Compiler-Driven Performance on Heterogeneous Computing Platforms

by

Artem Chikin

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

Master of Science

Department of Computing Science

University of Alberta

© Artem Chikin, 2019

Abstract

Modern parallel programming languages such as Open Multi-Processing (OpenMP) provide simple, portable programming models that support offloading of computation to various accelerator devices. Coupled with the increasing prevalence of heterogeneous computing platforms and the battle for supremacy in the co-processor space, gives rise to additional challenges placed on compiler/runtime vendors to handle the increasing complexity and diversity of shared-memory parallel platforms.

To start, this thesis presents three kernel re-structuring ideas that focus on improving the execution of high-level parallel code in Graphics Processing Unit (GPU) devices. The first addresses programs that include multiple parallel blocks within a single region of GPU code. A proposed compiler transformation can split such regions into multiple regions, leading to the launching of multiple kernels, one for each parallel region. Second, is a code transformation that sets up a pipeline of kernel execution and asynchronous data transfers. This transformation enables the overlap of communication and computation. The third idea is that the selection of a grid geometry for the execution of a parallel region must balance the GPU occupancy with the potential saturation of the memory throughput in the GPU. Adding this additional parameter to the geometry selection heuristic can often yield better performance at lower occupancy levels.

This thesis next describes the Iteration Point Difference Analysis — a new static-analysis framework that can be used to determine the memory coalescing characteristics of parallel loops that target GPU offloading and to ascertain safety and profitability of loop transformations with the goal of improving their memory-access characteristics. GPU kernel execution time across the Polybench suite is improved by up to 25.5× on an Nvidia P100 with benchmark overall improvement of up to 3.2×. An opportunity detected in a SPEC ACCEL benchmark yields kernel speedup of 86.5× with a benchmark improvement of 3.4×, and a kernel speedup of 111.1× with a benchmark improvement of 2.3× on an Nvidia P100 and V100, respectively.

The task of modelling performance takes on an ever increasing importance as systems must make automated decisions on the most suitable offloading target. The third contribution of this thesis motivates the need with a study of cross-architectural changes in profitability of kernel offloading to GPU versus host CPU execution, and presents a prototype design for a hybrid computing device selection framework.

Preface

Chapter 3 of this thesis has been published as **A. Chikin**, T. Gobran. J.N. Amaral, "OpenMP Code Offloading: Splitting GPU Kernels, Pipelining Communication and Computation, and Selecting Better Grid Geometries", *Proceedings of the Fifth Workshop on Accelerator Programming Using Directives*, in November 2018. This work was presented at the workshop during the Supercomputing 2018 conference in Dallas, TX, on November 11, 2018, by the author. My role on the project was to formulate the research project, introduce ideas for compiler transformations that the project evaluated, and form the experimental methodology to evaluate these ideas. I was also responsible for supervision of the undergraduate student working on the project, educating him about the subject and guiding the execution of the ideas. T. Gobran was responsible for carrying out the manual prototyping of the proposed code transformations, performing the experimental evaluation, reporting the results to myself and J.N. Amaral and drafting the manuscript. J.N. Amaral was the supervisory member of the research project team and contributed to the overall structure, direction of the resulting manuscript and to manuscript edits.

Chapter 4 of this thesis has been submitted for publication as **A. Chikin**, T. Lloyd, J.N. Amaral, E. Tiotto, M. Usman, "Memory-access-aware safety and profitability analysis for transformation of accelerator-bound OpenMP loops". The contents of this chapter fall under a patent filed with the U.S. Patent Office as **A. Chikin**, T. Lloyd, J. N. Amaral, E. Tiotto "Compiler for Restructuring Code Using Iteration-Point Algebraic Difference Analysis", Patent Reference: P201706298US01, filed on March 12, 2018. I was responsible for designing, building, and experimentally evaluating the analysis framework (within the IBM XL Compiler), the loop dependence test, and the associated safety/profitability analyses that are used to guide relevant automated compiler transformations. T. Lloyd contributed many technical discussions and advice that helped evolve the ideas developed during this research, the work also directly extends T. Lloyd's prior work. J.N. Amaral was the supervisory author and contributed to guiding the experimental evaluation and editing the resulting manuscript. E. Tiotto was the IBM collaborator, and participated in technical discussions. M. Usman contributed the formal mathematical formulation for the algorithms presented in this work.

Chapter 5 of this thesis has been submitted for publication as **A. Chikin**, J.N. Amaral, K. Ali, E. Tiotto, "Towards Hybrid Execution Target Selection Through Analytical Performance Modeling". The contents of this chapter cover ideas that fall under a patent filed with the U.S. Patent Office as **A. Chikin**, J.N. Amaral, K. Ali, E. Tiotto, "Hybrid Compute Device Selection Analysis", Patent Reference: P201803063US01, filed on August 31, 2018. I was responsible for conceiving the research project idea, constructing the architecture of the hybrid analysis, prototyping its components, and experimentally evaluating its efficacy. J.N. Amaral and K. Ali were supervisory authors and contributed to the focus of the project and edits to the resulting manuscript. E. Tiotto was the IBM collaborator, and participated in technical discussions.

Another research result, which is not part of this thesis, but was completed during my M.Sc program at the University of Alberta is a research project published as T. Lloyd, **A. Chikin**, E. Ochoa, K. Ali, J.N. Amaral , "A Case for Better Integration of Host and Target Compilation When Using OpenCL for FPGAs", *Proceedings of Fourth International Workshop on FPGAs for Software Programmers*, in October 2017. This work introduced a series of interconnected compiler transformations aimed at improving performance of Field-Programmable Gate Array (FPGA) programs generated through high-level synthesis of Open Compute Language (OpenCL). My roles on the project included implementing the reduction dependence elimination transformation and participating in guiding the overall direction of the research.

To my Mom and Dad

For all the sacrifices they have made, endless love, support and inspiration.

Acknowledgements

I would like to thank my supervisor, Dr. J Nelson Amaral, for his continuing support through guidance, advice and direction over the years that I have had the privilege of working with and learning from him.

I would also like to thank the many people working at, and visiting, the University of Alberta systems laboratory, for insightful conversations, technical arguments, and moments of whimsy. I have learned so much from each and every one of you.

This research has been funded by IBM Center for Advanced Studies (CAS) Canada and by the National Science and Engineering Research Council (NSERC) of Canada through their Collaborative Research and Development (CRD) program and through the Undergraduate Student Research Awards (USRA) program.

Contents

List of Tables

List of Figures

List of Acronyms

ACF Arithmetic Control Form.

DDG Data Dependence Graph.

FPGA Field-Programmable Gate Array.

GPU Graphics Processing Unit.

HPC High-Performance Computing.

IPDA Iteration Point Difference Analysis.

IR Intermediate Representation.

LLVM Low-Level Virtual Machine.

MIC Many Integrated Core.

OpenACC Open Accelerators.

OpenCL Open Compute Language.

OpenMP Open Multi-Processing.

SIMD Single Instruction, Multiple Data.

SIMT Single Instruction, Multiple Thread.

SM Streaming Multiprocessor.

VEG Value Evolution Graph.

Chapter 1

Introduction

Heterogeneous computing platforms are increasingly commonplace across all major application domains. Accelerator devices can be found in systems that vary from mobile phones to supercomputers. In the most recent Top500 list [59], 7 out of the top 10 machines use accelerator devices. The two fastest computers in the USA, the Summit supercomputer at the Oak Ridge National Laboratory and the Sierra supercomputer at the Lawrence Livermore National Laboratory rely heavily on accelerators for their computation. Summit has more than $25,000$ NVIDIA V100 GPUs coupled with $9,200$ IBM POWER 9 CPUs [56]. The Cray XC40 Trinity supercomputer, at the Los Alamos National Laboratory, achieves 41.5 petaflops of peak performance with a different heterogeneous architecture that uses Many Integrated Core (MIC) Xeon Phi co-processors [60]. Various competing accelerator platforms co-exist, often within the same host machine, as each has its own strengths, weaknesses, and corresponding favorable application domains. Moreover, different accelerator architectures place different, often opposing, demands on attributes of computer programs. Such demands must be met in order to achieve acceptable levels of performance.

Proliferation of heterogeneous computing has spurred development of programming models that allow developers to write applications that target execution on accelerator devices. High-level programming models such as OpenMP [13] and Open Accelerators (OpenACC) [58] provide the means to write architecture-agnostic accelerator code. Programs written using such high-level programming models are meant to be written once in a generic fashion and to run on a variety of computational devices from general-purpose CPUs to GPU, FPGA, and MIC co-processors. Target-agnostic programming abstracts the details of accelerator architecture from the developer and makes it the prerogative of the compiler/runtime to handle architecture-specific code generation, optimization, and parameter tuning, within the limits allowed by the programming model. Furthermore, programming models like OpenMP are shifting towards being more descriptive, rather than prescriptive, with the next iteration of the standard poised to introduce constructs that allow compilers most freedom yet on how to generate code and where it should execute (e.g. `#pragma loop`).

The notion of *performance portability* is a core tenet behind the design of high-level parallel programming models. OpenMP promises a model that allows users to write programs that seamlessly scale from workstation PCs to supercomputers. Yet, performance portability is extremely difficult to achieve because compiler heuristics and analyses struggle to keep up with advancements in micro-architecture. To compensate, the OpenMP language specification has, over time, accumulated hints that developers may provide to the compiler to increase performance for a specific target. For instance, the `collapse` clause instructs the compiler to collapse the iteration-space of multiple nested loops in a prescribed manner. The resulting code exhibits memory access patterns that favour a specific architecture's memory subsystem. Such hints detract from portability. More capable compilers should be able to generate the same higher-performing code without portability-reducing annotations.

Chapter 2

Background

A typical heterogeneous system consists of a host machine that operates using an ordinary CPU and contains main memory modules. Attached to the host machine, via a data-transfer bus, are one or more accelerator devices. An example topology of a host computer with two accelerator devices attached is shown in Figure 2.1. The host — a general-purpose CPU machine — is responsible for the overall system's operation, memory management and control of execution among attached devices. During execution of a program that contains an accelerator kernel — a piece of computation specified to be offloaded to an accelerator — the host machine schedules execution of the kernel on a given computing device and performs the necessary data transfers to and from the computing device. A fallback scenario may occur in which the required accelerator device is unavailable or busy, in which case the host may instead schedule execution of the accelerator kernel on the host machine's CPU. If the programming model allows it, the host may elect to schedule kernel execution either on the host itself or any of the available accelerators.

2.1 GPU Architecture, Programming Model, and Execution Platform

GPUs are composed of a large number of Streaming Multiprocessors (SMs), each capable of executing thousands of threads in parallel. Such massive parallelism requires a strict Single Instruction, Multiple Thread (SIMT) data-parallel programming model in order to achieve performance. The Nvidia V100

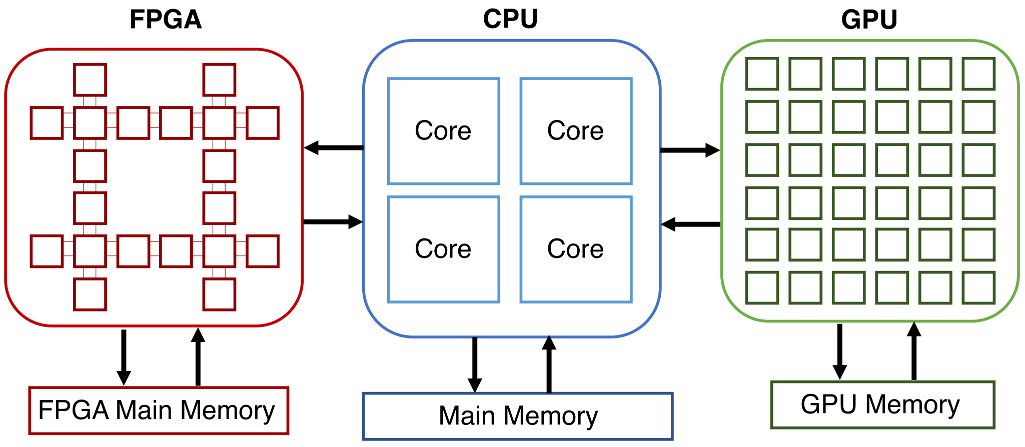


Figure 2.1: Example topology of a heterogeneous computing environment

is a state-of-the-art Nvidia GPU for high-performance computing composed of 80 SMs. Each SM can issue an instruction for 128 threads per cycle [44]. The V100 has enough resources to maintain the state of thousands of threads, which gives each SM the ability to context switch between threads with zero penalty — a key instruction-latency hiding mechanism. GPU threads are grouped into *warps*: all the threads that comprise a warp execute either the *same* instruction in lock-step or no instruction at all. Lock-step execution reduces the overhead of scheduling work across a large number of threads. Threads are further grouped into thread blocks or Cooperative Thread Arrays (CTA). All threads within a thread block must execute on a single SM. They can both share intermediate results via access to SM's shared memory banks and synchronize their execution. Threads from different thread blocks have no means of direct communication or synchronization. The number of threads per thread block and the number of thread blocks comprise the GPU *grid geometry*.[1]

[1]NVIDIA-specific terms such as *warp* and *SM* are used throughout this thesis for the sake of clarity and consistency.

2.1.1 Memory Coalescing

In a typical data-parallel kernel, thread identifiers are used in memory-access addressing expressions to load/store the data items for each thread. The number of memory requests issued by a warp in a given cycle can be as large as the number of threads in a warp (32 in current architectures) because the threads belonging to it execute the same instruction simultaneously. The GPU global memory subsystem has a limited amount of bandwidth available. As a means to reduce the overall number of requests, the GPU *coalesces* multiple same-cycle accesses to memory within the same cache line into a single request. Coalescing memory accesses into fewer requests can dramatically improve memory throughput because no thread in a warp can continue execution until the memory accesses of all threads have completed. The number of requests necessary to satisfy all the accesses in a warp of threads is equal to the number of distinct cache lines that are accessed. Each global memory request requires hundreds of cycles to be completed; thus, structuring GPU programs to avoid non-coalesced memory accesses is paramount for performance [1].

2.2 OpenMP and Accelerator Programming

Programming in CUDA or OpenCL is a painstaking process that requires developers to have thorough working knowledge of the accelerator architecture being targeted. An alternative are high-level directive-based parallel programming models such as OpenMP that support offloading of code regions to accelerators. These models reduce the amount of effort required to write accelerator code by abstracting the accelerator hardware specifics from the developer. Being platform-agnostic, these models promise code portability across existing and future accelerators. Figure 2.2 shows an example parallel loop written using OpenMP 4.x with accelerator offloading. The programmer specifies a `target` region, directing the compiler to offload the region to an accelerator device. The target region directive is annotated with data-transfer `map` clauses that indicate which arrays must be transferred to and from the device data environment. Full assortment of OpenMP parallelism constructs are supported inside `target` re-

```
void vecAdd(double *a, double *b, double *c, int n)
{
  #pragma omp target map(to: a[:n], b[:n]) map(from: c[:n])
  #pragma omp teams distribute parallel for
  for (int i = 0; i < n; i++)
    c[i] = a[i] + b[i];
}
```

Figure 2.2: Example vector addition OpenMP 4.x accelerator code

gions; however, task-level parallelism maps poorly to data-parallel devices such as GPUs. Thus, performance considerations limit the expression of parallelism in OpenMP for execution in GPUs to `parallel` and loop constructs.

Moreover, GPUs' reliance on a SIMT execution model has implications on the compiler implementations that generate GPU code from OpenMP. Where possible, parallel constructs must be mapped to data-parallel structures in order to achieve good performance. Other aspects of the accelerator architecture must be taken into account in for efficient hardware utilization, such as the selection of a *grid geometry*. In contrast with low-level languages — such as CUDA and OpenCL — where the selection of program runtime parameters such as grid geometry is the prerogative of the programmer, OpenMP assigns this task to the compiler/runtime-system designers. On one hand the rigid execution model that must be followed when generating GPU code makes the compiler designer's task challenging. On the other hand it affords considerable freedom for code generation from high-level descriptive models. Grid geometry selection in the OpenMP context was one of the research projects pursued as a component of this thesis by the author, in collaboration with T. Lloyd. [37].

OpenMP can express three levels of parallelism inside a `target` region: the `teams` construct declares a region of code to be executed by a league of threads, the `parallel` construct declares a task to be executed in parallel by threads within a league, and the `simd` construct declares vector-based execution of a loop. The first two provide a natural mapping to the GPU's notions of thread blocks and threads, respectively. Both `teams` and `parallel` constructs have associated loop work-distribution clauses: `teams distribute` and `parallel` `for`. `teams distribute` distributes iterations of the associated loop into equal-sized partitions spread among teams, and `parallel for` distributes loop iterations within a chunk to individual threads. Combined constructs are often used to capture all of the above levels of parallelism for a given loop into one prescription, such as `teams distribute parallel for` construct in Figure 2.2.

2.2.1 OpenMP 4.x GPU Code Generation

As implemented in OpenMP 4.x for LLVM/Clang as well as in the IBM XL C/C++/Fortran compilers, `target` regions are outlined into separate procedures. The outlined procedure is cloned into two versions: a device version and a host fallback version. CPU code is generated for the fallback variant. A kernel suitable for GPU execution is generated for the device version. To best take advantage of the GPU, data-parallel code is generated in place of parallel loops. However, full breadth of the OpenMP specification must be supported by a compliant compiler implementation. Thus, the GPU code generator must be able to handle a multitude of constructs that contain both serial and parallel code that may be nested or adjacent within a `target` region.

The compilers used in this work employ a cooperative threading model that utilizes the technique of warp specialization to generate data-parallel GPU code from parallel OpenMP regions [24]. Parallel work is performed by a collection of worker warps and coordinated by a single master warp (selected to be the last warp in a CTA). The coordination between warps is done through the use of a CTA-level synchronization primitive that allows for named barriers that apply to a compiler-specified number of warps to participate in the barrier (`bar.sync $0 $1`). When the master encounters a parallel region, it activates the required number of worker warps and suspends its own execution. Figure 2.3 shows a visualization of the process described above.

The resulting device kernel is translated into Nvidia's *Parallel Thread Execution*(PTX) pseudo-assembly language, using Nvidia's proprietary PTXAS assembler. The host code that previously contained the target region is rewritten to invoke the outlined device kernel through a runtime method call.

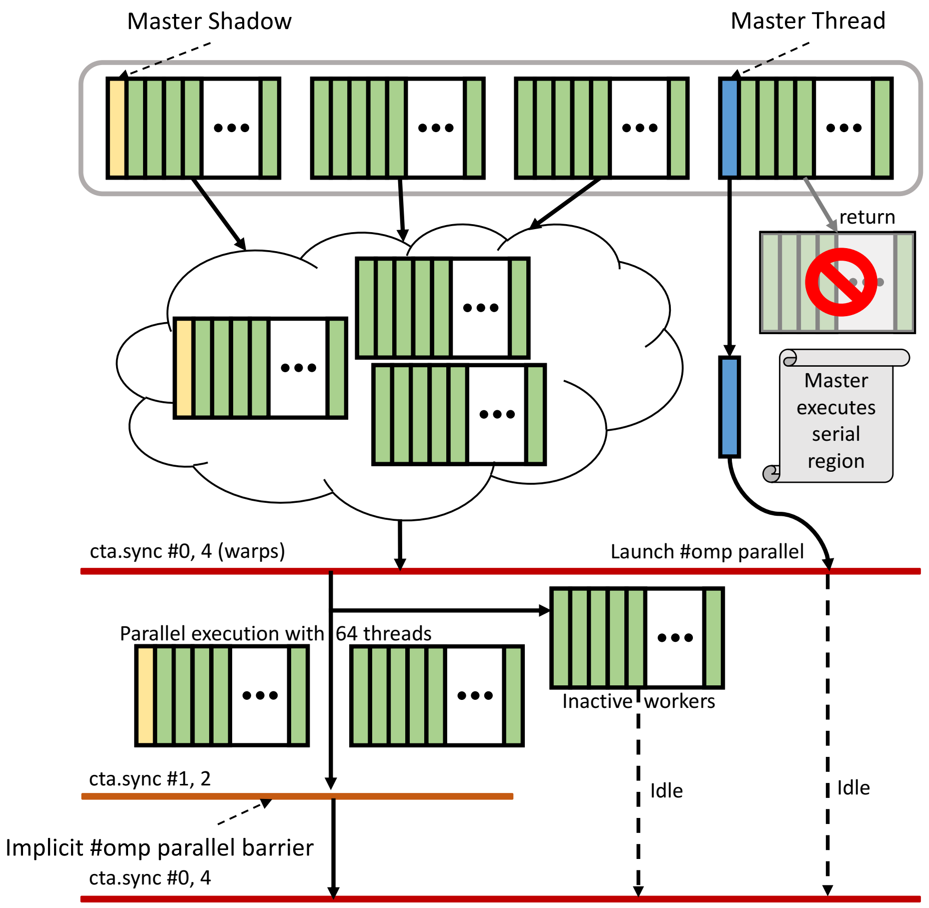


Figure 2.3: Visualization of the cooperative threading OpenMP 4 GPU code-generation scheme. (Adapted from similar figure in [24])

The runtime performs the required setup and data-transfer. Finally, the GPU runtime compiles the PTX code into machine instructions and launches kernel execution.

Warp Specialization Elision

While necessary to support the full breadth of possible OpenMP constructs that can occur in target regions, as well as serial code sections and sibling parallel regions, warp specialization code-generation scheme incurs a significant amount of runtime overhead that can be avoided in select special cases. Not all kernels require the full machinery of the cooperative code-generation scheme. For target regions that are comprised solely of a single parallel loop with no nested OpenMP constructs, and no serial code, the compiler optimizes the generated code by eliding the warp specialization and runtime-managed sections of the code. This optimization results in dramatically simpler generated data-parallel code that eliminates the mentioned overheads.

Elision of the cooperative code-generation scheme and its incurred synchronization points is enabled by target region splitting. Jacob et al. describe how this elision is handled automatically by the Clang compiler, and present a performance study of elision [24].

2.3 Symbolic Static Analysis: Arithmetic Control Form

The Arithmetic Control Form (ACF) static analysis framework, introduced by Lloyd et. al., is a way to capture linear and non-linear relationships between program statements [36]. ACF's main approach is to combine data and control flow by computing symbolic values for expressions of interest. Similarly to the work by Ferriere and Stoutchinin on ϕ-nodes [55] and prior efforts in if-conversion [39], ACF converts conditionally-executed statements into predicated statements, capturing definitions across all potential traces through the program. Resulting ACF expressions consist of binary operations on constants and symbols representing compile-time unknowns.

In the context of data-parallel programs, ACF's key strength lies in its ability to compute an algebraic difference on the symbolic representation of a statement. For instance, consider a statement $\mathcal{S}$ that is executed by different threads, and assume that $\mathcal{S}$ contains an addressing expression `A[f(i)]`, where `i` is the identifier of a thread executing the code. ACF constructs an algebraic expression for the difference between the symbolic value of the function `f` computed by two distinct threads. Then, by substituting actual constant thread identifiers into symbolic expressions, ACF can determine the memory-access stride between threads by solving the difference to a constant.

ACF replaces variable references with their dominating definitions wherever possible during the construction of symbolic expressions. ACF can perform this replacement without any additional considerations for potential performance impact of this replacement because ACF expressions are symbolic and are not actual Intermediate Representation (IR) of the program that will undergo transformation and code-generation.

