Compiler-Only Code Generation for Performant and Modular Matrix-Multiplication Micro Kernels Using Matrix Engines

by

Braedy Kuzma

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

Master of Science

Department of Computing Science

University of Alberta

Abstract

General Matrix-Matrix Multiplication (GEMM) is used widely in many high-performance application domains. In many cases, these applications repeatedly execute their matrix-multiplication subroutine, as is the case in the implementation of a particle-physics simulator or the repeated convolutions of many deep-learning models. This reliance on repeated executions causes matrix-multiplication operations to be a computational bottleneck in these applications, creating a strong motivation to improve the performance of GEMM.

The state of the art for the efficient computation of GEMM consists of manual, programmer-directed replacement of matrix multiplication with calls to highly optimised Basic Linear Algebra Subprograms (BLAS)-like libraries which contain kernels painstakingly written in assembly. Beyond a clear expertise barrier for porting each kernel to each iteration of a specific platform – and thus a maintenance issue – such a replacement creates a dependency on external code over which a developer has no control. Moreover, calls to an unknowable library function disable critical optimisations such as inlining and loop fusion that can enable further optimisations in the calling code.

The solution to these issues is to provide an alternative for the computation of matrix-multiplication, with competitive performance, directly within the compiler. An implementation in this style automatically generates a matrix-multiplication kernel that benefits from all applicable code transformations available in the compiler. This thesis addresses the lack of an efficient compiler-only path to generate code for GEMM by investigating and implementing a

high-performance matrix-multiplication kernel implementation directly within the LLVM™ compiler framework. Furthermore, the proposed solution integrates emerging technologies, namely the matrix engine, that provide hardware assistance for the computation of matrix multiplication. In particular, the recent POWER10 processor features one such extension named Matrix Math Assist (MMA). Its unique design choice to implement matrix multiplication through the computation of outer products presents new opportunities to improve performance.

The generation of efficient code for matrix multiplication in the LLVM compiler framework is divided into two levels: the macro kernel and the micro kernel. The main goal of the macro-kernel code generation is to make the best use of the memory hierarchy when bringing the operands from the main memory to the highest-level of cache memory. The focus of the micro-kernel code generation is to make efficient use of Single Instruction, Multiple Data (SIMD) functional units and to reduce the memory-register data-transfer requirements by increasing data reuse. This thesis focuses on the micro-kernel code generation, though a compiler-only macro-kernel code generation developed as part of a large work is available.

This thesis also contributes a detailed performance study that indicates that this new code-generation strategy results in speed improvements between 3.1 and 15.8 times when compared with the closest alternative compiler-only code-generation implementation for some data types. There is also strong indication that, given several improvements in the compiler assembly-code generation, the compiler-generated kernel can match the performance of an expert's handcrafted solution. This thesis also features a detailed analysis of the experimental results that reveals opportunities for changes in the compiler that have the potential to lead to improvements in the entire POWER compilation stack.

Preface

Chapter 3 and Section 6.3 contain content that has been extracted from a multi-author manuscript that is currently being revised [41]. The portions of Chapter 3 that have been used in this work were drafted by a co-author and edited by all authors. Due to the large amount of overlap between related sources, only a small amount of editing was performed to make the statements more relevant to this manuscript. Several new sources not in the multi-author document have also been added in Chapter 3.

The content in Section 6.3 that was duplicated from the original manuscript was written and drafted by myself before being edited by myself and my co-authors. The version contained in this thesis contains additional details in the description of the code-generation design and additional experimental results and analysis that are not included in the multi-author manuscript. The plan at the time of writing is for the manuscript to be further revised and to be submitted to a major international journal in the area of code generation.

This multi-author effort focuses on creating a more efficient compiler-only code-generation path for GEMM in the LLVM framework. The work in that manuscript addresses the much broader issue of efficient and large matrix multiplication on multiple platforms with varying memory and hardware constraints. My contribution to that work, the same contribution described in much greater detail in this thesis, is only one facet of the investigation developed by myself and my co-authors. Overall, my contribution to that work is the implementation of an efficient and performant matrix-multiplication kernel that makes use of MMA, allowing for the examination and comparison of the effects of matrix engines on large-scale matrix-multiplication kernels.

To my friends and family
For bearing through it all with me.

To Kirsten
For supporting me through to the end.

And especially, to my mother, Janice
For guiding me in all aspects of life.

Everyone who uses a computer frequently has had, from time to time, a mad desire to attack the precocious abacus with an axe.

– John Drury Clark, *Ignition!: An informal history of liquid rocket propellants*, 1972

Acknowledgements

First and foremost, I would like to thank my supervisor, J. Nelson Amaral, whose constant support has helped me finish the largest undertaking yet seen in my life. His passion for teaching has helped foster my love for all things computers more than I could have ever known.

Collectively, I would like to thank João Paulo Labegalini de Carvalho and Ivan Korostolev for being companions throughout my time as a student. They were always ready to be a sounding board for new ideas and offered encouragement and assistance when solving difficult problems. I looked forward to my time in the lab thanks to these two.

I would also like to thank the team at IBM who supported my research, notably Kit Barton and Jose Moreira, both of whom were essential in the completion of this thesis. Every resource and bit of knowledge they had was made available with a simple message. I must also acknowledge the backend team, led by Nemanja Ivanovic, for their aid and their tolerance as I played with and broke the things under their care.

Funding was necessary to complete the work contained in this thesis; without support from the IBM Canada Centre for Advanced Studies and the Natural Sciences and Engineering Research Council of Canada through their Collaborative Research and Development program, none of this would have been possible.

Contents

List of Tables

List of Figures

List of Algorithms

List of Listings

Glossary

A

AltiVec

A SIMD instruction set designed for use with single-precision floating point and integer values.

B

Basic Block

A straight-line code sequence where only the final statement is some sort of control flow and only the first statement may receive control flow. Alternatively, an atomic block of statements where if the first statement is executed, all statements in the block must be executed. Basic blocks form the nodes in a control flow graph.

Broadcast

The act of duplicating a scalar value to all lanes in a vector register.

Builtin

A function callable from a higher-level language (e.g. C/C++) which encapsulates a well-known functionality with well-defined semantics. Such a function may be implemented in header files, as language-specific keywords, or conditionally available functions.

D

Dead

A value in a program is dead if it is guaranteed that the value will never be used again. A static liveness analysis is performed to determine which values are dead at a given point in a program. Therefore, a value that is dead at a given point of the program must be dead for all possible executions of the program. A variable may be dead or a register may be dead.

I

Integrated Circuit

A set of electronic circuits built into a semiconductor, typically silicon. These circuits are set in place and cannot be changed.

Intrinsic

A generic function within an IR which encapsulates a well-known functionality with well-defined semantics. The function will eventually be lowered to either a call to an existing function or inline code. Some compilers (e.g. Microsoft Visual C++) may use this term to refer to a builtin.

L

Lane

A way to refer to an element of a vector register or operation. Derived from the way circuit pathways leave and enter vector functional units in groups corresponding to an element, like the lanes of a roadway. The number of lanes in a vector register, which has constant width, changes based on element width, i.e. a 128 bit register with 32 bit elements has four lanes, with 64 bit elements it has two lanes.

Linking

Achieved through a program called a "Linker", this is typically the final step in the compilation process. The procedure involves combining all object files into a single binary, resolving symbols, as well as several more involved processes.

Live

A value is live at a point in a program if there exists at least one path from this point that may use the value.

LLVM

The name of an umbrella project of compiler technologies including an IR, debugger, a variety of machine backends, etc. May also be used to refer specifically to the combination of optimiser and backend as part of the lowering process. See Section 2.3.

Lowering

The process of transforming one representation of a program to another representation that encodes the same semantics but is closer to a final product. Typically begins with source code in a high level language and ends with binary machine code; transitional steps often include a language agnostic IR and an ISA specific assembly listing.

M

Mangle

The act of encoding extra information into a string such that the result is unique and the information recoverable.

Matrix Engine

A general term describing a set of new facilities on the most recent generation of CPUs which focus on matrix operations.

O

Object File

The result of compiling a translation unit into a binary. Compilers may skip the object file and directly generate an executable if a program consists of a single translation unit.

P

Point

A point is a concept in program analysis defining places where a program can be paused and the state can be examined. A point exists between each logical statement (e.g. between instructions in assembly, between statements in a higher level language) as well as before and after control flow split and join points.

POWER

IBM's high performance computing architecture built for HPC and other applications.

POWER10

The tenth generation of IBM's POWER architecture announced in August 2020.

R

Rematerialisation

The process of computing a value multiple times rather than leaving it in a register or spilling it.

S

Spill

The act of storing a live register to memory temporarily in order to free the register for another value.

T

Translation Unit

A single input file to a compiler after the preprocessing step. All `include` directives have been resolved and replaced and preprocessor commands processed.

V

Vectorisation

The process of transforming a loop in order to perform multiple iterations simultaneously in a SIMD architecture.

Acronyms

A

Advanced Matrix Extension (AMX)

The matrix engine extension for the x86 ISA.

Application-Specific Integrated Circuit (ASIC)

An integrated circuit built for a specific use instead of general purpose use.

B

Basic Linear Algebra Subprograms (BLAS)

Originally a set of highly-optimised, low-level routines used as building blocks in many larger linear algebra operations [45]. It has since developed into a standard interface for higher (matrix-matrix) and lower level (scalar/vector, vector/vector) linear algebra operations with many open source and proprietary implementations.

C

Central Processing Unit (CPU)

The main component of a computer which interacts with and controls all other subcomponents according to the instructions contained in a program.

Common Subexpression Elimination (CSE)

An optimisation which searches for identical expressions and potentially replaces them with a single variable [13], [60].

Cycles Per Instruction (CPI)

The number of CPU clock cycles required to complete an instruction. Often used as a performance metric where it refers to the ratio of total instructions executed to total clock cycles.

F

Field-Programmable Gate Array (FPGA)

An integrated circuit which is configurable post-production via reconfigurable interconnects between logic blocks which perform operations.

Fused Multiply-Add (FMA)

An instruction which performs the operation $w = x \times y + z$ often used to perform accumulation operations as in $w \mathrel{+}= x \times y$. SIMD equivalents exist, performing the same operation per lane of the operand vectors.

G

General Matrix-Matrix Multiplication (GEMM)

A more general form of matrix multiplication with a scaling factor for accumulation and multiplication, i.e. $C' = \alpha AB + \beta C$ where C' is the new value of C after accumulation.

Graphics Processing Unit (GPU)

Originally designed to accelerate image creation for display devices, it was soon repurposed into a general purpose, high performance compute device.

H

High Performance Computing (HPC)

The area of computing concerned with performing large scale computations in the most efficient way possible through the minimisation of required time and resources.

I

Instruction Set Architecture (ISA)

The abstract model of a computer, including registers, supported data types, instructions, etc.

Intermediate Representation (IR)

A language-agnostic representation of a program encoding all of the semantics of the original program and potentially annotated with debugging and optimisation information.

International Business Machines (IBM)

A multinational technology company with a long history in the computing domain.

M

Matrix Math Assist (MMA)

An extension to the POWER ISA's SIMD capabilities introduced in POWER10. See Chapter 5.

P

Power PC (PPC)

An old name for the POWER ISA which is still in use in some contexts as a shorthand.

R

Reduced Instruction Set Computer (RISC)

A computer focused on a small, simple, and optimised instruction set meant to enable the processor's pipeline to have a low CPI.

S

Scalable Vector Extension (SVE)

An extension to ARM's NEON vector extension which enables programs to run on hardware with differing vector length without recompilation.

Single Instruction, Multiple Data (SIMD)

Used to describe an architecture with data-level parallelism capabilities, i.e. able to perform the same operation on multiple units of data simultaneously.

Static Single Assignment (SSA)

A programming paradigm common to IRs where each variable in a program is assigned to a single time in the program text. This format makes reasoning about code much easier for many optimisations. See Section 2.3.1.

T

Tensor Processing Unit (TPU)

An external ASIC for accelerating AI designed by Google [1].

Translation Lookaside Buffer (TLB)

The hardware cache responsible for aiding fast virtual to physical address translation. Typically implemented as a key-value map where the key is a page number and the value is the frame in which that page is resident.

V

Vector-Scalar Register (VSR)

The 64 128-bit registers defined as part of VSX. They can be interpreted as vectors of two 64-bit, four 32-bit, eight 16-bit, or 16 8-bit elements.

Vector Multimedia Extension (VMX)

An extension to the POWER ISA implementing the AltiVec standard.

Vector Scalar Extension (VSX)

An extension to the POWER VMX enabling even greater SIMD capabilities.

Chapter 1

Introduction

Matrix multiplication is a critical basic operation in many High Performance Computing (HPC) and AI workloads. Given the surge of popularity in AI and the growing scale of HPC tasks and simulations, the optimisation of matrix multiplication can mean an essential reduction in computing time. Waugh and McIntosh-Smith demonstrate that for small thread counts, General Matrix-Matrix Multiplication (GEMM) operations often dominate execution time in a set of benchmarks found to be representative of current workloads [71]. While multithreading reduces these functions' contribution to overall runtime significantly, they conclude that, in a soon-to-be-exascale future, applications will adapt to use more GEMM operations, expanding this portion considerably.

At the center of our capability to handle these expanding workloads is the matrix engine. Matrix engines are a new accelerator facility, found in the most recent generation of Central Processing Units (CPUs), that focuses on accelerating matrix multiplication. Currently, acceleration is done via external accelerators or via vector extensions to an Instruction Set Architecture (ISA). An external accelerator may be an Application-Specific Integrated Circuit (ASIC) specialised to the task (e.g. Google™'s Tensor Processing Unit (TPU)™ [1]), a Graphics Processing Unit (GPU), or a Field-Programmable Gate Array (FPGA). Vector extensions are part of the movement towards the Single Instruction, Multiple Data (SIMD) computing paradigm where multiple pieces of data are used and produced by a single instruction (see Section 2.1). An advantage of vector extensions in relation to external accelerators is that

a vector extension can access data through the same memory hierarchy used by the CPU while most external accelerators require data transfer through an interconnect with slower transfer speeds.

The current state of the art when working with matrix operations is to choose one of several libraries that implement the Basic Linear Algebra Subprograms (BLAS) interface (e.g. OpenBLAS [74], IBM's ESSL [33], Intel®'s MKL [36], [68], Nvidia®'s cuBLAS [55]). These libraries can provide incredible speedups and, in parallel architectures, automatic parallelisation: a very attractive feature for large workloads. It has been a natural extension of these libraries to include the usage of matrix engines when targeting CPUs. However, the development of most of these libraries rely on difficult-to-maintain assembly programs and their usage imposes extra requirements on systems.

All high-performance implementations of BLAS-like libraries have hand-written assembly kernels at their core, though the extent of the kernel varies [76]. Nevertheless, each of these kernels must be produced and hand tuned for each new CPU that the library needs to support. Creating these kernels requires engineers who are extremely knowledgeable about the ISA and the architectural details of a target CPU. Thus, maintaining the code surrounding such a kernel as well as the kernel itself requires significant ongoing effort as developers try to obtain greater performance from the implementation.

Additionally, using a library means that user code now has an external dependency. In applications where performance and correctness are critical and must be tightly controlled by the developer, using libraries may be an impossibility. Similarly, while finding a BLAS implementation on major platforms is often quite easy, some target platforms may not have an available implementation with which this dependency can be fulfilled [76]. This is a direct product of the manual kernel porting difficulties described above.

Therefore, as a counterpoint, a method of transparently accelerating matrix multiplication via a compiler-only path is important for portability and maintainability. When a matrix-multiplication kernel is created as part of a compiler, it no longer needs to be written in assembly. Relaxing this requirement means that kernels become *CPU agnostic* rather than *CPU dependent*.

It will not be *ISA agnostic* because a kernel, to obtain best performance, will still need to be written in terms of an ISA's vector extension, tying it to the architecture, but it will not be tied to a specific version of the ISA. The change to CPU agnostic does, however, mean that optimisations that are based on architectural details such as register count or available functional units, which would typically be done manually in a handwritten kernel, can now be automatically performed and tuned to the CPU by the compiler. This automation means that an appropriately written and parameterised kernel will also still be optimal in future hardware iterations.

Kernels written as part of a compiler are also implicitly subject to all of the optimisations available in the compiler, now and in the future. Currently, these include optimisations such as loop interchange, blocking, and unrolling which are known to have significant effects on matrix multiplication speed [4], [18], [61], [67]. For example, Velkoski et al. demonstrate a $\sim 2.5\times$ speedup using unrolling alone for certain parameters; interchange and blocking have a much greater effect on memory-heirarchy usage, providing greater benefits as the operation size grows. In the backend, improvements to processes such as register allocation and instruction scheduling (reordering/pipelining) will be retroactively available to these kernels as well, completely transparently to the user.

Furthermore, a crucial optimisation, loop fusion, is impossible with the current paradigm of library function calls. Mathematically, consecutive matrix operations (e.g. $D = ABC$) can be fused so as to perform both computations at once instead of producing a temporary result (i.e. $\mathrm{T} = AB; D = TC$). Fusing operations in this manner significantly reduces the memory movements required by removing the need to store and reload the whole-matrix temporary value. Moreover, given sufficiently large matrices, it is likely that portions of the matrix have already been forced out of the cache. A compiler has the foresight and tools to fuse these operations and then further optimise the resulting code.

A compiler-only solution for matrix multiplication, while benefitting from preexisting features of a compiler, nevertheless requires knowledge of the com-

piler framework, target operation, and the target architecture; forethought based on the constraints derived from this knowledge; and an actionable design built from this forethought. As in a handwritten solution, required knowledge includes an understanding of the matrix engine and vector extensions of the target architecture as well as how they interact with each other and with the memory hierarchy. However, because the kernel is written at a higher level within the compiler, it counters many of the issues associated with the hand-written assembly version. Authors are now more productive by only having to focus on writing the kernel: registers and the data in them are now automatically managed, the instruction schedule can be optimised by the compiler to hide latencies, loops are unrolled according to architecture capacity, and more. The produced kernels are more portable, functioning on CPUs with the same ISA (in some cases cross-ISA) with much of the infrastructure being portable between architectures. Finally, maintenance is also easier: higher level languages are more readable with less expertise required to interact with them.

The method of delivering the kernel must also be considered. Libraries are the currently preferred method of acceleration and therefore any replacement method should aim to require less effort if it hopes to be adopted. Library download and installation is oftentimes a simple barrier to overcome for the average user whereas the majority of users are loath to find and benchmark combinations of compiler flags in an effort to optimise their program. They would prefer instead to assume that the preset, curated optimisation levels (i.e. `-O1`, `-O2`, `-O3`) are sufficient. Therefore, a compiler-only solution should strive to be as transparent as possible when it comes to enabling easy adoption.

Chapter 2

Background

This chapter introduces concepts that are necessary for subsequent chapters. SIMD architectures and vectorisation are concepts critical to every facet of the work. This chapter also explains the LLVM framework, in which this work is implemented, as well as the Intermediate Representation (IR) used in the LLVM framework.

2.1 SIMD Architecture

First introduced as part of the ILLIAC IV [8], processors capable of SIMD computation were a crucial advancement in high-performance computing history. Flynn characterised a SIMD processor as having the ability to apply a single "master" instruction over a vector of related operands [20]. Such an architecture is desirable for its ability to increase the throughput of repeated operations through data-level parallelism and is thus highly applicable to one of the most standard computing control flow mechanisms: loops. This process of increasing the throughput of loops, through what is essentially a reduction in the number of instructions executed, has developed into what is now the well-known and well-studied process of "vectorisation".

Vectorisation is the process by which a compiler merges multiple scalar operations into a single operation on a vector. Unrolling a loop exposes repeated operations that can be rescheduled and grouped. This extends loads and stores which, when combined, produce only a single (wider) memory operation. Repeated memory requests are queued in a memory controller, waiting

for earlier requests to complete before being started. Requiring only a single request for the same amount of data can significantly reduce memory latency.

2.2 The IBM Power Architecture

International Business Machines (IBM)® has decades of history in computing, tracing their roots back to the 1880s; first incorporated as the Computing-Tabulating-Recording Company in 1911, it was eventually renamed to IBM in 1924 [35]. Throughout this history, IBM has made a point of innovating and pioneering numerous technologies in hardware, software, and the intermingling of the two. One such innovation was the POWER ISA.

The POWER® ISA was first announced in 1990 along with its primary instantiation in the IBM System/6000 [51]. The System/6000, a Reduced Instruction Set Computer (RISC), implemented new features such as register renaming and out-of-order execution via the Tomasulo algorithm [64]. Previously these features were only available in the IBM System/360 mainframe.

Years later, in August 2020, IBM remains competitive in its technology offerings with the announcement of the tenth generation of the POWER ISA, aptly named POWER10.

2.2.1 SIMD History in IBM Power

Initially, the POWER architecture implemented its SIMD capabilities through a common standard named AltiVec. AltiVec was designed through a collaboration between IBM, Apple, and Motorola and described a SIMD instruction set for floating-point and integer values. This implementation, presented under the name Vector Multimedia Extension (VMX), was first instantiated by IBM as part of POWER6 (POWER ISA v2.03) in 2007 [17] despite the standard being presented in 1999 [65].

Improvements for POWER7 (POWER ISA v2.06) added a new facility, called the Vector Scalar Extension (VSX), designed to add even further manipulation capabilities when dealing with vectors. This included support for up to 64 vector registers, 64-bit integers, and double-precision floating point

values. Both VMX and VSX exist to this day in the most recent version of the ISA (POWER ISA v3.1), though many refer to them collectively under the second name, VSX.

2.2.2 Matrix Math Assist

As part of the new POWER10's offerings, IBM has implemented a new facility, dubbed Matrix Math Assist (MMA), into the ISA, continuing IBM's commitment to providing cutting edge hardware for the software needs of the current era. MMA is an addition to POWER's preexisting VMX and VSX SIMD facilities. It provides a high-throughput and low-latency method for calculating matrix multiplications.

The state of the art for hardware that performs matrix multiplication falls into two categories: **(1)** via external components (e.g. GPU, FPGA, or ASIC such as Google's TPU) and **(2)** via CPU ISA extensions (e.g. x86's Advanced Matrix Extension (AMX) or ARM®'s NEON™/Scalable Vector Extension (SVE)). MMA belongs to the second category given that the facility is built directly into the CPU. Thus, the facility is amenable to tasks that would normally incur avoidable overhead, for example, while sending data to an external component such as a GPU. See Chapter 5 for further discussion.

2.3 LLVM

LLVM, originally an initialism for Low-Level Virtual Machine, is a compilation framework devised by Chris Lattner [43], [44]. Lattner originally positions LLVM as a "unique multi-stage optimisation system" that aims to "support extensive inter-procedural and profile-driven optimisations, while being efficient enough for use in commercial compiler systems" [44]. LLVM, now a mononym, has evolved into a project covering a wide range of compilation-related tools including frontends for many languages, an optimiser, backends for many platforms, a linker, debugger, and several other projects.

