Summer 8-2020

Investigating Single Precision Floating General Matrix Multiply in Heterogeneous

Steven Harris

Washington University in St. Louis

School of Engineering and Applied Science

Department of Computer Science and Engineering

Thesis Examination Committee:
Professor Roger Chamberlain,
Professor Christopher Gill,
Professor Ning Zhang

Investigating Single Precision Floating General Matrix Multiply in Heterogeneous Hardware

by

Steven D. Harris

A thesis presented to the McKelvey School of Engineering
of Washington University in partial fulfillment of the
requirements for the degree of

Master of Science

May 2020
Saint Louis, Missouri

Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank the Washington University Department of Computer Science & Engineering and the Department of Mathematics and Statistics. These departments have been my home for a very long time and within its walls, I have had the opportunity to work with luminaries that for all their brilliance had equal measures of modesty and patience. I would like to give a very sincere acknowledgement to Dr. Roger Chamberlain, Dr. Chris Gill, Dr. Ning Zhang, and Dr. Mladen Wickerhauser for all of their time, energy, and insights. There support has been invaluable to my progress and the development of this manuscript. I would like to thank the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. I would also like to thank the National Science Foundation for supporting my work through NSF CCF1337218 XPS:FP: Real-Time Scheduling of Parallel Tasks, NSF CNS1527510 CSR: Small: Concurrent Accelerated Data Integration, NSF CNS1814739 CSR: Small: Dynamically Customizable Safety-Critical Embedded Systems, and NSF CNS1763503 CSR: Performant Architecturally Diverse Systems via Aspect Oriented Programming.

Steven D. Harris

Washington University in Saint Louis
May 2020

To My Companions

As I think about all the people, events, and perfectly timed synchronicities that have occurred in order to create this work, the metaphor of standing on the shoulders of giants comes to mind and I chuckle to myself. Instead of being so high that I can see the full landscape above the trees and know where I am headed, this work required a mad dash through a dense rain forest. There were twists, turns, and many holes. There were times when I dodged the branches and when I did not, I was either looking up at the clouds or face to face with flowers, vines, and a few startled eyes. I have reached a small clearing and as I look back, all the thrashing through the forest makes a clear path that very few could miss. I would like to thank my mother and teacher, Maria DeShields, who home-schooled my sister and I, enabling all three of us to graduate from Washington University. She told me to keep running but don't miss all the marvelous sights as I race by. I would like to thank my father, Alvin Harris, who gave me my very first computer, unassembled, in a box filled with components and told me to tuck in my chin, stop flailing like a bird, and read the manual. My sister, Salina Greene, who dragged me into the forest but told me to stop using my head to clear branches. My Rebbe, Randy Fleisher, who told me to always bring Tikkun Olam, even in the wilderness. These teachers are the giants that chuckled and cheered as I raced headlong through the tropics at breakneck speed. I had one other companion on this trip as well, my son Yochanan. Tucked in a tactical baby carrier, reaching for every berry, branch, and thorn; I hope that he can one day walk further down a well-trodden path. To them and all the others, along with the ancestors shaking their heads but patiently watching our travels through this forest, I say thank you.

ABSTRACT OF THE THESIS

Investigating Single Precision Floating General Matrix Multiply in Heterogeneous Hardware

by

Steven D. Harris

Master of Science in Computer Science

Washington University in St. Louis, May 2020

Research Advisors: Professor Roger Chamberlain & Professor Christopher Gill

The matrix, a rectangular array of numbers, expressions, or symbols, has become ubiquitous across many genres of science, industry, and business. It is fundamental to mathematics, physics, computer science, and engineering disciplines including cyber-physical systems in which precise simulation and control of safety-critical physical behavior must be managed computationally. By delineating data sets into neatly arranged rows and columns, these arrays can be manipulated to identify and exploit underlying relationships and correspondences between elements. For example, the structure provided by a matrix allows one to perform one of the most fundamental matrix operations: multiplication.

The form, function, and behavior of matrix multiplication has been studied consistently over the years and its properties are well-known. However, the standard matrix multiplication computation performed on modern computer system runs in cubic time O(n^3). As modern data sets become ever larger, matrix multiplication becomes ever more time-consuming. Therefore, with each new central processor unit (CPU) or graphics processing unit (GPU),

and with the introduction of accelerators such as Field Programmable Gate Arrays (FPGAs), researchers continue to investigate new optimizations to improve the performance of matrix multiplication on modern hardware.