The rest of this thesis builds on these core topics. Chapter 3 outlines several compiler transformations that aim at re-structuring OpenMP parallelism constructs to better suit the GPU architecture. Chapter 4 posits a static analysis framework for detecting memory access patterns of GPU-bound parallel loops and using it to power safety and profitability analyses guiding performance-improving loop transformations. Chapter 5 re-approaches the topic of accelerator offloading in high-level parallel programming models, detailing a hybrid-analysis decision framework for selecting a target processing element out of a plurality present in a heterogeneous compute node. Chapter 6 records the state-of-the-art related work in the relevant areas of study. Chapter 7 concludes this thesis.

Chapter 3

OpenMP Code Offloading: Splitting GPU Kernels, Pipelining Communication and Computation, and Selecting Better Grid Geometries

A natural way for an experienced OpenMP CPU programmer to write OpenMP GPU code is to offload to an accelerator sections of code that contain various parallelism-specifying constructs, that are often adjacent. However, this programming style generally leads to unnecessary overheads that are not apparent to programmers unfamiliar with GPU programming and mapping of high-level OpenMP code to GPUs. Experienced GPU Programmers will instead create a common device data environment and operate on data by invoking separate kernels for each required parallel operation. This technique results in more efficient code and often reduces the overall amount of host-device data transfer, as our work will demonstrate. The main goal of this investigation is to deliver better performance for the code written by experienced OpenMP programmers that are not necessarily GPU programming experts.

When an OpenMP `target` region contains a combination of parallel and serial work to be executed in a GPU, the compiler must map these computations to the GPU's native SIMT programming model. One approach is through a technique called warp specialization [3]. When specializing warps, the compiler

```
1  #pragma omp target teams map(to: B[:S]) map(tofrom: A[:S], C[:S])
     {
2    #pragma omp distribute parallel for
3    for () {
4      ... // Parallel work involving A and B
5    }
6    #pragma omp distribute parallel for
7    for () {
8      ... //Parallel work involving B and C
9    }
10 }
```

Figure 3.1: Example OpenMP GPU code with multiple parallel loops in a target region.

designates one warp as the master warp and all others as a pool of worker warps. In the Clang-YKT compiler OpenMP 4 implementation, the master warp is responsible both for executing serial code and for organization and synchronization of parallel sections [24] [23]. The synchronization between parallel and serial work is implemented through named warp barriers and an emulated stack in GPU global memory for the worker warps to access the master threads state. To handle activation and deactivation of worker warps, synchronization constructs are added to the target region.

Figure 3.1 is an example of OpenMP 4 code. The pragma in line `1` establishes that the following region of code will run on the default target accelerator — assumed to be a GPU in this work, and ensures that the data specified in `map` clauses is transferred to and from the GPU, respective to the (`to`,`from`) specifiers. The pragmas in lines `2` and `6` establish the associated work as parallel within the enclosing target region. There is an implicit synchronization point at the end of each parallel region.

Warp specialization introduces substantial overhead because the master and worker warps must synchronize execution when parallel region execution starts and finishes. Even when there is no sequential code between parallel regions, synchronization is required between the completion of one parallel region and the start of another.

The Clang-YKT compiler performs a transformation called *elision* that removes the warp specialization code, the master-thread stack emulation and the synchronization code, thus eliminating unnecessary overhead [24]. To be candidate for elision, a target region must contain only one parallel loop and this loop must not contain calls to the OpenMP runtime. A research question posed by our work is: what would be the performance effect of transforming an OpenMP 4 `target` region that contains multiple parallel regions, with or without serial code, into multiple target regions, each with a single parallel region. The goal is to enable the compiler to perform the elision transformation. Special care must be taken to avoid increasing the amount of data transfer between the host and device memory.

This chapter explores two additional transformation opportunities, both applicable to any OpenMP code where parallel loops are isolated into their own target regions. The first is the overlapping of data transfer and GPU kernel execution for multiple adjacent target regions. The target regions are wrapped in a common device data environment and through memory-use analysis, a compiler can determine which data is and is not needed until or after a certain point. The second opportunity is to overlap computation with data-transfer by pipelining the loop within a single-loop parallel region in a fashion similar to iterative modulo scheduling [49]. The loop iteration space can be divided into multiple tiles, each resulting in a separate kernel launch, execution of which happens asynchronously with the data transfer for the next tile.

Finally, this target region format allows for better selection of grid geometry tailored to the contained parallel loop. Grid geometry is the number of Cooperative Thread Arrays (CTAs), also known as thread blocks, and the number of threads per CTA that the GPU uses. Grid geometry strongly affects the overall occupancy of the GPU. Tailoring this selection to a specific parallel region can have a significant effect on the performance of that region. However, a single grid geometry must be selected for an entire target region. Therefore, multiple parallel regions in the same target region cannot have individually specialized geometry for each parallel region.

In the remainder of this chatper, Section 2.2.1 describes how kernel splitting enables the elision of runtime calls and barrier synchronization. Section 3.1

```
 1 #pragma omp target data map(to: B[:S]) map(tofrom: A[:S], C[:S]) {
 2   #pragma omp target teams distribute parallel for
 3   for () {
 4     ... // Parallel work involving A and B
 5   }
 6   #pragma omp target teams distribute parallel for
 7   for () {
 8     ... //Parallel work involving B and C
 9   }
10 }
```

Figure 3.2: Example OpenMP code following kernel splitting.

presents a sample code to demonstrate how kernel splitting is performed. Section 3.2 describes the implementation of asynchronous memory transfers and presents a study of their performance implications. Section 3.3 explains how these transfers can be used to establish a pipeline between computation and data transfers. Section 3.4 shows that custom grid geometry must take into consideration the potential saturation of memory bandwidth in the GPU. Section 3.5 presents the performance study that can be used to predict the potential benefits of the proposed transformations.

3.1 Fission of Multiple-Parallel-Region Target Regions

When a target region is separated into two target regions, as shown in Figure 3.2, each target region is then executed as a separate kernel on the GPU and therefore data transferred for the first region is no longer present for the second region to utilize as is the case when both exist in a single-target region. Figure 3.2 shows how the single-target region spanning lines `1-10` in Figure 3.1 can be split into two separate target regions, one spanning lines `2-5` and the other lines `6-9`. The parallel region directives (lines `2` and `6` of Figure 3.1) are combined with the target directives (lines `2` and `6` of Figure 3.2), transforming each parallel region into a stand-alone target construct. To avoid extra data transfers, the newly formed target regions are enclosed in a common device data environment containing all the implicit and explicit mappings of data from the original single-target region. Only the data items specified in the data environment persist in GPU global memory across multiple target regions. The motivation for this transformation to be performed by a compiler is further reinforced by the design of the `kernels` OpenACC construct [57]. `kernels` construct definition states:“The compiler will split the code in the kernels region into a sequence of accelerator kernels“, as deemed appropriate by the implementation. This design makes a strong argument for implementing the proposed transformation at the OpenMP level to further the efforts towards performance portability.

Furthermore, with a common device data environment, it is possible to overlap memory transfers with computation by analyzing when each data element is needed or produced. In our hand-implemented prototype for the transformation the OpenMP `target update` directive is used for these transfers, with the additional `nowait` clause added to allow for asynchronous memory transfers.

Safety measures must be taken when performing target fission, mainly to handle the presence of serial sections within the original single-target region. One concern to address is the possibility of variables being declared for the scope of the original single-target region. These variables reside in GPU memory and exist for the duration of the target region that is their scope, as a result the compiler must ensure that splitting does not interfere with any usages of them. One approach, if possible, is to move the variable declaration onto the CPU and map it to the common device data environment with an `alloc` map clause. Additional care must be taken to then mark such variables as `teams private`, to replicate the semantics of original code. Another approach is to limit the fission transformation such that all code from the declaration of the variable to its final usage resides within a single target region, though this can prevent elision.

A mitigating factor for this concern is that any such interfering declaration within the original single-target region scope must reside in a serial region at the target region scope. Variables declared inside parallel regions are assumed to be thread-local and expire when the parallel code block goes out of scope.

Another safety concern is that of serial code operating on data objects that are modified by previous parallel regions or are utilized by later parallel regions. The compiler must ensure that an updated variable is used by both the serial code and any later parallel regions on the GPU as would be the case with a single-target region wherein all code operates on the same GPU memory. One solution is to place serial code segments on the GPU in their own target regions. A drawback is paying the cost of additional kernel launch to execute serial code. An alternative approach is to execute the serial code on the CPU, with compiler analysis ensuring that any data object used in parallel regions are transferred to and from the device as needed for correctness. These transfers can become costly if they occur frequently, but in some cases run time can be improved significantly by executing serial code on the CPU.

Therefore the kernel splitting method should be applied with caution when the original single-target region has serial code or target region scoped local variables. Such scenarios did not appear in any of the benchmarks tested and likely do not represent a large portion of OpenMP code that can benefit from splitting.

3.2 Overlapping Data Transfer and Split Kernel Execution

Overlapping data transfer with computation can be an effective strategy to increase performance. Opportunities to benefit from asynchronous data transfers may arise from the splitting of a multi-parallel-region target into multiple single-parallel region targets. To enable the pipelining of data transfers and computation, the compiler must determine the first point of use of data and also when the computation of results is completed and the data is no longer used in the target. After such analysis, a schedule can be created for the pipelining with the overlapping effectively hiding the memory transfer time.

Figures 3.3 and 3.4 illustrate how this pipelining, enabled by asynchronous memory transfers, can reduce the overall execution time. In this example, if the runtime of the two kernels are long enough, this transformation results in

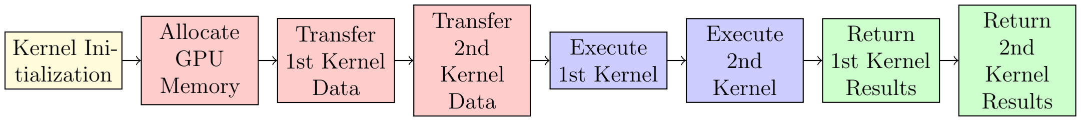


Figure 3.3: Two kernel GPU code structure before asynchronous memory transfer.

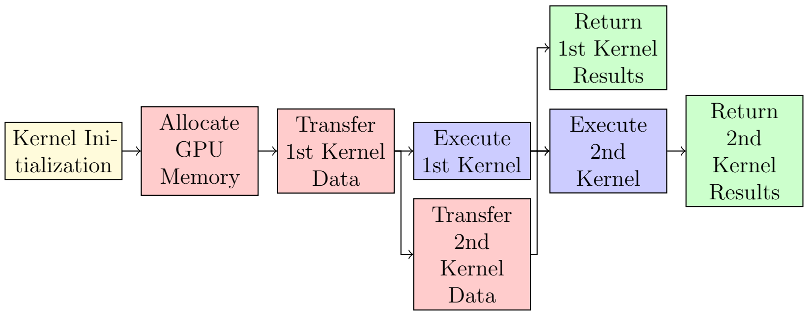


Figure 3.4: Two kernel GPU code structure with Asynchronous Memory Transfer.

the costs of the asynchronous memory transfers being entirely hidden.

Execution of asynchronous memory transfers and their synchronization with kernel execution can be specified manually by a programmer, using two OpenMP 4.5 clauses: `depend` and `nowait`. An OpenMP command with a `depend` clause with an `out` attribute must finish before any command with a `depend` clause with an `in` attribute with the same value. The `nowait` clause states that the specified OpenMP task can be run asynchronously with other tasks, thus allowing the update memory transfer to occur while a target region is executing. The combination of these clauses allows for the construction of GPU code that has asynchronous memory transfers to and from the GPU while also maintaining correct computation through clearly established task dependence relations by which these asynchronous transfers must finish.

Figure 3.5 is an example of split target region code with asynchronous memory transfers within a common device data environment. In this example the data element `C` is not needed until the target region at line `9`, thus its mapping in the `target data` region in line `2` is only to return to the host after all work finishes. The transfer to the GPU for `C` instead begins on line `3` where it is declared asynchronous by the `nowait` clause. With the pair of `depend` clauses in lines `3` and `9` ensuring the transfer must be completed before any computation on the target region in line `9` can begin. Furthermore the array `A`

```
1  int a;
2  #pragma omp targe data map(to: A[:S], B[:S]) map(from: C[:S]) {
3    #pragma omp target update to(C[:S]) depend(out: a) nowait
4    #pragma omp target teams distribute parallel for
5    for () {
6      ... // Parallel work involving A and B
7    }
8    #pragma omp target update from(A[:S]) nowait
9    #pragma omp target teams distribute parallel for depend(in: a)
10   for () {
11     ... //Parallel work involving B and C
12   }
13 }
```

Figure 3.5: The split OpenMP GPU code with asynchronous memory transfers.

can be transferred back to the host memory asynchronously as it is not used in the second target region. Thus the memory transfer of `A` back to the host is moved to line `8`, after the first target region computation and it is declared to be asynchronous.

As per vendor specification, asynchronous memory transfers require that the transferred data be page-locked i.e. *pinned* on the host. A pinned page cannot be swapped out to disk and enables DMA transfers via the memory controller, bypassing the CPU. To enable asynchronous transfers, the pinning must be done through the CUDA API to allocate/free pinned memory or to pin pre-allocated heap memory. The invocation of these API functions and the actual pinning of the memory introduce additional overheads but also leads to faster memory transfers. Memory capacity constraints of the target device are not affected by the transformed kernel. The amount of data required to be present on the device at a given time is reduced in the best case, and is left unaffected in the worst.

The experimental results shown in Figure 3.6 illustrate the cost of pinning memory using the CUDA API. As a point of comparison, the time taken to allocate non-pinned memory with a call to `malloc` and release with `free` is provided. `cudaHostAlloc` measurements include releasing memory with `cudaHostFree`. Finally, the `cudaHostRegister` results include the cost

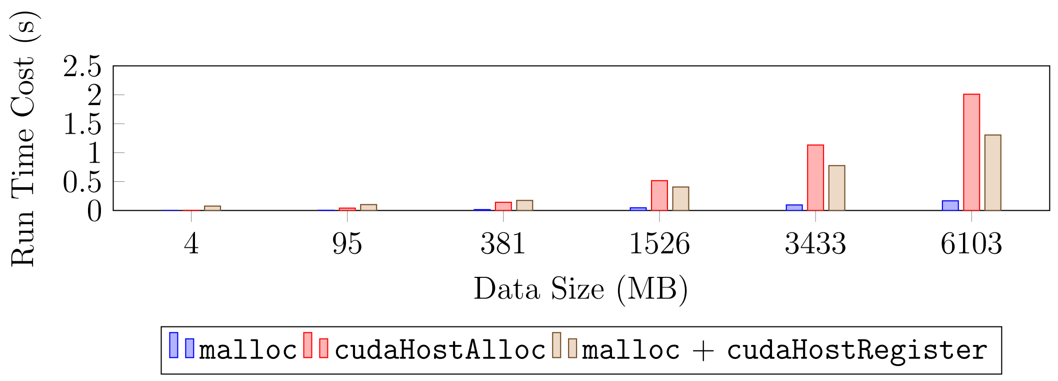


Figure 3.6: Run time cost of allocating and freeing memory with the three methods.

of allocating with `malloc`, pinning with `cudaHostRegister`, unpinning with `cudaHostFree` and releasing with `free`. Allocating and freeing non-pinned memory takes far less time compared to the same with pinned memory for both methods that the CUDA API provides at all data sizes. However the cost of the `cudaHostAlloc` method grows considerably with greater data sizes while the cost of the `cudaHostRegister` method grows less in comparison, overtaking the other in performance. This is, in part due to `cudaHostRegister` not zeroing the data it pins, unlike `cudaHostAlloc` which does.

We use the `cudaHostRegister` API to pin user-allocated memory in our experiments. The main trade-off to consider when implementing kernel asynchronous data transfers is to offset the overhead of pinning memory through faster transfers enabled by pinned memory and overlapping transfer with computation. Pinning memory also has the effect of reducing the overall memory available on the host for other processes, which can possibly stifle host computation. An important factor to consider when pinning memory is the operating system's default page size.We have found that pinning the same amount of memory was up to 10× faster on a POWER8 host with 64`KB` pages than on a x86 Haswell host with 4`KB` pages.

A synthetic experiment to illustrate the balancing of the costs and benefits of asynchronous memory transfer was designed with three simple GPU kernels ($\texttt{k}_1$, $\texttt{k}_2$, $\texttt{k}_3$) that execute within a shared data environment; $\texttt{k}_2$ modifies one data object from the CPU whose results must be returned, the object is not

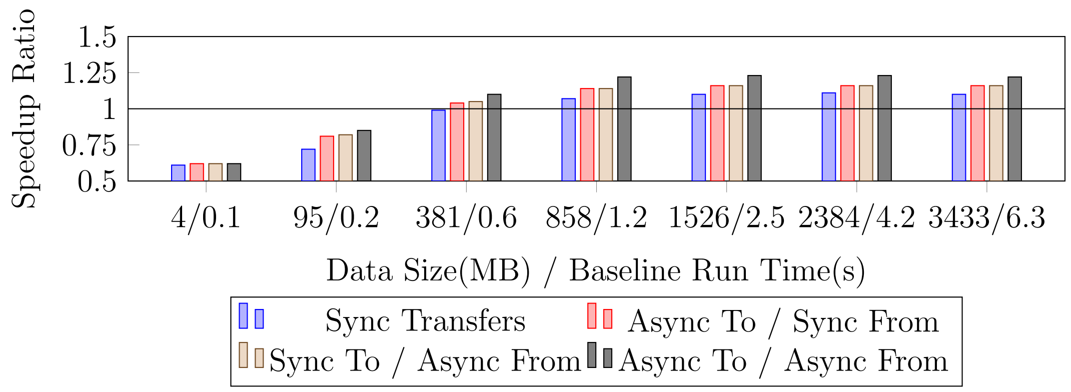


Figure 3.7: Speedup of the four versions pinning memory over the baseline version.

used by the first or third kernel. Thus, asynchronous transfer is possible both to transfer this data object to the GPU and back to the CPU. Furthermore, $\texttt{k}_1$ and $\texttt{k}_3$ both have enough computation to fully hide the asynchronous memory transfers. The experiment's results with a varying size of the object modified by $\texttt{k}_2$ are shown in Figure 3.7. The baseline version uses unpinned memory and synchronous transfers. Four versions using pinned memory were constructed for comparison: (1) sync transfers; (2) async to/sync from; (3) sync to/async from; (4) async to/async from. The run time measured includes the time needed to allocate and free memory. The graph outlines the speedup ratio in total execution time for each of the four pinned memory versions compared to the baseline version. The horizontal axis shows both the size of the object transferred and the baseline run time measured in seconds. The results show that as the size of the transferred object increases, the additional cost of pinning memory becomes less relevant. For larger objects, even though simply pinning the memory pages yields performance gains, asynchronous memory transfers produce additional benefits.

3.3 Pipelining Data Transfer and Parallel Loop Execution

A more ambitious code transformation that utilizes the faster transfer to/from pinned memory and asynchronous communication and computation consists of

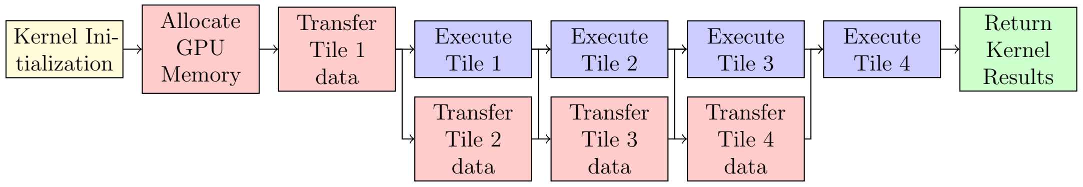


Figure 3.8: A GPU parallel regions structure after being broken up into 4 tiles.

breaking a singular parallel loop into multiple loops. Known as *tiling* in compiler literature, this transformation produces multiple sub-loops (tiles) which are then placed in separate target regions. After this transformation the data transfer required for the original loop may be split into several asynchronous data transfers for data elements required by the respective tiles. Ideally, each tile should use different, contiguous, large chunks of data. The goal is to overlap the transfer with computation. In the evaluation prototype OpenMP `depend` clauses are used to ensure that each data transfer is finished before the corresponding tile executes. Transmission of tile results back to the host can also be added to this pipeline. Pipelining can greatly improve the run-time performance of programs with large data transfers, when the execution time of the split loop is long enough to compensate for the overhead of setting up data transfers and pinning memory.

Figure 3.8 illustrates how the execution of a parallel region can be pipelined to overlap memory transfers with computation. The single parallel-loop GPU kernel is split into four tiles which allows the memory transfers required for the latter three tiles to be hidden underneath the previous tiles' execution with asynchronous transfers. Furthermore if the execution of the tiles are long enough to cover the runtime of the memory transfers then the total cost of the transfers may be as low as 1/4 of the original cost.

The Polybench benchmark `ATAX` is a good candidate to benefit from this transformation. The original benchmark's first parallel region, shown in Figure 3.9, has the majority of its runtime dependent on the memory transfer of the data object `A` to the GPU in line `1`.

Figure 3.10 shows the code after the loop is divided into four tiles and the transfer of `A` split into four OpenMP `target update` calls. The first

```
#pragma omp target teams distribute parallel for map(to: A[:NX*NY
   ], x[ :NY]) map(from: tmp[:NX]) {
  for(int i = 0; i < NX; i++) {
    tmp[i] = 0;
    for(int j = 0; j < NY; j++)
      tmp[i] = tmp[i] + A[i*NY+j] * x[j];
  }
}
```

Figure 3.9: First parallel region in `ATAX` before pipelining.

call in line 2 is not asynchronous as it must be done before the first tile execution starts. The remaining three transfers in line 6 are asynchronous and start before the preceding tile execution to overlap communication and computation. The `depend` clauses in the asynchronous transfers are needed to synchronize the end of the data transmission with the execution of the corresponding tile. Figure 3.10 shows a proof-of-concept manually implemented code change. A sufficiently-capable compiler should be able to apply a similar code transformation when equipped with memory access-pattern analysis to be able to separate tile data chunks, among other code safety analyses.

```
 1 int S[4];
 2 #pragma omp target update to(A[0:(NX/4)*NY])
 3 for(int s = 0; s < 4; s++)
 4 {
 5   if (s < 3)
 6     #pragma omp target update to(A[((s+1)*NX/4)*NY:((s+2)*NX/4)*NY
         ]) depend(out: S[s+1]) nowait
 7   #pragma omp target teams distribute parallel for depend(in: S[s
       ])
 8   for(int i = (s*NX/4); i < ((s+1)*NX/4); i++) {
 9     tmp[i] = 0;
10     for(int j = 0; j < NY; j++)
11       tmp[i] = tmp[i] + A[i*NY+j] * x[j];
12   }
13 }
```

Figure 3.10: `ATAX` region after being broken up into four tiles for pipelining.

3.4 Custom Grid Geometry

A grid geometry defines the number of CTAs and the number of threads per CTA assigned to execute a GPU kernel. A typical GPU has a number of Streaming Multiprocessor (SM) cores that can each issue instructions for two groups of 32 threads (warps) in each cycle. An SM can maintain the state of thousands of threads in-flight, and thus can context switch execution from a warp waiting on data accesses to other warps in order to hide memory-access latency.

Each SM has a fixed-size register file, giving each CTA a register budget. At any given time the number of CTAs that can be scheduled is limited by the size of the register file. Similarly, each SM has a fixed amount of shared memory which is shared by all CTAs running on the SM. Thus, the number of CTAs simultaneously executing on an SM is also constrained by the individual CTA's shared memory use. Additional CTAs that cannot be scheduled due to these and other hardware resource limitations are queued for later execution. GPU occupancy is the percentage of available GPU threads that are used by a given kernel.