Frontends exist for a multitude of languages including C/C++ (`clang`™), Fortran (`flang`), Swift (`swiftc`), and Rust (`rustc`). Backends also exist for

a wide variety of platforms, such as x86, ARM, PowerPC, and WebAssembly. The LLVM optimiser, named `opt`, is a popular target for compiler research of all varieties, seeing advancements in various areas like register allocation [49], [57], pointer analysis [30], [63], and polyhedral optimisation [6], [26]. The framework has also been adopted by several large companies who have based their own products off the ever-improving set of open-source tools. Two examples of such products are IBM's XL C/C++ compiler and Nvidia's CUDA® compiler.

2.3.1 LLVM's Intermediate Representation

The main pillar upon which the LLVM's compilation pipeline is built is its IR. An IR is a programming language in its own right, though its intended use is a frontend-language-agnostic and backend-target-agnostic intermediary language. In this way, all frontends may target their lowering toward producing a single shared language and all backends may consume a single shared language to produce their platform specific assembly. Given this shared middle point, it is easy to create an optimiser which both consumes and produces the single intermediate language. Thus, any combination of high-level language and destination platform may benefit from new or improved optimisations in the optimiser.

LLVM IR is known for being strongly typed and maintaining much of the high level type information from the original input. It can also be annotated with large amounts of debugging information all while remaining easily serialisable. A critical feature of the IR is that it is in Static Single Assignment (SSA) form.

SSA was formalised by Cytron et al. [14] though its use was seen as a side effect in previous works [5], [60]. Efficient methods for its construction followed shortly after [11], [15]. Intuitively, the main property of a program in SSA form is that each variable in the program text (static) is assigned to exactly once (single assignment); this property does not preclude executing an assignment multiple times at runtime, potentially with different values.

```
int foo(int a) {
  int x = a + 2;
  int y = x + 10;
  x = y * 4;
  return x;
}
```

Listing 2.1: An simple example C program, pre-conversion to SSA.

```
define i32 @foo(i32 %a) {
entry:
  %x.1 = add i32 %a, 2
  %y.1 = add i32 %x.1, 10
  %x.2 = mul i32 %y.1, 4
  ret i32 %x.2
}
```

Listing 2.2: The same simple program, converted to LLVM IR in SSA format.

A simple C program, shown in Listing 2.1 is converted to the SSA format in Listing 2.2 via a simple renaming scheme. The first reference to a variable is suffixed with a one, and all future references increment the counter by one. This process continues for the entirety of a function body, regardless of any control flow. There are further complications in the construction algorithm in the presence of control flow but because the process presented in this thesis functions within a single basic block, this explanation is left to Cytron et al.'s work. This format is significantly more conducive to analysis and optimisation than the source language, allowing for a much easier data flow analysis.

Chapter 3

Related Work

The sections below present work related to separate facets of this thesis. Each section contains text derived from work currently under revision [41].

3.1 Matrix Multiplication

In a seminal work, Goto and van de Geijn detail a layered approach to improve cache and vector-register utilization on CPUs [24]. Using this approach, modern linear-algebra libraries, such as Eigen and OpenBLAS, achieve high performance on HPC workloads. Goto and van de Geijn show that modelling both L2 cache and Translation Lookaside Buffer (TLB) — and not only L1 cache as considered earlier — is crucial for cache performance. Their work is seminal because it publicly explained practical strategies for optimal cache and vector register utilization on CPUs; these strategies were previously only available in proprietary libraries. The layered strategy features two stages: **(1)** blocking input matrices and packing tiles of these blocks in such a way that tiles lay in main memory in the order that they will be accessed; and **(2)** computing a small GEMM at the register level. Kuzma et al. [41] is the first work to create a compiler-only code generation for the layered approach and adapts blocking, tiling, and packing to create a data layout that is suitable for computing with MMA and to also improve utilization of the L3 cache. This thesis is an in-depth presentation of the method behind **(2)**.

Gareev et al. [22] implement tiling and packing within Polly [25], [26] without the need for an external library or automatic tuning. Their approach with

a hand-optimized x86 SSE kernel reports performance on par with BLIS on Intel Sandy Bridge. When not relying on an assembly kernel, their pass uses the default LLVM vectoriser that delivers only a small speedup over naïve code. The work in this thesis presents a compiler-only kernel that can replace Gareev et al.'s SSE kernel or the default LLVM vectoriser on POWER10. Furthermore, this work is more modular and can be reused by new passes or other code generation paths as a drop-in inner-kernel.

Uday Bondhugula presents an implementation of the BLAS strategy within the emerging MLIR framework [10]. He demonstrates that blocking, packing, register tiling, and unroll/jam yields code that achieves 34% of OpenBLAS' performance on Intel's Coffee Lake [10]. Bondhugula also implemented a custom vectorisation pass to replace the default LLVM vectoriser thus reaching 91% of the performance of OpenBLAS. This work also shows the weakness of the default LLVM vectoriser. As well, because LLVM IR is a dialect within MLIR, using LLVM IR intrinsics is inherently an accessible method of implementing an inner kernel in MLIR.

Carvalho et al. introduce KernelFaRer, a robust pattern recognition system that can identify matrix-multiplication patterns in the LLVM IR level and can replace the matrix multiplication with library calls [12]. While this approach can lead to speedups on the order of 1000s in comparison with non-optimized code, it has the drawback of requiring the use of libraries in a computer system that may not have them installed. Moreover, their experimental results indicate that, for smaller matrices, the overhead of invoking functions in the libraries leads to performance degradations. The solution in this thesis is orthogonal to KernelFaRer. The pattern recognition in KernelFaRer can identify GEMM kernels at the intermediate-level representation and then replace the inner-most kernel with the solution presented here.

3.2 Code Generation for New Hardware

When presenting the ILLIAC IV, one of the first SIMD machines, Barnes et al. advocated that data parallelism would be crucial for progress [8], citing

matrix operations as a critical target [40]. Nearly 50 years later, Barnes' direction culminated in the inclusion of vector extensions in all mainstream CPUs such as IBM's VSX [34], Intel's AVX-512 [37], and ARM's NEON/SVE [7]. Although fast vectorisation is powerful, matrix-multiplication performance could be improved further with specialized hardware units. This possibility is now realized with the introduction of what Domke et al. have dubbed "matrix engines" [16], now available in IBM's MMA [34], Intel's AMX [37], and ARM's NEON/SVE [7]. This thesis, in the same vein as Barnes et al., focuses on bringing high performance to new hardware facilities.

3.3 Other Matrix Engines

The advent of the "general purpose" GPU quickly saw study and performance analysis of matrix computations [19], [42]. This evolved into implementations of matrix multiplications on GPUs: manually [46], through libraries like BLAS [54], and through frameworks such as DistME [29]. Matrix multiplication is also central to the design of hardware for tensor-operation acceleration such as Google's Tensor Processing Unit [38], Nvidia's Tensor Core [50], and Huawei's Cube Unit [47]. Performance evaluations of GEMM in tensor hardware are difficult to find because the studies of these devices focus on the benchmarking of various flavours of neural network [38], [70]. Matrix-multiplication acceleration in standalone accelerators is not the focus of this thesis.

It is difficult to find works explicitly discussing the outer product because kernels are often described in terms of Goto and van de Geijn's work. However, a work by Naohito Nakasato [53] and another by Wu Jing and Joseph Jaja [73] both explicitly mention the use of the outer product to compute matrix multiplication. Both of these works use a GPU for their implementation while this work focuses on the emerging field of on-chip matrix engines.

Yu et al. implement a rank-one update algorithm on the GPU, but their research focus is improving the performance of bilinear pooling [75]. Their work aims to replace another method of singular value decomposition via an

iterative method requiring many matrix-matrix multiplications by using the outer product. The work presented in this thesis aims to improve matrix-matrix multiplication on the CPU using rank-r updates.

Pal et al. introduce an outer product accelerator designed for sparse-matrix multiplication [56]. Their design, called OuterSPACE, is an external accelerator focused on alleviating the memory issues associated with sparse matrix-matrix multiplication. The algorithm presented in this thesis uses MMA as an accelerator for dense matrix-matrix multiplication. Despite the focus of this thesis on matrix multiplication, MMA is not at all limited to dense operations. Work by Gu et al. develops a memory-efficient algorithm for sparse matrix-matrix multiplication using the outer product that may prove to be an excellent match with MMA [27].

3.4 Performance Evaluation

Robust performance benchmarking is critical for the evaluation of vector extensions. While there is extensive performance evaluation of matrix multiplication on vector extensions for Intel architectures [3], [31], [32], to the best of the author's knowledge, similar studies do not exist for the PowerPC or ARM platforms. Moreover, the introduction of matrix engines is recent in all platforms and therefore only simulated or theorized performance estimates exist for AMX, SVE, or MMA [16], [58]. Therefore, this work is among the first to present performance evaluation of a matrix engine on actual hardware.

Chapter 4

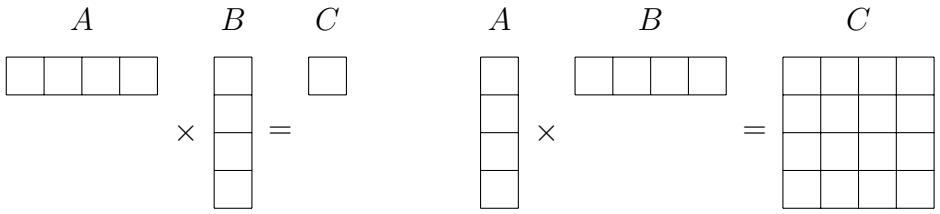
Matrix Multiplication

Matrix multiplication is used extensively in many fields of Science that rely on HPC, including several fields within physics [39], biology [2] and chemistry [72]. It is also possible to simulate quantum computing processes using matrix multiplication [77]. Moreover, many neural-network operations are based on matrix-matrix or vector-matrix multiplication [9], [59].

Due to its simple structure, tight loop layout, but potentially long execution time, matrix multiplication is well positioned for improvements through modifications to both software and hardware. Seminal research, such as that by Goto and van de Geijn [24], has resulted in high-performance software implementations, including the extensively used linear-algebra libraries Eigen [28] and OpenBLAS [74]. These implementations often outperform naïve versions of matrix multiplication by hundreds or thousands of times, turning hours of work into seconds or less. This speedup is possible through the use of hardware features, such as SIMD, and code transformations, such as loop blocking and data packing. Understanding matrix multiplication and how it is implemented is critical to achieving these improvements.

4.1 Inner and Outer Product

In the most abstract sense, matrix multiplication is a very simple operation to implement. As long as each element of A is multiplied with the correct element of B and the result is accumulated to the correct element of C, the order in which these operations happen does not matter. Thus, the simplest



(a) Inner product. (b) Outer product (rank-one update).

Figure 4.1: Example matrix-matrix multiplication computation styles.

implementation of matrix multiplication can be written as in Listing 4.1.

```
1 for i=0 to M, j=0 to N, k=0 to D do
2   C[i, j] += A[i, k] * B[k, j]
```

Listing 4.1: Pseudocode implementing matrix multiplication as simply as possible.

The order in which the loops are executed does not affect the output of the algorithm on the condition that each combination of i, j, and k is reached exactly once. However, mathematically speaking, choosing a certain order for the loops will produce functionality resembling certain common mathematical operations. A given order for the loops leads to matrix multiplication via the *inner product*, which is the classic method taught in linear algebra courses, while a different order leads to matrix multiplication via the *outer product*. While both choices arrive at the same final result, the efficacy with which they arrive can vary significantly, as is discussed later in Section 4.3 and Section 4.4.

4.1.1 The Inner Product

As taught in the classroom, matrix multiplication can be computed by repeatedly applying the inner product. The inner product takes two vectors and produces a single scalar value. Specifically, it is defined as the summation of the pair-wise products of two vectors (e.g. $v \cdot w = v_1 w_1 + v_2 w_2 + \ldots + v_D w_D$).

Within the context of matrix multiplication, the inner product is used to compute a single cell of C from a row of A and a column of B. More concretely, given the computation $\underset{(M\times N)}{C} = \underset{(M\times D)}{A} \times \underset{(D\times N)}{B}$, the inner product computes the cell $c_{i,j}$ by taking the inner product of the ith row of A and the jth column of

B (Figure 4.1a). In this way, matrix multiplication is composed of one inner product for each cell of C.

We can see this process in the following implementation of matrix multiplication:

```
1 for (uint64_t i = 0; i < M; ++i)
2   for (uint64_t j = 0; j < N; ++j)
3       for (uint64_t k = 0; k < D; ++k)
4         C[i][j] += A[i][k] * B[k][j];
```

Listing 4.2: A basic matrix multiplication via inner product.

Referring back to Listing 4.1, the loop order has been concretised to (i, j, k). In order to implement the mathematical process outlined above, the loops on lines 1 and 2 of Listing 4.2 select the cell from C, the row of A, and the column of B that will be used. The innermost loop on line 3 implements the inner product by multiplying elements from the chosen vectors in A and B and then reducing them into the chosen cell of C. One can thus view the implementation in Listing 4.2 as a loop over i and j that selects an output cell and repeatedly calls an inner product function. Therefore, with the choice of loop order (i, j, k), matrix multiplication can be conceptualised as shown in Listing 4.3.

```
1 for i=0 to M, j=0 to N do
2   C[i, j] = innerProduct(A.row(i), B.column(j))
```

Listing 4.3: A basic matrix multiplication via inner product broken into logical concepts.

4.1.2 The Outer Product

Matrix multiplication can also be computed through repeated application of the outer product. In contrast to the inner product, the outer product takes two vectors and produces a matrix. Specifically, given v and w that have lengths m and n respectively, the outer product produces an $m \times n$ matrix where cell (i, j) is the product of v_i and w_j (Figure 4.1b).

Where, given a vector from A and and a vector from B, the inner product fully computes a single cell, the outer product partially computes every cell of

the output C. Moreover, each vector of A and B is used only once. Specifically, given the computation $\underset{(M\times N)}{C} = \underset{(M\times D)}{A} \times \underset{(D\times N)}{B}$, matrix multiplication is computed by taking the outer product of column k of A and row k of B and summing the resulting partial matrices. Because the dimension D dictates both the number of columns in A and the number of rows in B, the full matrix multiplication can be computed by performing a total of D outer products.

Using Listing 4.1 as a model, this process implements matrix multiplication with a loop order of (k, i, j):

```
1  for (uint64_t k = 0; k < D; ++k)
2    for (uint64_t i = 0; i < M; ++i)
3      for (uint64_t j = 0; j < N; ++j)
4          C[i][j] += A[i][k] * B[k][j];
```

Listing 4.4: A basic matrix multiplication via outer product.

Here, the loop on line 1 selects a column from A and a row from B while the loops on lines 2 and 3 compute the outer product of those vectors. Thus, in a tranformation similar to the one between Listing 4.2 and Listing 4.3, Listing 4.4 can be rewritten like so:

```
1  for k=0 to D do
2    C += outerProduct(A.column(k), B.row(k))
```

Listing 4.5: A basic matrix multiplication via outer product broken into logical concepts.

Rank-r Updates

Some mathematical operations, such as lower-upper factorization, depend on a changing data set, updating a model as new data points are added [62]. This update process is performed by taking a matrix derived from the model and updating it with the outer product of two vectors. Because the result of the outer product of two vectors, a matrix, will always be of rank one, this operation is called a *rank-one update*.[1] Matrix multiplication can be computed in the same way by updating a zeroed-out matrix with outer products

[1] All rows and columns are a linear combination of one vector. For example, given $M = vw^T$ all columns in M are a multiple of v.

(the rank-one update), eventually arriving at a fully computed multiplication: $C^1 = 0; C^{k+1} = C^k + A_{*,k}B_{k,*}$. With this model, the outer-product matrix multiplication in Listing 4.5 is rewritten as such:

```
1 for k=0 to D do
2   rankOneUpdate(C, A.column(k), B.row(k));
```

Listing 4.6: Matrix multiplication using rank-one update.

It is not necessary for a complicated model to update itself one data point at a time. Should, for example, two data points become simultaneously available, two rank-one updates can be performed at the same time. Updating with two rank-one updates means summing two rank-one matrices; by definition, the summation of two rank-one matrices creates a rank two matrix.[2] Therefore, this operation can now be described as a rank-two update. This same logic extends to creating updates with greater ranks, giving rise to the rank-r update.[3] Revisiting the process in Listing 4.6, this repeated rank-one update process is apparent. The loop simply performs D rank-one updates, implying that the algorithm can be once again reinterpreted in the form of one rank-D update like so:

```
1 // Assumption: A, B are arrays of D vectors
2 rankRUpdate(C, A, B, /*rank*/ D);
```

Listing 4.7: Matrix multiplication using rank-r update.

4.2 State of the Art

Matrix multiplication, as a critical operation in many applications, has been the focus of optimisation research for decades. Simple insights into matrix multiplication have enabled better data usage, both spatially and temporally. Contemporary libraries and their overall frameworks are derived from seminal work by Goto and van de Geijn [24]. More recent work [66], [76] have improved

[2] Given matrices $M_1, M_2 \in \mathcal{R}^{m \times n}$, each rank one, then each matrix has v_i, its only basis vector. If $v_1 \neq v_2$, then $M_1 + M_2$ has two basis vectors v_1, v_2, implying $M_1 + M_2$ has rank two.

[3] Common notation is rank-k update; it is changed to rank-r update in this work so as to avoid confusion with k in other uses.

upon that seminal work, but the overall structure remains the same. Each of these works aim to improve the portability to new architectures by providing a more modular approach for the micro kernel. This includes a work by Low et al. which shows that the tuning parameters of the kernels for each architecture can be derived analytically from the architecture specification [48].

Goto and van de Geijn's work concludes that matrix multiplication should be implemented in a *layered approach*. Each of these "layers" is typically one or more loops that perform some function before executing the enclosed layer. It can be convenient to think of each of the upper layers as a function with a nested function call within it. The innermost layer in this view performs the most basic computation on which the entire operation is built, essentially a small and fast matrix multiplication. The surrounding layers are concerned with the optimisation of data movement and layout.

Typically, the layered approach consists of two layers: **(1)** the blocking strategy responsible for breaking up the computation combined with the packing strategy responsible for optimising data movement between memory and cache as well as data layout within the cache (called the *outer kernel* or *macro kernel*); and **(2)** the computation strategy responsible for data movement between cache and register as well as for producing the actual result (called the *inner kernel* or *micro kernel*). This work focuses on building a modular innermost layer and therefore the outer layer is explained only briefly for completeness.

4.3 The Outer Kernel

There are two operations that are critical to the function of the outer kernel: blocking and packing. Blocking takes an operation and breaks it into smaller pieces, focusing on computing output in a piecewise manner rather than all at once. Division is necessary because matrix sizes found in applications are often much larger than what can fit in a cache memory. Even with this division, the distance between elements within different rows or columns can cause cache conflicts or TLB misses. To increase the spatial locality of accesses, packing

takes an operand matrix and copies a block of data into a smaller buffer where all elements fit in cache simultaneously, allowing for faster access times. Packing can also improve the data access pattern for a more efficient inner kernel, as discussed later in Section 4.3.1.

The outer kernel is a delicate dance between blocking and packing. If the blocking factor is too large (i.e. many small blocks) then there is an excessive amount of data movement, requiring packing more often and having less reuse of each packed element. However, when the blocking factor is too small (i.e. few large blocks) then blocks no longer fit in cache – in the worst case, causing page faults – and packing is ineffective. The dimensions of the blocking, and therefore the packing, can be determined analytically [48] and are dependent on the sizes of all levels of the cache present on the target architecture (L1, L2, L3 on modern consumer machines, uncommonly L4).

Goto and van de Geijn address blocking, caching, and TLB issues in their work by devising three ways of breaking up both operand and output matrices. When blocking, each new blocking loop breaks up a single dimension of the matrix multiplication. Dividing one dimension of a matrix produces a *panel*: a section with one long dimension and one short dimension. Further dividing a panel produces a section with two short dimensions called a *block*. Only two of the three dimensions (M, D, N) are blocked while the third is the inner most loop's iteration variable. This process produces $\binom{3}{2} = 3$ different choices for the innermost loops.[4] Different combinations of blocking orders also induce different packing choices that change which of A, B, and C are packed into L1 and L2 cache.

4.3.1 Blocking and Packing for Cache

The primary concern when blocking and packing for cache is optimising L2 cache bandwidth. When dimensions M and N are blocked, the innermost loop multiplies a panel by another panel, producing a block. This breakdown is affiliated with the inner product because it multiplies two panels, which resemble vectors, while producing a block, which resembles a cell. According

[4] "Three choose two".

to Goto and van de Geijn's proposed method, with this breakdown, C will be packed in L2 cache. Unfortunately, because GEMM is an accumulating operation (e.g. $C \mathrel{+}= AB$), C must be read from cache, accumulated into, and then written back. When either A or B are packed into L2 cache, as is the case in the other breakdowns, there are only read operations from the L2 cache. Therefore, the inner-product method is less bandwidth efficient because it requires both a read and a write from L2 cache.

An implementer can choose between the remaining two methods (blocking M and D or blocking D and N) based on the storage order of C. Within each of these methods, the order in which D and the second dimension are blocked creates two options. One of these options is considerably better than the other; to demonstrate, without loss of generality, consider the method that blocks M and D and iterates N.

Given that A is already packed in L2 cache, the first option, which blocks D then M, **(1)** streams C to and from memory and **(2)** packs B in L1 cache. Given again that A is packed in L2 cache, the second option, which blocks M followed by D, **(1)** streams B from memory and **(2)** computes C in a temporary in L1 cache which is eventually unpacked and merged with C in memory. The operations labeled **(1)** in each option can be assumed to have negligible impact because they can be pipelined with computation. However, the operations labeled **(2)** are effectively overhead. By virtue of the unpacking and merging of C being a more complex operation, the first option is concluded to be superior. The same logic can be applied to the method which blocks D and N by substituting N for M and swapping which of A and B are packed in L2 cache, arguing that packing D then N is superior.

Blocking and Packing for Registers

The dimensions chosen to maximise cache usage are likely to be too large for the registers to handle, thus an additional set of loops can be added to block for registers. These loops only block the pre-packed buffers in memory and do not perform any packing themselves. Instead, when packing for the cache, the order in which data is packed is modified. Rather than packing in the

same order relative to the original data, data is organised such that these sub-matrix register arguments are contiguous. Packing in this way results in only a single data copy and allows loads to register for calculation to be contiguous in memory, improving hardware prefetching effects.

Practicality

Naïve matrix multiplication (i.e. Listing 4.1) does not perform blocking nor packing; such a case can be viewed as a "bare" inner kernel, without an outer kernel. Given small matrices as operands, this lack of blocking and packing is often the correct choice. The overhead resulting from the data movement in the outer kernel is designed to be overshadowed by the speedup gained from improved cache performance. When input and output data are already small enough to be entirely resident in cache, the effort made to pack and block is shown to be an unnecessary expense.