Many optimizations are moving beyond the scope of general-purpose CPUs in order to target emerging architectures. To support this effort, the Khronos Group, a non-profit technology consortium, has developed an open framework called the Open Computing Language (OpenCL). OpenCL allows rapid development of programs, typically called kernels, that can execute across a wide range of heterogeneous platforms including but not limited to CPUs, GPUs, and FPGAs.

In the last few decades, CPU speeds have rapidly diverged from projections given by Moore's law as we approach the limitations of Dennard scaling. Consequently, researchers have put increasing efforts into optimizing GPUs for computationally intensive tasks such as matrix multiplication. The rapid adoption of GPUs for a wide range of computationally intensive tasks has infused the spheres of academia, industry, and engineering with a wealth of literature, implementations, and guidelines for developing performant kernels for matrix multiplication on the GPU.

While GPUs excel at most parallel workloads, these devices are not a panacea for complex workloads including computations which may require branching for data dependent conditions. Such workloads must be completed independently by the CPU. Yet, one of the newest emerging technologies attempts to combine the benefits of the CPU and other accelerators into a dynamic, hybrid, reconfigurable processing unit. One realization of this effort comes by way of the Heterogeneous Architecture Research Platform (HARP). This system consists of an Intel Broadwell Xeon CPU combined with an Intel Arria 10 GX1150 FPGA into a Multi-Chip Package (MCP) that enables shared DRAM memory through a single Intel

QuickPath Interconnect (QPI) and two Peripheral Component Interconnect Express (PCIe) channels.

With the advent of such hybrid architectures, one of the benefits of OpenCL is the ability to write code for a given task, once, and have it execute across a range of devices in a heterogeneous environment. With a suitable implementation in hand, one should be able to perform complex computations efficiently in any OpenCL compliant environment. As better hardware becomes available, one should be able to execute these kernels on new infrastructures with little, if any, modification to the existing kernels. Using the OpenCL framework as a vehicle of exploration, we have investigated the efficacy of using the current wealth of knowledge and best practices pertaining to matrix multiplication for OpenCL compliant devices on the HARP system and the implications of such methodologies for future heterogeneous architectures.

Chapter 1

21st Century Challenges in Matrix Multiplication

1.1 Introduction

Matrix multiplication is a well-known mathematical operation that is critical in numerous disciplines. Even with the nearly exponential increases in computer performance, matrix multiplication remains a persistent challenge and the advent of accelerators (both FPGA and GPU) reopens questions of both expression and optimization. In decades past, users could scale up simply by purchasing the next generation of higher performance processors, but with the breakdown of Dennard scaling, along with thermal and memory barriers, there has been a divergence from the performance predictions previously driven by Moore's law and CPU clock rates have reached a plateau around 4-5 GHz. Under these constraints, both users and manufacturers have elected to scale horizontally. Manufacturers increase the core counts on processors and users aggregate more of these processors into individual servers, and by extension into data center clusters. However, vertical and horizontal scaling of CPU resources has not been enough to keep up with the demands of modern workloads. This performance gap has motivated alternative architectures such as GPU, ASIC, and FPGA solutions.

Some users have bypassed CPU and GPU architectures entirely by adapting their workloads to Application-Specific Integrated Circuits (ASICs), which tend to be more performant than general purpose CPU/GPU devices for specific applications. However, these devices require

significant overhead and cannot be modified once created. For devices with strict timing and performance constraints, and requiring application modification in the future, the Field-Programmable Gate Array (FPGA) has become the option of choice. Although it has considerably less cost and development overhead than an ASIC, developing a comparable FPGA based solution may involve a steep learning curve that has traditionally only been accessible to those with knowledge of Hardware Description Languages (HDLs) and digital system design techniques.

That learning curve may be exacerbated in heterogeneous environments due to additional challenges, two of the most prominent being orchestration and data migration. With each device having different architectures, languages, and computational units, it can be challenging to wrangle all the disparate tools and functionalities to create a suitable implementation of an application. This may be further complicated by data migration, wherein pertinent data must shuttle from one device to another in a pipeline or streaming fashion, particularly, across memory hierarchies.