Some parallel regions with relatively low parallelism perform better when not using all available threads. A compiler can analyze parallel loops in a `target` region to select the most performant grid geometry. However, a single grid geometry has to be selected for an entire `target` region leading to a compromise that performs relatively well for all the loop nests in the region. Grid geometry specialized to each individual parallel loop, made possible by `target` region fission, can lead to significant performance improvements.

Lloyd et al. propose a compiler heuristic, based on static analysis and runtime loop tripcount data, for the selection of a grid geometry calculated by the amount of parallelism in each loop nest [37]. The heuristic takes into account the usage of registers and shared memory for each thread and CTA as it seeks to maximize the GPU occupancy. However, maximizing occupancy can often lead to far worse performance because it leads to saturation of other hardware resource, such as the memory subsystem in heavily memory-bound

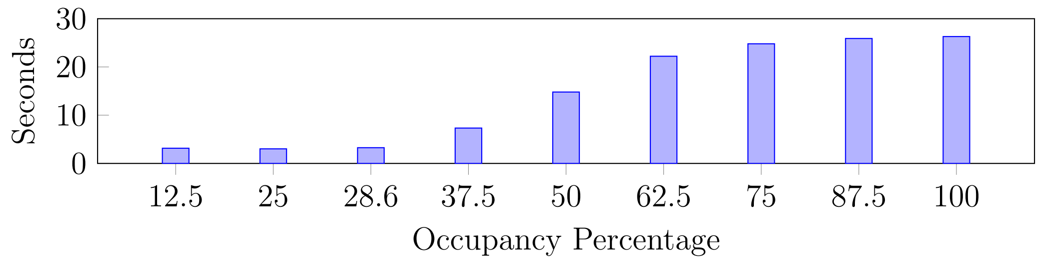


Figure 3.11: Runtime results by occupancy of `SYRK` at tripcount 4000.

codes. An example of this effect occurs in the `SYRK` benchmark shown in Figure 3.11. At a tripcount of 4000 the best performance is achieved around 25% occupancy which is close to the Clang-YKT default of roughly 28.6% (128 CTAs on this GPU). For this case the heuristic proposed by Lloyd makes a poor choice of geometry because in seeking to maximize occupancy it does not consider memory-bandwidth saturation. Maximum occupancy produces a Unified Cache throughput of 19.742 GB/s compared to a throughput of 183.001 GB/s at the optimal occupancy of 25%; moreover, the observed Global Load Throughput of 751.8 GB/s at optimal occupancy versus 81.5 GB/s at maximal, and the respective Global Store Throughput is 91.5 and 9.9 GB/s. These metrics support the intuition that memory bus saturation can severely limit performance at high occupancy.

This exception to the grid geometry formula led to the formulation of an improved grid-geometry selection strategy for the cases where the optimal occupancy is lower than the maximum. These cases fall into the broad category of parallel regions with a high amount of parallelism exposed by the program (high parallel-loop tripcounts) and result from memory-bandwidth saturation due to a large number of memory requests. The results of this performance study allows for the classification of these cases of massively parallel memory-bound kernels into two subcategories:

Uncoalesced Kernels are highly memory-bound due to uncoalesced memory accesses in large tripcount parallel loops. Uncoalesced memory accesses being loads and stores to global memory where data locations accessed by adjacent threads in a warp are not grouped together closely enough, hence the warp must perform several memory accesses to satisfy all the threads in a warp. This subcategory includes the benchmarks `SYRK` with tripcount of 1000 or higher and `COVAR` with tripcounts of 12000 or higher. `SYRK` falls into this subcategory due to the two high tripcount outer loops of its longest running parallel region being collapsed for high parallelism and an innermost loop containing an uncoalesced memory access which is performed sequentially by each thread. `COVAR` has a similar structure except without a collapse of the two outer loops and two uncoalesced memory accesses instead of one inside the inner loop. A close examination of the execution of the `SYRK` benchmark in the Nvidia Visual Profiler, reveals that the best performance is observed when the ratio between attempted memory transaction count and the memory throughput is the lowest — when the most data is transferred with the fewest requests. The grid geometry affects this ratio because more warps generate more requests when memory accesses are not coalesced.

The `SYRK` performance study shown in Figure 3.11 indicates that there is an opportunity to improve the grid-geometry selection by taking into consideration memory-bandwidth saturation. In a supplementary performance study we altered the ratio of requests/memory throughput in `SYRK` by adding and removing dummy uncoalesced memory accesses. This study yielded a pattern of optimal occupancy halving roughly when the number of uncoalesced memory accesses double. This insight can be used to predict the optimal occupancy for a parallel region. To analyze this pattern further a synthetic experiment was designed in which a more generalized program similar to `SYRK` was created consisting of a simple summation of the rows of k different $N \times N$ matrices to produce a single matrix. The summation statement is performed within a triple-nested loop with each tripcount being 5000 and the summation involves exclusively uncoalesced memory accesses (row-major matrix accesses). The experiment was then performed with different numbers of uncoalesced memory

Number of Accesses	1	2	3	4	5	6	7	8	9	10
Optimal Occupancy	25%	12.5%	6.3%	6.3%	6.3%	6.3%	6.3%	4.0%	4.0%	3.1%

Table 3.1: Optimal occupancy for a massively parallel memory-bound kernel at varying numbers of uncoalesced memory accesses with tripcount 5000.

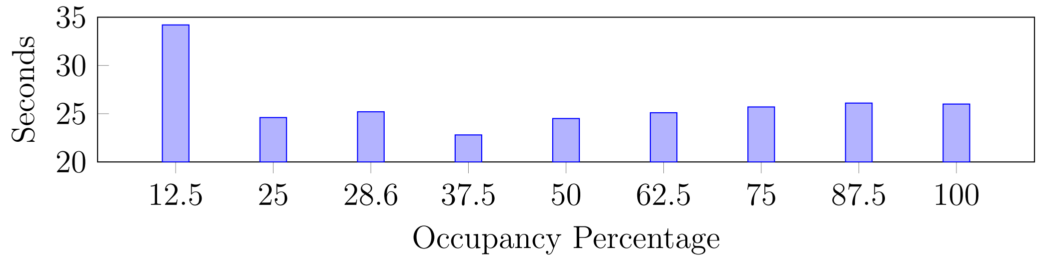


Figure 3.12: Runtime results by occupancy of `FDTD-2D` at tripcount 15000.

accesses to find the optimal occupancy for each. The results of the experiment in Table 3.1 show the similar optimal occupancy pattern that was found in the study of `SYRK`, indicating a general pattern. This study and experiment indicates that the heuristic for grid-geometry selection introduced by Lloyd et al should be augmented to account for memory-request saturation [37].

Coalesced Kernels have high memory utilization because of parallel loops with very large tripcounts and several memory accesses. Coalesced memory accesses are the opposite of uncoalesced and require only one access to bring over all data required by a warp of threads. This category includes the benchmarks `FDTD-2D` and `LUD` at high tripcounts. Lower occupancy results in better performance but the effect is less significant as shown in the results for the experiment study of `FDTD-2D` in Figure 3.12. This category should also be taken into consideration in an augmented version of the grid-geometry-selection heuristic.

3.5 Estimating Potential Benefits of Transformations

The goal of this experimental evaluation is to estimate the potential performance benefits of the proposed transformations to inform a design-team's decision to include them in a compiler. The results in this section are based on manually-implemented modifications to programs in the Polybench and Rodinia benchmark suites [11] [8]. Both suites have an initial OpenMP 4.0 implementation. Before performing the experiments, we modified some programs in both suites to fully utilize the GPU parallelism hierarchy with `teams` and `distribute` constructs. This experimental study uses benchmarks that contain parallel regions where the three transformations described in this chapter can be applied. SPEC ACCEL benchmarks, while available to us for experimentation, contain few to none such cases. Therefore, they do not make a good case for the transformation described in this work due to their already-extensive usage of `target data` data-sharing environemnts.

All performance results reported are the average of ten runs of the program under the same conditions. Measurement variances were monitored and stayed below 1% of the average and are not reported. Two exceptions are in the execution of `SYRK` and `COVAR` that saw up to 5% variance from the average because of the effects of memory saturation. Correctness of every transformation was verified using the benchmarks' output verification mechanisms.

This experimental study uses an x86 host equipped with an Intel i7-4770 processor, 32 GiB of RAM and an NVIDIA Titan X Pascal GPU with 28 SMs and 12 GiB of on-board memory that is attached via the PCIe bus. The clock rate is locked at 80% of the nominal clock rate for the GPU to prevent variance in performance due to frequency scaling [1]. Additional experiments are performed using an IBM POWER8 (8335-GTB) host with an Nvidia P100 GPU with 60 SMs that is attached via NVLINK.

3.5.1 Combining Kernel Splitting with Elision Improves Performance

The effect of the transformation on performance is studied on `2MM`, `3MM`, `FDTD-2D`, `SYRK`, `COVAR`, `ATAX`, `MVT` and `BICG` applications from the Polybench benchmark suite and `SRAD` and `LUD` from the Rodinia benchmark suite. All benchmarks chosen can be logically written with a singular target region by a naive GPU

[1]Dynamic frequency scaling makes achieving consitent, reproducible results very challenging due to high variance and increased effects of device warm-up.

Version	Kernel Splitting	Elision	Custom Grid Geometry	Asynchronous Memory Transfer
Baseline				
K	✓			
KE	✓	✓		
KG	✓		✓	
KEG	✓	✓	✓	
KA	✓			✓
KEGA	✓	✓	✓	✓

Table 3.2: The experimental evaluation versions for the splitting method.

OpenMP programmer. The experimental evaluation of the kernel-splitting technique includes seven different versions of each benchmark outlined in Table 3.2. Custom grid geometry was calculated using the heuristic by Lloyd et al. with the additional pattern for massively parallel memory-bound uncoalesced kernels described in Section 3.4 utilized for relevant cases [37].

Figure 3.13 displays the speedup over the baseline for each benchmark and each version shown in Table 3.2. Asynchronous transfer is not applicable (N/A) to the `SRAD`, `FDTD-2D` and `LUD` benchmarks as they all lack memory transfers that could be performed asynchronously. In the baseline, serial code is executed between any two parallel regions and the state of the master thread is propagated to all worker threads. Kernel splitting removes the serial code and workers' update. `LUD` has few worker threads because of its low level of parallelism, thus there is little benefit to the elimination of worker updating and the cost of launching a second kernel makes `LUD` slower after splitting (version K). `LUD`'s target region is executed within a loop, which amplifies the cost of the extra kernel launch. In contrast, `FDTD-2D` and `SRAD` have far higher levels of parallelism which leads to more expensive workers' state update. Thus they benefit the most from kernel splitting.

Benefits from adding elision to splitting (version KE) vary, with `2MM`, `3MM` and `SYRK` performing poorly because the runtime's default strategy selects an inefficient grid geometry. The removal of the warp specialization and sequential code overhead makes the memory bus saturation issue more relevant leading to

Benchmark	Base Time	K	KE	KG	KEG	KA	KEGA
2MM	36.6s	1.00	0.85	0.94	1.22	1.00	1.22
3MM	54.8s	1.00	0.85	0.94	1.22	1.00	1.22
FDTD-2D	11.8s	1.03	1.23	0.97	1.37	N/A	N/A
SYRK	40.9s	1.00	0.92	1.01	1.04	1.01	1.04
COVAR	54.9s	1.00	1.04	1.02	1.04	1.00	1.04
ATAX	0.16s	1.01	1.03	1.02	1.03	0.66	0.67
MVT	0.16s	1.01	1.00	1.02	1.02	0.66	0.66
BICG	0.16s	1.00	1.01	1.01	1.01	0.66	0.66
SRAD	8.90s	1.04	1.45	1.18	1.48	N/A	N/A
LUD	38.1s	0.91	1.57	0.92	1.57	N/A	N/A

Figure 3.13: The speedup ratio over the baseline for each experiment evaluation of the applicable Polybench and Rodinia benchmarks run at a tripcount set to 9600. `SRAD` executes on a 512by512 image with the encompassing iteration loop performed 9600 times. `LUD` operates on a 9600by9600 matrix.

the lower performance. In `SYRK` the main issue is that the default occupancy is too high. In `2MM` and `3MM` the number of threads is too low to exploit all available parallelism. `FDTD-2D`, `SRAD`, and `LUD` benefit greatly from elision because they contain a large number of kernel calls, accumulating the reduction in overhead of the elided kernels over time. Moreover, the amount of parallelism and the compute-bound nature of the kernels in these benchmarks suit the compiler's default grid geometry selection strategy.

Asynchronous memory transfer (version KA) by itself produces either negligible benefits or performance degradation. The degradation for `ATAX`, `MVT` and `BICG` results from the small size of data objects making the cost of pinning the data for transfer far greater then any hidden transfer cost and the short length of the benchmarks emphasizes this.

In general, significant performance improvements are achieved by the kernel-splitting technique combined with elision for the given benchmarks. Furthermore, any poor performance can be mitigated by additional procedures such as tuning grid geometry that are only available once the splitting technique is applied.

3.5.2 Elision Amplifies Benefits of Custom Grid Geometry

`SYRK` and `COVAR` are the benchmarks most affected by the grid-geometry selection. Both are highly memory-bound because they contain frequently executed uncoalesced memory accesses (`SYRK` has one, and `COVAR` has two) and as a result they both have lower than maximum optimal occupancies that produce large performance improvements. `COVAR` only has a lower than maximum optimal occupancy at higher tripcounts as it lacks the high parallelism of `SYRK`. These benchmarks' optimal occupancies decrease as the number of memory accesses rise with higher tripcounts, the optimal percentages following the optimal-occupancy trend outlined in Table 3.1.

Further experimental evaluation of `SYRK` and `COVAR` at multiple tripcounts for both a base unsplit version and a KEG version illustrates the effects of varying the grid geometry. The optimal occupancy, determined by the grid geometry, changes with the amount of parallelism for both benchmarks. For `SYRK` the optimal occupancy is 25% at lower tripcounts and 18.75% at higher tripcounts, while for `COVAR` the optimal is the default heuristic presented by Lloyd et al. that has the occupancy slowly grow towards the maximum for lower tripcounts when parallelism is low, with higher tripcounts having an optimal occupancy of 12.5% [37]. To illustrate this shift of optimal occupancy the experimental results shown in Figure 3.14 present the speedups of the two benchmarks' KEG version over the baseline for both of their optimal occupancies. The large improvement for `SYRK` at tripcount 3000 matches a similar effect in other programs with collapsed parallel loops that is caused by a sufficiently high parallelism. This performance is due to a combination of a GPU code that was simplified by elision and low impact of memory bus saturation because of still relatively low parallelism.

`2MM`, `3MM` and `FDTD-2D` present slight performance degradation when only custom grid geometry is applied (version KG) as with all three benchmarks the custom grid geometry is set to achieve full occupancy of the GPU SM's. With elision this is optimal but without elision additional warps are added on top of

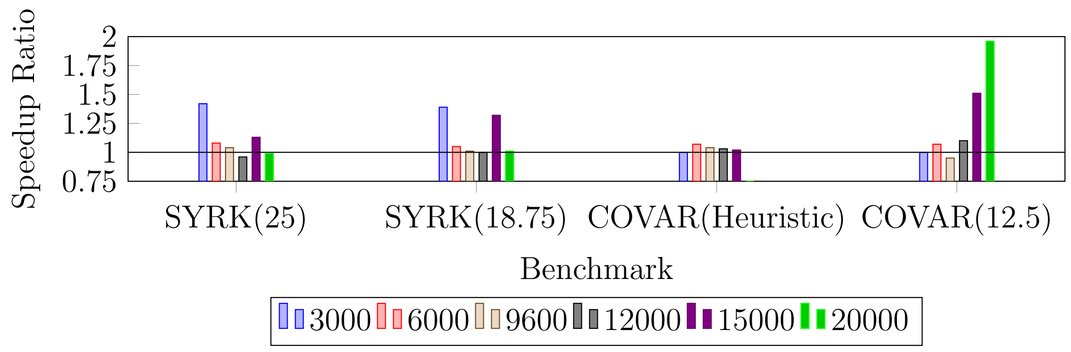


Figure 3.14: Speedup over the baseline for the KEG version of the two benchmarks at their two improved occupancies with varying tripcounts. Heuristic refers to that by Lloyd et al. [37]

the full occupancy for the master warps in each CTA. As a result the executed kernels request more warps then can be active on the GPU concurrently, thus additional scheduling of the warps is performed by the GPU to ensure all warps execute. This scheduling causes overhead that result in worse performance for the three benchmarks compared to when only kernel-splitting is applied. In comparison significant performance improvements for `SRAD` with version KG come from the high amount of computation compared to memory accesses in the program which take advantage of increased GPU occupancy from the heuristic.

The improvements brought by custom grid geometry (version KG) are amplified when combined with elision (version KEG) because the simplified execution for elided code better utilizes an optimized number of CTAs in terms of memory utilization and computation ability. Thus, the version KEG produces the best performance through this amplification of the benefits of elision and custom grid geometry.

Finally asynchronous transfers do not interact with grid geometry in any meaningful way, as such the KEGA results are only presented for completeness.

3.5.3 Pipelining Improves Performance for High Trip Counts

The pipelining transformation requires that a parallel region be broken into sub-loops that process separate data chunks of sufficiently large size to justify

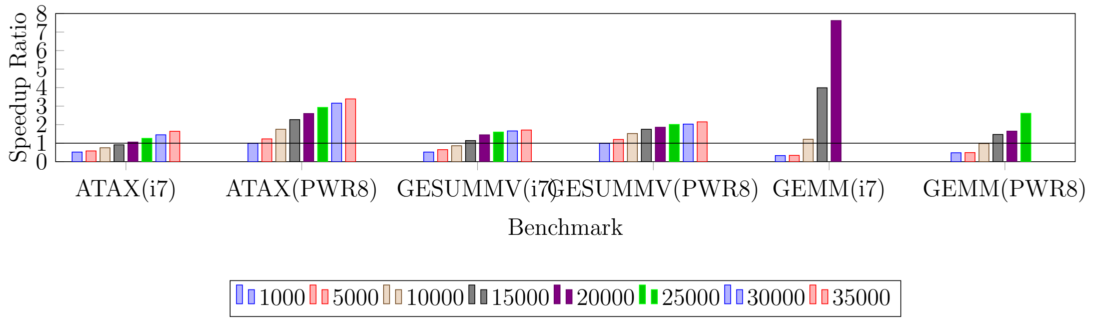


Figure 3.15: Speedup over the baseline with the kernel pipelining method on applicable Polybench benchmarks at varying tripcounts. GEMM is missing sizes due to time constraints.

the pipeline. Thus only the Polybench benchmarks `ATAX`, `GESUMMV` and `GEMM` are suitable for pipelining. Memory transfers to/from the GPU are dominant for the execution time for `ATAX` and `GESUMMV` resulting in significant improvements from pipelining. These improvements increase with the number of iterations as the additional computation amortizes the cost of pinning memory pages. Pipelining transfers plays a minor role in `GEMM` because kernel execution is dominant, instead the restructuring of computation caused by splitting the kernel in four improves performance. Each of the four resulting kernels have a quarter of the parallelism of the original kernel and thus a higher number of loop iterations can be executed before memory bandwidth saturation requires reduction in occupancy. A similar effect occurs for `ATAX` and `GESUMMV` but due to the dominant memory transfers the effect on performance is minimal.

For the experimental evaluation of kernel pipelining on the benchmarks `ATAX`, `GESUMMV` and `GEMM`, the baseline is a KE version of the benchmarks with non-pinned memory for all data objects. This baseline is compared to a version that has data transfer pipelined into four tiles with all pipelined data objects pinned. Both versions utilize the default Clang-YKT grid geometry formula. The evaluation is run on two machines: the Intel i7-4770 described above and a POWER8 host with a P100 GPU. The results in Figure 3.15 show significant improvements in performance with kernel pipelining for sufficiently large tripcounts. The POWER8 speedup is far larger because it uses 64KB pages compared to the 4KB pages in the Intel i7. The larger page size greatly reduces

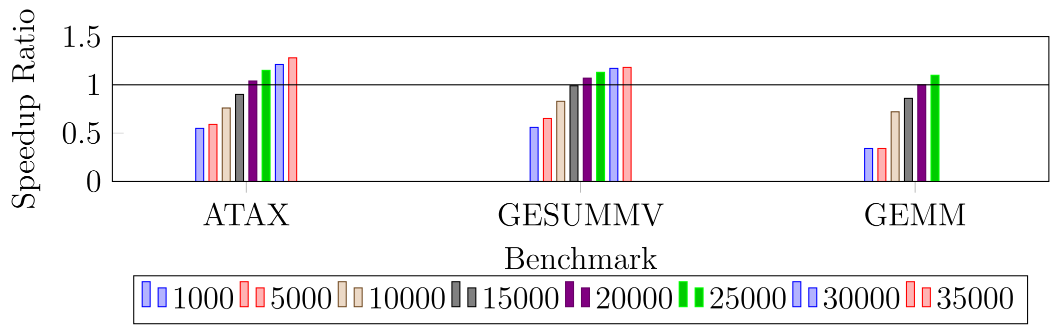


Figure 3.16: Speedup over the baseline for the kernel pipelining method on an Intel i7 machine at varying tripcounts with optimal occupancy applied to all kernels.

the cost of pinning memory. `GEMM` sees significant performance improvement because each individual tile processes a smaller chunk of data, thus allowing for higher utilization of the device without hitting the memory subsystem saturation performance barrier. In comparison, the benefits for `ATAX` and `GESUMMV` emerge from hiding transfer cost. At lower tripcounts the transformation degrades performance because there is not enough parallelism in the tiles to utilize as many GPU SMs, and the overhead of pinning memory and initializing additional kernels is not overcome.

In a second version of the experiment, on the x86 machine, the optimal occupancy at every tripcount for each kernel is applied to remove the influence of memory saturation. For `GESUMMV` and `GEMM` the optimal is 12.5% occupancy for the baseline and the Clang-YKT default formula for the pipelined version. While `ATAX` has an optimal occupancy of 18.75% for the baseline and the default formula for the pipelined version. Figure 3.16 shows this experiment's results with significantly better performance for the baseline that lowers the speedup from the pipelined version. However the pipelining still shows benefits due to pipelined memory transfers and later memory saturation at higher tripcounts of the benchmarks.

3.6 Concluding Remarks

This chapter puts forward the idea of splitting a singular OpenMP target region of GPU code with multiple parallel regions into multiple target regions each with a singular parallel region. The experimental evaluation using Polybench and Rodinia benchmarks indicates that there can be non-trivial performance gains from implementing this idea in future compilers targeting OpenMP `4.x`. Additionally the evaluation indicates that combining kernel splitting with synchronization elision and support for asynchronous memory transfers (with OpenMP 4.5) would lead to even more significant performance.

The study of grid geometry indicates that there is scope to improve existing grid-geometry selection strategy by considering the saturation of the GPU memory bandwidth due to uncoalesced memory accesses or to data-intensive parallel loop nests. This problem was recognized before with both dynamic runtime solutions and hardware changes proposed. However, the solution proposed here based on static analysis and compiler action is simpler, effective, and has lower overhead.