However, given large matrices, lacking a blocking and packing layer is disastrous for performance. With sufficiently large matrices, every load will result in a cache miss because the requested data will have been expelled from the cache by a more recent load simply by the pigeonhole principle. In the worst case, these issues are compounded with TLB misses as well. Therefore, while a single, optimised inner kernel is always critical to performance, it is important for library implementers or compiler designers to consider the size of the data when possible and offer multiple outer-kernel data-management strategies.

4.4 The Inner Kernel

The inner kernel is focused on maximising performance. Whether the computation itself has small dimensions or a larger computation has been blocked and packed, the inner kernel should be as tight and as efficient as possible. Therefore, the implementation of the inner kernel must consider optimisations such as loop unrolling and instruction rescheduling to improve operation pipelining. The instruction cache capacity, instruction issue rate, and functional unit availability must also be taken into account. First, however, the product

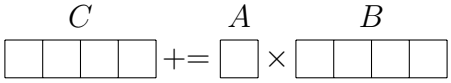
operation used to implement the inner kernel must be carefully chosen.

4.4.1 Inner Product vs Outer Product

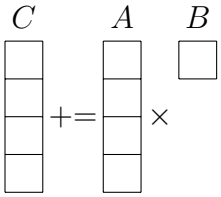
Matrix multiplication's inner kernel is implementable through two operations: inner product and outer product. Both operations arrive at the same destination albeit via different processes. At the end of the computation, both methods will have performed exactly the same number of multiplications and additions. In a hypothetical machine with unlimited resources, including vector registers of unlimited length, they would also perform the same number of stores and loads. In such a theoretical framework, at least in terms of computations performed, neither method can be said to be better. However, in a practical machine, it is impossible to load the entirety of a long row or column into vector registers in order to fully compute a portion of the matrix.

Long dimension lengths make tiling a necessity. Tiling, as discussed in Section 4.3.1, breaks the overall multiplication into smaller, more manageable computations. As discussed in that same section, on the macro scale, Goto and van de Geijn conclude that an inner-product-based tiling results in more costly data movement, making it a poor choice for the implementation of the outer kernel.

When considering the inner kernel, the inner-product approach also makes poor use of SIMD capabilities. Computing the inner product requires a horizontal reduction (or tree reduction): the reduction of all the lanes of a vector register into a single scalar element. Computing a partial result in this manner reduces the effectiveness of SIMD by creating a scalar value. SIMD was created specifically to alleviate the bottleneck associated with working with scalar values; reverting to scalars should therefore be avoided when possible. The outer product, on the other hand, uses a vertical reduction (or element-wise reduction) that allows partial products to be accumulated simultaneously in all lanes of the destination vector register.

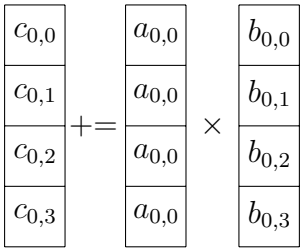


(a) Block times panel.

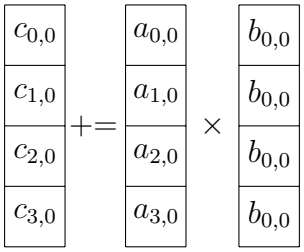


(b) Panel times block.

Figure 4.2: The two possible implementations for the inner kernel viewed mathematically.



(a) Block times panel.



(b) Panel times block.

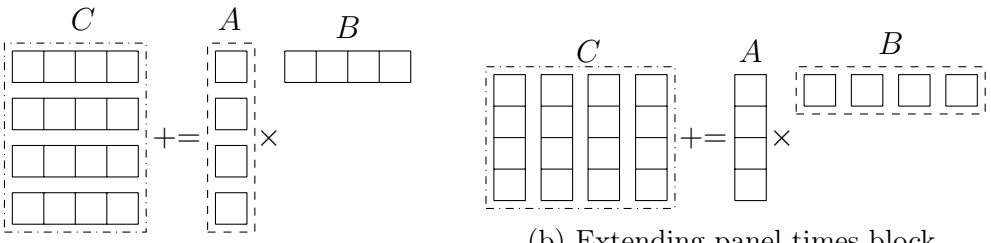
Figure 4.3: The two possible implementations for the inner kernel as computed by using broadcasting and FMA instructions.

4.4.2 Goto and van de Geijn's Methodology in the Inner Kernel

Goto and van de Geijn's methodology continues to be present within the inner kernel of libraries. As in the outer kernel, rather than implementing matrix multiplication in terms of the inner product, which some ISAs have directly supported for years[5], matrix multiplication is implemented through the multiplication of a combination of a block and a panel. While for the outer kernel a short dimension is optimally several hundred elements wide and a long dimension is the full extent of one of the matrix dimensions, the inner kernel must operate with registers. Therefore, a "short" dimension is typically a single element and a "long" dimension is the length of the vector register. Figure 4.2a and Figure 4.2b show, in the mathematical sense, what both combinations of block and panel multiplication look like if a vector register can hold four elements.

Next is a discussion of the typical library implementation of these concepts

[5]For example, the DPPS or DPPD instructions that were introduced in x86 SSE 4.1 in 2008.



(a) Extending block times panel.

(b) Extending panel times block.

Figure 4.4: Extending Goto and van de Geijn's inner kernel techniques to outer product.

using SIMD techniques in a simple but flawed outer product followed by a presentation of a method to resolve these flaws through an improved version of the outer product.

Emulating the Outer Product with SIMD Instructions in Libraries

Figure 4.3a and Figure 4.3b show how the outer product is emulated using an architecture's SIMD capabilties. First, the block's single element is broadcasted to each lane of a vector register. The elements of the panel are then loaded into a different vector register. Once both registers are ready, a Fused Multiply-Add (FMA) instruction multiplies the two vector registers together and accumulates the result into a third vector register representing a panel in C. This process is more effective than using the inner product because multiple output values are accumulated at the same time in each lane of the vector. However, in a kernel that is supposed to be as tight as possible, cycles are lost to duplicating values. Furthermore, the kernel should also use resources as effectively as possible, however, the duplicated values use all lanes of a vector, effectively reducing a vector register back to a single scalar, nullifying the benefits of SIMD.

Improving the Design

To improve upon this design, it must first be reexamined. In choosing this method, the implementer is performing a $\underset{(1\times n)}{C} = \underset{(1\times 1)}{A} \times \underset{(1\times n)}{B}$ or $\underset{(n\times 1)}{C} = \underset{(n\times 1)}{A} \times \underset{(1\times 1)}{B}$ matrix multiplication. While it is tempting to view this operation as a traditional inner-product, a different point of view can be more

beneficial. Instead, this multiplication can be regarded as a degenerate case of an outer product where one of the operand vector's length is one. With this understanding, the direction for improvement becomes apparent. The duplication issue can be resolved by replacing the duplicated values with unique values, extending the outer product's shortened dimension, bringing it closer to Figure 4.1b. Doing so stacks more of these operations, as shown in Figure 4.4. A dashed box represents one vector register, with each element therein being a distinct element from the matrix rather than a single broadcasted element. Given a register that could contain the much larger output data in the dot-dashed boxes, the overhead of broadcasting can be completely removed and the computational throughput of the operation quadrupled. Chapter 5 shows how this improvement can be implemented using POWER10's MMA.

4.4.3 Implementation

Having looked at which kernel to implement, the direction for the implementation must be decided. As discussed previously, libraries often have kernels handwritten in assembly. Kernel writers must implement all optimisations by hand. In doing so, they must review CPU properties as well as consider the design of their outer kernel to determine to what degree an optimisation will affect the hardware. Historically, with great effort, this process has produced excellent results.

Unrolling the kernel to a larger degree allows for greater opportunities in instruction rescheduling as well greater use of available registers. Computations can be moved closer to loads to enable pipelining as well as interleaved to hide stalls due to occupied functional units. Future iterations that would load the same data can also be moved such that the values are still in register and can be reused, reducing register pressure and stores.

However, all of these optimisations, and others, are readily available in a compiler, often implemented generically and with great care regarding applicability and benefit analysis. For example, the kernel can be written in terms of scalars, allowing the compiler to vectorise the kernel automatically. Automatic vectorisation means obvious optimisations such as using the best

available vector load and store instructions as well as unrolling to allow for pipelining of loads. Less obvious issues are also solved automatically: handling operations that may have dimensions that are not a multiple of vector register automatically creates additional code pathways that make use of vectors at half width or less, eventually reducing down to scalar computation. Issues that are impossible to solve for handwritten kernels without writing an entirely new kernel, such as new instructions added to an ISA, are also immediately solved because the compiler knows which architecture to compile for and can use any instruction available on that architecture. Therefore, a kernel written within the compiler can be written at a higher and more generic level, as in an IR, and achieve the same, or better, performance. It is for this reason that Chapter 6 presents a method built fully within a compiler without a handwritten kernel.

4.5 Summary

This chapter delves in-depth into the intricacies of matrix multiplication, an operation that seems quite simple at first glance. First to consider are the two main ways in which matrix multiplication can be computed: inner product and outer product. Both are implemented using the same set of loops and operations and arrive at the same answer, but their performance can vary drastically based on transformations applied when optimising.

The modern approach to optimising matrix multiplication divides the operation into two levels of work. The first layer, the outer kernel, focuses on efficient manipulation of the memory hierarchy. This goal is achieved through the use of a blocking and packing strategy derived from seminal work by Goto and van de Geijn.

The inner kernel focuses on optimising the computation of matrix multiplication given the small amount of data provided by the outer kernel at each invocation. Goto and van de Geijn's influence extends into this layer as well, where they state that the inner product is an inefficient method for computing matrix multiplication. Despite indications that an outer-product

based method is superior, implementation within libraries must rely on emulation via SIMD methods. However, this method of emulation contains a piece of the outer product in it. To improve upon the SIMD method, a new hardware extension is required. Chapter 5 presents an implementation of that improvement.

Chapter 5

Matrix Math Assist

POWER10's MMA is what is being called a "matrix engine" [16]. Specifically, it is a facility in the CPU whose function is computing matrix multiplication. Matrix engines have appeared in multiple architectures, though their implementation and design choices differ. Likewise, the algorithm for optimally making use of each of these architectures differs. While this work focuses on MMA, a discussion of the differences in these architectural implementations follows in Section 5.3.

MMA adds a new architectural feature alongside the matrix engine: eight accumulator (ACC) registers. Accumulators and the new instructions for interacting with them enable the efficient matrix-matrix multiplication outlined in Section 4.4.2. Much of the content in this section can be found throughout the POWER10 ISA document though some of it comes as insight from engineers at IBM or external sources.

MMA's design is built around the choice to compute the outer product instead of the inner product. As discussed in Section 4.4.1, the inner product is inferior when compared with the outer product. MMA eschews the overhead revealed at the end of Section 4.4.2 by directly providing the outlined outer product capabilities along with the large register necessary for holding the result data.

5.1 Accumulator Assembly and Disassembly

An accumulator, in almost all interactions with MMA, can be thought of as a single register consisting of 4×4 32-bit elements. Accumulators may be assembled (initialised) in three manners: **(1)** set to zeroes (`xxsetaccz`, Set Acc to Zero) **(2)** constructed from four consecutive Vector-Scalar Registers (VSRs) (`xxmtacc`, Move To Acc) and **(3)** a multiplication instruction that does not accumulate (see Section 5.2). When constructing from VSRs, each VSR is moved into the ACC and becomes one row of the resulting accumulator.[1]

In its current iteration, accumulator n is the logical grouping of four 128-bit VSRs (`VSR[4n:4n+3]`) into a single register. As such, the four underlying registers used to construct the accumulator are unavailable for use while the accumulator is in an assembled state. For example, `VSR[0:3]` are unavailable while `ACC[0]` is assembled. This blocking effect also applies to the underlying registers when assembling an accumulator to zeroes.

Blocking the underlying registers is viewed as a reasonable restriction in POWER10. However, the ISA is written such that if, in future versions of the hardware, the blocking restriction is removed, no changes will need to be made to the ISA.

The reverse operation to assembly is disassembly (`xxmfacc`, Move From Acc). Disassembly moves each row from the accumulator into the same vector registers used for construction. Therefore, as before, the first row of `ACC[0]` is placed into `VSR[0]`, the second row is placed into `VSR[1]`, etc.

5.2 Matrix Multiplication in MMA

Table 5.1 shows all seven of the data types usable with MMA. Instructions are given three arguments: an accumulator and two VSRs. Considering the simplest example, single-precision floating point, a four-element row or column fills the entire VSR ($32\text{bits} \times 4 = 128\text{bits}$). Thus, two length-four vectors are

[1] Viewing the vectors as rows is intuitively convenient, but relatively arbitrary in reality. See Section 6.3.5 for use as columns.

[2] Google's Brain floating point format [69].

Instruction	Input Type	Output Type	Input Dim.	Smallest Computation
xvi4ger8	i4	i32	4×8	$\underset{(4\times4)}{C} = \underset{(4\times8)}{A} \times \underset{(8\times4)}{B}$
xvi8ger4	i8	i32	4×4	$\underset{(4\times4)}{C} = \underset{(4\times4)}{A} \times \underset{(4\times4)}{B}$
xvi16ger2	i16	i32	4×2	$\underset{(4\times4)}{C} = \underset{(4\times2)}{A} \times \underset{(2\times4)}{B}$
xvf16ger2	half	float	4×2	$\underset{(4\times4)}{C} = \underset{(4\times2)}{A} \times \underset{(2\times4)}{B}$
xvbf16ger2	bfloat16[2]	float	4×2	$\underset{(4\times4)}{C} = \underset{(4\times2)}{A} \times \underset{(2\times4)}{B}$
xvf32ger	float	float	4×1	$\underset{(4\times4)}{C} = \underset{(4\times1)}{A} \times \underset{(1\times4)}{B}$
xvf64ger	double	double	4×1	$\underset{(4\times2)}{C} = \underset{(4\times1)}{A} \times \underset{(1\times2)}{B}$

Table 5.1: A description of MMA instructions investigated. Input dimension is transposed for second argument.

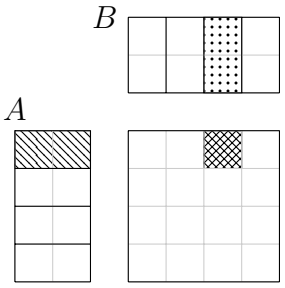
multiplied in an outer product, creating the computation $\underset{(4\times4)}{C} = \underset{(4\times1)}{A} \times \underset{(1\times4)}{B}$. Recalling Section 4.1.2, this operation is a rank-one update to the accumulator.

Halving the size of the type to 16 bits means the argument VSRs are underutilised at half capacity. Accumulators cannot be expanded and are always 4×4 matrices of 32-bit elements. Thus, to increase utilisation of VSRs, the number of elements contained in each VSR is doubled resulting in doubling the number of rows or columns in a VSR. Therefore, when using 16-bit values, the calculation performed is $\underset{(4\times4)}{C} = \underset{(4\times2)}{A} \times \underset{(2\times4)}{B}$, now performing a rank-two update to the accumulator. The same logic applies for eight- and four-bit types, computing rank-four and rank-eight updates respectively.

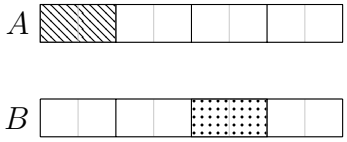
5.2.1 Arguments in Register

Once the instruction is performing a rank-two update or greater, the order of the elements in the register must be considered. The POWER ISA specifies that the first argument must have elements in row-major order while the second argument must have elements in column-major order. This enables simple circuit pathways where, for a rank-r update, element (i, j) in C accumulates with the inner product of the r-length group of elements in A at offset $i \times r$ and in B at $j \times r$.[3]

[3]Typical matrix notation uses one-indexing; this example uses zero-indexing.



(a) Arguments for an `i16` matrix shown contextually.



(b) Arguments for an `i16` shown in VSR.

Figure 5.1: Element ordering in types smaller than `float` demonstrated using `i16`.

	`-p`	`-n`
`p-`	$C' = C + AB$	$C' = -C + AB$
`n-`	$C' = C - AB$	$C' = -C - AB$

Table 5.2: MMA accumulation computations by instruction suffix. C is the accumulator before accumulation while C' is the accumulator after accumulation.

Figure 5.1a shows an example for `i16`. A and B are divided by dark lines according to their access order while lighter lines divide the elements within the vector. The first two elements of the operand from A, highlighted by diagonal lines, form a row while the first two elements from B, highlighted by dots, form a column.

In Figure 5.1b, the arguments are shown laid out in a VSR. The VSR is divided using dark and light lines in the same way as Figure 5.1a to show how the element positions are translated to a VSR. The inner product of these two two-element vectors, from A at offset $0 \times 2 = 0$ and from B at $2 \times 2 = 4$ in their respective VSRs, are accumulated with the value in C at $(0, 2)$, highlighted in a crosshatch pattern.

5.2.2 Instruction Variants

An instruction without a suffix (first column of Table 5.1) will both assemble the given accumulator and overwrite it with the outer product of the two VSRs. Afterward, to accumulate into the accumulator, each data type has a

"family" of instructions offering different semantics. Four instruction suffixes are available: `pp`, `pn`, `np`, `nn`. The `p` and `n` stand for positive and negative respectfully. The first character specifies the sign of multiplication (i.e. $\pm AB$) while the second specifies the sign of the accumulator (i.e. $\pm C$). The possible computations are shown in Table 5.2. Computations with 16-bit integers have an additional suffix, `s`, which replaces the regular overflow semantics with saturating semantics (e.g. 0xFFFF + 1 = 0xFFFF).

In combination with any of the suffixes, a single prefix exists: `pm`. This prefix indicates a "prefixed masked" instruction. These instructions take an additional three arguments, each of which is a mask that enables fine-grain control over the accumulation. Each of the masks allows disabling of one of the following: **(1)** any of the rows of the accumulation **(2)** any of the columns of the accumulation or **(3)** any of the ranks of the accumulation. Masks disable only the application of computed values to the accumulator, not the calculation; therefore, the execution time of a masked instruction is the same as that of an unmasked one.

5.2.3 Double Precision Differences

When all other computations produce a 32-bit value, essentially gaining precision or remaining at the same precision, it would be unreasonable to force computations with 64-bit values to lose precision. Therefore, when working with double-precision floats, accumulators become 4×2 arrays rather than the usual 4×4. The change in data size affects the argument VSRs as well: a single VSR now fits only two values. Thus, in order to compute a rank-one update, the first dimension requires four values, spread across two registers, while the second dimension now requires two values in a single register.

5.3 Matrix Engine Comparison

Currently, three prominent matrix engines exist, but, given the current importance placed on matrix multiplication, it is unlikely that this list remains small. Furthermore, each of the architectures have been designed in their own

Architecture	Location	Product Style	Arg/Dest	Supported Types
Power10 MMA	core	outer	VSR/ACC	`i4`, `i8`, `i16`, `bfloat16`, `half`, `float`, `double`
x86 AMX	off-core	inner	Tile	`i8`, `bfloat16`
ARM NEON/SVE	core	inner	Vector register	`i8`, `bfloat16`, `float`, `double`

Table 5.3: Comparison of features currently offered by cpu-based matrix engines.

way with extensibility in mind implying that the richness and diversity of features is sure to improve as well. The features of these three matrix engines are summarised in Table 5.3.

The x86 architecture has placed a strong focus on matrix multiplication by developing AMX. It is an off-core accelerator dedicated specifically to matrix multiplication. Furthermore, the accelerator makes use of an entirely new register file consisting of eight large (16×64 bytes) registers dubbed "tiles". The architecture supports only inner product computations and uses tiles as both arguments and destinations . While it supports less types than MMA, it does support two important types for artificial-intelligence workloads: `i8` and `bfloat16`.

ARM's NEON and SVE do not focus on matrix multiplication, providing them instead as a small part of a larger vector extension. The facility is implemented on-core and does not provide any new registers, relying on the architecture's already-present vector registers. These two factors, combined with an inner-product style computation mean relatively small computation sizes when compared with MMA or AMX. NEON, like AMX, supports `i8` and `bfloat16` while SVE adds support for `float` and `double`.

5.4 Summary

This chapter presented the new MMA extension in POWER10. The extension adds a new type of register called an accumulator which can be used to compute matrix multiplication using the outer product. Moreover, a total of seven types

are supported, several of which perform multiple outer products in a single instruction, increasing efficiency. Finally, this chapter compared MMA with the matrix engines available in other ISAs.

Chapter 6

Implementation Methodology

This chapter discusses the contributed implementation of a high-performance and library-free matrix-matrix multiplication algorithm which makes use of MMA. Due to its implementation in the LLVM compiler framework, the method has immediate performance-enhancing potential on the POWER10 platform.

6.1 Intrinsics in LLVM

An intrinsic is a function available in an IR and is therefore not directly available in higher-level languages. Select intrinsics are available in higher-level languages in the form of a builtin, though no builtins are used in this work.[1] These functions' most important role is to symbolise the concept of a common operation in a representation that must be lowered to many disparate forms while maintaining efficiency and correctness. They are therefore not platform-specific but their lowering may be if the semantics necessitate it or if the maintainer wants to provide a more efficient version. For example, the LLVM intrinsic `llvm.vector.reduce.add.*` may be generically lowered to a series of vector operations at the IR level, or, if the ISA provides a reduction instruction, it could be lowered more efficiently by the backend.

[1] Some compilers use this terminology differently; the definitions given here are the definitions according to LLVM.

```
declare i32 @llvm.smax.i32(i32 %a, i32 %b)
declare <4 x i32> @llvm.smax.v4i32(<4 x i32> %a, <4 x i32> %b)
```

Listing 6.1: A set of basic intrinsic declarations [21].

6.1.1 Intrinsic Format

Within LLVM IR, intrinsics appear simply as external function declarations whose names are prefixed with "`llvm.`". Names within LLVM are permitted to contain periods, a feature that intrinsics use to mangle families of intrinsics. Families arise from the need to provide generic intrinsics to a strongly typed language where overloading is not allowed. For instance, consider the `llvm.smax.*` intrinsic which takes two signed integers of the same type and returns the largest. The LLVM documentation uses the "`.*`" suffix to represent the mangled portion of the name. An actual definition will replace the asterisk with a type name; if the function is generic in multiple places (return value, arguments) then the other type names will be appended and separated with more periods. The `llvm.smax.*` intrinsic only has one generic type. Listing 6.1 shows a declaration that uses a scalar 32-bit integer, typed and mangled as `i32`, as well as one that uses a length-four vector of 32-bit integers, typed as `<4 x i32>` and mangled as `v4i32`.