To alleviate these challenges, the Khronos Group, a non-profit technology consortium, has developed an open framework called the Open Computing Language (OpenCL). OpenCL allows rapid development of programs that can execute across a wide range of heterogeneous platforms, including but not limited to, CPUs, GPUs, and FPGAs. The OpenCL framework enables computation orchestration on existing systems and its compatibility with the Intel High Level Synthesis compiler allows users to architect new designs for reconfigurable hardware using C/C++.

As the world shifts toward more application-specific accelerators for modern workloads, Intel has developed an alternative accelerator in the form of a unified hybrid CPU+FPGA. One realization of this effort is the Heterogeneous Architecture Research Platform (HARP). Version 2 of this system consists of an Intel Broadwell Xeon CPU combined with an Intel Arria 10 GX1150 FPGA into a Multi-Chip Package (MCP) that enables shared DRAM memory through a single Intel QuickPath Interconnect (QPI) and two Peripheral Component Interconnect Express (PCIe) channels. Using the HARPv2 as a vehicle for exploration, we investigate the design space of matrix multiplication, using several existing cache-oriented optimizations to better understand the performance portability of OpenCL and the implications for such optimizations on this and future heterogeneous architectures.

Across a range of matrix sizes, we show that several classic optimizations designed for traditional caches are also effective on the HARP system. This includes transposition, blocking, and loop unrolling. When all optimizations are included, our implementations consistently outperform the optimized standard library implementation (CBLAS). However, there are still considerable variations in performance, across both matrix size and various tuning parameters, that are not yet well understood and that warrant further investigation.

1.2 The Current State of the Art

Over the past decade, heterogeneous computing has typically implied a computation that is partitioned across one or more devices containing a combination of CPUs and GPUs. For CPU computations, there are a wealth of libraries and tools, such as Open Multi-Processing (OpenMP) and the Basic Linear Algebra System (BLAS) that facilitate the rapid development of high-performance computations. For GPUs, the dominant solutions come from industry leaders such as NVIDIA who manufactures the most popular GPUs as well as being the author of CUDA, the well-known parallel computing platform [1, 27, 38, 21, 14].

In a general sense, these solutions simplify development. Developers have invested considerable time and effort into understand the underlying hardware architectures, identifying optimal execution methods, and generalizing these operations. Instead of investigating the myriad of hardware specifications, developers have identified generalities between target architectures to use across various classes of devices. By having a solid grasp of the computational architectures, developers provide users with extensions, application programming interfaces, and tools that organize the data, computation, and communication efficiently across these devices. Yet, developers also share the burden of the revision process: particularly, as hardware evolves, users and developers alike are required to pivot to new hardware, frameworks, and specifications to take advantage of additional features incorporated into next-generation GPUs. Developers perform the heavy-lifting for users that may not have the time or background to understand how an algorithm may augment performance or the resources to create interfaces to operate these devices.

While users have the flexibility to perform tasks that may actually degrade performance, developers typically provide users with best-practices for a given class of tasks [19]. Instead

of having to reinvent the wheel, users with these tools can rapidly develop solutions that fully utilize these devices with little effort. However, even with such tools many challenges arise when users want to use multiple devices for a given computation. With different computational units, memory hierarchies, and communication methods, users can spend considerable time on data manipulation, coordination, and orchestration. Irrespective of the sophisticated programmatic solutions developed for a given computation, data migration also can have significant impacts on overall performance.

These challenges bring out new questions concerning algorithms, architectures, and computational workflows. For example, are there tools that allow user to orchestrate a computation and reconfigure all available resources in the most efficient manner? This is one goal that researchers are working towards with solutions such as OpenCL for orchestration and hybrid accelerators for reconfigurable architectures. However, such question pose a more fundamental question: Are we developing solutions that target the computation or the hardware?

1.3 Innovation under Compliance Constraints

Mathematical concepts such as systems of equations, linear transformations, and scalar products are prevalent in many areas of science. One of the most pertinent of these concepts is that of matrix multiplication. Matrix multiplication has been studied, analyzed, and established for decades. It sits at the core of many numerical algorithms, scientific computations, and big data workloads.