This chapter also studies the performance effect of pipelining memory transfers with kernel execution when there is sufficient data. Both kernel splitting and loop tiling can be used to enable pipelining. The results indicate that the performance gains can be significant especially in machines with larger page sizes such as the POWER architecture.

Chapter 4

Memory-access-aware safety and profitability analysis for transformation of accelerator-bound OpenMP loops

The paradigm shift toward heterogeneous platforms with accelerator devices adds an additional dimension to the portable code generation problem. The OpenMP 4.0 standard allows programmers to offload regions of code for execution on a coprocessor, making no assumptions about its memory or execution model [12]. Accelerator architectures operate on different assumptions of memory layout and locality, each requiring highly specialized program code.

Common pitfalls in GPU programming can be avoided when generating code from high-level languages through code analysis and transformation that would be difficult in lower-level programming models. For instance, *coalescing of memory accesses* leads to higher performance when threads in a warp access memory locations that map to few cache lines so that these accesses can be satisfied with fewer requests to the memory subsystem. A naive mapping of parallel loop nests to data-parallel code often results in non-coalesced accesses. A sufficiently capable optimizing compiler must be able to detect such code patterns and be able to reshape loop nests such that the resulting mapping exhibits better memory access characteristics.

This chapter introduces the IPDA that is able to capture OpenMP parallel loop access stride information and enable safety and profitability analyses that guide automatic interchange and collapse of loop nests. Loop-nest reshaping, informed by the results of IPDA, can yield dramatic performance improvements, demonstrated with an achieved speedup of up to 25.5× for a loop nest in the Polybench benchmark suite and up to 86.5× for a loop nest from the SPEC ACCEL suite. This new analysis builds on the ideas proposed in a novel static analysis framework called ACF [36]. ACF is unique in its ability to handle control-flow conditionals symbolically and statically determine access stride patterns in CUDA code. The IPDA framework introduces the ability to discover inter-loop-iteration symbolic differences statically. Stronger data-access analysis also enables the generation of efficient parallel code *without* requiring programmers to provide hints to the compiler. The performance study in this chapter demonstrates that removing `collapse` clauses from OpenMP 4.x programs can increase performance across diverse accelerator architectures if the compiler is capable of inferring the profitability of loop collapsing automatically.

This chapter also demonstrates the versatility of the analysis framework by building a loop-dependence test based on IPDA: the IPDA Test is introduced as an Data Dependence Graph (DDG) pruning algorithm that enables safety proofs on more loop nests than originally possible in the experimental compiler setup. This chapter describes the following contributions: i) IPDA – A static analysis framework for the computation of inter-iteration symbolic differences among expressions contained in loops. ii) A novel DDG pruning technique based on constructing inequality proofs over symbolic iteration-point algebraic difference equations. iii) A static analysis that identifies inter-thread memory access stride of addressing expressions contained in parallel OpenMP loops. iv) Safety and profitability analyses to guide loop collapse and interchange transformations on OpenMP parallel loops intended for GPU execution.

4.1 Loop Iteration Point Algebraic Differences

The Iteration Point Difference Analysis can symbolically calculate the difference in the expression's value across iterations of a loop, and is specially useful for analysis of induction-value-dependent addressing expressions. A compiler can use the results of IPDA to make decisions regarding both safety and profitability of classical loop transformations. IPDA can improve the generation of code that will execute either in the CPU or the GPU.

IPDA uses an ACF-like approach to compute the loop access stride of an addressing expression. While ACF relies strictly on the presence of a direct source of thread-dependent behaviour in the expression, IPDA uses the induction variables to examine iteration-point differences. In its evaluation prototype, ACF is applied to CUDA programs and is limited to stride-access analysis and branch-divergence detection on explicitly data-parallel programs. In that prototype, results are strictly used to advise CUDA programmers about potential opportunities for performance improvement.

Given a thread-dependent memory-addressing expression IPDA computes the inter-iteration access stride. For instance, consider the code snippet in Figure 4.1.

```
#define TSIZE 64
for (int i = 0; i < N; ++i) {
  int idx = 0;
  if (i < (TSIZE / 2))
    idx = TSIZE + i
  else
    idx = i
  B[idx] = foo()
}
```

Figure 4.1: Example loop to be analyzed by IPDA with a conditionally-defined indexing expression.

The symbolic expression computed for the address expression of the memory reference B[idx] in line 6 is:

$$\begin{aligned} IPD(\texttt{B[idx]}) &= (\texttt{[i]} < 32) \times (\texttt{[\&B]} + 8 \times (64 + \texttt{[i]})) + \\ &\quad (\texttt{[i]} >= 32) \times (\texttt{[\&B]} + 8 \times \texttt{[i]}) \end{aligned}$$

where references to idx were replaced with their definition in terms of i, which is the source of induction-variable-dependent behaviour. Symbolic propagation allows the analysis to compute the inter-iteration memory access stride by substituting constant parameters in place of induction variable identifiers and performing algebraic simplification:

$$\begin{aligned} & IPD_1(\texttt{B[idx]}) - IPD_0(\texttt{B[idx]}) = \\ & (1 < 32) \times (\texttt{[\&B]} + 8 \times 65) + (1 >= 32) \times (\texttt{[\&B]} + 8 \times 1) \\ & -(0 < 32) \times (\texttt{[\&B]} + 8 \times 64) + (0 >= 32) \times (\texttt{[\&B]} + 8 \times 0) \\ & = (\texttt{[\&B]} + 520) - (\texttt{[\&B]} + 512) \\ & = 8 \end{aligned}$$

Determining the inter-thread memory access stride to be 8 bytes. In this example constant iteration values of 0 and 1 are used. The actual analysis computes this difference for a sufficiently large number of iterations to arrive at a memory-access stride description. For example, when applying the analysis to calculate the inter-thread access stride of a GPU parallel loop, a number of iterations equal to the GPU thread-block size is tested.

A major strength of the IPDA framework is its reliance on symbolic value computation, which makes it highly independent of the transformation phase ordering: the analysis is likely to produce equally accurate results regardless of the current state of the loop code (e.g. before or after loop-invariant code motion). This has the effect of not only increasing the overall analysis accuracy but also its applicability at different stages of the compilation process.

Furthermore, IPDA's innovations make ideas first proposed in ACF relevant for non-data-parallel programs. Detection of induction-variable-dependent behaviour is also particularly useful for the analysis of parallel OpenMP loops because in such loops different iterations might be scheduled to be executed by different threads, affecting cache behaviour.

4.1.1 Focusing on Loop-Specific Analysis Demands

The transformation of loop nests often requires an analysis that can compute an induction-value-dependent difference between *distinct* addressing expressions computed at different iteration points of a loop nest. In contrast, the original design of ACF was intended only to compute the differences between the same expression as evaluated by multiple threads to detect divergent behaviour. For example, let E_s be the expression used to compute the address of the source of a loop-carried dependence relation that exists in the compiler's DDG and let E_t be the expression for the target of the same dependence. In many loops, E_s and E_t are similar enough that the difference between the ACF symbolic representations of E_t and E_s produces a simplified ΔE expression that yields useful information about potentially overlapping access ranges of the two statements. Symbolic differences between distinct expressions can often be used to determine that the dependence occurs under certain specific conditions or that it never actually occurs at runtime. Such information can often enable a multitude of compiler transformations previously prevented by a conservative or insufficiently capable safety analysis. Moreover, for the cases where the symbolic difference simplification framework does not provide information to increase the precision of the dependence relations, it does not affect the soundness of the results. In the case of a ΔE expression that could not be simplified or does not provide meaningful access range insights, the dependence is left as-is in the DDG.

4.2 Symbolic Representation

IPDA computes the algebraic difference of expression instances as accessed in different loop iterations. It models each access as a tree of symbolic values consisting of constants, statically unknown values, and operators. The symbolic representation for an expression is constructed in a way that lends easily into computing differences across iterations. For instance, whenever possible, a load is replaced directly with its reaching definition. IPDA iterates through instructions referencing the location of the load and avoids store-updates in the case of loops by means of dominance analysis.

An addressing expression of an arbitrary level of indirection is representable symbolically in a fashion similar to that of building an AST by traversing each index operation and creating offset addition operators for element accesses. Consider a data access expressed in C such as `A[4].x[2]` - the symbolic-expression tree is built by taking sums of each index operation (three in this case), each of which are offsets to the specified element.

Whenever there are multiple reaching defining expressions for a variable that is used in an IPDA address expression, a separate term is created for the IPDA expression for each reaching definition. Each term is multiplied by the set of predicates, extracted from the conditional statements along the path where the reaching definition lies, that must be true for that definition to reach the IPDA address expression. During the execution of the program, only one such path can be executed, and therefore only one such set of predicates can be true.

4.2.1 Algebraic Simplification

Two IPDA symbolic values for address expressions at different points in the iteration space are subtracted to determine if the two expressions could ever access the same memory location. IPDA canonicalizes the difference to a sum-of-products form and applies various algebraic simplification techniques to arrive at a simplified expression, such as simplifying constant sub-expressions, factoring constant multiplier variables and folding of special cases such multiplication by zero or one. In many practical cases, applying this step to the subtraction results in a tree where common expressions are cancelled leaving only a constant value that indicates the memory access stride.

- **Canonicalization** Symbolic trees are refactored into a canonical, sum-of-products form using a fixed point algorithm. Canonicalization is bottom-up process: the innermost expressions are distributed (e.g. $(v + w) * (x)$ becomes $vx + wx$) before successive operations.

- **Simplification** Expression rewriting is used to bring the canonicalized form into the most simplified form possible. For instance, constant values are eliminated whenever possible: $\frac{x}{1}$ and $x + 0$ both are re-written to x. Constants are folded, e.g. $(x * y + c1) + c2$ becomes $x * y + c$ where $c = c1 + c2$. Various included factoring techniques are also implemented as fixed-point processes.

4.2.2 Algebraic Difference Cancellation

The following are the steps taken to compute a difference between two address expressions, which are now represented as canonicalized symbolic-expression trees:

1. A new binary tree root is created, joining the two trees with a subtraction operator

2. The canonical sum-of-products form allows for a simple way to split each of the two subtraction operands into a series of constituting product summands in order to identify common expressions on both sides that can be cancelled.

3. Once the cancelled sub-expressions are removed from the summand lists, the remaining expressions are rebuilt back into a tree which is further simplified as outlined in Section 4.2.1.

If the resulting tree has been resolved to a constant value, IPDA returns the number of requests to global memory that an expression will incur.

4.3 Data Dependence Graph Pruning with Iteration Point Differences

The IPDA test constructs address value ranges that encompass the full scope of memory locations accessed by a potential dependence source and sink expressions across all dimensions of the iteration space. Consider a potential dependence in a given single loop where the dependence source is a store instruction to a memory location whose address is bounded by $E_{src} = [a, b]$. Similarly, the sink of the dependence is a load instruction whose address is in the range $E_{sink} = [c, d]$. If the ranges E_{src} and E_{sink} do not overlap, then IPDA determines that source and sink operations do not create a dependence; thus, the potential dependence is false and can be pruned from the DDG.

```
for (i=0; i<6; i++) {
  A[(i+8)*N] = A[i*N] + x;
  x = ...;
}
```

Figure 4.2: Example loop array access with a potential dependence.

To determine the value range of an address expression, IPDA propagates the value ranges of individual variables up the expression trees. Let $R1$ =$[a, b]$ and $R2$ =$[c, d]$ be two ranges. The following list outlines the various operations on ranges and how IPDA evaluates them to produce a resulting range.

- $R1 + R2 = [a + c, b + d]$
- $R1 - R2 = [a - d, b - c]$
- $R1 \times R2 = [min(min(ac, ad), min(bc, bd)), max(max(ac, ad), max(bc, bd))]$
- $R1 \div R2 = (a >= 0 \wedge b >= 0) \times [a/d, b/c] + (a < 0 \vee b < 0) \times [minInt, maxInt]$
- $R1 \% R2 = (a >= 0 \wedge c >= 0) \times [0, d] + (a < 0 \vee c < 0) \times [minInt, maxInt]$
- $R1\ and|or|xor\ R2 = [min(0, a), min(b, d)]$
- $R1 = | \neq | < | \leq | > | \geq R2 = [0, 1]$

We outline the procedure IPDA uses to verify dependencies in non-nested loops, then describe the more complex case of nested loops.

4.3.1 Single-Loop Dependence Checking

Let i and i' be two distinct values for the induction variable of a single-nested loop. The source and sink expressions of a potential dependence are formalized as functions of the loop induction variable. IPDA constructs symbolic, canonicalized expressions for $f(i)$ and $g(i')$ for the source and sink expressions, respectively. Functions f and g map the induction variable to an interval of memory addresses that may be accessed by the source and the sink. Therefore, IPDA difference $f(i) - g(i')$ is the interval difference of memory accesses by the source and the sink. If the difference is an empty interval, then distinct ranges of memory addresses are accessed by the source and by the sink and there is no real dependence. On the other hand, if the difference is resolved to a non-empty interval then a range overlap exists.

Consider the illustrative example in Figure 4.2, where `A` is an array of integers and `x` is a value whose value is reassigned in the body of the loop. There is a potential dependence whose source is the write of an element of `A` on the left-hand side of the first statement in the loop body and whose sink is the read of an element of `A` on the left-hand side of the same statement. IPDA begins by identifying the source and sink of a dependence, then constructs their symbolic representations as per Section 4.2. The symbolic difference $f(i) - g(i')$ is constructed and then factored such that the induction variables appear exclusively as a term of difference on each other: $(i' - i)$ (or $(i - i')$, it is immaterial which). The IPDA test applies the following algebraic steps:

$$\begin{aligned}
f(i) =& IPD(\texttt{A[(}i\texttt{+8)*N]}) = \texttt{[\&A]} + 4\times(i+8)\times\texttt{[N]}\\
=& \texttt{[\&A]} + (4i+32)\times\texttt{[N]}\\
=& \texttt{[\&A]} + 4i\texttt{[N]} + 32\texttt{[N]}\\
g(i') =& IPD(\texttt{A[}i'\texttt{*N]}) = \texttt{[\&A]} + 4i'\texttt{[N]}\\
f(i)-g(i') =& (\texttt{[\&A]} + 4i\texttt{[N]} + 32\texttt{[N]}) - (\texttt{[\&A]} + 4i'\texttt{[N]})\\
=& 4i\texttt{[N]} + 32\texttt{[N]} - 4i'\texttt{[N]}\\
=& 4\texttt{[N]}(i-i') + 32\texttt{[N]}\\
=& 4\texttt{[N]}\Delta i + 32\texttt{[N]}
\end{aligned}$$

With the simplified symbolic difference, the analysis verifies whether the difference can possibly equal to zero. The equation is rewritten into an inequality, as follows for our example: $4\texttt{[N]}\Delta i + 32\texttt{[N]} \neq 0$. Should this equation possibly have solutions, then a dependence exists because $f(i)$ overlaps with $g(i')$, ie. $f(i) - g(i') \neq 0$. IPDA rewrites the inequality by splitting the equation into a right operand (RO) and a left operand (LO) and isolating either into one side of the inequality. In particular, the three cases that are checked by splitting into the two operands are as follows:

$$\begin{aligned}
RO + LO \neq 0 &\implies RO \neq -LO\\
RO - LO \neq 0 &\implies RO \neq LO\\
RO \times LO \neq 0 &\implies RO \neq 0 \wedge RO \neq 0
\end{aligned}$$

In the example, the inequality $4\texttt{[N]}\Delta i + 32\texttt{[N]} \neq 0$ is evidently true iff $4\texttt{[N]}\Delta i \neq -32\texttt{[N]}$. Observe that if `N` equals 0, the inequality is proven to be false and a dependence exists. Suppose `N` is the size of a dimension of the array in question; then, a value range analysis would determine that the range of `N` is, conservatively $[1, maxint]$. This fact allows IPDA to further simplify the ineuqality into: $\Delta i \neq -8$. By definition of normalized loops, the induction variables initialize at 0 and exclusively increment. Should IPDA be able to statically determine the upper bounds of the loop, it can substitute in ranges for induction variables i and i'. Since the range is $[0, 5]$ for i in our example, then the value range of $(i - i')$ is either $[1, 5]$ or $[-5, -1]$, by construction of i'. The range of the right-hand-side expression scalar value is $[-8, -8]$. The dependence check is then reduced to verifying whether ranges $[1, 5]$ and $[-8, -8]$ or $[-5, -1]$ and $[-8, -8]$ overlap. This verification can be performed via a simple bounds check. Arbitrary precision integers are used in our implementation in order to handle various *maxint* range scenarios. In the case of intra-iteration dependencies, IPDA will not be concerned because loop-independent dependencies may be executed in parallel and are therefore not tested by the analysis.

4.3.2 Loop-Nest Dependence Checking

When dependence relation source and sink are contained in a loop nest, the IPDA Test must ensure that their access ranges do not overlap in *any* two points of the iteration space. To do so, IPDA repeatedly applies the access overlap test described in Section 4.3 to differences across all combinations of loops that contain the dependence.

IPDA collects all induction variables, along with their upper bounds whenever possible, to create a set $I = \{i_1, i_2, ..., i_n\}$. Another set $I' = (\{i'_1, i'_2, ..., i'_n\} | i'_k \neq i_k)$, holds the set of induction variables that represent arbitrary values for the induction variables in the n-dimensional iteration space. The subscript indicates the IV of a specific loop in the loop nest.

In the same manner that functions f and g are used in 4.3.1, let $f(i'_1, i'_2, ..., i'_n)$ and $g(i'_1, i'_2, ..., i'_n)$ be functions that map arbitrary iteration points to the location in memory being accessed by the source and sink, respectively. IPDA constructs symbolic, canonicalized expressions for f and g, joined by a symbolic subtraction operator which represents $f - g$. This total expression is factored such that each i_k and i'_k term appears *exclusively* as a difference $\Delta i_k = (i_k - i'_k)$.

In order to prove that f and g do not map to the same memory location in *any two iterations*, IPDA evalutes the access range overlap across *every* dimension in the iteration space as well as every combination of dimensions. Consider the n-degree loop in Figure 4.3. The iteration space is composed of

```
for (i1 = 0; i1 < N1; i1++) {
  for (i2 = 0; i2 < N2; i2++) {
    ...
      for (in = 0; in < Nn; in++)
        x = A[i1*N1 + ... + in*Nn]
        A[src] = x
    ...
  }
}
```

Figure 4.3: Example n-degree loop nest with a potential loop dependence.

n axes: $i_1, i_2, ...i_n$. A trivial example of data dependencies arises if the source expressions accesses memory at A[i_1*N_1+ ... +i_n*N_n-1]. In such a case, the next iteration in the innermost loop depends on its previous iteration, but the IV value for every other loop is constant. Consider also that a source could access memory *across* dimensions, unlike the previous example where the source and sink occur only in a single dimension (the innermost loop). Since dependencies may occur across any possible combination of dimensions in a nested parallel loop, IPDA performs range-overlap analysis (4.3) on each combination by *fixing* the Δi_k values to 0 for each loop at depth k which does not participate in the dependencies for the combination.

The power-set $\mathcal{P}(D)$ is the set of all subsets of D. For each set $S \in \mathcal{P}(D)$, each $\Delta i_k \in D \wedge \Delta i_k \notin S$ indicates a loop at depth k that is fixed ($\Delta i_k = 0$), so that all $\Delta i_k \in S$ identify the combination of loops in S that are part of the current iteration space being evaluated for memory-access overlap. IPDA fixes Δi_k values not in a given S by substituting a value of zero for the difference that appears as $(i_k - i'_k)$ in the symbolic expression. The resulting reduced difference expression is then evaluated with the range analysis overlap method as discussed in 4.3.1. If and only if, for every S, the analysis is able to prove via the range overlap analysis that the difference expression is *never* equal to zero, the dependence relation is pruned from the DDG.

4.3.3 Symbolic Differences of Control-Dependent Expressions Improve Dependence Testing

The IPDA Test's ability to incorporate conditionally-defined values into symbolic expressions, means to perform algebraic simplification on differences of such expressions, and ability to propagate variable definitions to their uses across control flow, distinguishes it from other symbolic analyses.

Many competing analyses require the dependence source and sink expressions to be defined in terms of the induction variables of their containing loops. However, this reliance is often broken by other compiler transformations, such as the ones that canonicalize the representation of loops. For instance, Figure 4.4 (b) shows the normalized version of a loop. After normalization the addressing expressions are no longer expressed in terms of the canonical loop induction variables. Some compilers may rely on expression re-materialization to obtain the expressions in terms of the canonical induction variables; however, a cost function may prevent the propagation of expressions into the body of a hot loop. The result is addressing expressions still expressed in terms of a mix of both original and canonical induction variables. Such mixed indexing expressions may stymie dependence analyses, as is the case in the example in Figure 4.4 (c) where propagating `CIVJ+1` and `CIVI+1` to the indexing expression, in place of `i` and `j` would hurt performance. IPDA's symbolic propagation of variable definitions to their references eliminates the problem.

Common loop dependence analysis algorithms have difficulty processing addressing expressions with non-constant induction-variable coefficients. Consider the loop nest in Figure 4.6 (a), and suppose `NI`, `NJ`, and `NK` are runtime parameters. The Greatest Common Divisor (GCD) Test, for example, checks dependences by verifying that the induction variable coefficients divide the constant factor of the respective Diophantine equation [63]. This test cannot be performed if one of its operands is an unknown runtime value. Similarly, the Banerjee Test is not able to compute the coefficient sums in order to evaluate the constraint condition inequality [2]. However, the dependence source and sink are often likely to contain the same induction-variable coefficients, and therefore the IPDA's difference calculation engine will factor and cancel them , leaving the resulting expression in terms of the induction variables and other constants.

```
for(j=1;j<NJ-1;++j){
  for(i=1;i<NI-1;++i){
    for(k=1;k<NK -1;++k){
      B[i*(NK*NJ)+j*NK+k] = foo(j,i,(CIVK+1));
    }
  }
}
```

a. Original Source Code

```
for(CIVJ=0;CIVJ<NJ-2;++CIVJ){
  j = CIVJ+1;
  for(CIVI=0;CIVI<NI-2;++CIVI){
    i = CIVI+1;
    for(CIVK=0; CIVK<NK-2;++CIVK){
      k = CIVK+1;
      B[i*(NK*NJ)+j*NK+k] = foo(j,i,(CIVK+1));
    }
  }
}
```

b. After Loop-Normalization

```
for(CIVJ=0;CIVJ<NJ-2;++CIVJ){
  j = CIVJ+1;
  for(CIVI=0;CIVI<NI-2;++CIVI){
    i = CIVI+1;
    for(CIVK=0;CIVK<NK-2;++CIVK){
      B[i*(NK*NJ)+j*NK+(CIVK+1)] = foo(j,i,(CIVK+1));
    }
  }
}

```

c. After Copy-Propagation

Figure 4.4: Example Loop Nest at various stages of compilation/optimization.

4.3.4 Prototype Implementation Demonstrates That the IPDA Test is Essential For Safety Analysis

The IPDA Test was implemented in the IBM XL compiler. As in LLVM, XL's loop dependence analysis is based almost entirely on the seminal work by Goff et. al. [18], and includes an assortment of exact and approximate tests such as the Lamport Test [30], GCD Test [63], Banerjee Test [2], and the Delta Test [18]. The IPDA Test is appended as an additional step in the DDG pruning pipeline. Implementation of the IPDA Test is highly specific to the compiler infrastructure it is built within. Thus, the only comparison to this first prototype is the original XL loop dependence analysis, which is a mature infrastructure from a major vendor. Upon implementation, the IPDA Test became an essential component of the safety analysis for the loop transformations described in Section 4.5, reducing the DDG further than the compiler's existing analyses and allowing transformations previously deemed unsafe. The techniques described here are applicable in a much wider variety of applications. We invite researchers and developers to explore those.