6.2 The `llvm.matrix.multiply.*` Intrinsic

The `llvm.matrix.multiply.*` intrinsic existed prior to this work and warrants a brief explanation to provide context [21]. The intrinsic has three generic types: the return-value type and the type of the two input matrices. Each of the three matrices is typed as flattened vectors and therefore the signature requires three extra arguments to describe the dimensions as in $\underset{(M\times N)}{C} = \underset{(M\times D)}{A} \times \underset{(D\times N)}{B}$.

The original implementers required that input dimensions be statically known constants. As well, those familiar with the analogous BLAS routines may notice that arguments describing data-access order are conspicuously ab-

```
1  declare <128 x float>
2    @llvm.matrix.multiply.v128f32.v40f32.v80f32(
3      <40 x float>, <80 x float>, i32, i32, i32
4    )
5
6  define void @foo() {
7    ; Declaration and construction of %A and %B...
8    %C = call <128 x float>
9      @llvm.matrix.multiply.v128f32.v40f32.v80f32(
10       <40 x float> %A, <80 x float> %B, i32 8, i32 5, i32 16)
11     )
12   ; Function continues...
13 }
```

Listing 6.2: An example declaration and usage of the `llvm.matrix.multiply.*` intrinsic.

sent. The original interface assumes that all input matrices have a column-major access order. Users may include a specific flag in their compiler invocation to change this setting to row-major order.

The original implementation also includes one other significant constraint: the intrinsic does not use pointers and must be called with the input matrices already loaded from memory into a virtual register. Each value in LLVM's SSA IR exists in a virtual register; there is an infinite number of virtual registers and each virtual register has infinite width. Values which are loaded from memory into a virtual register will be broken up by the backend code generator to fit the available hardware registers. Thus it is possible to load a matrix with a certain access order from memory and then implement a process to reorder the elements to the opposite access order before calling the intrinsic, though the process will add considerable overhead. Hence, while it is possible to use matrices with different access orders in memory, they must be transformed to have matching access orders before calling the `llvm.matrix.multiply.*` intrinsic. Listing 6.2 provides an example declaration and invocation of the intrinsic which computes $\underset{(8\times16)}{C} = \underset{(8\times5)}{A} \times \underset{(5\times16)}{B}$.

During lowering, the `call` statement is replaced with a series of LLVM IR

vector operations by the `LowerMatrixIntrinsics` pass. This version of the lowering is what is referred to as the default "vectorisation" method. Because the dimensions of the operation are known statically, the computation is completely unrolled: no loops will be generated. Such a lowering is ideal because it removes the need for loop analysis and allows later passes to further vectorise or pipeline the operation. This property makes it ideal for creating small and efficient kernels focused on rapid computation.

These kernels, because fully unrolling them creates a large amount of code, cannot be used to compute large matrix multiplications. For example, unrolling a kernel for $\underset{(128\times128)}{C} = \underset{(128\times128)}{A} \times \underset{(128\times128)}{B}$ takes several minutes of compilation time and results in an IR file slightly larger than a gigabyte with over 12 million lines. Producing a binary from such a large kernel has never been accomplished due to the inordinate compilation time. Therefore, wrapping the kernel in an outer kernel, which breaks down the operation while efficiently handling memory movement, is a necessity. For further discussion see Section 4.3 or for a functional implementation and deeper discussion see the work by Kuzma et al. [41].

6.3 An Alternate Lowering Using MMA

We look to implement an improvement to this lowering on the POWER platform using MMA. Content in Section 6.3.1, Section 6.3.3, and Section 6.3.4 is derived from work the work by Kuzma et al. [41].

This work focuses on a lowering for floating point values when computing $\underset{(8\times16)}{C} = \underset{(8\times5)}{A} \times \underset{(5\times16)}{B}$. The 8×16 output size is exactly the output dimensions produced when fully utilising the architecture's available accumulators. This property makes it the ideal size for the intrinsic's role as a tight inner kernel. Nevertheless, these restrictions create a simple scenario enabling a foundational understanding of this thesis' core algorithm; removing these restrictions is discussed in Section 6.3.3, Section 6.3.4, and Section 6.3.5.

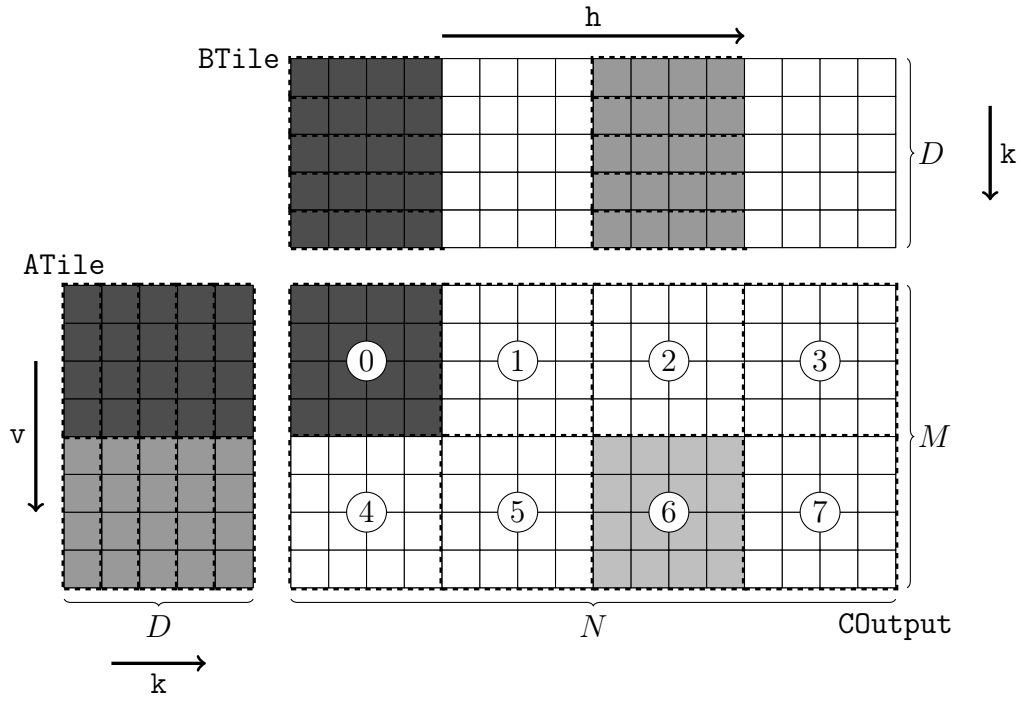


Figure 6.1: Division of `ATile` and `BTile` into operands and `COutput` into MMA accumulators.

6.3.1 Base Case

In addition to the software constraints described in Section 6.2, an efficient MMA lowering must take into account the following hardware constraints: [1] eight accumulators are available per thread and for each accumulator that is used, the usage of four VSRs are blocked; [2] there are 64 VSRs, thus if eight accumulators are used, there are 32 VSRs remaining to contain data from input matrices; [3] two multiply-and-accumulate outer-product instructions can be issued per single cycle; [4] the issue-to-issue latency for the same accumulator is four cycles; and [5] spilling an accumulator to memory is an expensive operation because it requires an instruction to disassemble the accumulator into four VSRs, four vector store instructions and, later, four vector load instructions.

Figure 6.1 illustrates how `COutput` is divided into portions that are assigned to the MMA accumulators. `ATile`, `BTile`, and `COutput` are represented in two dimensions to illustrate the position of the elements in the matrices. Each

small square in the figure represents one 32-bit element of a matrix. A circled number indicates that the corresponding portion of `COutput` is assigned to that accumulator number. When the intrinsic is executed each accumulator computes D outer products using an MMA outer-product instruction. The two tones of gray colour in Figure 6.1 illustrate that a strip of `ATile` and a strip of `BTile` are used for the accumulation of each portion of `COutput`. Each strip is reused for all of the accumulations in the same row or column of accumulators. Each outer-product computation needs two four-element operands, one from `ATile` and one from `BTile`. These operands are surrounded by dashed lines for the two accumulations highlighted in gray. The arrows indicate how the loop indices in Algorithm 6.1 iterate for the example in the figure.

Algorithm 6.1 describes the lowering of the intrinsic computation for MMA. This is the algorithm within the compiler that produces LLVM IR, not the code executed on the target machine. The produced code is discussed in Section 6.3.2.

The compile-time constants `V` and `H` (used on lines 8, 10, 12, 13) specify the layout of the accumulators for the computation. Given constraint $\boxed{1}$, the largest amount of data reuse can be obtained when `V = 2` and `H = 4` or vice versa (see Figure 6.1). The only other configuration with full accumulator usage (`V = 1`, `H = 8` or vice versa) uses more operand registers for a single accumulation – nine instead of six – and demonstrates reuse along only a single axis. These constants in the compiler generalise the lowering and make it applicable to future architectures where the ideal arrangement to increase data reuse may be different from the 2×4 arrangement in the POWER10 processor.

First, on line 2, an MMA intrinsic is chosen based on the element type of the operation (e.g. `float`, `i32`).[2] Next, an array is created to contain the eventual output (line 3) followed by assembling zeroed-out accumulators (line 4). The following loop (line 5) iterates from 0 to $D - 1$, extracting operands and

[2]This is an operation internal to the matrix multiplication meaning all negations have been applied prior to the intrinsic. Therefore, the positive multiply, positive accumulate variant is always chosen.

Algorithm 6.1 Algorithm for lowering `llvm.matrix.multiply.*` with MMA.

```
1:  function LLVM.MATRIX.MULTIPLY.*(ATile, BTile, N, D, M)
2:      MMAIntrinsic ← intrinsic chosen based on element type
3:      COutput ← M × N empty array
4:      Accs ← V × H ACCs assembled and initialised to zero
5:      for k = 0 to D − 1 do
6:          AOps ← V empty vector operands
7:          BOps ← H empty vector operands
8:          for v = 0 to V − 1 do
9:              AOps[v] ← Extract operand at ATile[v × 4][k]
10:         for h = 0 to H − 1 do
11:             BOps[h] ← Extract operand at BTile[h × 4][k]
12:         for v = 0 to V − 1 do
13:             for h = 0 to H − 1 do
14:                 createCall(MMAIntrinsic, Accs[v][h], AOps[v], BOps[h])
15:     Disassemble ACCs and store VSRs into COutput
16:     return COutput
```

performing accumulations. For each value of `k`, using the accumulator assignment shown in Figure 6.1, the algorithm extracts two operands from `ATile` (lines 8-9) and four operands from `BTile` (lines 10-11). For `k` = 0 the operands are extracted from the leftmost column of `ATile` and from the topmost row of `BTile` in Figure 6.1. The algorithmic presentation in Algorithm 6.1 uses the notation `ATile[v` × 4`][k]` and `BTile[h` × 4`][k]` to denote the index where operand vector starts. These operands are extracted to virtual IR registers; the actual VSRs to be used will be determined later by a register-allocation pass.

In Figure 6.1 each operand is formed by four elements and, once extracted, occupies one 128-bit VSR. Given constraints 1 and 2, with the choice of $D = 5$, there are enough non-blocked VSRs to contain all the thirty operands needed for the computation illustrated in Figure 6.1. Thus, laying out the accumulators in this 2×4 pattern maximises the reuse of values loaded into the VSRs: operands extracted from `ATile` are reused four times and operands extracted from `BTile` are reused two times.

Once this iteration's operands are extracted, the algorithm then iterates

over the accumulators (lines 12-13) and creates a call to the MMA intrinsic (line 14), computing a single accumulation in each. Following constraint [3], two outer-product instructions can be issued each cycle. Four pairs of accumulators can be scheduled before circling back to the first pair, thus satisfying constraint [4]. The assignment of a portion of `COutput` to a single accumulator eliminates the need to spill accumulators, thus increasing the performance according to constraint [5].

6.3.2 The Resulting IR

As discussed in Section 6.2, the fully unrolled IR code for the lowering of an intrinsic can be extremely long. Listing 6.3 presents the code resulting from unrolling $\underset{(8\times16)}{C} = \underset{(8\times1)}{A} \times \underset{(1\times16)}{B}$. The result has been simplified but preserves the process presented in Algorithm 6.1. The listing shows all operations starting from the initialisation of the accumulators to the disassembling. The matter of storing vectors to memory is easily addressed and so is left out for brevity.

First, on line 1 of Listing 6.3, a call to the Power PC (PPC) intrinsic `xxsetaccz`.[3] Only a single assembly is required in the IR because the IR is in SSA form. Because the intrinsic takes no arguments and is functionally pure[4]within the IR, the result of multiple repeated calls are seen by the optimiser as identical and will be removed by a Common Subexpression Elimination (CSE) pass. Therefore, it is up to the register allocation pass to later recognise the need for and to provide multiple simultaneous accumulators. Thus, all operations begin from the "same" zeroed out accumulator, at least within the IR.

The `<512 x i1>` return type of `xxsetaccz` normally refers to a 512-length vector of one-bit integers. However, backend developers have co-opted the type to represent an MMA accumulator when certain flags are set. There also exists a separate type which represents a VSR for use with MMA intrinsics

[3]The latest version of the algorithm begins from a non-accumulating outer-product instruction (Section 5.2) instead of a `xxmtacc` or `xxsetaccz` (Section 5.1). The `xxsetaccz` remains in this listing to connect with Algorithm 6.1.

[4]A pure function is one which: (1) has no side effects (e.g. writes to memory) and (2) returns the same output given the same input.

```
1   %ACC = call <512 x i1> @llvm.ppc.mma.xxsetaccz()
2   %ATile.0_0 = shufflevector <8 x float> %ATile, <8 x float>
        undef, <4 x i32> <i32 0, i32 1, i32 2, i32 3>
3   %ATile.0_4 = shufflevector <8 x float> %ATile, <8 x float>
        undef, <4 x i32> <i32 4, i32 5, i32 6, i32 7>
4   %BTile.0_0 = shufflevector <16 x float> %BTile, <16 x float>
        undef, <4 x i32> <i32 0, i32 1, i32 2, i32 3>
5   %BTile.0_4 = shufflevector <16 x float> %BTile, <16 x float>
        undef, <4 x i32> <i32 4, i32 5, i32 6, i32 7>
6   %BTile.0_8 = shufflevector <16 x float> %BTile, <16 x float>
        undef, <4 x i32> <i32 8, i32 9, i32 10, i32 11>
7   %BTile.0_12 = shufflevector <16 x float> %BTile, <16 x float>
        undef, <4 x i32> <i32 12, i32 13, i32 14, i32 15>
8   %ACCProd.0_0.0 = call <512 x i1> @llvm.ppc.mma.xvf32gerpp(<512 x
        i1> %ACC, <4 x float> %ATile.0_0, <4 x float> %BTile.0_0)
9   %ACCProd.0_1.0 = call <512 x i1> @llvm.ppc.mma.xvf32gerpp(<512 x
        i1> %ACC, <4 x float> %ATile.0_0, <4 x float> %BTile.0_4)
10  ...
11  %ACCProd.1_3.0 = call <512 x i1> @llvm.ppc.mma.xvf32gerpp(<512 x
        i1> %ACC, <4 x float> %ATile.0_4, <4 x float> %BTile.0_12)
12  %ACCDis.0_0 = call { <4 x float>, <4 x float>, <4 x float>, <4 x
        float> } @llvm.ppc.mma.disassemble.acc(<512 x i1>
        %ACCProd.0_0.0)
13  %ACCDis.0_1 = call { <4 x float>, <4 x float>, <4 x float>, <4 x
        float> } @llvm.ppc.mma.disassemble.acc(<512 x i1>
        %ACCProd.0_1.0)
14  ...
15  %ACCDis.1_3 = call { <4 x float>, <4 x float>, <4 x float>, <4 x
        float> } @llvm.ppc.mma.disassemble.acc(<512 x i1>
        %ACCProd.1_3.0)
```

Listing 6.3: An example lowering of the `llvm.matrix.multiply.*` intrinsic for a $8 \times 1 \times 16$ computation.

but its use was removed from Listing 6.3 to facilitate reading.[5]

Line 2 shows the extraction of the first operand from `ATile` via a shuffle instruction. Conceptually, the shuffle instruction takes two equally sized vector operands, logically concatenates them, and labels each element from 0 to $2N$. A third argument, the shuffle mask, is a vector of integers corresponding to labels defining which element from the concatenation should be in the output vector's same position. For example, with two four-length vectors as arguments and a mask of `<i32 7, i32 3, i32 0>`, a shuffle would produce a three-element vector starting with the second vector's final element followed by the first vector's final and first elements. Furthermore, the special value `undef` can be used to replace one of the input vectors to allow easily manipulation of the elements of a single vector. Selecting an element from the `undef` vector results in undefined behaviour.

Thus, using a shuffle, line 2 combines `ATile` and `undef` and extracts the first four elements of `ATile`; the result variable is named `ATile.0_0` indicating the vector starts at $(0, 0)$. The next four elements are extracted on line 3 for the second operand vector. All four operand vectors are extracted from `BTile` in a similar process on lines 4-7.

Outer products and accumulations begin on line 8 and end on line 11. Each operation takes an accumulator as the first argument and multiplies the two extracted operands found in the second and third arguments. The `xvf32gerpp` intrinsic returns a value with type `<512 x i1>`, the accumulator type. This represents the input accumulator with the operation accumulated on top. In this first iteration, the zeroed out accumulator will be duplicated for each operation. In the next iteration of accumulations, the returned accumulators will be used as the first argument. This process continues with each accumulation, creating a chain of return values to arguments, effectively showing the accumulation process.

Finally, each of the accumulators are disassembled between lines 12 and

[5] VSRs are represented by `<16 x i8>`, a 16-element vector of bytes. In the true result IR, shuffles produce four-element vectors of floats which are bit-casted (same bit-width) to the VSR type which is then consumed by the intrinsics.

15. The disassembly intrinsic takes an accumulator and returns a vector of four length-four `float` vectors representing the accumulators underlying VSRs after disassembly. The code which follows the disassemblies can extract each of the vectors and store them directly to memory.

6.3.3 Other Data Types

The presentation so far has assumed 32-bit data types where each operand VSR contains four elements and an MMA instruction computes a rank-one update. As discussed in Section 5.2, halving the data-type size doubles the number of elements in each VSR and therefore doubles the rank of the update. The packing of more elements into a single VSR and the accumulation of multiple outer products by a single MMA instruction requires changes to Algorithm 6.1. Let r be the number of outer products performed by an MMA instruction — i.e. the rank of the update. Now the step size of the loop on line 5 must be r because, in Figure 6.1, r rows of a vertical section of `BTile` and r columns of horizontal section of `ATile` are packed into each VSR. The extraction of operands on lines 9 and 11 is now a strided access. For instance, for $r = 2$ (16-bit data types), four consecutive elements are extracted both from row k and from row $k+1$ to form the 128-bit VSR. The length of D must increase by r times to provide enough data to populate the VSRs. The effect is that more partial-product accumulations can be computed per micro-kernel invocation given the same number of assemblies and disassemblies because the number of multiplications per outer product increases by r.

For the double-precision floating-point data type, an accumulator contains 4×2 64-bit elements. The operand extracted from `ATile` is placed into a combination of two VSRs that each contain two elements, collectively four, while the operand extracted from `BTile`, now only two elements, is placed into a single 128-bit VSR. Therefore, for double-precision floats, the value of N should be reduced by half to reflect the number of VSRs available. With this reduction, an `ATile` tile occupies 16 VSRs and a `BTile` tile also occupies 16 VSRs. The extraction of operands into vector registers in lines 9 and 11 of Algorithm 6.1 must be changed accordingly.

6.3.4 Arbitrary Values for M, D, N

Until now, the algorithms have used values of M, N, and D selected such that a micro kernel with the accumulator arrangement shown in Figure 6.1 could be computed with a single set of assemble and disassemble instructions. However, the implementation of Algorithm 6.1 in LLVM must handle any `llvm.matrix.multiply.*` intrinsic created by any compilation path and thus must handle arbitrary values for M, N and D.

To handle larger values of M and N, the micro-level code-lowering algorithm has an additional outer double-nested loop that logically divides the `COutput` tile into 8×16-element sections as shown in Figure 6.1. Each of these sections can then be handled as shown in Algorithm 6.1. The disadvantage of an input size that spans multiple accumulator sections is that the extraction of data into vector registers becomes more complex. For example, consider a 32-bit data multiplication as shown in Figure 6.1 but with the values of N and M double of what is shown in the figure. The rows of `ATile` and `BTile` shown in Figure 6.1 are now a portion of the rows of larger tiles and the data extraction must gather the correct data into the vector registers that will be used by the accumulators. This data gathering adds additional code and may impact access locality if the tiles are large enough.

Arbitrary values for D are implicitly handled in Algorithm 6.1. The loop on line 5 unrolls the loop as many times as is necessary to perform all accumulations. Thus the number of loads and outer products performed changes automatically as D increases or decreases. The only restriction is that when smaller data types are used and the rank r of the update does not perfectly divide D, the final accumulation will not have enough data to fill the argument registers. In this case, a masked instruction (see Section 5.2.2) must be used to disabled the unused ranks of the input arguments.

6.3.5 Arbitrary Access Order

The code-lowering algorithm also supports inputs and outputs in any access order through modifications to the functions that extract operands and store

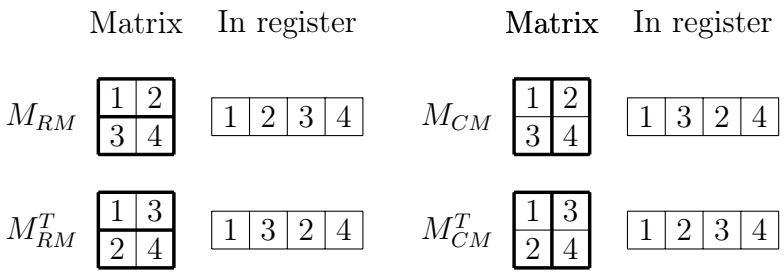


Figure 6.2: Demonstration of transposition access-order modification on data in register..

the results in the accumulators to memory.

As shown in Section 5.2.1, operands in VSR from `ATile` must always be in row-major order while operands from `BTile` must always be in column-major order. Without loss of generality, consider extracting an operand from `ATile`. Using the notation from Section 6.3.3, the final operand VSR must be a row-major $4 \times r$ matrix. Given a column-major matrix argument to the intrinsic, one can extract r length-four vectors from the columns and then interleave (also called zip) them to produce the VSR. The interleaving essentially converts the data access order from column major to row major. If the operand matrix is row major, one can also extract four length-r vectors from the rows and then concatenate to produce the VSR. Extracting operands from `BTile` follows the same logic, except that arguments are column-major and the extraction logic for column-major and row-major matrices are exchanged.