Much like fundamental arithmetic operations, many would consider the "problem" of matrix multiplication to be solved: we do not often consider the efficiency of arithmetic operations or review new developments in their optimizations, assuming suitable performance of such arithmetic operations. That is, we assume that these operations can be performed as fast as the hardware (i.e. silicon switching) will allow. When we consider multiplication and division, we often focus more on precision than on efficiency. We would like to think that the operations may run slightly slower than addition or subtraction, but that ultimately the main challenge is one of precision. However, this comparison could not be farther from the truth: the performance of matrix multiplication remains an open problem.

The challenge of optimizing the performance of matrix multiplication has been investigated by Strassen [18], Bini [4], and Coppersmith & Winograd [8], among others [55]. Many resulting algorithms have been implemented on computer systems since the 1970s, and yet each year the scientific communities of the world still try to find more efficient ways to perform this well-known mathematical operation [22, 5, 7, 2, 50, 23, 16, 34, 33, 36, 45]. Given that the efficiency of matrix multiplication on modern computer systems is still in a state of flux, what does it mean when a new method arrives that is touted to be more efficient, simpler, faster, or offering more calculations per unit of time? These definitions are all suitable classifications for efficiency. However, efficiency in the context of matrix multiplication has a particular metric that complements any other measurement of efficiency: utilization. The algorithm with the highest level of efficiency is often tightly coupled with the utilization of the underlying hardware resources. Because of the relationship between efficiency and utilization, each matrix multiplication algorithm has to be specifically tailored to the underlying architecture.

Given the multitude of computer systems on the market and the likely new architectures of the future, it thus seems necessary to continue refactoring implementations indefinitely. There are many reasons why we cannot now simply write a program once and (without further modification) see improvements in precision, performance, and efficiency as better technology on which to run the program becomes available: we write our programs to a specific architecture as opposed to writing code solely to address the computational problem; our primitive data types may be selected from predefined capacities; our algorithms reflect the parallelism of our compute units; and our function calls, methods, and tools are often selected *a priori* based on available languages, operating systems, or architectures.

To overcome those limitations of the current state of the art, it is appropriate to consider what computations would look like if we could simply write a sufficient algorithm and the system then could dynamically orchestrate the underlying architecture to fulfill the algorithmic requirements.

The experiments presented in this work offer a preliminary step towards that vision, with a particular emphasis on domain specific architectures and heterogeneous solutions, by examining performance of matrix multiplication across different relevant design and architectural

factors. By investigating state of the art techniques for a subset of matrix multiplication challenges, we have identified some areas where new ideas about architectures and assumptions affecting matrix multiplication may prove useful.

Chapter 2

Modern Architecture

In 1972, Intel released the first commercial 8-bit processor called the 8008. This processor was formed from 3,500 transistors that ran at speeds up to 800 kHz. Using the (then advanced) 8-bit architecture, this processor was capable of accessing up to 16 KB of RAM. In stark contrast, today's modern processors are comprised of over 8,000,000,000 transistors and can reach speeds of up to 4,400,000 kHz (4.4 GHz). Using modern 64-bit architectures, these processors can access over 2,000,000,000 KB (2TB) of RAM. Since the inception of the 8008, the performance of microprocessors has increased dramatically. A well-known predictor of processor performance has been Moore's law which proposed that the number of transistors on microprocessors would double every 18 months. For a time, this prediction held true and society reaped the benefit of increased performance year after year. However, since about the mid-2000s processor speeds have plateaued due to constraints imposed by physics and the laws of thermodynamics. While many would argue that Moore's law is dead, each time this argument arises engineers and scientists develop new ways to bring its predictive power back to life. In this incarnation, Moore's law is reborn again via multi-core processors.