4.4 IPDA GPU Global Memory Coalescing Analysis on parallel OpenMP loops

Equipped with the ability to calculate inter-iteration access stride of addressing expressions, the compiler can infer the inter-thread access pattern of an addressing expression contained in a parallel loop. If the loop in question is destined for GPU offloading, then the inter-thread access pattern can be used to determine the coalescing characteristics of the memory access. The original ACF, as implemented in GPUCheck, utilized explicit sources of thread dependence to examine the degree of coalescing in a given memory access operation. ACF employed taint analysis where thread identifiers and their propagated uses were marked so that thread-dependent accesses to global memory could be analyzed and therefore inform the programmer of possible non-coalesced accesses. The IPDA test makes use of a similar approach but with regards to memory accesses within the body of parallel loops.

```
Constants ← { };
Unknowns ← 0;
**for** instruction I ∈ loop body **do**
    E ← symbolic(I);
    **for** thread t ∈ num(threads) **do**
        **if** diff.isConstant() **then**
            Constants.append(diff);
        **else**
            Unknowns += 1;
    **end**
**end**
**Function** *numRequests*(Constants, Unknown)
    Requests ← { };
    **for** c ∈ C **do**
        fit ← false;
        **for** r ∈ Requests **do**
            **if** $c \geq$ r.low && $c \leq$ r.high **then**
                fit ← true;
            **else if** $c \geq$ r.high - 256 && $c \leq$ r.high **then**
                r.low ← c;
                fit ← true;
            **else if** $c \leq$ r.low + 256 && $c \geq$ r.low **then**
                r.high ← c + 8;
                fit ← true;
        **end**
        **if** fit $\neq$ true **then**
            Requests.append( (low: c, high: c+8) );
        **end**
    **end**
    **return** (Requests.size, Requests.size + Unknown);
```

Figure 4.5: Computing the number of coalesced accesses

Only OpenMP loops that specify a `schedule` clause set to `static` with a compile-time constant chunk size parameter are analyzed. In practice, this restriction does not seriously limit the usefulness of the analysis because loops without a user-specified schedule are common and can be treated as having a schedule that the compiler deems beneficial. Taking the schedule chunk size into account, IPDA maps the induction variable of a parallel OpenMP loop to threads and employs a similar coalescing analysis as ACF. The analysis collects load and store instructions in the body of a given loop nest that are marked as tainted and outputs the number of memory requests that are required per warp to satisfy the memory access. Each tainted access instruction in a given loop is represented as a symbolic expression E. The difference $E_t - E_0$, as computed by IPDA for thread t, indicates the memory access stride for the instruction of interest. Algebraic simplification and difference cancellation techniques outlined in 4.2.1 are applied in an attempt to simplify inter-thread difference results to constant values. For instructions where IPDA is successfully able to compute constant-value access strides, the analysis employs the algorithm outlined in Figure 4.5 to greedily fit the solved results into a memory request.

The IPDA test creates a list holding potential constant *symbolic* differences and keeps a count of non-constant instances. The list `Requests` indicates the overall requests to global memory in the loop body, where each individual element depicts the range of accesses that compose a single coalesced access. An insertion into `Requests` creates an access of stride $sizeof(accesstype)$ bytes, with a limit of 256, the size of the cache line in the current generation of NVIDIA cards.

For symbolic differences that cannot be solved to a constant value, IPDA coalescing analysis conservatively assumes distinct requests. Non-constant values may arise in the occurrence of runtime-only known values in the symbolic difference computation, in which case no static analysis would be able to deduce the value at compile time.

4.5 Improving GPU Memory Access Patterns with Loop Transformations

```
#pragma omp target teams distribute parallel for
for (i = 0; i < NI; i++) {
  for (j = 0; j < NJ; j++) {
    C[i*NJ + j] *= BETA;
    for (k = 0; k < NK; ++k) {
      C[i*NJ + j] += ALPHA * A[i*NK + k]
                         * B[k*NJ + j];
    }
  }
}
```

a. Original benchmark source code

```
#pragma omp target teams distribute parallel for
for (c = 0; c < NI * NJ; c++) {
  i = c / NI;
  j = c % NI;
  C[i*NJ + j] *= BETA;
  for (k = 0; k < NK; ++k) {
    C[i*NJ + j] += ALPHA * A[i*NK + k]
                       * B[k*NJ + j];
  }
}
```

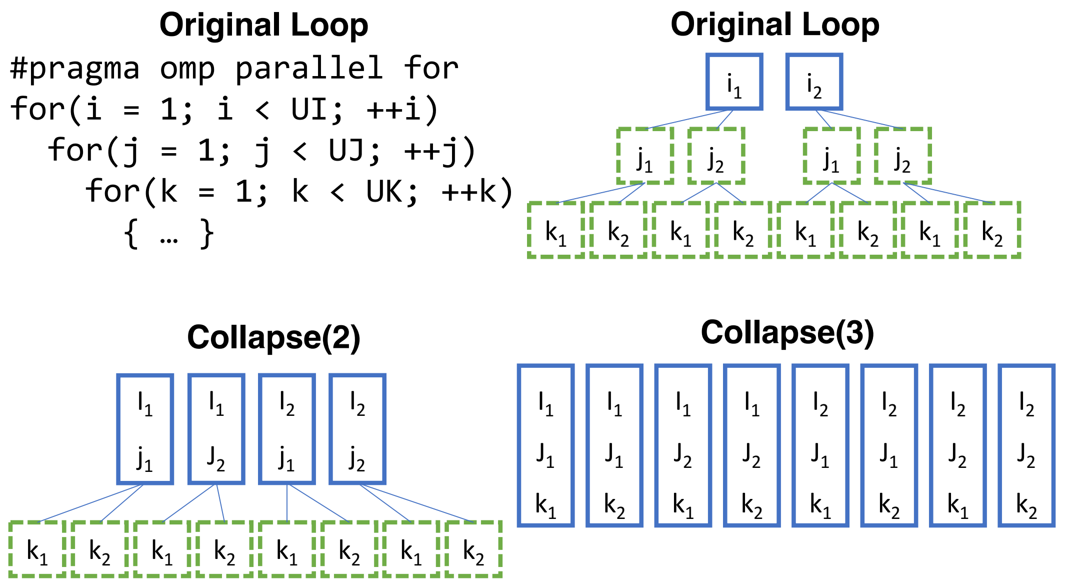
b. Collapsed `i-j` nest.

Figure 4.6: Example `target` region from GEMM benchmark

Naively translating parallel OpenMP loops directly into data-parallel code can lead to an inefficient kernel that poorly utilizes the GPU memory subsystem. Consider the OpenMP target region extracted from the GEMM benchmark from the Polybench suite shown in Figure 4.6 (a). The iteration space of the `i` loop is first divided into chunks in conformance with the `teams distribute` construct directive, and iterations of each chunk are then scheduled to run in parallel, as prescribed by the `parallel for` construct. Each thread executing an iteration of the `i` loop sequentially executes the `j, k` loop nest. In this example, for a given memory access, the inter-thread stride is the size of each array, which result in an inter-thread stride of 4096 bytes and none of the accesses can be coalesced.

Memory access patterns for a GPU kernel that uses high-dimensional data structures or otherwise non-trivial addressing expressions are often not obvious even to experienced developers and require expert knowledge of the compiler's code-generation scheme and mapping from loop-parallel to data-parallel code. A particular loop layout may also benefit one accelerator architecture over others, leading to loss of performance portability no matter which selection is made. IPDA's memory-access analyses enable and guide loop transformations that improve GPU memory utilization by increasing access coalescing.

4.5.1 Loop Collapse



```
#pragma omp parallel for
for(i = 1; i < UI; ++i)
  for(j = 1; j < UJ; ++j)
    for(k = 1; k < UK; ++k)
      { … }
```

Figure 4.7: Pictorial representation of a parallel loop nest's iteration space.
□ – Loop iterations comprising units of parallel work.
⬚ – Loop Iterations executed sequentially.

The OpenMP `collapse(n)` clause merges `n` nested loops into a single parallel iteration space. Collapsing parallel loops for execution on a GPU has two performance-sensitive effects: the number of parallel work items to be scheduled increases; and, the thread memory access pattern changes.

- **Increased parallelism**

  The total number of iterations of the collapsed parallel loop is equal to the product of the trip counts of all the loops in the collapsed nest, increasing the amount of parallelism available. Figure 4.7 shows an example parallel loop nest and the mapping of iterations to units of parallel work and sequential iterations.

- **Improved memory access pattern**

  Collapsing changes the inter-thread access stride because the outer loop induction variable no longer maps to the thread identifiers. If the sequential execution order of the collapsed loops is used to determine the order of the iterations in the collapsed iteration space, then the stride of the innermost collapsed loop becomes the inter-thread access stride.

Collapsing can be benefitial even when no `collapse` clause is present. For example, in the loop shown in Figure 4.6 (a) consecutive iterations of the `j` loop access adjacent elements of arrays `C` at line 4, `C` at line 6, and `B` at line 7 and have an inter-thread stride of 4096 bytes with no coalescing. After collapsing the `i-j` loop nest (Figure 4.6 (b)) all accesses are perfectly coalesced. Figure 4.8 illustrates how this collapse enables coalescing by changing the access to a two-dimensional array from row-major order to column-major order.

Loop Collapse Safety:

In the absence of a `collapse` clause, the compiler must prove that collapsing is safe for a loop nest of depth n with a parallel outermost loop. Such nest must satisfy the following conditions to be safely collapsed:

- It must be perfect: all statements must be inside the innermost loop.

- Iteration spaces of the loops in the nest cannot be affected by the values of the induction variables of the other loops in the nest. Loop boundaries for all loops must not change after entry into the loop nest.

- There must be no loop-carried dependencies among iterations of any of the loops in the nest.

The parallel clause in the outermost loop implies that the iterations of this loop are independent within the specified schedule chunk size. Therefore, the required dependence analysis must only check for dependencies across loops nested in the outermost loop, but not across the outermost loop itself. A parallel loop contained in a target region follows aliasing restrictions on data mapped into the device data environment. These restrictions often make dependence analysis feasible where it would not be for an identical loop not contained within a target region. For a given loop nest, all possible collapse depths are tested for safety. For the set of provably safe collapse depth levels, profitability analysis determines which, if any, should be performed by the compiler.

Loop Collapse Profitability:

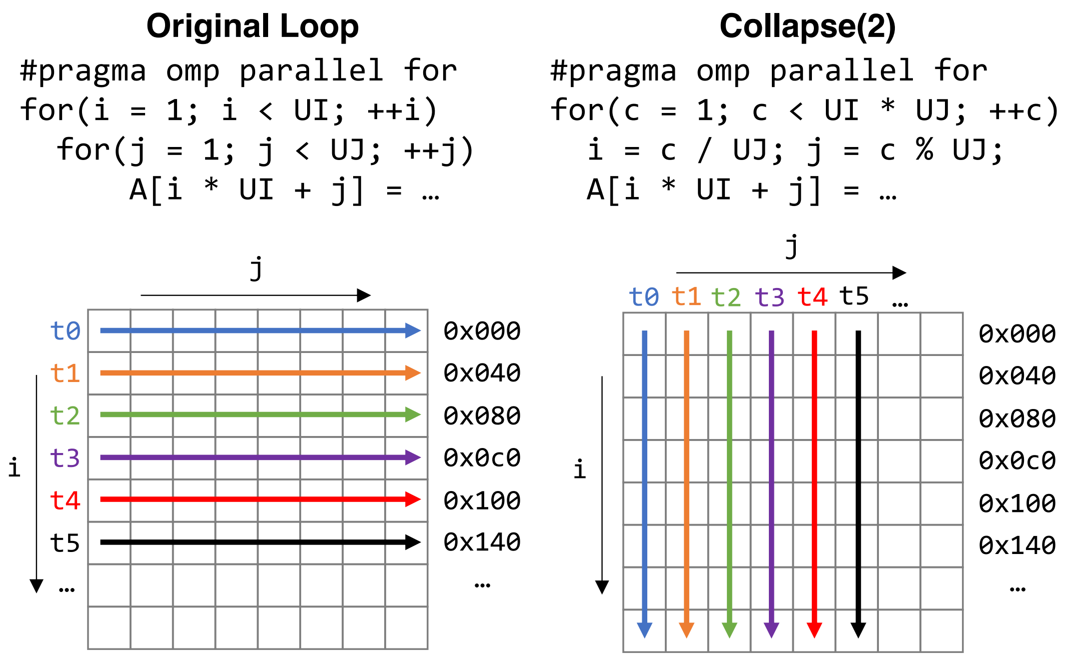
The main performance benefit of loop collapse stems from improved access patterns. Thus, the reduction in the number of memory requests executed per warp can be used to estimate the profitability of collapsing a loop nest. Using the IPDA framework, the profitability is computed as follows:

1. Compute the number of memory requests for every access in the original loop nest.

2. For each nest level κ, compute the number of memory requests for every access in the nest by rewriting the addressing expressions to emulate the effect of collapsing the nest to that level.

3. A collapse is *profitable* if, for any nest level, the total number of memory requests per warp in the kernel is reduced.

These steps are performed for all collapse depth levels considered safe and the most *profitable* level is chosen for actual code transformation.



```
#pragma omp parallel for
for(i = 1; i < UI; ++i)
  for(j = 1; j < UJ; ++j)
      A[i * UI + j] = …
```

```
#pragma omp parallel for
for(c = 1; c < UI * UJ; ++c)
  i = c / UJ; j = c % UJ;
  A[i * UI + j] = …
```

Figure 4.8: Pictorial representation of a parallel loop nest's memory access pattern.

4.5.2 Loop Interchange

Loop interchange is a classical loop transformation wherein the order of two iteration variables in a loop nest is exchanged. In loop nests of dimension higher than two, several interchanges may occur, for example, moving the innermost loop to the outermost position in a 3-dimensional nest.Loop interchange is typically performed to improve spatial locality and cache utilization of array accesses in loop nests. The aim of loop interchange here is to improve access coalescing by changing the mapping of loop induction variable differences to inter-thread memory access stride. For example, consider the loop nest excerpt shown in Figure 4.9. IPDA memory coalescing analysis finds that the resulting inter-thread stride leads to non-coalesced accesses. It also shows that loop `i`, when used as a source of thread-dependence, i.e. when iteration-point differences of the `i`-loop are treated as thread-difference values for addressing expressions, would result in perfectly coalesced loads and stores. However, the loop nest cannot be collapsed to a depth of three, as described previously, because it contains a loop-carried dependence across iterations of the `j` loop. An alternative method to achieve a mapping to data-parallel code that distributes individual iterations of the `i` loop to GPU threads, loops `i` and `j` can be first interchanged without affecting the semantics of the program. Post-interchange, the outermost 2-dimensional `k-i` loop nest can be collapsed, according to the collapse profitability analysis outlined above.

Loop Interchange Safety:

The proposed interchange applies to a loop nest where the outermost loop is labeled as parallel by a programmer, and consists of moving a loop from inside that nest to the outermost level. Thus, the two outermost loops of the transformed nest can be collapsed, as described in section 4.5.1. Finding loops within a given loop nest for which this transformation is legal requires finding loops within the nest that are independent. Moving an independent loop to the outermost level preserves loop-carried dependences of all other loops in a nest. Similar to the analysis performed for loop collapse, aliasing restrictions on data mapped to the target region data environment often result in a more precise dependence analysis. In both collapse and interchange safety analysis, the compiler uses the IPDA Test to further reduce the DDG. Every loop found to be independent in a parallel loop nest is considered a candidate for loop interchange.

Loop Interchange Profitability:

For all candidates, profitability is computed in a fashion identical to the profitability of collapsing the nest to depth of up to and including the loop in question. The profitability of loop interchange takes into account the subsequent collapse transformation, which is required to create a mapping from induction variable differences in addressing expressions to inter-thread stride that results in better coalescing characteristics. The most profitable of the candidate loops is selected for actual code transformation.

4.6 Evaluation

Coalescing of memory requests is a key performance consideration when writing or generating GPU code. It is increasingly difficult for developers to infer memory access characteristics of OpenMP GPU code as it gets translated into a data-parallel form. Moreover, explicitly committing the code to a specific access pattern that would suit a specific type of accelerator can hurt *performance portability*. Thus, high-level accelerator programming models make such considerations the prerogative of the compiler designer rather than the developers.

The analysis framework described in this chapter has been implemented in the IBM XL compiler. The efficacy of the analysis framework is evaluated in two ways. First, the potential performance impact of the two proposed loop transformations is demonstrated on a set of representative OpenMP 4.x programs. The second evaluation demonstrates that a compiler equipped with loop transformations informed by the IPDA analysis allows a higher degree of performance portability in OpenMP code. Generality and architecture-independence of OpenMP code can often be improved by removing developer-specified clauses intended as optimization prescriptions that commit generated code to specifically target GPU accelerators. The results of this evaluation demonstrate (1) that equipped with the IPDA analysis and loop transformation framework, the compiler is able to re-capture the performance impact of such clauses by automatically performing the required optimization when generating GPU code; and (2) that omitting performance-guiding clauses results in a performance improvement when targeting other accelerator architectures.

4.6.1 Informed Loop Reshaping Performance Impact

The Polybench [50] and SPEC ACCEL [26] OpenMP 4 benchmark suites are used to evaluate the efficacy of the coalescing-analysis-informed loop reshaping of OpenMP 4.x parallel loop nests. Execution times are reported for two experimental setup machines: an IBM POWER8 host with an Nvidia P100 GPU, and an IBM POWER9 host with an Nvidia V100 GPU accelerator. Table

4.1 shows speedup of benchmarks with IPDA-guided collapse and interchange transformations enabled in the compiler. The analysis detected transformation opportunities in three Polybench benchmarks: `MVT`, `2DCONV`, `3DCONV`, and one SPEC ACCEL benchmark: `557.pcsp`.

The matrix multiplication `GEMM` contains a single 3-deep parallel loop nest. Collapsing the nest to depth 2 was found by the analysis to have the effect of transforming a kernel with completely non-coalesceable accesses into a kernel with perfectly coalesced accesses. The kernel execution time improves by a factor of 25.5× in the P100 and 20.9× in the V100. The benchmark execution time improves by a factor of 3.18× in the P100 and by 8% in the V100 machine. The much lower overall benchmark improvement in the V100, in spite of the higher kernel improvement, is due to higher kernel launch overhead as explained below.

`2DCONV` and `3DCONV` convolution benchmarks contain a single parallel loop nest of depth 2 and 3, respectively. The original compiler failed to prove that the `3DCONV` loop nest is free of loop-carried dependences. The IPDA Test reduced the DDG further, ultimately proving the nest as independent and safe to collapse. Profitability analysis on the two parallel nests indicated that automatic collapse would result in turning both from non-coalesceable into completely coalesced GPU kernels. The transformed code for `2DCONV` improves kernel execution time by a factor of 14.75× and 6.37× and benchmark execution time improves by a factor of 1.65× and 5%, on the P100 and V100 machines, respectively. Transformed `3DCONV` code results in kernel execution time speedup of 19.54× and 18.3× and benchmark execution time speedup of 2.03× and 3% on the P100 and V100 machines, respectively.

The SPEC ACCEL benchmark suite consists of highly-tuned OpenMP code written in a way that maximizes GPU performance. Aggressive use of `collapse` clauses by the benchmark developers limits the opportunities available for automatically inferring the need to interchange or collapse loop nests. Still, our analysis identified one such opportunity. `557.pcsp`, a pentadiagonal software application, is an OpenMP port of a pentadiagonal solver software developed by the Center for Manycore Programming at Seoul National University, derived from an application developed by NAS. It contains > 60 OpenMP `target` regions, all made up of parallel loops. One parallel loop, performing the forward elimination operation according to the Thomas algorithm, was identified by the IPDA analysis to be a safe candidate for transformation.An excerpt from the 3-dimensional `k-j-i` parallel loop nest in question is shown in Figure 4.9. The coalescing analysis identifies the opportunity to reduce the number of memory requests per warp present in the kernel by interchanging loops `j` and `i`, and collapsing the resulting `k-i` nest. Observe that a `collapse(3)` transformation of the original code is not possible due to loop-carried dependences across iterations of the `j` loop. Post `j-i` interchange and `k-i` collapse, the `j` loop is executed sequentially by each thread, preserving the `j`-loop-carried dependence. The combination of transformations reshapes the resulting GPU kernel in a way that makes every memory access contained within fully-coalesced. The transformed version yields an improvement in kernel execution time of 86.5× on an Nvidia P100 and 111.1× on an Nvidia V100 accelerators. Across a run of the benchmark, the forward elimination kernel is invoked 401 times. The untransformed kernel's poor performance characteristics make it the biggest contributor to the overall benchmark execution time, of which it constitutes 41%. Applying the transformations described above to just 1 out of > 60 parallel loops present in the benchmark results in overall speedup of 3.38× in the P100 and 2.3× in the V100.

The dramatically lower improvement in benchmark execution time compared to kernel execution time in Polybench benchmarks stems from their small default input data sets. Thus, kernel execution time comprises only a small part of the total time with kernel initialization and data transfer taking up most of the benchmark execution time. Higher improvements should occur with larger input sizes. For example, the `GEMM` benchmark performance improvement begins to approach the kernel speedup as input size is increased, as can be seen in Table 4.1. As input size is increased, `CONV` benchmark speedups remain fixed, despite a massive improvement in kernel execution time because memory transfer time dominates the overall execution time.

The relative decrease in overall benchmark execution time on a V100 is due to an increased overhead of kernel launch when using a Volta GPU with CUDA 9. The CUDA runtime creates a unifying context at first kernel launch in order to maintain state for consecutive launches. The context creation operations takes up to 0.7 seconds versus up to 0.1 seconds for the machine with a Pascal GPU. We hypothesized that the difference is due to Volta's support for unified memory which would require allocation of pinned host memory. The impact of this overhead is most pronounced in single-kernel programs with short runtimes. To test this hypothesis, we created high-throughput versions of the Polybench programs in which the benchmark kernel is invoked 100 times in a single launch, on different data sets, amortizing the CUDA context creation overhead. Results are reported in the rightmost columns of Table 4.1. The high-throughput version of `GEMM` confirmed our intuition with overall benchmark speedup of 15.9× and 7.14× on the P100 and V100. Memory-transfer dominated `3DCONV` also demonstrated Volta improvement start to approach Pascal figures with speedups of 1.82× and 1.56×. Based on these experiments, we believe that in real-world computation-heavy code, the improvement in overall program performance on Volta is likely to scale more closely with kernel performance improvement.

No other Polybench or SPEC benchmarks contain loop nests that are legal to automatically collapse because they all contain loop-carried dependences. A large number of benchmarks contain parallel loop nests collapsed by the programmer by specifying a `collapse` clause on the loop construct directive. In benchmarks where no safe/profitable opportunities were found, performance remained unchanged; as such, their results are not presented in Table 4.1. Benchmarks that do not contain `collapse` clauses and benchmarks that could not be compiled with the current compiler versions are not presented in Table 4.2. `557.pcsp` currently crashes when compiled with the ICC.

4.6.2 Code Portability Impact

The OpenMP `collapse` clause is prescriptive and requires the compiler to generate a specific code structure. Its goal is to exploit performance characteristics of a particular architecture. The issue is that memory access patterns that lead to maximum coalescing in a data-parallel GPU result in poor spatial locality on a multi-core CPU architecture.

