
  72    CL_BGRA           CL_SNORM_INT8
  73    CL_BGRA           CL_SNORM_INT16
  74    CL_ARGB           CL_SNORM_INT8
  75    CL_ARGB           CL_SNORM_INT16

oclDeviceQuery, Platform Name = NVIDIA CUDA, Platform Version = OpenCL 1.2 CUDA 8.0.0, SDK
    ↪  Revision = 7027912, NumDevs = 2, Device = GeForce GTX 690, Device = GeForce GTX
    ↪ 690

System Info:

 Local Time/Date = 1:37:9, 9/15/2016
 CPU Arch: 9
 CPU Level: 6
 # of CPU processors: 12
 Windows Build: 7601
 Windows Ver: 6.1 (Windows Vista / Windows 7)
```