Accumulators can change orientations using a simple mathematical theorem: given matrices X, Y, Z, if $Z = XY$ then $Z^T = Y^T X^T$. In order to apply this theorem to MMA, first consider Figure 6.2. The top left of the figure shows a matrix M in row-major order (M_{RM}) as well as its data laid out in register $(1, 2, 3, 4)$. The bottom left shows M mathematically transposed (M^T_{RM}) while the top right shows M laid out in memory after changing to a column-major access order (M_{CM}). Examining the in-register representation of M^T_{RM} and M_{CM} shows that either transposing the matrix or reorganising its underlying access order produces identical data layouts in register $(1, 3, 2, 4)$. Furthermore, combining the two transformations to produce a transposed M with column-major access order (M^T_{CM}) results in the original data layout in

register, as shown in the bottom right of Figure 6.2. This proves the lemma that M_{RM} is equivalent to M_{CM}^T and that M_{CM} is equivalent to M_{RM}^T in regards to data in register.

The final goal of this proof is to show that there is a simple method by which an accumulator can produce untransposed data in column-major order when the default is row-major. Section 5.2.1 shows how the first argument to an MMA outer-product instruction must be in row-major order while the second argument must be column-major order. Therefore, the original equation can be annotated with access orders like so: $Z_{RM} = X_{RM}Y_{CM}$. Applying the theorem above produces $Z_{RM}^T = Y_{CM}^T X_{RM}^T$. First, consider the right side of the equation. The first argument must still be in row-major order and the second must be in column-major order for the outer product to produce correct results. By the previously proven lemma, a transposed matrix of one access order is equivalent to the untransposed matrix in the opposing order in regards to data in register. Thus, the equation can be simplified to $Z_{RM}^T = Y_{RM}X_{CM}$. Recall that the first outer-product argument must be in row-major order and the second must be in column major order. This simplification shows that after swapping the arguments, each argument's in-register data is already organised correctly for the outer product instruction.

Now, considering the left side of the equation, the end goal is to produce an untransposed Z in column-major order. By the same lemma, the left side can be simplified to produce $Z_{CM} = Y_{RM}X_{CM}$. This completes the proof that simply by swapping the order of the register operands to the MMA outer-product instruction, the data-access orientation of the accumulator is changed without transposing the result matrix.

6.4 Summary

This chapter presented a compiler-only method for the code generation of a high-performance matrix-multiplication kernel. It first introduced LLVM intrinsics and their format with a specific focus on the `llvm.matrix.multiply.*` intrinsic. This particular intrinsic serves as the starting point for the gener-

ation of the matrix-multiplication kernel. As an example, the intrinsic's pre-existing vectorisation lowering serves to introduce the usage and constraints of the intrinsic.

Then, the chapter discussed how the implementation of the intrinsic can produce code targeting a matrix engine, specifically MMA. The examination of an 8×16 kernel formed by `float` elements serves as the computation's base case, demonstrating how arguments are extracted from input matrices and how accumulators can be used to efficiently produce the computation results. The resulting IR is briefly examined in order to show its effectiveness before examining the methods through which the limitations in the base case can be lifted.

The first limitation present in the base case is the lack of usage of the other types supported by MMA. This limitation is lifted through modifications to the loops that extract arguments as well as new logic to place multiple sets of elements into a single VSR. Next, the chapter addressed the dependence on the 8×16 kernel output dimensions as well as an arbitrarily large D dimension. Larger kernel outputs are handled simply by breaking up the problem into tiles that resemble the base case; all computation remains exactly the same. An arbitrarily large D dimension is implicitly handled in the presented method; the only change is that a larger number of loads and outer products are executed. Finally, the issue of arbitrary data access order is resolved. For input matrices, handling a different access order is simply a matter of extracting values from the correct place in the input data and using a shuffle instruction to create an argument that matches the outer product's expected input. For output matrices, this chapter presented the argument that simply exchanging the outer-product argument positions can change the data access order, allowing the kernel to write the output matrix without loss in performance.

Chapter 7

Evaluation

This chapter presents an evaluation of the implementation developed in Chapter 6. The results presented also represent one of the first evaluations of POWER10's MMA facility. The evaluation was performed on a first-silicon version of POWER10 and, as such, the frequency may be different than what will be featured in commercially available versions. The results presented in this chapter are expected to be representative of the hardware that will eventually be available to consumers but the performance of the commercial machines may differ. Possible changes between this version and commercial versions include clock frequency, firmware updates or even silicon updates.

7.1 Experimental Setup

This section presents details on resources and processes that are necessary to produce the results in Section 7.2 and onwards.

7.1.1 Machine Details

The experimental platform is a pre-commercial IBM POWER10 machine that was made available through a research collaboration with IBM. The processor is not yet available to the public but relevant details are presented in Table 7.1. The listed core counts are per socket while the thread counts are per core. Only the L1 caches are listed as all tested matrices collectively fit within the L1 cache. The machine runs Linux with a 64-bit kernel at version 5.10.0-17496-g41bc5268c5e8.

	IBM POWER10
Cores/Threads	15/8
L1i cache	48KiB
L1d cache	32KiB
Frequency	4.00 GHz

Table 7.1: POWER10 experimental environment machine statistics.

7.1.2 Compilation

All binaries are compiled with Clang version 13.0.0[1] at the highest optimisation level (`-O3`) and are tuned to the processor (`-mcpu=pwr10`). All executions are also single threaded; parallelism exists only in the form of SIMD instructions.

Because support for POWER10 was not available in mainstream LLVM when this work began, it has been implemented as part of IBM's variant of LLVM which did have support for MMA instructions. This variant contains several platform-specific optimisations to improve POWER code; all of these specific optimizations have been disabled so that results are representative of mainstream LLVM.

7.1.3 Experimental Methodology

Every measurement presented in this chapter was produced using the Google benchmark library [23].[2] A single measurement is the result of timing the execution of a kernel many times until statistical stability is obtained. The framework implements a method of measuring a kernel by placing it in a loop whose exit condition is dynamically determined by the benchmark being executed. Algorithm 7.1 summarises the algorithm for determining the iteration count.

The body of function `RunBenchmark` is an infinite loop that exits only when the results are statistically stable. Every iteration of the loop calls the function `RunNIterations`. `RunNIterations` executes and times the tested kernel `iters` times and returns statistics. These statistics are passed to the function `ShouldReportResults` that decides if the results of the latest set of

[1] At time of writing this is a development version.
[2] At commit ab74ae5e104f72fa957c1712707a06a781a974a6.

Algorithm 7.1 Algorithm for dynamically determining a statistically stable kernel timing loop iteration count.

```
1: minTime = 1e9                                          ▷ 1 × 10^9 ns →1 s
2: function ShouldReportResults(Results r)
3:     return (r.iters >= 1e9) || (r.totalCpuTime >= minTime)
4: function PredictItersNeeded(Results r, iters)
5:     multiplier = minTime * 1.4 / r.totalCpuTime
6:     isSignificant = (r.totalCpuTime / minTime) > 0.1
7:     multiplier = isSignificant ? multiplier : 10
8:     nextIters = min(round(iters * multiplier), 1e9)
9:     return nextIters
10: function RunBenchmark(Results r)
11:     iters = 1
12:     while true do
13:         results = RunNIterations(iters)
14:         if ShouldReportResults(results) then
15:             reportResults(results)
16:             return
17:         iters = PredictItersNeeded(results, iters)
```

kernel executions have reached stability. Stability is reached when the kernel has executed more than a maximum iteration count (one billion iterations) or the total time spent executing on the CPU (`totalCpuTime`) is longer than a threshold (`minTime`, one second). The framework's default `minTime` is set at half a second; the value was doubled to one second in each experiment in this chapter to simulate a longer running matrix multiplication with many inner kernel executions.

If `ShouldReportResults` returns false, then a new iteration count is decided using `PredictItersNeeded`. `PredictItersNeeded`, in conjunction with `ShouldReportResults`, tries to find the number of iterations that will cause the sum of CPU execution time to be between `minTime` and 1.4 × `minTime`. For benchmarks with variable execution times, this target window is an important part of conservatively determining statistical stability according to the minimum execution time. Aiming for close to or equal to `minTime` can result in less than `minTime` CPU execution time in the final set of kernel executions which can in turn return unstable results. The framework thus aims to overestimate the number of required iterations to ensure that execution time meets

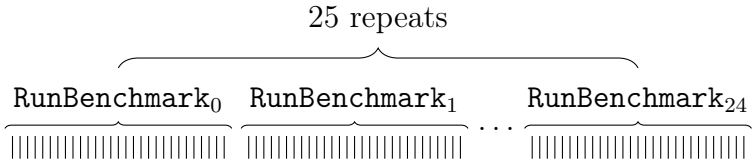


Figure 7.1: Experiment measurement production methodology.

or exceeds `minTime`.

To implement this strategy, `PredictItersNeeded` first creates a multiplier that, if the kernel execution time is very consistent, would perfectly scale the total execution time to be $1.4 \times$ `minTime`. Next, the function decides if the previous total execution time was a significant portion of the desired `minTime` (more than 10%). If it was a significant portion, then the multiplier is unchanged, with the assumption that `iters` $\times$ `multiplier` iterations is enough for the total execution time to be greater than `minTime`. Otherwise, if the total time was not significant, the multiplier becomes ten. Finally, the chosen multiplier is applied to the original iteration count to produce the next iteration count – up to a maximum of one billion iterations – and the process is restarted.

Thus, because the execution time of the kernels presented in this thesis are very consistent, a reasonable summary of the process is as follows: (1) execute the kernel with one iteration; (2) if the total execution time of step one is less than a tenth of a second, repeat step one with ten times the iterations, else go to step three; (3) determine a multiplier using $1.4/\texttt{cpuTime}_{\text{prev}}$; (4) execute the kernel $\texttt{iters}_{\text{prev}} \times \texttt{multiplier}$ times; (5) produce results. As mentioned before, because the execution times of the tested kernels are very consistent, the total execution time of the final set of kernel executions is roughly 1.4 seconds. The final iteration count used in this process is named n for future reference.

One measurement is the result of executing `RunBenchmark` once. In Figure 7.1, producing one measurement corresponds to a single of the lower braces labeled "$\texttt{RunBenchmark}_i$". A total of 25 measurements are produced in this way, corresponding to the upper brace in Figure 7.1, to reduce the impact of

system fluctuations in any execution. Repetitions vary n slightly because the timing iterations may produce slightly different overall times, changing the final multiplier.

Each measurement produces two values alongside the iteration count: the mean CPU execution time and the mean of the number of cycles required by each kernel execution. Vertical bars in plots represent the mean number of cycles and the attached error bars represent the 95% confidence interval across the 25 repetitions. Tables accompany each plot with mean iteration count (n), mean CPU time, mean number of cycles, and their respective confidence intervals.

As an extra precaution against system fluctuations, all kernels execute in a random order, producing a single measurement before having their order reshuffled. Specifically, the process is as follows: (1) shuffle all kernels; (2) execute `RunBenchmark` for each kernel once, producing a single measurement for each kernel; (3) repeat (1) and (2) until 25 measurements have been produced.

7.1.4 Human-Crafted Assembly Code

The following sections compare matrix-multiplication-kernel assembly code generated by the compiler backend with human-crafted assembly code. Moreira et al. describe both the human-crafted assembly code kernel created by a team of IBM engineers and the methodology for producing that kernel [52]. They present information about MMA as well as early results comparing the performance of several kernels when compiled with MMA and VSX on POWER10 and POWER9. The kernels used for comparison in this thesis are updated and improved versions that were provided by the authors of that report.

7.1.5 Types Missing From Analysis

The performance analysis in this section does not include the types `i4` and `bfloat` because of limitations in the backend code generator and the microkernel code generation for `double` is not within the scope of this thesis. The `i4` and `bfloat` types use the same IR code-generation code paths and the same

logic as every other type except `double`. Thus, the code generated for them is likely to be functional and as performant as the code for these other types. However, while translating to assembly, the backend produces incorrect code for `i4` or an error when using `bfloat`. Both issues have been acknowledged by the developer team and will be fixed in future versions of LLVM.

For `double`, the addition of a new operand to the instruction, as described in Section 5.2.3, cause a significant divergence in code-generation logic that otherwise applies to all other types usable with MMA. Within the scope of this thesis, the development efforts focus on an algorithm that worked for six of the seven types. The method presented in Chapter 6 applies to `double` but its lowering implementation will require changes in relation to the original code.

7.2 Caveats

There are some unexpected performance results reported in the experiments detailed in the remaining sections of this chapter. The unexpected performance is caused either by issues in the backend code generator or by limitations inherent to the compilation framework, both of which are outside of the scope of this thesis.

Before presenting the results, this section discusses some of the causes for the unexpected performance results. These uncovered issues are addressable in the future and represent opportunities for improving code generation for the entire POWER software stack. The presented MMA kernel is a new type of kernel never before seen by the backend and therefore the assembly code generation has not had the opportunity to be tuned as other kernels have had. Only after stripping away bottlenecks present in the naïve kernel can one learn about new issues that are still to be addressed by the backend design team.

7.2.1 Spilling

In compiler terminology, a live value is a value that may be used by a statement in the future. Ideally, once a value is brought from memory into a register or it is computed and placed into a register, it should remain in the register until it

is dead — a value is dead when it is guaranteed that it will never be used again. However, there are often not enough registers available in an architecture to keep all live values in register. Therefore, the compiler performs analysis to decide how to judiciously select some live values to be temporarily spilled to memory in order to temporarily free registers for new values. Thus, spilling occurs when a new value must be placed in a register but all registers at that point in the program have a live value. For small matrix-multiplication kernels, all values may be resident in register simultaneously or may only be used once and thus do no remain live across future loads. Efficient spilling requires careful scheduling but in the case of MMA the problem is made more difficult by the following issues.

Framework Vector Loads

The implementation described in Chapter 6 is integrated within a preexisting framework that provides the default vectorisation lowering method described in Section 6.2. This framework contains data structures to support operations with matrices. These data structures transform the load of a single long vector representing a flattened matrix into several vector loads representing the matrix's column or row vectors, depending on the original access order. For example, a floating-point column-major matrix $\underset{(4\times2)}{A}$ is originally represented by `<8 x float>` and then transformed into two `<4 x float>` column-vector loads. After the transformation, these load instructions in the IR remain consecutive and occur before any computation. An examination of the assembly code generated by the backend reveals that such a program is translated by the backend into assembly that brings *all* of the matrix's elements to registers immediately and consecutively. Given that (1) the matrix may be larger than all available registers; or (2) some registers may contain values from other matrices, many of the loaded values are immediately spilled back to memory. These values must be reloaded later when they are needed, effectively tripling the time spent on memory accesses. Memory operations are considerably slower than MMA operations, even though the values may be going to and from the L1 cache, and thus the runtime of the kernel becomes dominated by memory

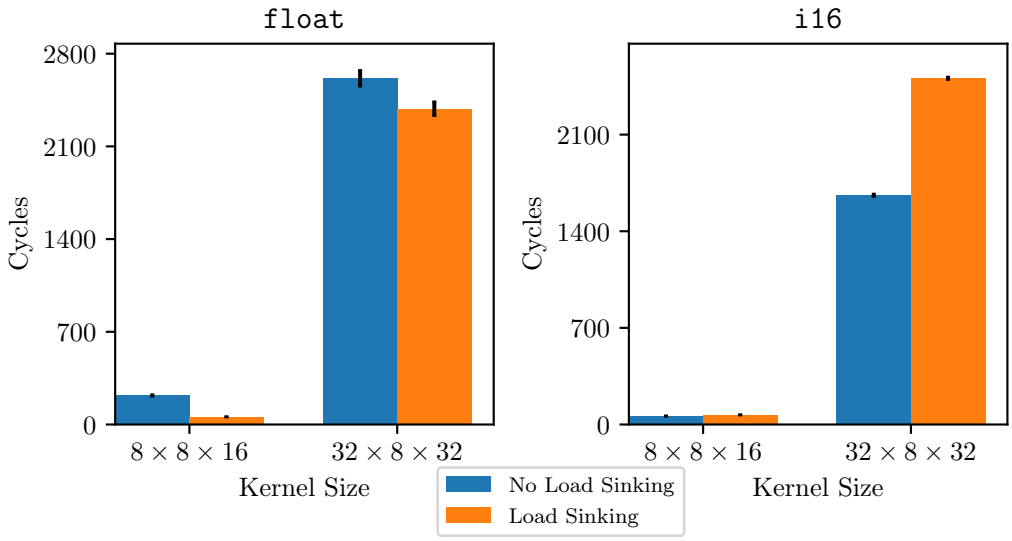


Figure 7.2: The effect of load sinking on `float` and `i16` matrix multiplications.

operations.

This spilling issue must be resolved by the backend scheduler: the scheduler needs to move loads closer to the point in the program that contains the first use of the loaded value to reduce register pressure by using a technique called "load sinking."[3] Load sinking may not reduce register pressure in many other programs, but in the case of this matrix-multiplication kernel it certainly does. A workaround for the spilling that results from unsunk loads, which can be used until the backend scheduler improves, consists of modifying the IR by further breaking up the vector loads into smaller vector loads and sinking these smaller loads closer to the first use of the loaded values.

In preparation for the performance evaluation in this thesis, a short targeted IR transformation was implemented using this tactic. Its goal was to produce results that are more representative of the code that will be generated by a final production-ready backend. Consider the pair of bars labeled "$8 \times 8 \times 16$" ($\underset{(8\times16)}{C} = \underset{(8\times8)}{A} \times \underset{(8\times16)}{B}$) for `float` (left) in Figure 7.2. This matrix size represents the best-case scenario for such an optimisation for three

[3]The backend design team is currently investigating how to implement this change to the scheduler.

Type	Size	Load Sinking	n (millions)	CPU Time (ns)	Cycles
`float`	small	no	25.6 ± 0.62	54.84 ± 1.39	219.11 ± 5.57
`float`	small	yes	97.3 ± 0.14	14.44 ± 0.05	57.67 ± 0.19
`float`	large	no	2.1 ± 0.05	654.30 ± 14.86	2614.07 ± 59.39
`float`	large	yes	2.4 ± 0.05	596.81 ± 13.12	2384.39 ± 52.42
`i16`	small	no	91.4 ± 0.04	15.31 ± 0.00	61.18 ± 0.02
`i16`	small	yes	80.4 ± 0.15	17.42 ± 0.01	69.61 ± 0.03
`i16`	large	no	3.4 ± 0.02	415.63 ± 2.21	1660.53 ± 8.85
`i16`	large	yes	2.2 ± 0.00	627.73 ± 2.19	2507.92 ± 8.75

Table 7.2: The effect of load sinking on `float` and `i16` matrix multiplications. See Figure 7.2 for graphical presentation.

reasons: (1) A's eight-element column-vector load can be broken up into two four-element vector loads, each of which fills an entire VSR (similar logic applies to B's rows); (2) only a single load is necessary to create an outer-product operand (rank-one update); and (3) each operand is used only once during the accumulation step ($2 + 4 = 6$ registers live at once). Given such a perfect case, the runtime of the original version (labeled "No Load Sinking" in Figure 7.2, first row in Table 7.2) is roughly quartered in the optimised version ("Load Sinking", second row). If the operation were to have more accumulations, then this gap should continue to grow as more and more spills would be required in the unoptimised version.

However, by examining the bars labeled "$8 \times 8 \times 16$" for `i16` (right) in Figure 7.2 one can see that for smaller types load sinking actually causes a slight slowdown (Table 7.2, rows 5 and 6). This slowdown can be attributed to the invalidation of properties (1) and (2): loading an `i16` operand no longer fills an entire register and two loads are necessary to create an operand for `i16` (rank-two update). In fact, the assembly produced without load sinking is much closer to the handwritten version because it loads multiple vectors into a single register and uses this register as a source for multiple shuffle instructions. Loading multiple vectors into a single register is possible because the IR of the version without load sinking has all of vectors being loaded simulataneously. When lowered by the backend, this single contiguous load is

automatically broken into register-sized pieces that are then, importantly, also spilled in the same format. Thus when the VSRs are reloaded, they contain two `i16` vectors. The load-sinking version avoids spills by breaking the single contiguous load into individual vector loads, removing the IR opportunity to have the loads merged. It is possible for the backend to recombine the loads but this recombination has yet to be observed in any kernel compilation. These individual vector loads increase register pressure and each loaded VSR is used in a single shuffle instruction. The performance degradation because of load-sinking in this case is small only because the overall register pressure has been decreased by halving the element width.

Operand Spilling and Rematerialisation

After addressing the load-sinking issue in this way, another, separate but related, issue appears. As can be seen from the pair of bars labeled "$32 \times 8 \times 32$" for `float` in Figure 7.2, the same fix that produced significant speedup for a smaller kernel has significantly reduced effectiveness for a larger kernel. An examination of the assembly reveals that the large number of loads that led to spills at the beginning of the kernel prior to sinking the loads are indeed no longer there. However, while there are no spills throughout the body of the smaller kernel, the larger kernel has multiple spills or reloads per set of outer products. The issue here is that, for the larger kernel, property (3) no longer holds true.

Consider the example in Figure 7.3. The figure shows an example large kernel (16×32) in the style of Figure 6.1; to illustrate how spills occur, the figure does not divide individual elements nor does it show B. On the right, darker lines divide the output into blocks of accumulators while the lighter lines divide blocks into the eight accumulators as in Figure 6.1. Each of the smaller boxes on the left represents an operand composed of four `float` elements used in an outer-product computation.

An example operand from A, highlighted by parallel diagonal lines, must be used in the accumulation of the first set of accumulators, highlighted by a crosshatch pattern. In the computation of the smaller kernel in Figure 7.2,

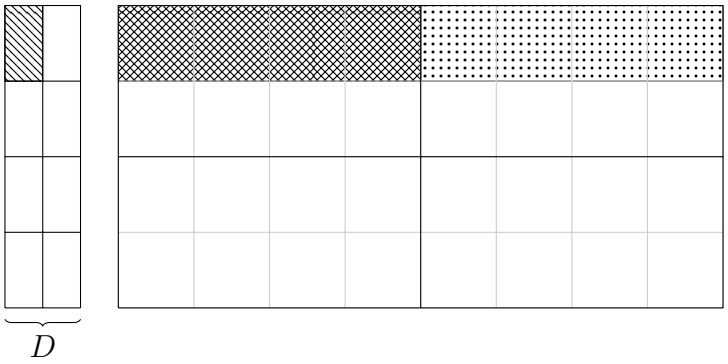


Figure 7.3: Demonstration of operand reuse in large kernels.

this value is now dead. The larger kernel, however, must use the value again in computing the second set of accumulators, highlighted by a dotted pattern.

According to constraint 5 in Section 6.3.1, spilling an accumulator is expensive, and therefore an efficient solution must finish computing a block of accumulators before moving on to a different area of the matrix. Doing so requires that every operand along the dimension D be brought to a register before any operand can be reused with a new block of accumulators. For instance, an accumulator layout like that in Figure 7.3 requires six operand registers for each accumulation to be performed. Using only the 32 registers available while the accumulators are assembled, if $D \geq \left\lceil \frac{32}{6} \right\rceil = 6$, then there is not enough space for all values to remain in registers. And thus the cause for the multiple spills and reloads throughout the computation becomes apparent: operands from A that are still live are being forced out of registers and the compiler is choosing to spill them.[4]

This code generation evokes the well-studied compiler technique "rematerialisation". Essentially, the compiler analysis must answer the question "is it more efficient to store a value to memory and retrieve it later (spill) or to recompute it later using the same process?"