Forcing developers to make such trade offs in a prescriptive manner reduces *performance portability.* A capable compiler must be able to detect when such transformation is beneficial for the target architecture. This experimental evaluation demonstrates that the IPDA-based loop collapse safety and profitability analyses are effective at capturing the same insights made by expert developers for a specific architecture, while allowing the code to remain generic.

To test this claim we remove all `collapse` clauses from the Polybench and SPEC ACCELL benchmarks available to us. Then we let the IPDA-based framework automatically perform the same transformations. The evaluation shows that the resulting, more-generic, code has *the same* performance as the architecture-specific code and has *better* performance on platforms for which it wasn't hand-tuned — this evaluation uses the Intel MIC Xeon Phi 7250 accelerator. Table 4.2 shows the impact of removing the `collapse` clauses from benchmark code. The performance implications of this directive in GPUs are clear from the slowdowns of up to 33× on `3MM` and 16× on `SYR2K` when it is removed. `COVAR` slowdown is not as significant because only 1 out of 3 parallel loops in the benchmark is annotated with collapse. The IPDA-enabled GPU Speedup column shows that the original performance is recovered when IPDA-based transformations are applied to the code from which the clauses were removed.

In multi-core CPU platforms each processor has a local cache. Adjacent threads accessing adjacent memory locations, which is the effect of collapsing, results in false sharing, causing unnecessary coherence traffic and degrading performance. Intel's OpenMP performance guidelines recommend avoiding this usage pattern at all cost [21]. Yet, even highly-tuned benchmark implementations have programmers inserting prescriptive clauses that maximize false sharing. The 'No Collapse MIC Speedup' column of Table 4.2 shows that simply removing the collapse clauses can significantly improve performance on an x86-based accelerator (up to 26%). Analysis and transformation capabilities enabled by IPDA go a long way towards removing the need for specializing the code to a given architecture.

```
#pragma omp target teams distribute parallel for private(i,j,k,m,
    fac1,j1,j2)
for (k = 1; k <= gp2-2; k++) {
  for (j = 0; j <= gp1-3; j++) {
    j1 = j + 1;
    j2 = j + 2;
    for (i = 1; i <= gp0-2; i++) {
      fac1 = 1.0/lhsY[2][k][j][i];
      lhsY[3][k][j][i] = fac1*lhsY[3][k][j][i];
      lhsY[4][k][j][i] = fac1*lhsY[4][k][j][i];
      for (m = 0; m < 3; m++) {
        rhs[m][k][j][i] = fac1*rhs[m][k][j][i];
      }
      lhsY[2][k][j1][i] = lhsY[2][k][j1][i] - lhsY[1][k][j1][i] *
          lhsY[3][k][j][i];
      ...
    }
  }
}
```

Figure 4.9: Excerpt from a target region in 557.pcsp

4.7 Concluding Remarks

Architecture-specific compiler optimization is key for achieving performance portability for high-level parallel programs. Conflicting demands of current accelerator architectures when it comes to efficient use of memory hierarchies mean compilers demand stronger program analyses and heuristics in order to generate optimal code for a given target. This chapter introduced a static analysis framework capable of identifying memory access strides of parallel accelerator code using Iteration Point Difference Analysis. The evaluation of a prototype implementation of a framework that uses IPDA to guide the safety and profitability decisions required for improving performance through loop transformations, demonstrated the potential for dramatic performance improvement in GPU-bound OpenMP code. Moreover, this chapter also demonstrated that informed compiler transformation can further advance the goal of performance portability by reducing the reliance on programmer hints used to hand-tune OpenMP loop code. Making such hints redundant both increases performance across a greater variety of target architectures and increases abstraction of the underlying computing platform, making parallel programs more generic and allowing the developer to focus instead on the problem at hand.

Setup / Bmk	Original				Increased Input Size ($16\times - 64\times$)				High-Throughput	
	P8 + P100		**P9 + V100**		**P8 + P100**		**P9 + V100**		**P8 + P100**	**P9 + V100**
	Kernel	**Overall**	**Kernel**	**Overall**	**Kernel**	**Overall**	**Kernel**	**Overall**	**Overall**	**Overall**
GEMM	25.5×	3.18×	20.9×	1.08×	8.07×	6.79×	6.99×	3.64×	15.9×	7.14×
2DCONV	14.75×	1.65×	6.37×	1.05×	31.88×	1.43×	11.1×	1.10×	1.41×	1.11×
3DCONV	22.84×	1.43×	18.3×	1.03×	19.49×	1.30×	10.35×	1.20×	1.82×	1.56×
557.pcsp	86.5×	3.38×	111.1×	2.3×	——	——	——	——	——-	——-

Table 4.1: Benchmark execution time speedup with automatic loop interchange and collapse enabled compared to the default code-generation scheme.

Benchmark	**Parallel OpenMP 4 loops**	**User-specified `collapse` clauses**	**IPDA automatic collapse transformations**	**No Collapse GPU Speedup**	**IPDA-enabled GPU Speedup**	**No Collapse MIC Speedup**
`3MM`	3	3	3	0.03×	1.0×	1.26×
`COVAR`	3	1	1	0.98×	1.0×	1.04×
`SYR2K`	1	1	1	0.06×	1.0×	1.13×
`SYRK`	2	2	2	0.12×	1.0×	1.15×
`503.postencil`	1	1	0	0.33×	0.33×	1.83×
`555.pseismic`	14	13	0	0.09×	0.09×	1.01×
`563.pswim`	17	8	0	0.84×	0.84×	1.19×
`570.pbt`	42	36	26	0.78×	0.96×	0.97×
`557.pcsp`	60	60	50	0.04×	0.76×	`na`

Table 4.2: Loop `collapse` clauses were removed from benchmarks which contain them. This table shows the portion of the loop collapses re-discovered to be benefitial and automatically applied by the compiler. Execution ratio columns show the performance of the code stripped of collapse clauses versus the code with collapse clauses present. GPU code is executed on an Nvidia P100. The MIC (Many Integrated Core) relative execution column compares the performance of the same two versions of the kernel executed on an Intel Xeon Phi 7250 Processor, compiled with ICC ver. 17.0.2

Chapter 5

Toward Hybrid Execution Target Selection Through Analytical Performance Modeling

Analytical performance modelling, a mature field of research, has been the focus of work in tuning software systems and guiding compiler optimizations. Due to the increasing prevalence of heterogeneous compute platforms, architecture-specific performance modelling becomes a progressively important topic due to the role it has to play when deploying target-agnostic applications. The ability to choose the processing unit which will execute a given section of code can result in a critical performance advantage that can be offered by compiler/runtime systems. Existing analytical models strive to capture the complexity of the architectures they are modelling, and the interplay between the levels of abstraction used to represent said architectures.

A critically important challenge faced by analytical performance predictors for CPU execution is to model the specifics of CPU resource allocation and how it impacts instruction latencies. To improve the accuracy of CPU instruction mix latency modelling, we propose an elegant solution that leverages LLVM-MCA - a predictor that uses the compiler's built-in instruction scheduling algorithms [38]. The tool is integrated into an existing analytical model in order to increase its accuracy. In the realm of GPU performance models, Hong's performance model is a seminal approach to runtime prediction [20]. One of the model's biggest losses in abstraction is in characterizing the coalescing characteristics of memory accesses - a critical factor for GPU code performance. We introduce an improvement to the model that applies IPDA, a hybrid symbolic analysis framework that captures the precise coalescing characteristics of OpenMP parallel loops set for GPU code-generation, in order to generate better estimates of the GPU's memory-warp parallelism.

This chapter addresses the following research problem: *What should be the paradigm of constructing runtime target device selection heuristics, what are the biggest challenges involved, and how to make such heuristics suitable to production environments.* Modelling execution of compute kernels on a variety of accelerator architectures is a notoriously challenging task, we highlight this by examining cross-generational GPU architectural differences having a significant impact on the outcome of deciding whether to execute a kernel on an accelerator target or to keep execution on the host. We position analytical performance modelling in addressing this research question as the approach to making better decisions fast and efficiently. Machine Learning-based algorithms may achieve high degrees of accuracy, but suffer from important drawbacks that limit their applications. In particular, the high level of dependence on runtime parameters in order to make an informed decision requires that the learned model be evaluated immediately prior to kernel launch, which can be a prohibitively expensive procedure. Moreover, a classical problem of learning approaches — their black box nature — is a much more serious limitation in compiler/runtime systems due to its effects on understandability, reproducibility and susceptibility to non-linear, and sometimes non-contiguous, relations between model parameters and performance. Finally, we present a decision framework for profitability analysis of offloading GPU versions of OpenMP parallel loops that indicates that analytical models based on a combination of static analysis and runtime parameters may be most suitable for such decisions.

5.1 Comparative Offloading Performance Change Across GPU Generations

Significant differences among generations of GPU architecture and bus interconnects mean that performance models must be fine-tuned to the most intricate details of the platform they aim to abstract. Table 5.1 displays our experimental measurement of GPU offloading benefit for a series of Polybench OpenMP kernels. The data was collected on two experimental platforms: 1) POWER8 Host + Nvidia Tesla K80 (PCI-E) and 2) POWER9 Host + NVidia Tesla V100 (NVlink 2). The k80 and V100 host's CPUs was clocked at 3000Mhz. All programs were compiled using the IBM XL C/C++ compiler ver. 16.1. Each kernel was evaluated in two execution modes, *test* and *benchmark*, which differ only in the size of the program's input, being 1100×1100, and 9600×9600, respectively, in most programs. Each benchmark was executed 10 times and the average execution time of each kernel is used for relative performance measurements. Kernel execution time includes data transfer, but does not include the CUDA context initialization that occurs on the first kernel launch by a given program. The context creation is an overhead paid once by a program that may repeatedly launch many kernels. Omitting context initialization overhead presents a more typical case of executing a kernel of computation and prevents the results from being skewed on single-kernel benchmarks. In our experiments, on Volta architecture, CUDA context initialization can take upwads of 0.5 seconds. The recorded kernel execution time is used to present speedup over the host execution time of the same `target` region.

This data shows that a single GPU generation may sway the offloading profitability decision in a drastic fashion. For example, the `3DCONV` kernel, in benchmark configuration is a far better fit for execution on the CPU when the accelerator choice is Kepler, with GPU offloading resulting in a slowdown of of 2.1×. Yet, a Volta equipped machine with an even more capable CPU sees a dramatic speedup of 4.41× when offloading the same computation to the GPU. The benchmark's computation kernel has low arithmetic intensity and is heavily memory-bound; thus, benefiting greatly from the Volta's card memory

Platform	GEMM1	MVT1	MVT2	3MM1	3MM2	3MM3	2MM1	2MM2	ATAX1	ATAX2	BICG1	BICG2	2DCONV1	3DCONV1	COVAR1	COVAR2	COVAR3	GESUMMV1	SYR2K1	SYRK1	SYRK2	CORR1	CORR2	CORR3	CORR4
Test GPU Offloading speedup on Host with 4 OMP Threads																									
P8+k80	4.01	5.12	0.67	5.24	28.07	24.26	6.68	14.70	5.43	0.79	5.23	0.53	0.11	0.13	5.42	0.71	3.44	0.09	1.68	3.21	3.58	5.55	0.67	3.26	2.95
P9+v100	1.54	23.26	36.67	186.59	183.67	187.78	182.88	180.95	19.05	1.36	25.36	1.15	0.39	0.53	25.19	2.19	12.97	0.06	3.94	16.93	71.67	19.43	1.36	4.54	12.10
Benchmark GPU Offloading speedup on Host with 4 OMP Threads																									
P8+k80	128.3	0.44	3.66	132.76	132.54	133.97	137.86	137.70	0.42	2.77	2.79	0.44	0.54	0.98	2.85	1.12	80.79	0.17	0.98	0.64	2.49	2.96	4.21	5.04	79.85
P8+v100	332.85	4.31	3.91	362.36	389.38	373.92	342.03	384.55	1.67	4.30	4.62	1.44	5.18	23.73	4.37	1.75	145.62	0.30	4.10	1.70	4.62	4.56	3.93	2.88	142.25
Test GPU Offloading speedup on Host with 160 OMP Threads																									
P8+k80	3.31	52.25	1.37	3.84	10.34	11.04	3.52	9.78	49.84	1.24	52.46	48.43	1.47	0.87	53.18	37.57	3.05	0.96	1.03	26.69	1.69	70.44	46.97	3.82	2.47
P9+v100	0.49	66.67	88.92	62.48	52.72	53.22	51.89	36.24	51.64	40.69	71.53	33.44	1.11	0.81	66.92	0.66	5.54	0.17	0.61	41.83	8.91	75.76	0.70	1.49	5.15
Benchmark GPU Offloading speedup on Host with 160 OMP Threads																									
P8+k80	35.45	0.89	2.53	31.64	34.75	32.37	45.54	42.82	0.80	2.90	2.98	0.23	0.35	0.47	2.21	0.72	64.94	0.30	0.21	0.79	0.42	2.45	3.21	2.33	65.09
P8+v100	32.07	1.44	1.54	36.26	32.18	29.39	35.55	34.43	1.15	1.59	2.61	0.17	1.48	4.41	2.52	0.44	41.088	0.18	0.43	0.91	0.20	1.66	1.18	0.73	35.03

Table 5.1: Cross-architectural changes in GPU sffloading speedup vs. host execution.

bandwidth of 900GB/s, nearly double of the K80's peak 480GB/s. An example to the contrary is the `CORR` kernel, which, in benchmark execution mode, is a good candidate for acceleration for a POWER8 host, but should not be offloaded on a POWER9 machine. This outcome holds despite a more capable GPU on a faster interconnect. The four kernels invoked by the benchmark contain sequential loops to be executed by each parallel worker, which are well-suited for SIMD vectorization and stand to benefit from POWER9's broader vector operation support and newly introduced VSX3 operations. In several other cases, despite the decision whether a `target` region should be offloaded remaining the same, the magnitude of change of speedup is colossal: `ATAX2` kernel, in a test run, compared to a 160-thread host, saw an offloading speedup go from 1.24× on K80 to 40.69× on a V100 due to a combination of faster data transfer rates and architectural improvements.

5.1.1 Generational Performance Gaps Require Fine-Tuned Performance Estimates

Year-over-year advances in GPU generations are far out-pacing development of CPU architecture. This pace of innovation coupled with rise in domain-specific applications particularly well-suited to data-parallel computation mean that rapid evolution of accelerator architectures presents significant challenges for both the compiler developers, and the research community working on analytical performance modelling. Both are chasing a moving target for code optimization and analysis. Meanwhile, CPU platforms too are gaining new features and ever-increasing facilities for vector computation, adapting to the emerging workloads through application-specific gadgets. The increasing importance of performance models means that they need to capture greater amounts of detail intricacies of their target architectures; meanwhile, a growing variability across computing-device architecture types calls for more domain-specific expertise on behalf of those who attempt to model them, attracting more hardware experts to the problem.

5.2 A New Hybrid Analysis Framework for Deciding the profitability of GPU Offloading

Designing a compiler/runtime framework for a heterogeneous system that combines multiple processing units is a challenging task. Such framework must amalgamate multiple compilation backends that generate code for several targets and bundle all versions into a single binary, a collection of static analyses that extract relevant program features and characteristics, a means to aggregate relevant dynamic information at a program point prior to the relevant target section, a runtime machine description query mechanism, and detailed performance models that would inform the final offloading selection decision. We describe a prototype of such a framework and outline our approach in detail below, concluding the description by providing some early results of applying the framework on an OpenMP 4 microkernel benchmark suite — Polybench.

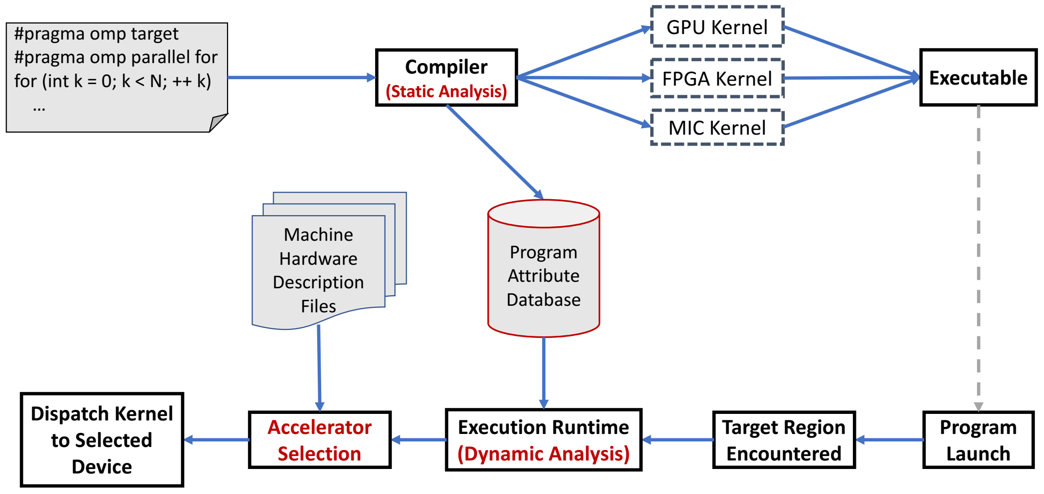


Figure 5.1: Example compilation and execution flow of an offloading decision compiler/runtime framework.

Figure 5.1 shows the flow of program compilation and execution. The IBM XL Compiler is used for this prototype. This compiler is a fully compliant implementation of the OpenMP 4.5 standard, capable of outlining `target` regions specified in the program and translating them into GPU kernels. The outlined region is duplicated prior to code-generation, and a host-bound CPU-parallel version is generated to provide a fallback mechanism in case a GPU device is unavailable. After the creation of optimized versions of the compute kernel for CPU and GPU, the compiler was augmented with a static analysis that collects relevant program features that will form skeletons of respective performance models. The evaluation of these models may depend on values that cannot be known at compilation time and that can be only discovered at runtime; thus, statically constructed performance predictors are inherently incomplete. During program execution, on reaching the target region, the OpenMP runtime is invoked that initializes the accelerator, queues up required data-transfers and launches kernel execution. In our proposed method, the runtime is augmented to instead extract the compiler-collected program features from the program attribute database, and collect runtime values that were missing from the static attributes. A compiler transformation is required that supplies the OpenMP runtime with dynamic information e.g. array sizes, loop trip counts, arbitrary variable values that may be required to determine memory access stride/characteristics. The above data is then used to generate predictions of potential performance gain or loss of offloading the `target` region to the GPU. Finally, based on the decision, either of the two generated versions of the region code is invoked for execution.

The measurements were performed on an IBM POWER9 (AC922) machine with an Nvidia V100 GPU accelerator connected via the NVlink 2 interface. The machine runs RHEL Server 7.3 Operating System with CUDA version V9.2.88. Due to the requirements placed on the LLVM's instruction scheduler by LLVM-MCA, POWER9 is the only viable host architecture for our experiments at the time of writing. OpenMP loops from the Polybench benchmark suite, representing kernels of the more common high-level computation operations is used to demonstrate the performance model's efficacy and applicability in deciding GPU offloading profitability [50].

$$
\begin{aligned}
Parallel\_Region_c &= Fork_c + \\
&\sum_{j=1}^{m}[maximum(Thread_0\_exe\_j_c, ..., Thread_{n-1}\_exe\_j_c)] + \\
&Join_c \\
Thread_i\_exe\_j_c &= Worksharing_c + Synchronization_c \\
Worksharing_c &= Parallel\_for_c | Parallel\_section_c | Single_c \\
Synchronization_c &= Master_c | Critical_c | Barrier_c | Atomic_c | Flush_c | Lock_c \\
Parallel\_for_c &= Schedule_times \times (Schedule_c + Loop_chunk_c \\
&+ Ordered_c + Reduction_c) \\
Loop\_chunk_c &= Machine_c\_per\_iter \times Chunk\_size + Cache_c \\
&+ Loop\_overhead_c
\end{aligned}
$$

Figure 5.2: Equations of Cost Model for OpenMP from [33].

5.2.1 OpenMP CPU Performance Model

In this work, we leverage a compile-time cost model for OpenMP proposed by Liao and Chapman [33]. The cost model was originally built to augment existing performance estimators of the OpenUH optimizing open-source OpenMP compiler for C/C++ and Fortran programs [34]. OpenUH, in turn, inherits its performance models largely from the Open64 loop nest optimizer infrastructure [62]. Liao's adaptation of the compile-time model implements extensions that account for specifics of OpenMP work-sharing constructs, estimating execution time of a parallel region as determined by the execution time of the most time- consuming thread between each pair of synchronization points. It also adds factors such as scheduling overhead cycles and parallel loop chunk size overheads. An appealing quality of this model is that values of its parameters can be obtained from micro-benchmarks [6], [7], [54]. Figure 5.2 contains the OpenMP model's equations, directly derived from the equations of the original OpenUH parallel model. Our input kernels consist of strictly parallel loop code and therefore other types of work-sharing constructs described by the model are not exercised. Table 5.2 contains various parameters used in the model. Some obtained from the POWER9 Processor User Manual [48]. The TLB miss penalty is estimated using the TLB cost measurement tools included in the Linux libhugetlbfs utility [35]. We used the EPCC OpenMP micro-benchmark suite to measure scheduling and synchronization overhead parameters of the execution model on our hardware configuration [15].

Cycles Per Iteration of a Parallel Loop

A key metric in Liao's model is the $Machine_c\_per\_iter$ value, computed based on cycles from FP and ALU units, processor memory units, and issue units. Deep ties to the OpenUH compiler's inner instruction scheduler have made it challenging to obtain this estimate in other contexts until recently. We forego the original model's calculations on processor resource, dependency latency and register allocation cycle estimates in favour of the LLVM Machine Code Analyzer (MCA). Spearheaded by SONY, MCA is a performance analysis tool that uses the LLVM infrastructure's rich hardware backend ecosystem to estimate the value of IPC for a given sequence of assembly instructions [38]. Both the compiler-instruction-scheduler-driven analysis and the tool's reporting style were heavily influenced by Intel's IACA tool [22]. The prediction for the number of cycles required to execute an assembly sequence is based on throughput and processor resource consumption as the backend's instruction scheduling model already specifies. The backend module is used to emulate execution of machine code sequence, while collecting a number of statistics which are then presented as a report. The tool is able to handle the presence of long data dependency chains and other bottlenecks. Due to its reliance on the instruction scheduler, it is limited by the quality of the information present in the scheduler. For example, common machine instruction schedulers omit information on the number of retired instructions per cycle or the processor's number of read/write ports in the register file. The tool's known limitations also include a lack of a cache hierarchy and memory type model.

In our experimental implementation, the tool is integrated into the compilation process. The body of a parallel loop is extracted and MCA is used to estimate the total number of cycles required to execute it, yielding the number of cycles spent by a thread participating in the parallel region to do the work of one iteration — $Machine_c\_per\_iter$ — in the performance model. The cache hierarchy model, missing from the analysis tool, remains a limitation of the performance model described here and is a primary future work direction to improve the model's accuracy.

CPU Frequency	3 Ghz
TLB Entries	1024
TLB Miss Penalty	14 Cycles
Loop_overhead_per_iter	4 Cycles
Par. Schedule Overhead static	10154 Cycles
Synchronization Overhead	4000 Cycles
Parallel Startup	3000 Cycles

Table 5.2: CPU processor/parallel parameters as used in the execution model.