2.1 Parallelism and Concurrency

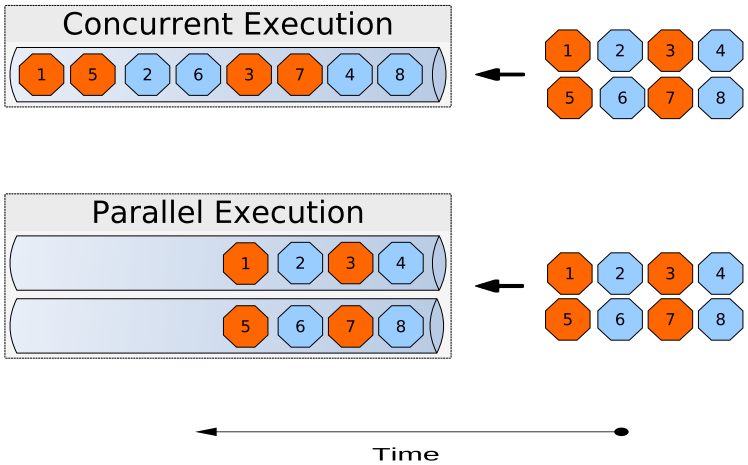

Figure 2.1: Parallelism vs Concurrency

While many programs have segments of sequential code, there are often opportunities to take advantage of parallelism in code segments. An important part of computer processing involve the concepts of *parallelism* and *concurrency*. While these two terms are often used interchangeably, parallelism and concurrency are not equivalent. Parallelism is an execution model wherein two or more tasks can be executed simultaneously. However, the concurrent task model allows tasks to progress at the same rate even if executed at different times. Concurrent tasks are typically executed in an interleaved fashion which allows two or more programs to execute in lock-step with each other. These execution models are shown in figure 2.1. In the most general sense, a computer consists of a combination of CPU, Memory, Storage, and various input and output devices which utilize a communication system (a.k.a. a bus) that is used to transfer data between components inside the computer system.

Traditionally, all computer processing units consist of control, arithmetic logic, and memory management units. The control unit orchestrates all the processes occurring inside the CPU. From execution of program instructions to the ALU and memory management, all processes are governed by the control unit. The ALU performs all arithmetic and bitwise logic operations. The memory management unit controls access and communication with memory. CPUs are typically synchronous circuits which utilize a clock signal to synchronize

their operations. Each and every operation occurs on a "tick" of a clock cycle. This clock signal arises from external oscillator circuits that generate a fixed number of pulses per second in the form of periodic square waves. The frequency of these pulses is how we derive the speed of the processor and the number of instructions that the CPU can perform per second. To simplify things, we will define a processor as containing a Processing Element (PE) which manages the execution of instructions, an Instruction Pool which stores those instructions, and a Data Pool which stores the data which the instructions will act upon. Over time, processors have improved to take advantage of parallelism using different methods to control both data and execution within the CPU.

In 1966, Dr. Flynn proposed a method for classifying digital computers, which is now referred to as Flynn's Taxonomy [13]. The primary classifications can be subdivided into two groups: Single Data Stream (SISD, MISD) and Multiple Data Stream (SIMD, MIMD) architectures.

2.1.1 Single Data Stream Architectures

Single Instruction Stream, Single Data Stream (SISD)

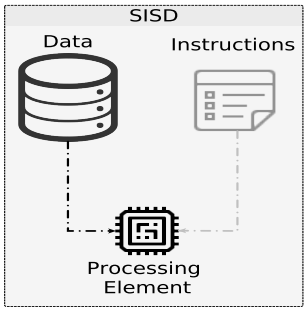

Figure 2.2: Single Instruction Stream, Single Data Stream (SISD)

As is shown in figure 2.2, a traditional single-core processor would fall into the Single Instruction Stream, Single Data Stream (SISD) category. This specific architecture processes a single stream of data in sequential order. To process large amounts of data, the user would be limited to using recursive techniques or loops in order to iterate through the data. This architecture has no parallelism.

Multiple Instruction Streams, Single Data Stream (MISD)

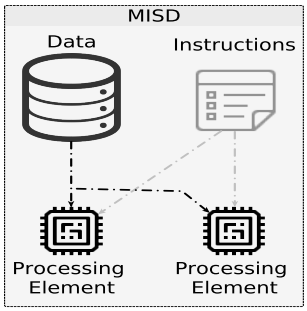

Figure 2.3: Multiple Instruction Streams, Single Data Stream (MISD)

In contrast to this is the Multiple Instruction Streams, Single Data Stream (MISD) architecture, a type of parallelism akin to pipe-lining. Multiple processing units perform separate operations on the same data set. This method is fault-tolerant as the same instruction can be executed by different PEs in order to ensure reliability. However, this architecture has been superseded by multiple data stream architectures.