In the case of floats, producing the operand is a simple load and therefore, if an operand *must* be evicted from its register, rematerialising the value is a simple load. Thus, the compiler's choice to spill the operand is incorrect and, because spills are saved on the stack, not only is it wasting memory by

[4]The issue is identical for operands of B, though the reuse distance is greater.

duplicating the value, it is also potentially evicting values from the cache. Regardless, the larger kernel's execution, optimised or not, is significantly influenced by memory operations, though the gains from reducing the initial spills still have significant impact.

For smaller data types, however, spilling is the correct choice. Materialising an `i16` involves two loads and at least one instruction to shuffle the loaded data together. Reproducing the value after spilling becomes a simple load, significantly reducing register pressure and required memory bandwidth. Both the original version and the load-sinking version correctly choose to spill their values instead of rematerialising them. As with `float`, the increased number of memory operations explains the significant increase in execution time for the larger `i16` kernel in Figure 7.2, but it does not, however, explain the large difference between the two version. The next section investigates this discrepancy.

Smaller Types

While the issues affecting `float` matrices presented above remain roughly identical for `double`-typed operations, small data types have a separate source of slowdown. The difference in performance is actually an exacerbation of an already-discussed issue, namely the invalidation of properties (1) and (2). For `i16`, in the load-sinking version, a single 64-bit double-word load instruction loads four `i16` elements at the same time, filling only half of a vector register. The original version loads 16 elements at a time using 256-bit vector-pair loads, meaning that the load-sinking version quadruples the number of loads required to load the values needed for the computation (excluding loads from the original spilling issue). Changing the scheduling to remove the unnecessary loads and stores resulting from spilling would likely lead to a much more significant performance gap between the two versions.

This issue extends to the other data types, which perform rank-two, -four, or -eight updates. The issue is aggravated when the number of memory requests increases because of the shrinking data width that requires more vectors for shuffling. For example, where, for `i16`, the load-sinking version takes

roughly 114% of the execution time of the original version, the same comparison for `i8` shows that the load-sinking version takes 292% of the execution time of the original version.

A more complicated load breaking and sinking strategy may counteract this slowdown, but only to a certain degree. Combining certain loads – for example two four-element `i16` column loads, making a full VSR – would decrease register pressure because the upper and lower halves of the register can be used in separate shuffles. However, this load combination still does not quite match the 256-bit paired-vector load in terms of memory-request efficiency. It is impossible for smaller types to use this 256-bit load because the columns of A are consecutive: the paired-vector load would load values that are not used until the second row of accumulator blocks, inevitably introducing new spills.

The issue worsens for `i8` and `i4` where the eight consecutive vertical elements fill less and less of a VSR. For larger kernels, the smaller data types thus imply a increasing number of loads, further diluting the speedup gained from MMA.

7.2.2 Shuffling

For data types smaller than 32 bits, producing the appropriate operand for a rank-r update requires several vectors be merged via shuffling. The POWER ISA offers several methods to move vector elements, ranging from register shifting and rotating combined with masking to efficient vector upper and lower half merges based on data type size. Transforming IR statements in an efficient code for shuffling is an involved design process.

There are a multitude ways to express the same shuffling operation in IR, each depending on multiple implementation choices made throughout the process. Certain variants are interpreted by the backend and translated to efficient code while others are interpreted very conservatively. This discrepancy in translation can be the difference between an idiomatic assembly of several instructions, often close to Moreira et al.'s handwritten version, and the same process being expressed in hundreds or thousands of instructions, often with spills throughout. Several factors that lead shuffling to cause performance

degradation are discussed below.

Data Type: `half`

The `i16` and `half` types have identical widths: a halfword. Because CPUs and their accompanying ISAs are type-agnostic and sensitive only to the width of a data element when implementing vector shuffles, shuffle code in the IR that differs only in choice of 16-bit type should produce identical assembly. However, the assembly code currently generated by the compiler backend for `i16` executes roughly six vector loads and six vector permute instructions to produce all operands for a single accumulation in a full 2×4 accumulator layout. For `half`, the generated assembly code has gone from six instructions to almost 150 instructions because the computation is partially scalarised by moving elements from vector registers to general-purpose registers and back again.

This explosion in code size is most apparent in the calculation epilogue where, in the IR, the accumulators are disassembled and stored to memory. First, the resulting accumulator vectors are truncated to the output data type. They are then concatenated together into an output flattened matrix and stored to memory. Concatenation is also achieved through shuffling and therefore experiences the same conservative code generation as before. The added truncation operation worsens the effect. All told, in the assembly code currently generated for `half`, the epilogue is roughly ten times the length of the same operation for `i16`.

Data Type: `i4`

The POWER10 ISA has instructions that merge vectors at a per-byte granularity, but not at a sub-byte granularity as is required for `i4` code generation. Therefore, to shuffle a vector of `i4` elements using the same IR code generation, the backend generates significantly less performant code that includes vector rotations and scalarisation, with single bytes being moved individually to and from general purpose registers. The consequent slowdown negates much of the speedup that could be achieved via the MMA rank-eight update.

7.2.3 Solutions

This section presents potential solutions to the performance degradations uncovered by the experimental evaluation. The successful integration of these and other ideas in the compiler requires further investigation and cooperation with backend developers.

Spilling

The simplest solution to the spilling issues is for developers to focus on using only small kernels and, indeed, this is not so unreasonable a proposal as one might perceive it at first. LLVM requires that valid IR code is always able to be lowered to assembly. However, it does not require that the generated code be as performant as possible, as is demonstrated by the above issues with shuffling `i4` vectors. Choosing to avoid larger kernels may seem like a defeatist notion but is actually a tried-and-true approach to performance in compilers: make the common case fast.

The average user interacts with a compiler through a high-level language, not its IR. Thus, the frontend will choose the fastest solution to implement a user's code; if the fastest implementation is a small kernel encapsulated by an outer kernel then that is what will be chosen. With this notion, optimisation can be focused on this most common case for which a solution is known. A larger inner-kernel will always remain usable but it will be less performant.

Furthermore, the most performant library implementations focus on using small unrolled assembly kernels, often computing kernels with dimensions of four to eight elements. These small kernels are then surrounded by memory-managing outer kernels as described in Section 4.3. This strategy is used by Kuzma et al. [41] while using the inner kernel presented in this thesis with results comparable to libraries that feature kernels written in assembly. Thus, using only smaller kernels can be seen as a well-worn path to performance.

If a larger kernel *must* be used, significant investigation must be done to test load scheduling, value rematerialisation, load breaking, and, if necessary, load recombination. It is unlikely that such a solution will be found within

the IR because (1) it is in SSA form which often folds identical values together without consideration for the times when a value is live; and (2) for better or worse, the backend does not have to respect IR operation order. Such a solution will have to be found in the backend.

Shuffling

Unlike issues with spilling, it is likely that performance degradations caused by misbehaving shuffles can be solved by creating type-specific shuffling code. While injecting the proposed solution code into the original vectorisation framework is possible, it would be much more effective to first refactor the framework so that such a modification is easier.

For `half` vectors, it may be possible for the compiler to generate code in a similar fashion as is done for `i16` simply by reinterpreting data as `i16` via casts and manipulating the data in this state. The cast is not a conversion, simply a reinterpretation of the bits in a register and it is meaningful only within the LLVM framework. Such casting should produce no new instructions. The data can be casted back to `half` at no cost when it needs to be used in a computation. In theory, this casting produces much better assembly but may also trigger a very conservative response in the back-end of the compiler, worsening the code significantly with scalar element movement derived from a perceived conversion hazard.

The sub-byte granularity of `i4` vectors poses a different problem. Performant shuffling code for `i4` in the same style as the larger types is an impossibility. Instead, clever use of bitwise operations must explicitly be used to implement efficient element extraction and movement. Bitwise operations can be applied across an entire vector register, removing the need for scalarisation and improving register usage by reducing the number of required intermediate registers.

Evaluation Disclaimer

The load-sinking and shuffling code transformations are not applied to any of the experimental evaluations presented in the remaining sections of this

chapter because no tested transformation positively affected all cases. Advantaging one type or one size of kernel would give a distorted view of the performance that can be achieved in the machine. Thus, all test cases use identical IR generation logic and are equally advantaged and disadvantaged in code generation.

7.3 Data Access Orders

As discussed in Section 4.3, packing is very important for matrix-multiplication performance. An important part of packing, beyond choosing dimensions, is the choice of data access order. Section 5.2.1 shows that MMA is sensitive to the access order of elements in VSRs and, therefore, also sensitive to the access order of the matrices in memory from which values are loaded into registers.

7.3.1 Setup

When using an unmodified version of LLVM, the access orders of A, B, and C are tied together and must be equal. Thus two test cases, one all-column-major ($\mathrm{C} = \mathrm{C} \times \mathrm{C}$) and another all-row-major ($\mathrm{R} = \mathrm{R} \times \mathrm{R}$), constitute the baseline for the experiment. In order to test the performance of the hypothesised best mix of orientations, further modifications to the lowering framework were performed to enable the mixing of orientations for the MMA lowering strategy. With this possibility enabled, two cases are added to the comparison: both use the proposed optimal layout for A and B but vary the layout of C ($\mathrm{C} = \mathrm{C} \times \mathrm{R}$ and $\mathrm{R} = \mathrm{C} \times \mathrm{R}$)

Each kernel uses a 2×4 accumulator setup and, therefore, in keeping with the suggestion in Section 7.2.3, uses a small 8×16 output. The inner most dimension of the kernel is 32, making the overall computation $\underset{(8\times16)}{C} = \underset{(8\times32)}{A} \times \underset{(32\times16)}{B}$. An inner dimension of 32 requires 32 accumulations but smaller data types perform rank-r updates, effectively reducing the number of operations required to compute the kernel. It may seem that 32 accumulations is relatively large, but work by Kuzma et al. [41] determines that, even given the poor load scheduling, up to 128 accumulations per kernel gives good speedup, with

performance plateauing with any more accumulations. Given the discussion in Section 7.3.2, more accumulations implies more opportunities for a "good" orientation to outperform a bad orientation, effectively highlighting differences in performance.

7.3.2 Expectation

A column-major orientation data layout for A should be optimal. For `float`, a VSR, which is a row-major 4×1 matrix according to the ISA, is indistinguishable from a single column of four elements when loaded from memory. Thus, if A is laid out in column-major order, an operand can be created with a single load and no waste. However, if A is laid out in row-major order, it is likely that $D \neq 1$ and thus the elements in the first column of A are not consecutive in memory. Therefore, to create an operand, four loads must be issued, one for each element of the operand. Further overhead is incurred to move elements into the vector register either from scalar registers if four scalar loads were issued or to shuffle elements from four vectors that were loaded. Thus, a column-major order layout is highly preferred. The same logic can be applied to B, except that a row-major orientation is preferred.

For smaller types, the logic is more complicated because the byte length of the overall loaded data also shrinks. In the interest of not repeating much of the discussion in Section 7.2, the same logic as above can be applied while acknowledging that as register usage efficiency decreases speedups also decrease.

An interesting point of discussion exists for types `i8` and `i4` whose in-register dimensions are 4×4 and 4×8 respectively. With two `i4` elements packed into a byte, an eight-element vector only occupies four bytes and, therefore, `i8` and `i4` require loading four four-byte vectors to provide the elements to fill a VSR. Thus, four loads must be issued when extracting operands from either a row-major or a column-major order matrix. In this case, either orientation is acceptable when operands are being built in isolation. However, if several operands are being built simultaneously, loads can be combined to increase efficiency. The need for shuffle instructions does not decrease, only the indices from which shuffles extract elements in a vector change. This is com-

Type	Order	n (millions)	CPU Time (ns)	Cycles
`float`	$C = C \times C$	10.7 ± 0.04	130.96 ± 0.47	523.21 ± 1.86
`float`	$R = R \times R$	13.8 ± 0.33	100.21 ± 0.33	400.36 ± 1.32
`float`	$C = C \times R$	17.6 ± 0.33	79.64 ± 1.50	318.18 ± 6.01
`float`	$R = C \times R$	18.0 ± 0.30	77.92 ± 1.31	311.31 ± 5.21
`i16`	$C = C \times C$	22.0 ± 0.04	63.79 ± 0.14	254.86 ± 0.57
`i16`	$R = R \times R$	27.7 ± 0.06	50.64 ± 0.12	202.32 ± 0.49
`i16`	$C = C \times R$	31.5 ± 0.10	44.48 ± 0.15	177.70 ± 0.59
`i16`	$R = C \times R$	31.5 ± 0.08	44.44 ± 0.11	177.57 ± 0.44
`half`	$C = C \times C$	11.6 ± 0.00	120.50 ± 0.03	481.41 ± 0.11
`half`	$R = R \times R$	5.5 ± 0.00	256.06 ± 0.18	1023.01 ± 0.71
`half`	$C = C \times R$	2.0 ± 0.01	691.33 ± 4.31	2762.04 ± 17.23
`half`	$R = C \times R$	3.2 ± 0.03	438.00 ± 4.54	1749.89 ± 18.12
`i8`	$C = C \times C$	37.9 ± 0.03	36.98 ± 0.02	147.75 ± 0.10
`i8`	$R = R \times R$	40.8 ± 0.44	34.38 ± 0.39	137.36 ± 1.57
`i8`	$C = C \times R$	41.8 ± 0.19	33.50 ± 0.15	133.82 ± 0.61
`i8`	$R = C \times R$	41.8 ± 0.20	33.49 ± 0.16	133.79 ± 0.64

Table 7.3: The effect of matrix access order on performance. See Figure 7.4 for graphical presentation.

bination transformation is only possible if access orders match the expected access orders for larger types, i.e. column major for A and row major for B. This change can have a small, but noticeable, impact on the results.

The initial expectation is that C should be stored in row-major orientation. This is simply because the simplest method of using MMA places the operand from A as the first argument and the operand from B as the second argument in the outer product instruction. When used in this way, the elements in the accumulator are oriented in a row-major fashion and, after disassembling, there is no extra work required before storing the underlying vectors to memory. Transposing the output so that it can be stored into a column-major output-matrix requires extra shuffles. However, the theorem discussed in Section 6.3.5 states that reversing the order of the arguments causes the result in the accumulator to be equivalent but in a column-major order. Thus, either access order can be used for C without loss of performance.

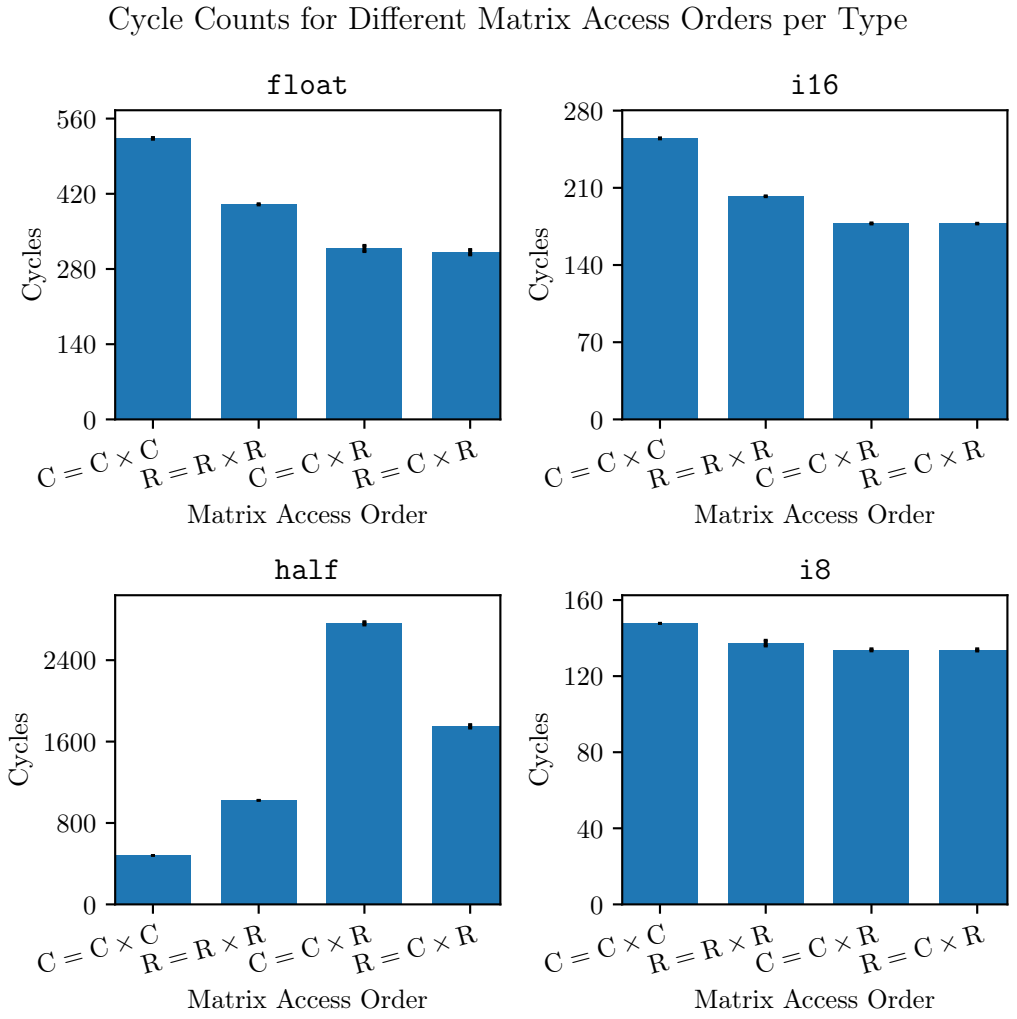


Figure 7.4: The effect of matrix access order on performance.

7.3.3 Analysis

Figure 7.4 and Table 7.3 show the effect of matrix access order on performance. For `float`, `i16`, and `i8`, the numbers are inline with the performance hypothesised in Section 7.3.2 (see below for discussion of `half`). The two order-locked variants ($C = C \times C$ and $R = R \times R$) perform worse than the two variants with optimal layouts ($C = C \times R$ and $R = C \times R$). Furthermore, between the third and fourth bars, representing the two kernels whose A and B are laid out optimally but with differing access orders for C, varying the access order of C has no perceptible effect on performance.

Excluding `half`, there is a noticeable difference in performance between

the two access-order-locked variants, R = R × R and C = C × C, represented in the first two columns of the plots. Based on the discussion in Section 7.3.2, the source of the discrepancy can be derived easily. Given that the order of C does not affect performance, the difference must be found in A and B. In each case, the access order of either A or B matches the preferred order while the access order for the other matrix does not. Because B is twice the size of A, when B has an incorrect data layout, the generated assembly performs a significantly greater number of inefficient loads than when A has an incorrect layout. Examining the generated assembly shows that for `float`, the column-only variant executes roughly 450 more instructions of "setup" (loading and shuffling values) before executing the first outer-product instruction; the generated code is effectively identical afterwards. This gap lessens as types become smaller due to the smaller overall amount of bytes being loaded, but the difference in performance remains.

Evaluating `half`

As discussed in Section 7.2.2, `half` and `i16` can and should be manipulated in exactly the same manner. The only difference in their generated assembly should be the outer-product instruction which interprets a register as a `half` or as an `i16`. The POWER ISA indicates that the two different outer product instructions also have identical latencies. Thus, given identical load, store, and shuffle schedules along with identical outer-product latencies, the performance of the two types should be identical in every instance.

However, given the results in Figure 7.4, it is clearly not the case that `i16` and `float` are identical in terms of performance. The most performant case for `half`, C = C × C, still requires roughly three times the cycles of `i16`'s least performant cast, also C = C × C. Results for `half` that do not follow the anticipated trend are a common thread in this chapter and are caused by a code length explosion due to extremely conservative assembly code generation (see Section 7.2.2). For this reason, performance can be expected to drastically increase and trends expected to follow that of `i16` when this issue is resolved. Such a change is thus likely to invalidate any of the results presented for `half`

in this and future experiments. Results for `half` will continue to be presented for completeness.

7.4 Varied Accumulator Layout

The layout of accumulators can significantly affect the performance of a kernel. Both the input and output memory efficiency as well as the in-register performance can degrade due to a suboptimal layout.

7.4.1 Setup

Intuitively, leaving an accumulator unused means unused functional units and therefore wasted performance; thus, any layout which aspires to be a contender for best performance must use all eight accumulators. The implemented framework does not allow for irregularly shaped layouts. Combining these two restrictions creates four test cases: two rectangular layouts, 2×4 and 4×2, and two linear layouts, 1×8 and 8×1.

Again, using the logic of Section 7.2.3, the kernel size is fit to the accumulator layout. The dimensions of a kernel are, therefore, four times the length of the same dimension of the accumulator layout (e.g. layout 8×1 has kernel dimensions 32×4). As in the previous experiment, each kernel performs 32 accumulations, though smaller types perform fewer outer-product instructions. Therefore, the overall kernel size is $\underset{((\texttt{V}\times 4)\times(\texttt{H}\times 4))}{C} = \underset{((\texttt{V}\times 4)\times 32)}{A} \times \underset{(32\times(\texttt{H}\times 4))}{B}$ where `V` and `H` correspond to the number of vertical and horizontal accumulators in the layout as described in Section 6.3.1. Additionally, using the result from Section 7.3, all kernels use a R = C × R layout for their matrices.

7.4.2 Expectation

A layout and its transpose (2×4 and 4×2, 1×8 and 8×1) are expected to have equal performance. An accumulation consists of: loads from A; loads from B; shuffles for each if necessary; and finally computation of the outer product. Transposing a layout means swapping the number of loads from A and B, hence, the overall number of loads remains the same. The number of

shuffles and outer products also remain the same because the same number of operands must be produced for the same number of outer-product accumulations. Thus, the total work remains the same and therefore the performance should be identical. Any difference in performance will certainly indicate a missed opportunity in one of the compilations.

The highest factor of reuse for the eight accumulators available in POWER10 MMA occurs for the 2×4 layout and its transpose 4×2 (see Section 6.3.1) with operands from one matrix being reused four times and an operand from the other matrix being used twice. These layouts reduce register pressure by requiring only six operands to be live before performing eight consecutive accumulations. With 32 registers available, using only six of these registers per accumulation allows for the operands needed for future accumulations to be brought to registers ahead of time or even for multiple sets of accumulations to be performed without intervening loads.

However, the argument in Section 6.3.1 is made under the assumption that the most efficient schedule for outer-product instructions is a large set of loads followed by as many outer products as possible. In this case, the rectangular layouts should be more performant because they require less operands per accumulation (six instead of eight) and therefore allow more accumulations to be setup simultaneously. If it is not the case that a group of loads followed by outer products is the most performant option, and instead interleaving loads with outer products is more performant, then either set of layouts can potentially be more performant, as long as the latency for the larger number of loads required to setup the linear layout can be effectively hidden.