5.2.2 GPU Performance Model

An analytical model for a GPU architecture with Memory-level and Thread-level parallelism awareness by Hong and Kim is a seminal approach to performance prediction of GPGPU kernels [20]. Our work implements their model adapted to the Volta architecture by combining static-analysis-driven feature gathering, dynamic kernel information acquired on encountering a target region, and micro-benchmark acquired hardware parameters for values not directly disclosed by the vendor.

Static features

The IBM XL compiler generates a GPU kernel version of encountered target regions. Static analyses were integrated into the compilation process that gather program features that are required by the model or are otherwise important indicators of performance.

Instruction Loadout

A key factor in the performance model is the amount of work performed by individual threads. For example, if the amount of computation done by each thread is very small, threads will finish execution very quickly and will have

If (MWP == N == CWP):

$$\begin{aligned} Exec\_Cycles = (&Mem\_Cycles + Comp\_Cycles + \frac{Comp\_Cycles}{\#Mem\_Inst} \\ &\times (MWP - 1)) \times \#Rep \times \#\boldsymbol{OMP\_Rep} \end{aligned}$$

If (MWP > CWP):

$$\begin{aligned} Exec\_Cycles = (&Mem\_Cycles \times \frac{N}{MWP} + \frac{Comp\_Cycles}{\#Mem\_insts} \\ &\times (MWP - 1)) \times \#Rep \times \#\boldsymbol{OMP\_Rep} \end{aligned}$$

If (CWP > WWP):

$$\begin{aligned} Exec\_Cycles = (&Mem\_L + Comp\_cycles \times N) \times \#Rep \\ &\times \#\boldsymbol{OMP\_Rep} \end{aligned}$$

Figure 5.3: Hong and Kim performance model program exection prediction [20]. Highlighted is the additional factor that describes the side-effects of OpenMP thread-loop-iteration scheduling.

to be queued to be scheduled for more work. In this case, the overhead of scheduling more work to be performed on a GPU in a very small time will be larger than the actual kernel computation, most likely leading to poor performance. The model's thread execution cycle estimate is computed using the number of dynamic instructions. We implement a simple static analysis to count the number of IR instructions, which will be translated into native micro-instructions later. Given the closed nature of the true GPU assembly ISA, this serves as a good estimate. Our static analysis groups collected instructions into IO and CMPUT categories. Control-flow constructs are abstracted in an identical way across CPU and GPU analyses: all loops are assumed to execute 128 iterations and all conditional blocks of code are executed half of the time. While the absolute prediction accuracy of this approach might suffer, it should provide a reasonable point of comparison of *relative* performance between the two platforms.

Architectural Model Parameters

Nvidia Tesla V100	
#SMs	84
Processor Cores	5376
Graphics Clock	1.312 Ghz
Processor Clock	1.53 Ghz
Memory Size	16 GB
Memory Bandwidth	900 GB/s
NVLink Transfer Rate	25 GB/s
Max Warps/SM	64
Max Threads/SM	2048
Issue Rage	1 cycle
Int Cmput Inst. Latency	4 cycles
Float Cmput Inst. Latency	8 cycles
Memory Access Latency	1029 cycles
Access on TLB Hit	375 cycles
Access on L2 Hit	193 cycles
Access on L1 Hit	28 cycles

Table 5.3: GPU device/bus parameters as used in the execution model.

Figure 5.3 shows the Volta architecture-specific values used by the model. These values were gathered from either the CUDA API queries, vendor manuals, and the excellent technical report by Zhe Jia who obtained them in a deep examination of the architecture through micro-benchmarking [25].

Runtime Model Parameters

The dynamic aspect of the hybrid approach to performance estimation is essential because only with runtime values the analytical models can be complete. The sizes of kernel inputs prescribe the amount of data that will be sent to the device and back over the interconnect. The size of the iteration space of the original parallel loop affects the number of parallel work-items in the resulting data-parallel program and the grid geometry the runtime will select. In order for the runtime to obtain these values, they are stored into a Program Attribute Database which is queried at execution time, indexed by the target region's program and location.

The original cycle count estimate from the Hong model needed to be modified to adjust for one OpenMP specific aspect of GPU code-generation. The $\#OMP\_Rep$ parameter, highlighted in Figure 5.3, represents cases where the maximum grid-geometry selected by the runtime does not result in a sufficient threads to cover all parallel work items — iterations of the original parallel loop. In that case, a thread performs the work that comprises the body of the original parallel loop, then, depending on the specified schedule of the parallel loop, is assigned another iteration to execute, either by advancing by a static chunk size, or querying the OpenMP runtime. This parameter is set to account for the number of distinct loop iterations a single thread will execute if the number of loop iterations is higher than the product of $num\_thread\_blocks \times threads\_per\_block$.

5.2.3 GPU Memory Access Pattern: Improved Coalescing Detection

Given the different memory organizations in different accelerators, the memory access pattern is an important input to the performance model. Existing approaches to performance modelling rely on either crude estimates or trace and profile driven analysis that requies an application to be executed in order to determine its coalescing characteristics The latter require the code to execute prior to the model being able to generate an accurate prediction. This constitutes a key shortcoming in a production runtime and is a key improvement of this approach in relation to solutions that appear in related work.

To further improve the accuracy of memory-throughput related model parameters by increasing the accuracy of memory-coalescing characteristics of code, we build memory-access related parameters of the model using the IPDA analysis framework [10]. Our prototype deploys IPDA to construct a symbolic equation for the inter-thread stride of each memory access. For example, suppose the kernel in question contains the following parallel loop:

```
#pragma omp teams distribute parallel for
for (int a=0; a<max; a++) {
    A[max*a] = ...
}
```

IPDA creates a symbolic expression for the inter-thread access stride on the store to array `A` in line 3:

$$\begin{aligned} & IPD_{t1}(\texttt{A[max * a]}) - IPD_{t0}(\texttt{A[max * a]}) \\ & = \texttt{[max]} \times 1 - \texttt{[max]} \times 0 \\ & = \texttt{[max]} \end{aligned}$$

where a value contained in [] indicates a symbolic unknown. Two possibilities exist in which the framework determines the stride for this memory access:

1. the value of `max` is known at compile-time and IPDA is able to statically determine, for example, whether or not this kernel would result in coalesced GPU code.

2. the value of `max` is not known statically, but is known at runtime, prior to kernel launch.

In our proposed compiler/runtime framework, the IPDA symbolic expression of the access stride is stored in a Program Attribute Database (Figure 5.1). At the program point when the target region is encountered, the unknown values are extracted and used in the symbolic expression to compute the actual stride, informing the analytical model with whether or not the kernel's accesses are coalesced and to what degree ($\#Uncoal\_Mem\_inst$ and $\#Coal\_Mem\_inst$ values used to compute Mem_Cycles).

5.2.4 Putting It All Together

With both the CPU and GPU analytical performance models defined in the OpenMP runtime system, the compiler must alter the code generated for invoking an encountered `target` region. Instead of simply launching GPU kernel execution, the generated code configures the runtime to extract static features of the generated versions of the region, feeds in the necessary runtime values, and queries the results of performance models. The model that results in the lowest predicted runtime is chosen as the winner and execution is queued up on the architecture the model describes, either the host CPU or GPU. Because

$$MWP = min(MWP\_Without\_BW, MWP\_peak\_BW, N)$$

$$BW\_per\_warp = \frac{Freq \times Load\_bytes\_per\_warp}{Mem\_L}$$

$$MW\_peak\_BW = \frac{Mem\_Bandwidth}{BW\_per\_warp \times \#ActiveSM}$$

$$CWP\_full = \frac{Mem\_Cycles + Comp\_Cycles}{Comp\_Cycles}$$

$$CWP = MIN(CWP\_full, N)$$

Figure 5.4: Equations of Memory-Warp and Compute-Warp Parallelism used in GPU execution Cost Model from [20].

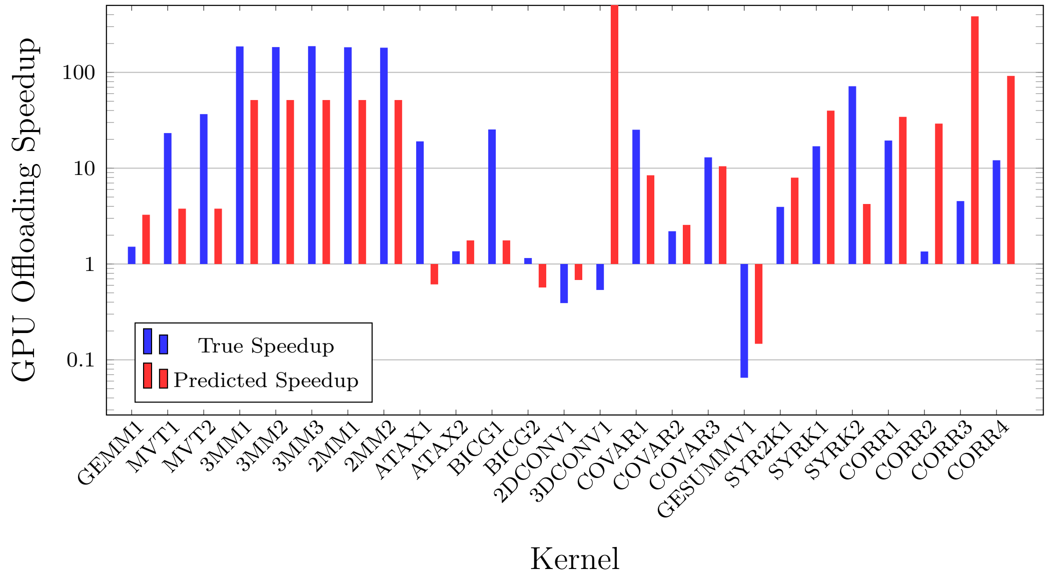


Figure 5.5: Actual versus predicted GPU offloading speedup for *test* kernel execution mode versus a host using 4 threads.

of the analytical nature of the model, generating a prediction for either target is equivalent to solving an equation, making decision time negligible. This goes in a stark contrast to an approach that would employ machine learning to perform model inference at runtime, a step that may, in fact, take longer than the kernel execution itself [37].

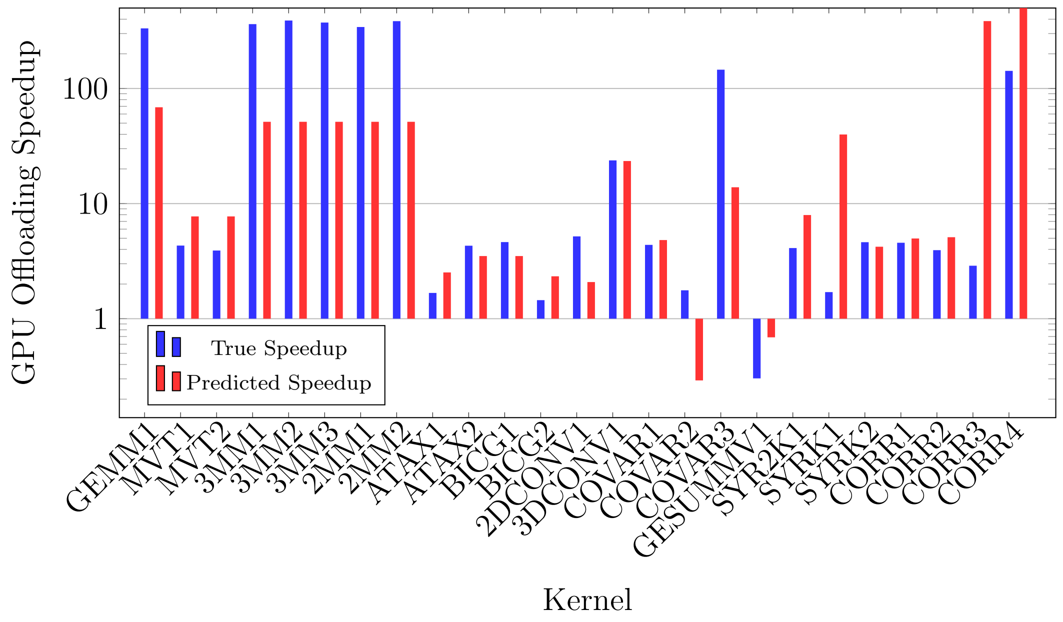


Figure 5.6: Actual versus predicted GPU offloading speedup for *benchmark* kernel execution mode versus a host using 4 threads.

5.2.5 Evaluation

We present some early results of the hybrid decision analysis by evaluating it on a collection of parallel OpenMP loops found in the Polybench benchmark suite. These parallel loops represent common atoms of computation found across a variety of applications. 25 kernels from are executed from 12 different benchmarks: `GEMM`, `MVT`, `3MM`, `2MM`, `ATAX`, `BICG`, `2DCONV`, `3DCONV`, `COVAR`, `GESUMMV`, `SYR2K`, `SYRK`, and `CORR`. Each benchmark in the suite has two modes of execution, test and benchmark. The execution modes differ only in the size of the program's input, being 1100×1100, and 9600×9600, respectively. We also include results for restricting the host execution environment to just 4 threads to demonstrate both the adaptability characteristics of the models, and a scenario that resembles a more typical execution environment, when compared to our experimental machine's 20-core 8-SMT CPU running at full capacity of 160 threads. Execution runtimes were recorded as average kernel runtimes across 10 runs of each benchmark. For the sake of a fair head-to-head comparison of kernel execution times across platforms, this overhead is omitted from the evaluation in order to demonstrate a generic case of computation offloading in a running application, similarly to the experiment described in Section 5.1.

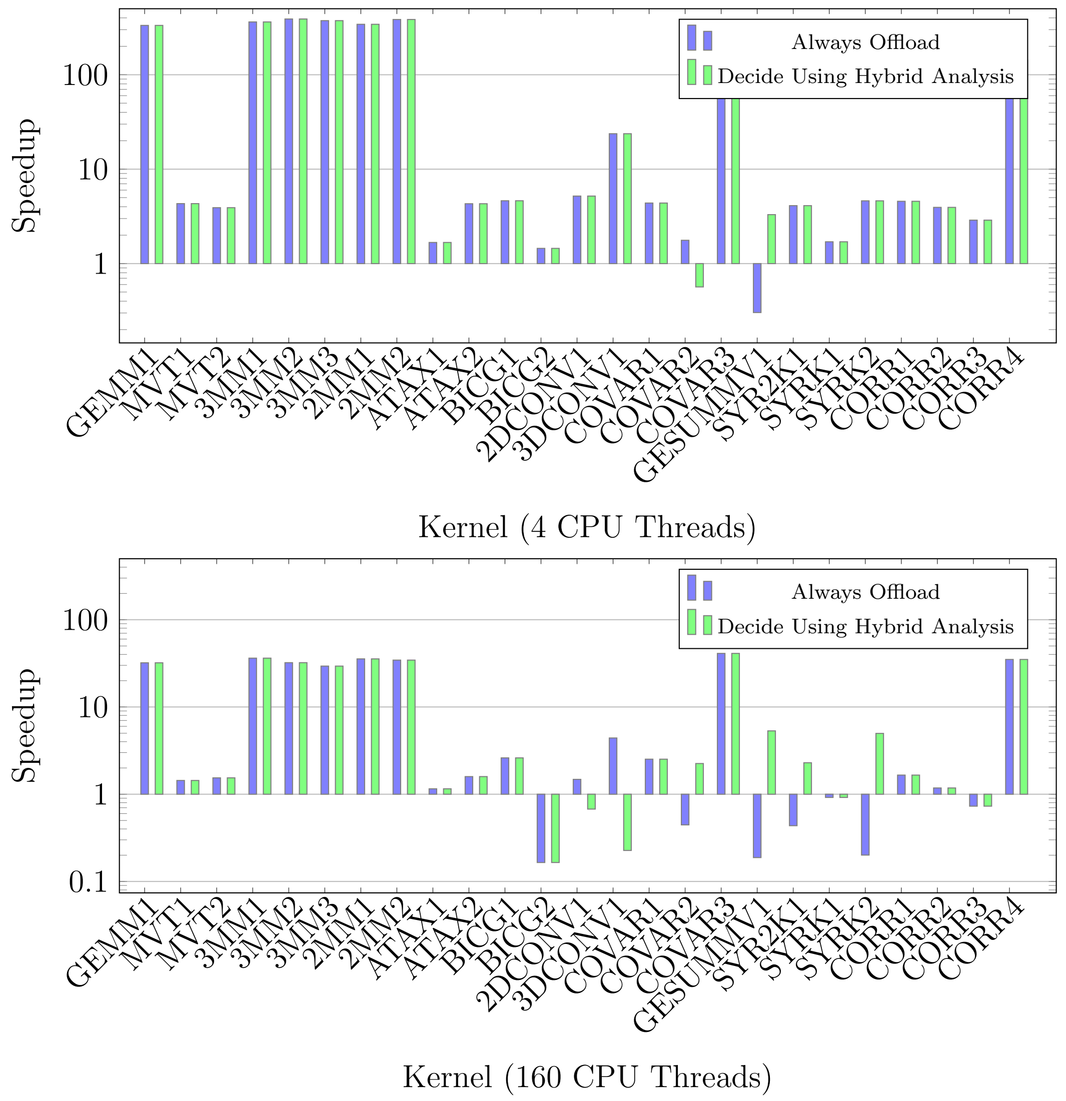


Figure 5.7: Speedup achieved by always offloading to GPU versus offloading when determined profitable by the analytical hybrid decision model. Benchmarks executed in *Benchmark* mode.

Figure 5.5 and Figure 5.6 demonstrate the true versus predicted speedup of offloading the kernel execution to the GPU in *test* and *benchmark* execution modes. While the overall magnitude of the predictions is often off due to the abstraction of control-flow constructs in the performance model (iterations of all loops counted 128 times), the predicted outcome is correct for 23/25 kernels.

When deploying the decision analysis framework to select the execution target, overall benchmark suite execution time is improved. When following the compiler's default policy of always offloading `target` regions to an accelerator, GPU offloading of all kernels yields a geometric mean speedup of 10.2× and 2.9× versus a host using 160 CPU threads (*Test* and *Benchmark* execution modes, respectively). Switching the runtime to evaluate the relative performance of GPU offloading through analytical modelling and only do so when predicted to be profitable profitable results in a geomean speedup of 14.2× and 3.7× on an otherwise the same configuration. Figure 5.7 shows the speedups achieved under both experimental setups. Note that the speedup provided by the GPU is captured in most cases, with few notable outliers: in the 160-thread *Benchmark* execution mode, the model's decision on the convolution kernels is incorrect, predicting a speedup of 0.913×, whereas the true offloading speedup is 1.48×, in the 2D case. Discrepancies in scenarios where the decision is a close one, such as these, require further tuning of the model to increase its accuracy. Improved representation of the memory hierarchy impacts is a sure way to improve prediction efficacy for these scenarios. The `SYRK2` kernel in *Test* execution mode has the performance model severely over-estimate the GPU execution time relative the the CPU running at 160 threads, likely due to over-accounting for the kernel's poor coalescing characteristics without taking the details of cache hierarchy into account.

While the OpenMP specification does not, currently, allow compliant runtime systems to elect to not offload `target` regions, there is a clear need to provide this ability to runtime vendors. Even among highly-regular OpenMP parallel loops — a construct best-suited for translation into data-parallel code, there are computation patterns ill-suited to GPU acceleration. While more difficult to model, common OpenMP programs that utilize mixtures of construct types to express parallelism alongside sequential code within `target` regions are even more likely to see better performance on the host fallback path. We demonstrate an early but successful attempt at guiding compile/runtime system architecture to handle a more descriptive programming model approach. The upcoming OpenMP 5.0 standard is set to introduce new constructs that allow implementors exactly this kind of freedom [47].

5.3 Discussion and Future Work

The task of constructing performance models is one of increasing importance. It requires deep understanding of intricate details of target architectures that go far beyond spec sheets. It also represents a chase after a running target as hardware architectures iterate at an increasing pace. As the challenge of the task rises, so does its importance; rise of heterogeneous computing platforms that combine radically different processing units in a single system makes these models crucial for a new realm of optimizing programs to the machine at-large, rather than a singular micro-architecture.

OpenMP 4.0 standard greatly expands the functionality of the programming model by introducing support for programming heterogeneous computing systems. Newly written applications can take advantage of powerful accelerators like GPUs by annotating the code with appropriate `target` constructs. Meanwhile, a great wealth of existing OpenMP code can be upgraded by users through fairly minor modifications and additions of new directives to existing constructs. This work takes the notion of upgrading existing OpenMP 3.x code further by taking the developer out of the equation and proposing an architecture for a supporting compiler implementation to automatically offload suitable parallel loops. This reduces the effort of porting legacy code to state-of-the-art heterogeneous computing platforms to a simple act of recompilation. A hybrid approach to a profitability analysis of offloading parallel loops to a GPU is essential due to the complexities of the trade-off made when sending both program code and data to a GPU. This approach demonstrates encouraging results, showing significant performance gain is possible through application of such analysis by choosing the correct architecture present in the system for execution of parallel code. Looking ahead, the upcoming OpenMP 5.0 will introduce the `concurrent` loop construct directive, which asks the compiler to make a decision on how to parallelize the loop and, more importantly, where to execute it. In light of this development, the work in profitability analysis of offloading OpenMP code to GPUs becomes ever so prudent.

Chapter 6

Related Work

The work presented in this thesis spans multiple areas of inquiry, well-studied by researchers. Related work is presented in this chapter, organized by fields related to the projects that comprise this thesis.

6.1 Overlapping GPU Computation and Memory Transfers

Asynchronous transfers are used for BigKernel, by Mokhtari et al., which breaks up a kernel into smaller kernels and pipelines memory transfer in a similar fashion to our kernel pipelining process [42]. BigKernel is a coding framework wherein a memory transfer that would be too large for the available GPU memory is partitioned into segments which are then transferred onto the GPU as needed. The data segments are laid out by BigKernel in host memory by analyzing the GPU kernel and organizing all data items that are used into a prefetch buffer in the order of their access by the GPU threads, creating more coalesced memory accesses in the GPU as memory accessed at the same time is placed beside each other. As with our pipelining method these transfers are performed asynchronously and are overlapped with unrelated kernel computation. However, BigKernel requires a programmer to specify different GPU calls as opposed to the compiler transformation that we propose. Furthermore, BigKernel focuses on very large data and thus its design requires double the original number of threads for a kernel, with half the threads utilized for calculating prefetch addresses. These additional overheads are not present in our method.

A common approach to pipeline GPU execution uses double buffering. Komoda et al. present a OpenCL library that optimize CPU-GPU communication by overlapping computation and memory transfers based on simple program-descriptions written by the programmer [29]. Komoda's work is limited to pipelining memory transfers with existing GPU kernels, and requires programmer specification. Our approach, in contrast, creates multiple kernels out of a single description of a GPU program (a single `target` region) to enable pipelining.