2.1.2 Multiple Data Stream Architectures

Single Instruction Stream, Multiple Data Streams (SIMD)

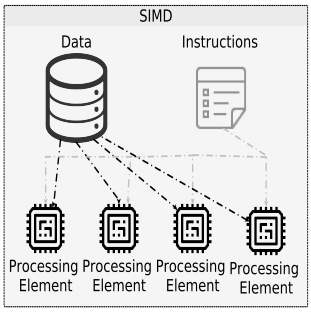

Figure 2.4: Single Instruction Stream, Multiple Data Streams (SIMD)

In terms of modern CPUs, one would be hard-pressed to find a system which did not contain more than one core. That is to say, most modern CPUs are multi-core processors. These processors have a Single Instruction Stream, Multiple Data Streams (SIMD) architecture, shown in figure 2.4, which allows them to perform the same instruction on multiple data sets simultaneously. This architecture exploits *data level parallelism*.

Multiple Instruction Streams, Multiple Data Streams (MIMD)

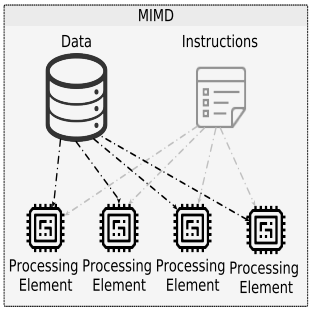

Figure 2.5: Multiple Instruction Streams, Multiple Data Streams (MIMD)

SIMD was expanded to take advantage of multiple instructions as well as data. CPUs of this type employ Multiple Instruction, Multiple Data architectures (MIMD) as shown in figure 2.5. These processors contain multiple PEs which work independently and asynchronously allowing the execution of various instruction on unique data sets simultaneously. This is typically referred to as *task level parallelism* according to Flynn's taxonomy. A subcategory of MIMD is Single Program, Multiple Data (SPMD) wherein a single program is executed across all PEs.

2.2 Single-Core to Multi-Core Evolution

2.2.1 Von Neumann Architecture

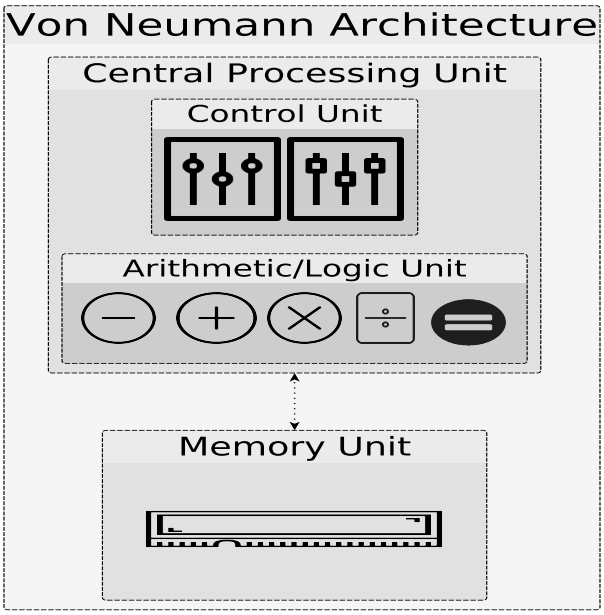

Figure 2.6: Von Neumann Architecture

General purpose CPUs have advanced significantly with the advent of additional cores, low power usage, and additional extensions. As shown in figure 2.6, the CPU consists of a Control Unit and Arithmetic Logic Unit (ALU) connected via data bus to the Memory Unit. The Control Unit manages communication with the ALU and interprets program instructions. The ALU receives data/commands from the Control Unit in order to execute arithmetic and logic instructions on the received data. This design is known as the *Von Neumann architecture*. Data and instructions are stored in the memory subsystem in the same format. This allows the contents of memory to be defined by the system interpreting the memory location. For instance, a program compiler can translate user-defined programming pragmas into machine language that is stored as ordinary data but can also be executed by the CPU directly as instruction directives. This makes Von Neumann architecture flexible and versatile but it has one glaring disadvantage, a phenomenon known as the *Von Neumann bottleneck* which will be discussed later. This architecture places specific limits on the programmatic

solutions as the execution of instructions are inherently sequential. Fundamentally, modern general purpose CPUs have not diverged from the original Von Neumann architecture since its inception which shows both the longevity of this architecture as well as the legacy of persistence computational constraints.