An important cause for slowdown is also the difference in raw element count between linear and rectangular layouts. A rectangular layout loads a total of $8 \times 32 + 16 \times 32 = 768$ elements while a linear layout loads a total of $4 \times 32 + 32 \times 32 = 1152$ elements. Both the rectangular and linear layouts result in 128 elements each with 32 accumulations applied, albeit in different orientations. This is an important consideration for an outer kernel which, when using a rectangular layout, can pack less elements to achieve the same throughput or pack the larger number of elements (if it can afford it) and

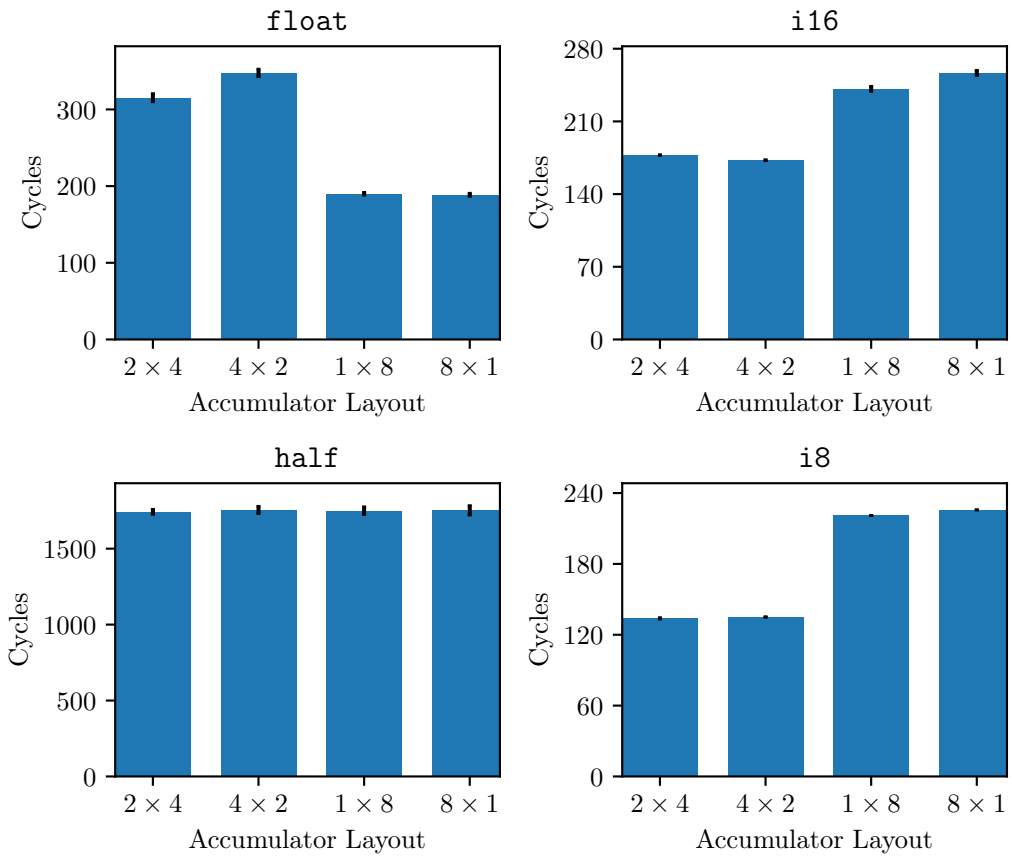


Figure 7.5: The effect of accumulator layout on performance.

receive 48 accumulations instead of 32. In this experiment, the lower number of total loads should result in a smaller execution time for the rectangular layouts.

7.4.3 Analysis

Figure 7.5 and Table 7.4 show the effect of accumulator layout on performance. The results for each of `float`, `i16`, and `i8` follow the expectation in Section 7.4.2 that both rectangular layouts (2×4, 4×2) have similar performance and both linear layouts (1×8, 8×1) have similar performance. However, where `i16` and `i8` agree with the theory that rectangular layouts should outperform linear layouts, `float` shows opposing results.

An initial analysis of this difference may hypothesise that, for `float`, the

Type	Layout	n (millions)	CPU Time (ns)	Cycles
`float`	2×4	17.8 ± 0.30	78.91 ± 1.34	315.27 ± 5.34
`float`	4×2	16.1 ± 0.23	87.01 ± 1.25	347.62 ± 5.00
`float`	1×8	29.5 ± 0.30	47.56 ± 0.47	190.00 ± 1.88
`float`	8×1	29.7 ± 0.33	47.21 ± 0.53	188.61 ± 2.13
`i16`	2×4	31.5 ± 0.07	44.45 ± 0.10	177.59 ± 0.38
`i16`	4×2	32.4 ± 0.10	43.22 ± 0.13	172.67 ± 0.52
`i16`	1×8	23.2 ± 0.24	60.40 ± 0.62	241.30 ± 2.47
`i16`	8×1	21.8 ± 0.22	64.25 ± 0.63	256.70 ± 2.50
`half`	2×4	3.2 ± 0.03	436.14 ± 4.18	1742.47 ± 16.70
`half`	4×2	3.2 ± 0.04	439.36 ± 5.97	1755.33 ± 23.86
`half`	1×8	3.2 ± 0.05	438.08 ± 6.38	1750.21 ± 25.50
`half`	8×1	3.2 ± 0.06	438.61 ± 7.87	1752.32 ± 31.46
`i8`	2×4	41.8 ± 0.20	33.51 ± 0.16	133.87 ± 0.64
`i8`	4×2	41.4 ± 0.11	33.79 ± 0.08	135.02 ± 0.32
`i8`	1×8	25.3 ± 0.01	55.31 ± 0.02	220.98 ± 0.08
`i8`	8×1	24.8 ± 0.02	56.50 ± 0.05	225.75 ± 0.18

Table 7.4: The effect of accumulator layout on performance. See Figure 7.5 for graphical presentation.

1×8 layout and its transpose had found a better load schedule or that it had activated hardware such as the stream prefetcher in a way that was not possible with the data types that loaded less data overall. It is true that the code generator found a better schedule, though not in the way that was anticipated. Instead, changing the the source and usage of operand registers has simplified the backend code generator's analysis, likely its liveness analysis, allowing it to find ways to reduce register interference and better manage the live values in order to create a simpler load schedule. This simplification has resulted in the removal of the initial spills present in every test case presented so far. In fact, given the kernel laid out for it, the assembly is very close to the handwritten assembly from Moreira et al., lacking only the small code transformations to combine 128-bit vector loads which use consecutive addresses into a paired-vector 256-bit load. For comparison's sake, activating the load-sinking code transformation for the rectangular layouts produces assembly resembling al-

most exactly the assembly in the tested linear layouts.[5] By examining the assembly for the rectangular test case with load sinking, however, it is easy to see that the total number of vectors loaded between accumulations is less than that in the linear layout. In both cases the only instructions present are the loads and the outer products which means there are no instructions with which to hide a stall due to load latency. Thus it can be expected that equivalently-scheduled `float` testcases would follow the expected trend because, with less loads, there is less likely to be a load-based stall. Following the results in Figure 7.2, fixing the spilling issue for the `float` rectangular layouts should quarter the number of cycles required. Such a decrease in the 2×4 and 4×2 bars would cause the `float` plot to follow the same trend as `i16` and `i8`.

Regarding `i16` and `i8`, the shuffles necessary to build operands serve to add extra complication to the liveness analysis and thus the same simplified schedule cannot be found for the linear layouts. Therefore, both the rectangular and linear layouts are affected by spills and can be compared directly. As explained in Section 7.4.2, the linear layouts were significantly slower. The lower amount of operand reuse and significantly larger number of required loads significantly slowed down the execution of the kernels.

7.5 Vectorisation and MMA

The closest compiler-only method to generate generic matrix-multiplication code is the preexisting vectorised lowering of the `llvm.matrix.multiply.*` intrinsic. Thus, that method is used as the baseline for speedup induced by the implementation of the MMA-targeted intrinsic lowering presented in this thesis.

7.5.1 Setup

Using the knowledge gained from the previous experiments, this experimental evaluation uses the optimal settings for the MMA lowering: an $\mathrm{R} = \mathrm{C} \times \mathrm{R}$ data

[5] The load sinking assemblies also contain the more efficient paired-vector loads though it is likely due to a code-transformation pass ordering issue, not deliberate choice.

orientation and a 2×4 accumulator layout.[6] This accumulator layout implies an 8×16 kernel output. For consistency, 32 accumulations are performed, making the overall computation once again $\underset{(8\times16)}{C} = \underset{(8\times32)}{A} \times \underset{(32\times16)}{B}$.

While the vectorisation method performs the same computation, some elements must be changed. As discussed in Section 6.2, the vectorisation method supports only all-column-major or all-row-major data layouts; both are tested. The implementation is based on the outer-product emulation method described in Section 4.4.2 and depends on this order. Two extra lowering options that apply only to the vectorisation method have been set: one enables the use of FMA instructions and the other enables operation fusing. While there is not a second matrix operation with which to fuse the matrix multiplication, operation fusing transformation has the side effect of sinking all loads used by the lowering. Because this lowering uses the outer-product-emulation method, load sinking does not break up loads in the same way it does for the MMA lowering, making it always beneficial.

7.5.2 Expectation

The comparison of human-crafted VSX and MMA kernels reported by the team of IBM engineers indicates that MMA attains approximately a two-times speedup over the same kernel using VSX [52]. It is unlikely that either of the two kernels compared in this experiment achieve the maximum performance possible because these kernels are automatically generated by lowering the `llvm.matrix.multiply.*` intrinsic and were not inspected for further improvements by a human expert. However, both kernels are equally disadvantaged when it comes to the initial spilling issue, both are subject to the same kernel unrolling scheme (i.e. completely unrolled), and both use the same register-allocation policy. Therefore, any code-improvement opportunity is either present or missing in both kernels and thus the two-times improvement found by Moreira et al. is a reasonable expectation for this experimental comparison as well.

[6] Despite the results for `float` in Section 7.4.3, the analysis indicates that 2×4 is optimal. Choosing 2×4 also ensures that all types are equally disadvantaged by spilling.

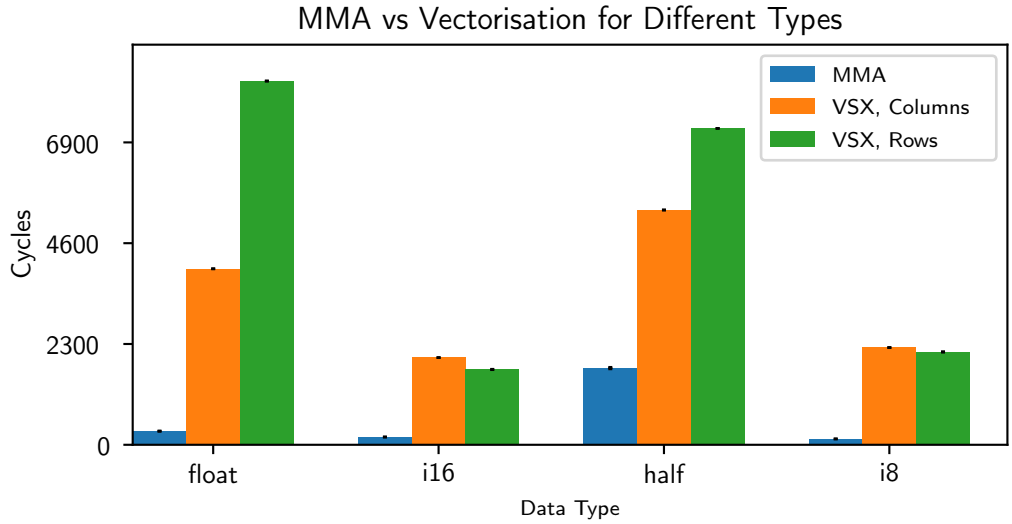


Figure 7.6: Comparison of MMA and vectorised lowerings of `llvm.matrix.multiply.*`.

The change in data access order in the vectorised version is not necessarily a disadvantage. The lowering is already set up to extract vectors from A, scalars from B, and insert results in to C. All extractions and operations are created with the vectoriser in mind. The lowering decouples operations from each other as best as possible, enabling the vectoriser to effectively combine operations into a single vector operation.

7.5.3 Analysis

Figure 7.6 and Table 7.5 show the effect of MMA on the lowering of the `llvm.matrix.multiply.*` intrinsic. The lowering for all four investigated types are significantly improved by using MMA. Comparing for each type the MMA lowering and the most performant vectorised lowering, there is a speedup, on average, of 12.9 times for `float`, 9.7 times for `i16`, 3.1 times for `half`, and 15.8 for `i8`.

While the speedup for `half` is likely to be changed due to improvements for both the vectorised and MMA lowerings, the fact remains that there is still a significant speedup. Fixing the code generation issues for `half` should bring these results to match those presented for `i16`.

Type	Compilation	n (millions)	CPU Time (ns)	Cycles
`float`	MMA	18.0 ± 0.30	77.92 ± 1.31	311.31 ± 5.21
`float`	VSX, Columns	1.4 ± 0.00	1005.16 ± 0.24	4015.84 ± 0.96
`float`	VSX, Rows	0.7 ± 0.00	2077.55 ± 0.40	8300.11 ± 1.59
`i16`	MMA	31.5 ± 0.08	44.44 ± 0.11	177.57 ± 0.44
`i16`	VSX, Columns	2.8 ± 0.00	497.98 ± 0.26	1989.60 ± 1.02
`i16`	VSX, Rows	3.3 ± 0.01	430.18 ± 1.29	1718.67 ± 5.14
`half`	MMA	3.2 ± 0.03	438.00 ± 4.54	1749.89 ± 18.12
`half`	VSX, Columns	1.0 ± 0.00	1340.99 ± 0.41	5357.58 ± 1.63
`half`	VSX, Rows	0.8 ± 0.00	1807.24 ± 0.22	7220.21 ± 0.92
`i8`	MMA	41.8 ± 0.20	33.49 ± 0.16	133.79 ± 0.64
`i8`	VSX, Columns	2.5 ± 0.00	555.32 ± 0.70	2218.66 ± 2.81
`i8`	VSX, Rows	2.6 ± 0.01	530.25 ± 1.56	2118.45 ± 6.23

Table 7.5: Comparison of MMA and vectorised lowerings of `llvm.matrix.multiply.*`. See Figure 7.6 for graphical presentation.

As for the three remaining types, the large difference in cycle counts between MMA and the vectorised lowering is due to inefficiencies in the vectorised lowering rather than exceptional performance increases from MMA. The results presented by Moreira et al. are likely at, or near, the theoretical peak performance for MMA and these results do not match them due, in large part, to the spilling issue. Thus, if these results do not improve upon Moreira et al., the difference certainly must come from suboptimal code generation for the vectorised method. Indeed, examining the vectorised lowering's assembly code reveals a spill or a reload between nearly every calculation, roughly every three instructions, even with the load-sinking transformation enabled. The only improvement, which is inherent to the method and not a result of code-generation improvement, is the lack of shuffling instructions for the smaller types.

Regardless of the reason for a particularly slow vectorised schedule, the results presented here represent the current best intrinsic lowering possible by the POWER10 code generation backend. Likewise, in spite of the issues and possible improvements noted in Section 7.2, the results for the MMA lowering represent a significant and tangible improvement in performance for matrix multiplication on POWER10. Both methods will improve in parallel

with general improvements to the backend, but the MMA lowering will always show a noteworthy improvement over the vectorised version.

7.6 Human-crafted Kernel Comparison

While the vectorised lowering of `llvm.matrix.multiply.*` is the most comparable point of reference when it comes to improving upon compiler-only methodologies for matrix multiplication, another important point to study is overall achievable performance. To this end, the investigation looks to comparisons with the kernels developed by the IBM engineers [52].

7.6.1 Setup

Because the human-crafted kernels do not fall victim to the spill issues, it is important to choose reference points that can demonstrate both the current state of the proposed lowering as well as the future potential within the kernels. As a result, three points of comparison are chosen.

The first two are the intrinsic lowering for `float`, one with no transformations applied and the second with the load-sinking transformation applied. These two executables demonstrate the current and future state, respectively, of the proposed methodology.

The final point of comparison is the handwritten single precision floating point kernel from Moreira et al. Since publishing their work on MMA, the kernels in question have been improved by the authors and thus are expected to surpass the performance presented in that work. This experiment compares against the performance of the newer version of the kernels.

Unfortunately, it is impossible to integrate the handwritten kernel directly into the benchmarking framework loop due to the explicit use of registers which results in overwritten values and segmentation faults. Therefore, the kernel is separated into its own function. This separation, however, means that the benchmarking framework is timing both the kernel and a function call, offsetting the timing slightly. To counteract this, a benchmark that times only a function call was added. The results in Section 7.6.3 for the handwritten

Name	n (millions)	CPU Time (ns)	Cycles
No Load Sinking	18.0 ± 0.30	77.92 ± 1.31	311.31 ± 5.21
Load Sinking	35.1 ± 0.04	39.89 ± 0.01	159.36 ± 0.05
Handwritten	31.5 ± 0.05	41.15 ± 0.07	164.39 ± 0.27

Table 7.6: Comparison of default and load-sinking lowerings of `llvm.matrix.multiply.*` with a handwritten MMA kernel. See Figure 7.7 for graphical presentation.

kernel thus report the average of `kernelCycles` − `fnCallCycles`.

In all cases, the kernels use `float` elements with a R = C × R data layout to compute a $\underset{(8\times16)}{C} = \underset{(8\times32)}{A} \times \underset{(32\times16)}{B}$ kernel.

7.6.2 Expectation

Given the results in Figure 7.2, the intrinsic version without load sinking should be significantly slower than the version with load sinking applied. This experiment, however, has quadrupled the number of accumulations that were performed during the analysis performed in Section 7.2. The analysis in the same section expects the gap between the intrinsic versions to have grown because a greater number of spills should be avoided by load sinking.

The handwritten kernel should represent the epitome of instruction choice and scheduling for this matrix-multiplication kernel. With this in mind, it should be expected that the handwritten kernel outperforms both of the intrinsic lowerings. However, previous examinations of the assembly resulting from a compilation with load sinking has shown that the assembly contains no spurious instructions, only outer product and load instructions. Therefore, the intrinsic lowering with load sinking should be very close in performance to the handwritten kernel.

7.6.3 Analysis

The experiment produced the results in Figure 7.7 and Table 7.6. The difference between the two intrinsic lowerings is significant as expected, though the proportion of the difference is unexpected. In Figure 7.2, the load-sinking

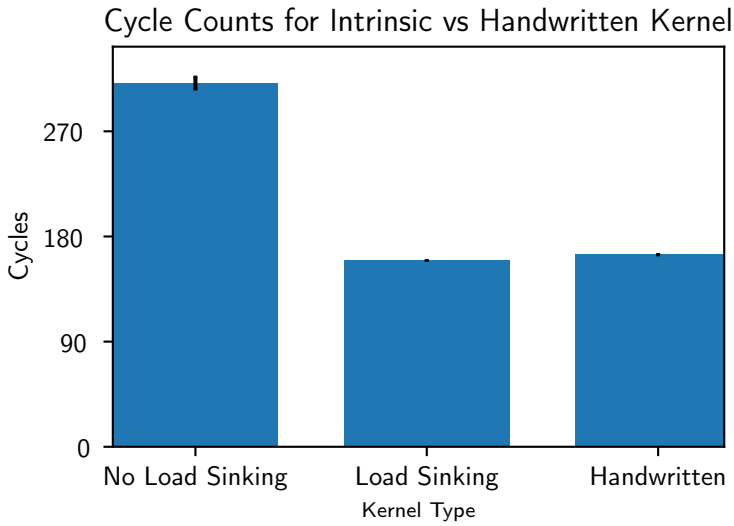


Figure 7.7: Comparison of default and load-sinking lowerings of `llvm.matrix.multiply.*` with a handwritten MMA kernel.

version takes roughly a quarter of the cycles that the non-load-sinking version takes to execute. This proportion was expected to increase as the number of accumulations increased. Contrarily, the proportion has decreased to about one half, suggesting that, as accumulations increase, the non-load-sinking instruction schedule improves or the load-sinking instruction schedule degrades performance. A possible explanation is that in the assembly, after the spills in the non-load-sinking version, load instructions are interleaved much more freely with outer product instructions than in the load-sinking version, which has clear distinctions between load and outer-product portions of the code. This finding points to a potential direction for improvement after the spilling issues are solved.

When comparing the load-sinking lowering and the handwritten kernel, the load-sinking lowering performs better than expected. In fact, according to Table 7.6, the generated kernel outperforms the handwritten kernel by several cycles. However, upon inspecting the assembly, the instructions emitted in each kernel are identical and only the registers used differ. This observation points to a potential great achievement by the code-generation algorithm because it implies that, once the spilling issues are resolved, the code generated

by the compiler will be on-par with handwritten code. Given that there is still the potential for improvements to scheduling that may be derived from the difference between the two generated lowerings, there is also the potential that the generated kernel will eventually exceed the performance of Moreira et al.

While the slight advantage in Table 7.6 remains a significant and promising result, given that the two assembly codes are nearly identical, it is difficult to conclude that one truly outmatches the other. It is possible that different register choices may save several cycles by removing output or write-after-read dependences, but it is far more likely that the side effects of calling a function have not been fully accounted for and are adding cycles to the handwritten kernel's average cycle count. Overall, the proposed method can produce a matrix-multiplication kernel that is as performant as a human-crafted assembly, and that is a great accomplishment in code generation.

7.7 Summary

This chapter presented an in-depth analysis and evaluation of the matrix-multiplication micro-kernel code-generation strategy presented in Chapter 6. Beginning the chapter, details of the environment and methodology for each experiment, for example, creating a cycle measurement using the Google benchmark library, are explained.

Following the experimental setup is a discussion of two important issues that must be addressed in future versions of the assembly-code generator in the backend: spilling and shuffling. These two issues pervade and influence each of the experiments that follow.

The first two experiments investigate the optimal parameters for the MMA micro-kernel. The experiments determine that a rectangular accumulator layout (2×4, 4×2) coupled with an A matrix in a column-major orientation and an B matrix in a row-major orientation are the most performant choices. Furthermore, as anticipated, the orientation of C is shown to have no effect on performance

Following these two experiments are two performance comparisons demonstrating the capabilities of the MMA lowering. First, a comparison with the vectorised lowering shows between a 3.1 and 15.8 times speedup for all supported types. A second experiment comparing the code lowered by the compiler with a human-crafted kernel shows that the performance of the MMA lowering takes approximately double the cycle count that the handwritten kernel takes. After applying a load-sinking transformation the cycle counts are nearly identical, providing strong indication that, once the listed issues in the backend are resolved, the performance of the kernel will match that of an expert's handcrafted solution.