6.2 GPU Occupancy / Grid Geometry

GPU occupancy is the focus of Kayıran et al.'s DYNCTA, a dynamic solution similar to ours that accounts for memory saturation by reducing occupancy [27]. DYNCTA analyzes each GPU SM's utilization and memory latency during execution and adjusts the occupancy within the SM to avoid memory-bandwidth saturation by keeping occupancy lower than the maximum. Changing the defined grid geometry for a kernel is impossible, as a result occupancy adjustment is achieved by assigning additional CTAs to a SM that have already been allocated to the kernel at the start of execution. Once assigned to an SM a CTA cannot be removed, as a result adjustment is performed by prioritizing or deprioritizing CTAs. A prioritized CTA has any available warps executed before a deprioritized CTA's warps, as a result with memory intensive programs wherein all warps stall on memory accesses the deprioritized CTAs will eventually be utilized after all prioritized warps stall. Performance is improved by having the occupancy just below the threshold where memory saturation causes negative effects, ensuring the SM remains utilized while avoiding the punishing effects of memory saturation. Analysis is recorded in two hardware counters within each SM, that record how long each SM has been under utilized and how the often the SM has stalled due to memory access waiting. Sethia et al. describe a similar approach with Equalizer, a heuristic that dynamically adjusts the number of CTAs based on four hardware counters [53]. Lee et al. propose a slightly different strategy with “Lazy CTA Scheduling” (LCS) wherein the workload of an initial prioritized CTA is calculated by a hardware performance counter and that data is used to calculate an improved number of CTAs for each SM [31]. In contrast, our grid geometry proposal is based on a simple hybrid analysis with a low runtime cost and is suitable for simple GPU kernels, which represent the majority of benchmarks we have tested. The benefits of a simple heuristic approach over heavyweight dynamic mechanisms, as outlined by Lloyd et al. allow for a practical deployment in a production system, even if it does sacrifice some optimility [37].

Sethia et al. present the Mascar system, which approaches memory saturation by prioritizing the accesses of a single warp instead of a round robin approach [52]. The single warp starts computation earlier to help hide the latency of other accesses, with the scheduler additionally prioritizing warps with computation over memory-accessing warps when memory saturation is detected. A queue for failed L1 cache access attempts is also added to the GPU hardware, holding the accesses for later execution, it prevents warps from saturating the cache controller with repeated access requests so that other warps can attempt their accesses. Mascar requires hardware design and warp scheduling changes. In contrast, our custom grid geometry based on static analysis is far less intrusive.

Other dynamic approaches include Oh et al.’s APRES, a predictive warp scheduler that prioritizes the scheduling of groups of warps with likely cache hits [46]. Kim et al. suggest an additional P-mode for warps waiting on long memory accesses wherein later instructions that are independent of the long accesses are pre-executed while any dependent operations are skipped [28]. Lee et al.’s CAWA reduces the disparity in execution time between warps by providing the slower running warps with more time to execute and a reserved area of the L1 cache [32]. All these approaches have additional run time costs when compared to a static analysis and compile-time selection of custom grid geometry and re-distribution of work across multiple kernels enabled by pipelining. Furthermore, warp scheduling approaches and custom grid geometries are complimentary and can be combined.

6.3 Performance Portability

The quest for performance portability of high-level parallel programming models has attracted much research attention to the areas of modeling and optimization of parallel-program performance [16], [40], [45], [61], [64]. In the context of OpenACC, Miles et. al. argue that true performance portability can only be achieved through compiler transformations guided by the specific demands of the target platform [41]. They observe that parallel loop nests must be structured differently depending on whether they are to be executed on a homogeneous multi-core machine, or on a highly parallel, throughput-optimized, accelerator. Both paradigms currently coexist in the domains of high-performance and scientific computing; thus, the continued development of compiler technology is key to achieve performance portability.

6.4 Symbolic Memory Reference Analysis

Symbolic analysis of loop code is a concept that dates as far back as 1976, when Cheatham and Townley proposed symbolic execution as a tool for loop analysis for the EL1 programming language [9]. This analysis expressed a set of facts about an execution of a loop across iterations captured as recurrence equations with symbolic unknowns. This research paved the way for decades of work that iterated on the idea. Haghighat et. al. use numerical finite differences in order to detect generalized induction-variable expressions and to reduce inter-iteration access stride to a recurrence, solving which can yield dependence information (Paraphase 2 compiler) [19]. Gerlek, Stoltz and Wolfe apply a technique based on a generalization of demand-driven constant propagation to detect strongly-connected components in the SSA graph with the goal of identifying sequence variables in program code [17]. They demonstrate how solving recurrences that occur in loop expressions can be used to replace update statements with the respective closed form.

Motivated by the need to analyze addressing expressions that cannot be captured as a solvable recurrence, Rus, Zhang and Rauchwerger introduced a framework for analysis of memory reference sets addressed by induction variables without closed forms [51]. The framework relies on a data structure called the Value Evolution Graph (VEG). Based on *Gated Static Single Assignment* representation, the VEG augments the GSA data-flow graph by representing values as ranges of possible actual values. Sequences of data-flow edges `p` →...→`q` form *evolutions*, which are unioned across all paths from `p` to `q` to form an aggregate evolution. Similarly to IPDA, the VEG can be used to compute iteration distance between two consecutively accessed elements. Representing evolutions as graph paths restricts the evolutions that it can represent and the kinds of operations that can be performed. IPDA's symbolic representation allows it to scale to large indexing expressions because the algebraic differences lead to simpler expressions due to term cancellations.

Moon et al. proposed a technique called *predicated array data-flow analysis* that associates predicates with data-flow values that represent control-flow paths taken to arrive at the values [43]. The predicates are formulated into executable program statements that form tests which guard parallelized versions of computation. They capture control-flow into the data-flow representation at runtime for a specific control-flow path. In contrast, The IPDA framework encodes facts about all possible control-flow paths into its symbolic representations of program statements without losing accuracy. A result similar to Moon et al.'s can, in principle, be achieved by the IPDA framework.

6.5 Loop Dependence Analysis

Industrial-strength compilers, such as the LLVM Compiler Infrastructure and the IBM XL C/C++/Fortran Compilers, use a near-complete implementation of Goff-Kennedy-Tseng dependence testing [18]. For a single-index addressing expression, exact tests are typically used that treat most commonly occurring single-index expressions as special cases for which efficient closed-form solutions are implemented. For linear addressing expressions, dependence testing is often reduced to finding integer solutions to systems of linear Diophantine equations. Implementations of the Goff-Kennedy-Tseng work include a limited variety of 'symbolic' tests. One such test processes addressing expressions that contain no index variables and can be symbolically tested for equality. Another handles expressions of the form ¡$ai + c_1, ai' + c_2$¿ that contain a single index variable i, with loop-invariant symbolic additive constants c_1 and c_2, where the difference $c_2 - c_1$ can be reduced to a constant. These techniques are restricted to a small subset of addressing expressions that conform to a very specific format. In contrast, the IPDA analysis scales to arbitrary addressing expressions and to many index variables present in these expressions, enabling the analysis of deep loop nests.

The work most similar to our proposed DDG algorithm technique is the Range Test by Blume and Eigenmann [4]. The Range Test propagates ranges to symbolic values to determine potential overlap of two addressing expressions across iterations of a given loop. It then computes the minimum and the maximum difference between addressing expressions across multiple loops and checks whether the maximum value for one expression is less than or equal to the minimum value of the other. Whereas the IPDA Test first computes algebraic differences between symbolic representations of the two subscripts, and then methodically reduces iteration point differences for all loop subsets in a given nest to verify if the difference can be zero. Moreover, the Range Test fails for subscript expressions that reference conditionally defined variables - a limitation the IPDA framework does not have.

Engelen et. al. propose a symbolic loop analysis framework for nonlinear dependence testing based on a representation of symbolic expressions with chains of recurrences (CRs) [14]. Their framework handles variable and pointer updates in conditional paths inside the loop body by constructing a set of CR forms for a conditionally defined variable, where each set element corresponds to the CR form of a given program path. Bounding functions for the range of the given variable are then constructed for a set of CR forms, instead of an individual CR form. Indexing expression range analysis is then performed, similarly to the Range Test, over sets of characteristic functions. The CR-set technique may handle some code patterns described in the paper that are common to DSP codes (boundary checks); however, range analysis over sets of characteristic functions has significant drawbacks that do not affect IPDA. Consider the case of:

```
int x; if (c) { x = 0; } ... if (c) { A[x] = ...; }
```

The set of CRs for the conditionally-defined variable `x` through code paths that lead to the array access `A[x]` will have the value of `x` range both 0 or any other possible integer value (for the case that the if condition does not hold). As a result, the analysis would not be able to infer any information about the array access. IPDA analysis is capable of capturing variable reference's dominating definitions and will determine the access to always be `A[0]`. Even without the definition propagation, the algebraic difference computation on the values of `A[x]` computed by different iterations would be solved to a 0 because the condition expressions would be canceled out. The symbolic difference simplification process often makes range computation much simpler.

Another popular methodology for dependence analysis is the Polyhedral model [5]. The polyhedral model treats loop iterations within loop nests as points in a lattice inside a polytope. This representation allows geometric modelling of any affine functions of indices that comprise the polytope. Dependence relations can then be established based on overlap of the resulting polytopes of memory location subscripts. A key limitation of the polyhedral representation, one that does not impact IPDA, is its restriction to spaces of affine functions of index variables. Moreover, expressions containing variables defined in conditional execution paths are intractable by the Polyhedral model.

Chapter 7

Conclusion

This thesis presents a collection of ideas and techniques aimed at extracting performance from accellerator-enabled heterogeneous computing machines through application of compile-time and runtime techniques.

Chapter 3 explored a series of code-restructuring transformations that can be executed by the programmer or the compiler, to better fit the structure of program code to the peculiarities of GPU architecture. Advantages of the proposed approaches include the opportunity to tailor grid geometry of each kernel to the parallel region that it executes and the elimination of the overheads imposed by a code-generation scheme meant to handle multiple nested parallel regions.

Chapter 4 championed a symbolic static analysis framework capable of characterizing memory access statements in GPU-bound parallel loops. This analysis can propagate definitions through control flow, works for non-affine expressions, and is capable of analyzing expressions that reference conditionally-defined values. Experimental results demonstrated potential for dramatic performance improvements. A highlight performance improvement result is an opportunity detected by the analysis framework in a SPEC ACCEL benchmark yielding kernel speedup of 111.1$\times$ with a benchmark improvement of 2.3$\times$ on an Nvidia V100. This thesis also demonstrated how architecture-aware compilers improve code portability and reduce programmer effort.

Finally, Chapter 5 argued that the transition to accelerator-based platforms demands an analytical approach for performance modeling, aiming to select the most suited accelerator out of those present in a system, to execute a given computation. The use of a hybrid analytical performance modelling is positioned as the most practical way forward in building fast and efficient methods to select an appropriate target for a given computation kernel. The target selection problem had been addressed in the literature, however there had been a strong emphasis on building empirical models with machine-learning techniques. We argued that the practical applicability of such solutions is severely limited in production systems. A comprehensive comparison evaluation of difference in GPU kernel performance on devices of multiple generations of architecture strengthened the need for accurate analytical performance models and provided insights in the evolution of GPU accelerators as a predominant computational platform. This thesis also highlighted a drawback of existing approaches to modelling GPU performance — accurate modelling of memory coalescing characteristics. To that end, we demonstrated a novel application of an inter-thread difference analysis to further improve analytical models. Finally, this thesis presents a prototype study of an OpenMP runtime framework for target-offloading target selection.

References

[1] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Intern. symp. on performance analysis of systems and software (ISPASS)*, Apr. 2009, pp. 163–174. 5

[2] U. K. Banerjee, *Dependence analysis for supercomputing*. Norwell, MA, USA: Kluwer Academic Publishers, 1988. 47, 49

[3] M. Bauer, H. Cook, and B. Khailany, "CudaDMA: Optimizing GPU memory bandwidth via warp specialization," in *High performance computing, networking, storage and analysis SC*, Seattle, WA, USA, 2011, pp. 1–11. 11

[4] W. Blume and R. Eigenmann, "The range test: A dependence test for symbolic, non-linear expressions," in *Proceedings of supercomputing '94*, Washington, DC, USA, Nov. 1994, pp. 528–537. 91

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Programming language design and implementation (PLDI)*, Tucson, AZ, USA, 2008, pp. 101–113. 92

[6] J. Bull, "Measuring synchronization and scheduling over- heads in openmp," in *The european workshop of OpenMP(EWOMP)*, Lund, Sweden, 1999. 74

[7] J. M. Bull, F. Reid, and N. McDonnell, "A microbenchmark suite for OpenMP tasks," in *Proceedings of the 8th international conference on OpenMP in a heterogeneous world(IWOMP)*, Rome, Italy, 2012. 74

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE international symposium on workload characterization (IISWC)*, 2009, pp. 44–54. 27

[9] T. E. Cheatham and J. A. Townley, "Symbolic evaluation of programs: A look at loop analysis," in *Symposium on symbolic and algebraic computation*, Yorktown Heights, New York, USA, 1976. 89

[10] A. Chikin, T. Lloyd, J. N. Amaral, and E. Tiotto, "Compiler for restructuring code using interation-point algebraic difference analysis," Patent Reference: P201706298US01, Filled on March 12, 2018, Apr. 2018. 79

[11] A. Chikin, *Unibench for OpenMP 4.0*, `https://github.com/artemcm/Unibench`. 27

[12] O. L. Committee, *OpenMP application program interface version 4.0*, `http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf`, Accessed: 2018-03-13. (visited on 03/13/2018). 35

[13] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE computational science and engineering*, Jan. 1998. 1

[14] R. A. van Engelen, J. Birch, Y. Shou, B. Walsh, and K. A. Gallivan, "A unified framework for nonlinear dependence testing and symbolic analysis," in *International conference on supercomputing (ICS)*, Malo, France, 2004, pp. 106–115. 91

[15] *EPCC OpenMP micro-benchmark suite*, Accessed: 2018-01-01. [Online]. Available: `https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openmp-micro-benchmark-suite`. 75

[16] T. Fahringer, M. Gerndt, G. Riley, and J. L. Träff, "Formalizing OpenMP performance properties with ASL," in *High performance computing*, 2000, pp. 428–439. 89

[17] M. P. Gerlek, E. Stoltz, and M. Wolfe, "Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form," *ACM transactions on programming languages and systems (TOPLAS)*, vol. 17, no. 1, pp. 85–122, Jan. 1995. 89

[18] G. Goff, K. Kennedy, and C.-W. Tseng, "Practical dependence testing," in *Programming language design and implementation (PLDI)*, Toronto, ON, Canada, 1991, pp. 15–29. 49, 90

[19] M. Haghighat and C. Polychronopoulos, "Symbolic program analysis and optimization for parallelizing compilers," in *Workshop on languages and compilers and parallel computing (LCPC)*, Portland, OR, USA, 1993, pp. 538–562. 89

[20] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th annual international symposium on computer architecture*, ser. ISCA '09, Austin, TX, USA, 2009, pp. 152–163. 67, 76, 77, 81

[21] Intel, *Avoiding and identifying false sharing among threads*, `https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads`, Accessed: 2018-03-13. 62

[22] *Intel® architecture code analyzer*, Accessed: 2018-09-01. [Online]. Available: `https://software.intel.com/en-us/articles/intel-architecture-code-analyzer`. 75

[23] A. C. Jacob, A. E. Eichenberger, H. Sung, S. F. Antao, G. T. Bercea, C. Bertolli, A. Bataev, T. Jin, T. Chen, Z. Sura, G. Rokos, and K. O'Brien, *Clang-YKT source-code repository*, `https://github.com/clang-ykt`. 12

[24] ——, "Efficient fork-join on GPUs through warp specialization," in *High performance computing HiPC*, Jaipur, India, 2017, pp. 358–367. 7–9, 12, 13

[25] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking," *Arxiv e-prints*, Apr. 2018. arXiv: `1804.06826`. 78

[26] G. Juckeland, W. Brantley, S. Chandrasekaran, B. Chapman, S. Che, M. Colgrove, H. Feng, A. Grund, R. Henschel, W.-M. W. Hwu, H. Li, M. S. Müller, W. E. Nagel, M. Perminov, P. Shelepugin, K. Skadron, J. Stratton, A. Titov, K. Wang, M. van Waveren, B. Whitney, S. Wienke, R. Xu, and K. Kumaran, "SPEC ACCEL: A standard application suite for measuring hardware accelerator performance," in *High performance computing systems. performance modeling, benchmarking, and simulation*, S. A. Jarvis, S. A. Wright, and S. D. Hammond, Eds., 2015. 58

[27] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for GPGPUs," in *Parallel architectures and compilation techniques PACT*, Piscataway, NJ, USA, 2013, pp. 157–166. 87

[28] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram, "Warped-preexecution: A GPU pre-execution approach for improving latency hiding," in *High performance computer architecture HPCA*, Barcelona, Spain, 2016, pp. 163–175. 88

[29] T. Komoda, S. Miwa, and H. Nakamura, "Communication library to overlap computation and communication for OpenCL application," in *Parallel and distributed processing symposium workshops IPDPSW*, Shanghai, China, 2012, pp. 567–573. 87

[30] L. Lamport, "The parallel execution of do loops," *Communications of the ACM*, vol. 17, no. 2, pp. 83–93, Feb. 1974. 49

[31] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving GPGPU resource utilization through alternative thread block scheduling," in *High performance computer architecture HPCA*, Orlando, FL, USA, 2014, pp. 260–271. 88

[32] S.-Y. Lee, A. Arunkumar, and C.-J. Wu, "CAWA: Coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads," in *International symposium on computer architecture (isca)*, ACM, Portland, Oregon, 2015, pp. 515–527. 88

[33] C. Liao and B. Chapman, "Invited paper: A compile-time cost model for OpenMP," in *2007 IEEE international parallel and distributed processing symposium*, Mar. 2007, pp. 1–8. 74

[34] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng, “OpenUH: An optimizing, portable OpenMP compiler: Research articles,” *Concurr. comput. : Pract. exper.*, pp. 2317–2332, Dec. 2007. 74

[35] *Libhugetlbfs - preload library to back text, data, malloc() or shared memory with hugepages*, Accessed: 2018-09-01. [Online]. Available: `https://linux.die.net/man/7/libhugetlbfs`. 75

[36] T. Lloyd, K. Ali, and J. N. Amaral, “GPUCheck: Detecting CUDA performance problems with static analysis,” Manuscript under review, Aug. 2018. 9, 36

[37] T. Lloyd, A. Chikin, J. N. Amaral, and E.Tiotto, “Automated GPU grid geometry selection for OpenMP kernels,” in *Workshop on applications for multi-core architectures*, ser. WAMCA 2018, Pre-print Manuscript. Available: `https://webdocs.cs.ualberta.ca/~amaral/papers/LloydWAMCA18.pdf`, Lyon, France, Sep. 2018. 6, 23, 26, 28, 30, 31, 81,

[38] *LLVM-MCA - LLVM Machine Code Analyzer*, `https://llvm.org/docs/CommandGuide/llvm-mca.html`, Accessed: 2018-09-13. 67, 75

[39] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, “Effective compiler support for predicated execution using the hyperblock,” in *Micro*, Portland, OR, USA, 1992, pp. 45–54. 9

[40] J. Meng and K. Skadron, “Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs,” in *International conference on supercomputing (ICS)*, Yorktown Heights, NY, USA, 2009, pp. 256–265. 89

[41] D. Miles, D. Norton, and M. Wolfe, “Performance portability and OpenACC,” in *Conference of cray user group (CUG)*, Lugano, Switzerland, 2014. 89

[42] R. Mokhtari and M. Stumm, “Bigkernel – high performance CPU-GPU communication pipelining for big data-style applications,” in *International parallel and distributed processing symposium IPDPS*, Phoenix, AZ, USA, 2014, pp. 819–828. 86

[43] S. Moon, M. W. Hall, and B. R. Murphy, “Predicated array data-flow analysis for run-time parallelization,” in *International conference on supercomputing (ICS)*, Melbourne, Australia, 1998, pp. 204–211. 90

[44] Nvidia, *NVIDIA TESLA V100 GPU ARCHITECTURE – The World’s Most Advanced Data Center GPU.* `http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, Accessed: 2018-01-01. 4

[45] M. F. P. O’Boyle, Z. Wang, and D. Grewe, “Portable mapping of data parallel programs to OpenCL for heterogeneous systems,” in *Intern. symp. on code generation and optimization (CGO)*, Shenzhen, China, 2013, pp. 1–10. 89

[46] Y. Oh, K. Kim, M. K. Yoon, J. H. Park, Y. Park, W. W. Ro, and M. Annavaram, "APRES: Improving cache efficiency by exploiting load characteristics on GPUs," *ACM SIGARCH computer architecture news*, vol. 44, no. 3, pp. 191–203, 2016. 88

[47] *OpenMP Technical Report 6: Version 5.0 Preview 2*, Accessed: 2018-09-01. [Online]. Available: `https://www.openmp.org/wp-content/uploads/openmp-TR6.pdf`. 84

[48] *Power9 Processor User's Manual*, Accessed: 2018-09-01. [Online]. Available: `https://openpowerfoundation.org/?resource_lib=power9-processor-users-manual`. 74

[49] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th annual international symposium on microarchitecture*, ser. MICRO 27, San Jose, California, USA: ACM, 1994, pp. 63–74. 13

[50] D. Rolls, C. Joslin, and S.-B. Scholz, "Unibench: A tool for automated and collaborative benchmarking," in *International conference on program comprehension (ICPC)*, Braga, Minho, Portugal, 2010, pp. 50–51. 58, 73

[51] S. Rus, D. Zhang, and L. Rauchwerger, "The value evolution graph and its use in memory reference analysis," in *Parallel architectures and compilation techniques (PACT)*, Antibes Juan-les-Pins, France, 2004, pp. 243–254. 90

[52] A. Sethia, D. A. Jamshidi, and S. Mahlke, "Mascar: Speeding up GPU warps by reducing memory pitstops," in *High performance computer architecture HPCA*, San Francisco, CA, USA, 2015, pp. 174–185. 88

[53] A. Sethia and S. Mahlke, "Equalizer: Dynamic tuning of GPU resources for efficient execution," in *International symposium on microarchitecture (MICRO)*, Cambridge, UK, 2014, pp. 647–658. 87

[54] *Sphinx*, Accessed: 2018-01-01. [Online]. Available: `http://www.llnl.gov/casc/sphinx/sphinx.html`. 74

[55] A. Stoutchinin and F. de Ferriere, "Efficient static single assignment form for predication," in *Micro*, Austin, TX, USA, 2001, pp. 172–181. 9

[56] *Oak Ridge Readies Summit supercomputer for 2018 debut*, `https://www.top500.org/news/oak-ridge-readies-summit-supercomputer-for-2018-debut/`, Accessed: 2018-03-13. (visited on 01/01/2018). 1

[57] *The OpenACC application programming interface*, `https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf`, Accessed: 2018-01-01. 15

[58] *The OpenACC application programming interface version 2.5*, Accessed: 2017-10-15. [Online]. Available: `http://www.openacc.org/sites/default/files/inline-files/OpenACC_2pt5.pdf`. 1

[59] *Top500 supercomputers*, `https://www.top500.org/`, Accessed: 2018-01-01, 2017. 1

[60] *Trinity*, `https://lanl.gov/projects/trinity/`, Accessed: 2018-08-08, 2018. 1

[61] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: A machine learning based approach," in *Principles and practice of parallel programming (PPoPP)*, Raleigh, NC, USA, 2009. 89

[62] M. E. Wolf, D. E. Maydan, and D.-K. Chen, "Combining loop transformations considering caches and scheduling," in *International symposium on microarchitecture (MICRO)*, Dec. 1996, pp. 274–286. 74

[63] M. J. Wolfe, "Optimizing supercompilers for supercomputers," AAI8303027, PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1982. 47, 49

[64] Z. Zheng, X. Chen, Z. Wang, L. Shen, and J. Li, "Performance model for OpenMP parallelized loops," in *Transportation, mechanical, and electrical engineering (TMEE)*, Changchun, China, Dec. 2011. 89