Chapter 8

Conclusion

Beginning by highlighting code-improvement opportunities that are unrealisable while using an external library for computation, this thesis presents a performant solution to the attractive alternative of a compiler-only path to generate code to compute matrix-multiplication. A comprehensive review of state-of-the-art matrix-multiplication micro-kernel implementation strategies in Chapter 4 reveals an opportunity for significant performance improvements in the inner kernel. This improvement comes in the form of extending operation dimensions in order to perform a full 4×4 outer-product, an operation that is supported by POWER10's MMA extension, as discussed in Chapter 5. The matrix engine offers significant speedup for matrix multiplication when compared with previous SIMD methods using VSX. Further insight in the form of the use of the `llvm.matrix.multiply.*` intrinsic enables the proposed method to be well positioned to enter common use as a standardised lowering for any frontend that compiles matrix-multiplication operations.

Chapter 6 details a procedure for emitting a matrix-multiplication kernel in LLVM IR that makes use of all observations made above. Deep understanding of the requirements of the hardware and the software led to the creation of several constraints that were used to create an efficient and performant lowering strategy. Each of these constraints is addressed throughout Section 6.3 and directly influences the lowering algorithm presented in Algorithm 6.1. The implemented lowering generates an efficient kernel for operands of any dimension, data orientation, and several data types.

A thorough performance evaluation and analysis of the algorithm follows in Chapter 7. An investigation and discussion of several shortcomings in the compiler backend for POWER indicates where performance can be improved in future iterations of the LLVM framework. Accounting for these deficiencies, the remaining experiments narrow kernel parameters to find an optimal setting. This optimal setting is compared with the closest point of reference: the vectorised lowering of the same intrinsic which makes use of the POWER ISA's SIMD capabilities in VSX. The comparison shows a speedup of at least 3.1 times for `half`, which is expected to improve drastically with improvements to the backend, up to a maximum of 15.8 times for `i8`. Moreover, a human-crafted kernel is only twice as fast as the current best lowering of the `llvm.matrix.multiply.*` intrinsic, a significant achievement. Furthering this comparison, there are strong indications that, after the noted spill issues are resolved, the handwritten and compiler-generated kernels will have identical performance.

The contribution of this thesis is part of a methodology that relies on a multi-level code generation strategy that separates the efficient utilization of the memory hierarchy at the macro level and the efficient utilization of the processing units and vector registers in the micro level. The design of the micro-kernel code generation presented in this thesis uses the `llvm.matrix.multiply.*` intrinsic in LLVM as an interface. In the future, this intrinsic will present an ideal entry point for the implementation of similar methodologies for new micro kernels targetting matrix engines in other ISAs. The combination of the simplicity of switching between kernels when compiling for different architectures as well as the isntant modularity within a macro kernel present clear and instantaneous benefits for implementations using this method. Thus, the methodology presented in this thesis shall be relevant for these other machine architectures as well.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Association, Nov. 2016, pp. 265–283, ISBN: 978-1-931971-33-1. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi. xix, 1

[2] T. Akutsu, S. Miyano, and S. Kuhara, "Algorithms for identifying boolean networks and related biological networks based on matrix multiplication and fingerprint function," *Journal of Computational Biology*, vol. 7, no. 3-4, pp. 331–343, 2000, PMID: 11108466. DOI: 10.1089/106652700750050817. eprint: https://doi.org/10.1089/106652700750050817. [Online]. Available: https://doi.org/10.1089/106652700750050817. 14

[3] C. L. Alappat, J. Hofmann, G. Hager, H. Fehske, A. R. Bishop, and G. Wellein, "Understanding HPC benchmark performance on Intel Broadwell and Cascade lake processors," in *High Performance Computing*, P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, Eds., Frankfurt am Main, Germany, 2020, pp. 412–433, ISBN: 978-3-030-50743-5. 13

[4] J. R. Allen and K. Kennedy, "Automatic loop interchange," in *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, ser. SIGPLAN '84, Montreal, Canada: Association for Computing Machinery, 1984, 233–246, ISBN: 0897911393. DOI: 10.1145/502874.502897. [Online]. Available: https://doi.org/10.1145/502874.502897. 3

[5] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1988, pp. 1–11. 8

[6] P. Alves, F. Gruber, J. Doerfert, A. Lamprineas, T. Grosser, F. Rastello, and F. M. Q. a. Pereira, "Runtime pointer disambiguation," *SIGPLAN Not.*, vol. 50, no. 10, 589–606, Oct. 2015, ISSN: 0362-1340. DOI: 10.1145/2858965.2814285. [Online]. Available: https://doi.org/10.1145/2858965.2814285. 8

[7] *Arm® Architecture Reference Manual armv8, for armv8-a Architecture Profile*, Arm Limited, Jan. 2021. [Online]. Available: https://developer.arm.com/documentation/ddi0487/latest/. 12

[8] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The ILLIAC IV Computer," *IEEE Transactions on Computers*, vol. C-17, no. 8, pp. 746–757, 1968. DOI: 10.1109/TC.1968.229158. 5, 11

[9] J. L. Blue and P. J. Grother, "Training feed-forward neural networks using conjugate gradients," in *Machine Vision Applications in Character Recognition and Industrial Inspection*, D. P. D'Amato, W.-E. Blanz, B. E. Dom, and S. N. Srihari, Eds., International Society for Optics and Photonics, vol. 1661, SPIE, 1992, pp. 179 –190. DOI: 10.1117/12.130286. [Online]. Available: https://doi.org/10.1117/12.130286. 14

[10] U. Bondhugula, "High performance code generation in MLIR: an early case study with GEMM," *CoRR*, vol. abs/2003.00532, 2020. arXiv: 2003.00532. [Online]. Available: https://arxiv.org/abs/2003.00532. 11

[11] M. M. Brandis and H. Mössenböck, "Single-pass generation of static single-assignment form for structured languages," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, 1684–1698, Nov. 1994, ISSN: 0164-0925. DOI: 10.1145/197320.197331. [Online]. Available: https://doi.org/10.1145/197320.197331. 8

[12] J. P. L. de Carvalho, B. Kuzma, I. Korostelev, J. N. Amaral, C. Barton, J. Moreira, and G. Araujo, "KernelFaRer: Replacing native-code idioms with high-performance library calls," *ACM Transactions On Architecture And Code Optimization (TACO)*, 2021. 11

[13] J. Cocke, "Global common subexpression elimination," *SIGPLAN Not.*, vol. 5, no. 7, 20–24, Jul. 1970, ISSN: 0362-1340. DOI: 10.1145/390013.808480. [Online]. Available: https://doi.org/10.1145/390013.808480. xvii

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1989, pp. 25–35. 8

[15] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, 451–490, Oct. 1991, ISSN: 0164-0925. DOI: 10.1145/115372.115320. [Online]. Available: https://doi.org/10.1145/115372.115320. 8

[16] J. Domke, E. Vatai, A. Drozd, P. ChenT, Y. Oyama, L. Zhang, S. Salaria, D. Mukunoki, A. Podobas, M. WahibT, and S. Matsuoka, "Matrix engines for high performance computing: A paragon of performance or grasping at straws?" In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Los Alamitos, CA, USA: IEEE Computer Society, 2021, pp. 1056–1065. DOI: 10.1109/IPDPS49936.2021.00114. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IPDPS49936.2021.00114. 12, 13, 29

[17] L. Eisen, J. W. Ward, H.-W. Tast, N. Mading, J. Leenstra, S. M. Mueller, C. Jacobi, J. Preiss, E. M. Schwarz, and S. R. Carlough, "Ibm power6 accelerators: Vmx and dfu," *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 1–21, 2007. DOI: 10.1147/rd.516.0663. 6

[18] J. F. Fabeiro, D. Andrade, and B. B. Fraguela, "Writing a performance-portable matrix multiplication," *Parallel Computing*, vol. 52, pp. 65–77, 2016, ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2015.12.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819115001611. 3

[19] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of gpu algorithms for matrix-matrix multiplication," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, ser. HWWS '04, Grenoble, France: Association for Computing Machinery, 2004, 133–137, ISBN: 3905673150. DOI: 10.1145/1058129.1058148. [Online]. Available: https://doi.org/10.1145/1058129.1058148. 12

[20] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972. DOI: 10.1109/TC.1972.5009071. 5

[21] LLVM Foundation. "LLVM language reference manual." (2020), [Online]. Available: https://llvm.org/docs/LangRef.html (visited on 01/04/2021). 37

[22] R. Gareev, T. Grosser, and M. Kruse, "High-performance generalized tensor operations: A compiler-oriented approach," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 3, Sep. 2018, ISSN: 1544-3566. DOI: 10.1145/3235029. [Online]. Available: https://doi.org/10.1145/3235029. 10

[23] "Google/benchmark: A microbenchmark support library," Google. (Jul. 21, 2021), [Online]. Available: https://github.com/google/benchmark (visited on 07/18/2021). 52

[24] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, May 2008, ISSN: 0098-3500. DOI: 10.1145/1356052.1356053. [Online]. Available: https://doi.org/10.1145/1356052.1356053. 10, 14, 18

[25] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly — performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1 250 010, 2012. DOI: 10.1142/S0129626412500107. eprint: https://doi.org/10.1142/S0129626412500107. [Online]. Available: https://doi.org/10.1142/S0129626412500107. 10

[26] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly-polyhedral optimization in llvm," in *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, vol. 2011, 2011, p. 1. 8, 10

[27] Z. Gu, J. Moreira, D. Edelsohn, and A. Azad, "Bandwidth optimized parallel algorithms for sparse matrix-matrix multiplication using propagation blocking," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '20, Virtual Event, USA: Association for Computing Machinery, 2020, 293–303, ISBN: 9781450369350. DOI: 10.1145/3350755.3400216. [Online]. Available: https://doi-org.login.ezproxy.library.ualberta.ca/10.1145/3350755.3400216. 13

[28] G. Guennebaud, B. Jacob, *et al.* "Eigen v3." (Jun. 19, 2021), [Online]. Available: http://eigen.tuxfamily.org (visited on 06/22/2021). 14

[29] D. Han, Y.-M. Nam, J. Lee, K. Park, H. Kim, and M.-S. Kim, "Distme: A fast and elastic distributed matrix computation engine using gpus," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19, Amsterdam, Netherlands: Association for Computing Machinery, 2019, 759–774, ISBN: 9781450356435. DOI: 10.1145/3299869.3319865. [Online]. Available: https://doi.org/10.1145/3299869.3319865. 12

[30] B. Hardekopf and C. Lin, "Semi-sparse flow-sensitive pointer analysis," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '09, Savannah, GA, USA: Association for Computing Machinery, 2009, 226–238, ISBN: 9781605583792. DOI: 10.1145/1480881.1480911. [Online]. Available: https://doi.org/10.1145/1480881.1480911. 8

[31] S. A. Hassan, A. Hemeida, and M. M. Mahmoud, "Performance evaluation of matrix-matrix multiplications using intel's advanced vector extensions (avx)," *Microprocessors and Microsystems*, vol. 47, pp. 369–374, 2016, ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2016.10.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933116302502. 13

[32] A. Hemeida, S. Hassan, S. Alkhalaf, M. Mahmoud, M. Saber, A. M. Bahaa Eldin, T. Senjyu, and A. H. Alayed, "Optimizing matrix-matrix multiplication on intel's advanced vector extensions multicore processor," *Ain Shams Engineering Journal*, vol. 11, no. 4, pp. 1179–1190, 2020,

ISSN: 2090-4479. DOI: https://doi.org/10.1016/j.asej.2020.01.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2090447920300058. 13

[33] IBM. "Engineering and scientific subroutine library 6.3." (Jun. 2020), [Online]. Available: https://www.ibm.com/docs/en/essl/6.3 (visited on 06/23/2021). 2

[34] ——, *Power® ISA version 3.1*, May 2020. [Online]. Available: https://ibm.ent.box.com/s/hhjfw0x0lrbtyzmiaffnbxh2fuo0fog0. 12

[35] "IBM - Archives - History of IBM - United States," International Business Machines. (Jan. 23, 2003), [Online]. Available: https://www.ibm.com/ibm/history/history/history_intro.html (visited on 09/15/2020). 6

[36] Intel. "Accelerate fast math with intel® oneapi math kernel library." (2021), [Online]. Available: https://software.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top.html (visited on 06/23/2021). 2

[37] *Intel® Architecture Instruction Set Extensions and Future Features programming reference*, Intel Corporation, Feb. 2021. [Online]. Available: https://software.intel.com/content/dam/develop/external/us/en/documents-tps/architecture-instruction-set-extensions-programming-reference.pdf. 12

[38] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, 1–12, Jun. 2017, ISSN: 0163-5964. DOI: 10.1145/3140659.3080246. [Online]. Available: https://doi.org/10.1145/3140659.3080246. 12

[39] D. Krol, D. Zydek, and H. Selvaraj, "Matrix multiplication in multiphysics systems using cuda," in *Advances in Systems Science*, J. Swiątek, A. Grzech, P. Swiątek, and J. M. Tomczak, Eds., Cham: Springer International Publishing, 2014, pp. 493–502, ISBN: 978-3-319-01857-7. 14

[40] D. Kuck, "Illiac iv software and application programming," *IEEE Transactions on Computers*, vol. 17, no. 08, pp. 758–770, Aug. 1968, ISSN: 1557-9956. DOI: 10.1109/TC.1968.229159. 12

[41] B. Kuzma, I. Korostelev, J. P. L. de Carvalho, J. Moreira, C. Barton, G. Araujo, and J. N. Amaral, "Fast matrix multiplication via compiler-only layered data reorganization and intrinsic lowering," under revision. iv, 10, 39, 65, 67

[42] E. S. Larsen and D. McAllister, "Fast matrix multiplies using graphics hardware," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, ser. SC '01, Denver, Colorado: Association for Computing Machinery, 2001, p. 55, ISBN: 158113293X. DOI: 10.1145/582034.582089. [Online]. Available: https://doi.org/10.1145/582034.582089. 12

[43] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665. 7

[44] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," *See* http://llvm.cs.uiuc.edu., M.S. thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec. 2002. 7

[45] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979. xvii

[46] J. Li, S. Ranka, and S. Sahni, "Strassen's matrix multiplication on gpus," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, 2011, pp. 157–164. DOI: 10.1109/ICPADS.2011.130. 12

[47] H. Liao, J. Tu, J. Xia, and X. Zhou, "Davinci: A scalable architecture for neural network computing," in *2019 IEEE Hot Chips 31 Symposium (HCS)*, IEEE Computer Society, 2019, pp. 1–44. 12

[48] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, "Analytical modeling is enough for high-performance blis," *ACM Transactions on Mathematical Software (TOMS)*, vol. 43, no. 2, pp. 1–18, 2016. 19, 20

[49] R. C. n. Lozano, M. Carlsson, G. H. Blindell, and C. Schulte, "Combinatorial register allocation and instruction scheduling," *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 3, Jul. 2019, ISSN: 0164-0925. DOI: 10.1145/3332373. [Online]. Available: https://doi.org/10.1145/3332373. 8

[50] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia tensor core programmability, performance precision," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 522–531. DOI: 10.1109/IPDPSW.2018.00091. 12

[51] R. K. Montoye, E. Hokenek, and S. L. Runyon, "Design of the ibm risc system/6000 floating-point execution unit," *IBM Journal of Research and Development*, vol. 34, no. 1, pp. 59–70, 1990. 6

[52] J. E. Moreira, K. Barton, S. Battle, P. Bergner, R. Bertran, P. Bhat, P. Caldeira, D. Edelsohn, G. Fossum, B. Frey, N. Ivanovic, C. Kerchner, V. Lim, S. Kapoor, T. M. Filho, S. M. Mueller, B. Olsson, S. Sadasivam, B. Saleil, B. Schmidt, R. Srinivasaraghavan, S. Srivatsan, B. W. Thompto, A. Wagner, and N. Wu, "A matrix math facility for power ISA(TM) processors," *CoRR*, vol. abs/2104.03142, 2021. arXiv: 2104.03142. [Online]. Available: https://arxiv.org/abs/2104.03142. 55, 77, 80

[53] N. Nakasato, "A fast gemm implementation on the cypress gpu," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 4, 50–55, Mar. 2011, ISSN: 0163-5999. DOI: 10.1145/1964218.1964227. [Online]. Available: https://doi.org/10.1145/1964218.1964227. 12

[54] R. Nath, S. Tomov, and J. Dongarra, "Accelerating gpu kernels for dense linear algebra," in *High Performance Computing for Computational Science – VECPAR 2010*, J. M. L. M. Palma, M. Daydé, O. Marques, and J. C. Lopes, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 83–92, ISBN: 978-3-642-19328-6. 12

[55] NVIDIA. "Cublas :: Cuda toolkit documentation." (May 20, 2021), [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html (visited on 06/23/2021). 2

[56] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 724–736. DOI: 10.1109/HPCA.2018.00067. 13

[57] F. M. Quintão Pereira and J. Palsberg, "Register allocation by puzzle solving," *SIGPLAN Not.*, vol. 43, no. 6, 216–226, Jun. 2008, ISSN: 0362-1340. DOI: 10.1145/1379022.1375609. [Online]. Available: https://doi.org/10.1145/1379022.1375609. 8

[58] A. Poenaru and S. McIntosh-Smith, "Evaluating the effectiveness of a vector-length-agnostic instruction set," in *Euro-Par 2020: Parallel Processing*, M. Malawski and K. Rzadca, Eds., Cham: Springer International Publishing, 2020, pp. 98–114, ISBN: 978-3-030-57675-2. 13

[59] R. Rojas, *Neural networks: a systematic introduction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, ISBN: 978-3-642-61068-4. DOI: 10.1007/978-3-642-61068-4_7. [Online]. Available: https://doi.org/10.1007/978-3-642-61068-4_7. 14

[60] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1988, pp. 12–27. xvii, 8

[61] R. Schreiber, J. J. Dongarra, *et al.*, *Automatic blocking of nested loops*. Research Institute for Advanced Computer Science, NASA Ames Research Center, 1990. 3

[62] P. Stange, A. Griewank, and M. Bollhöfer, "On the efficient update of rectangular lu-factorizations subject to low rank modifications," *Electron. Trans. Numer. Anal.*, vol. 26, pp. 161–177, 2007. 17

[63] Y. Sui and J. Xue, "Svf: Interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016, Barcelona, Spain: Association for Computing Machinery, 2016, 265–266, ISBN: 9781450342414. DOI: 10.1145/2892208.2892235. [Online]. Available: https://doi.org/10.1145/2892208.2892235. 8

[64] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, 1967. 6

[65] J. Tyler, J. Lent, A. Mather, and Huy Nguyen, "AltiVec/sup TM/: bringing vector technology to the PowerPC/sup TM/ processor family," in *1999 IEEE International Performance, Computing and Communications Conference (Cat. No.99CH36305)*, 1999, pp. 437–444. DOI: 10.1109/PCCC.1999.749469. 6

[66] F. G. Van Zee and R. A. van de Geijn, "Blis: A framework for rapidly instantiating blas functionality," *ACM Trans. Math. Softw.*, vol. 41, no. 3, Jun. 2015, ISSN: 0098-3500. DOI: 10.1145/2764454. [Online]. Available: https://doi.org/10.1145/2764454. 18

[67] G. Velkoski, M. Gusev, and S. Ristov, "The performance impact analysis of loop unrolling," in *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2014, pp. 307–312. DOI: 10.1109/MIPRO.2014.6859582. 3

[68] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel math kernel library," in *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures*. Cham: Springer International Publishing, 2014, pp. 167–188, ISBN: 978-3-319-06486-4. DOI: 10.1007/978-3-319-06486-4_7. [Online]. Available: https://doi.org/10.1007/978-3-319-06486-4_7. 2

[69] S. Wang and P. Kanwar, "Bfloat16: The secret to high performance on cloud tpus," *Google Cloud Blog*, 2019. 30

[70] Y. Wang, G. Wei, and D. Brooks, "Benchmarking tpu, gpu, and CPU platforms for deep learning," *CoRR*, vol. abs/1907.10701, 2019. arXiv: 1907.10701. [Online]. Available: http://arxiv.org/abs/1907.10701. 12

[71] H. Waugh and S. McIntosh-Smith, "On the use of blas libraries in modern scientific codes at scale," in *Driving Scientific and Engineering Discoveries Through the Convergence of HPC, Big Data and AI*, J. Nichols, B. Verastegui, A. B. Maccabe, O. Hernandez, S. Parete-Koon, and T. Ahearn, Eds., Cham: Springer International Publishing, 2020, pp. 67–79, ISBN: 978-3-030-63393-6. 1

[72] V. Weber, T. Laino, A. Pozdneev, I. Fedulova, and A. Curioni, "Semiempirical molecular dynamics (semd) i: Midpoint-based parallel sparse matrix–matrix multiplication algorithm for matrices with decay," *Journal of Chemical Theory and Computation*, vol. 11, no. 7, pp. 3145–3152, 2015, PMID: 26575751. DOI: 10.1021/acs.jctc.5b00382. eprint: https://doi.org/10.1021/acs.jctc.5b00382. [Online]. Available: https://doi.org/10.1021/acs.jctc.5b00382. 14

[73] J. Wu and J. Jaja, "Achieving native gpu performance for out-of-card large dense matrix multiplication," *Parallel Processing Letters*, vol. 26, no. 02, p. 1 650 007, 2016. DOI: 10.1142/S0129626416500079. eprint: https://doi.org/10.1142/S0129626416500079. [Online]. Available: https://doi.org/10.1142/S0129626416500079. 12

[74] Z. Xianyi, W. Qian, and Z. Yunquan, "Model-driven level 3 blas performance optimization on loongson 3a processor," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, Dec. 2012, pp. 684–691. DOI: 10.1109/ICPADS.2012.97. 2, 14

[75] T. Yu, Y. Cai, and P. Li, "Toward faster and simpler matrix normalization via rank-1 update," in *Computer Vision – ECCV 2020*, A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm, Eds., Cham: Springer International Publishing, 2020, pp. 203–219, ISBN: 978-3-030-58529-7. 12

[76] F. G. V. Zee, T. M. Smith, B. Marker, T. M. Low, R. A. V. D. Geijn, F. D. Igual, M. Smelyanskiy, X. Zhang, M. Kistler, V. Austel, J. A. Gunnels, and L. Killough, "The blis framework: Experiments in portability," *ACM Trans. Math. Softw.*, vol. 42, no. 2, Jun. 2016, ISSN: 0098-3500. DOI: 10.1145/2755561. [Online]. Available: https://doi.org/10.1145/2755561. 2, 18

[77] A. Zulehner and R. Wille, "Matrix-vector vs. matrix-matrix multiplication: Potential in dd-based simulation of quantum computations," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 90–95. DOI: 10.23919/DATE.2019.8714836. 14