2.2.2 Computing at Scale

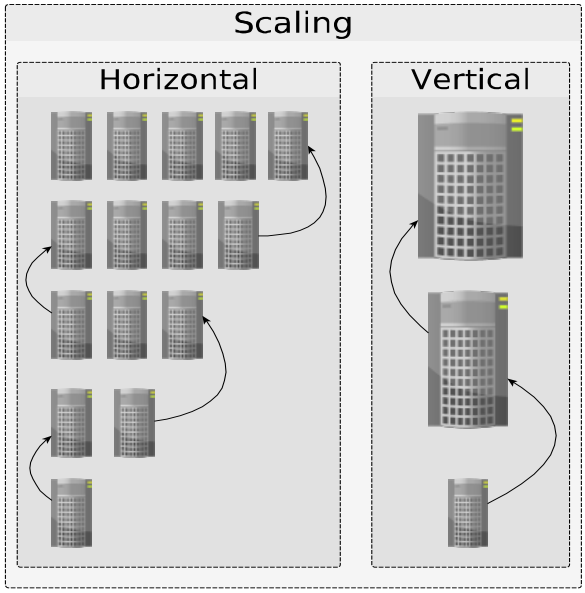

Figure 2.7: Horizontal vs Vertical Scaling

General-purpose architectures have developed over the years and are deployed in many industries based on the workload targets. However, a little over a decade ago, in an era dominated by the seemingly ever-increasing CPU frequencies, many industries found that if their computer systems could not keep up with the computational demands, they would defer to the old adage of "wait until next year", in order to upgrade to a faster CPU by utilizing vertical scaling. Some industries would take a slightly different approach by taking advantage of horizontal scaling which would incorporate additional CPUs in networked servers to increase processing power, as is often found in High Performance Computing (HPC) environments.

However, horizontal scaling in this fashion is expensive and cost prohibitive for many industries. Such vertical and horizontal scaling, shown in figure 2.7, were not limited to the industries which adopted the processors, but were also adopted in processor fabrication as well.

The rapid advances in processing power were due to the ability to take advantage of transistor scaling that enabled the minimization of transistors. Consequentially, the semiconductor industry took advantage of horizontal scaling afforded by transistor minimization in order to pack more transistors into integrated circuits (ICs) and thereby improve performance. For several decades, the number of transistors per square inch on an integrated circuit (IC) doubled nearly every two years, corresponding roughly to the projections by Dr. Moore. Such rapid increases in transistor scaling elicited an unparalleled effervescence of computational power and processor diversity. As the size of transistors became smaller, transistor density and switching frequency increased. Coupling the transistor improvement with significantly cheaper manufacturing costs created a perfect storm of events that led to rapid expansion in computer technology to such an extent that system integrators were barely able to assemble existing hardware before newer and more powerful integrated circuits (ICs) were introduced to the market.

Dr. Moor's projections seemed likely to extend far into the future, but two challenges emerged that caused actual results to diverge from the long held predictions: thermal runaway and quantum effects. As subsequent chips incorporated smaller and more tightly packed transistors, it was determined that these chips could maintain the same amount of power while operating at higher frequency while consuming less voltage. That is, as transistor feature size decreased, the power density remained relatively constant. This idea is commonly referred to as *Dennard Scaling*. However, the tremendous benefit of Dennard Scaling overlooked a major component: leakage current. Dennard scaling did not take into account leakage current which serves as a baseline for power per transistor. Consequently, the tightly packed chips with electrons moving at increasing speeds through smaller and smaller silicon circuits, contributed to higher and higher power densities. Given that power density does not scale with size, this generated substantial amounts of heat that would become impossible to dissipate with common airflow cooling methods. The only option for manufacturers was to limit either the number of transistors or the frequency of the processor. Due to Dennard Scaling, processor frequencies have not exceeded roughly 5 GHz.

With further transistor scaling, further challenges emerged which are related to smaller feature sizes. Transistors lie at the heart of a processor and are comprised of electrical leads called the emitter, collector, and base. In computer systems, these transistors are responsible for switching: that is, moving between on and off states to encode ones and zeroes. In switching implementations, the electrical leads take on additional terminology with the emitter referred to as the source, the collector as the drain, and the base as the gate. These terms evoke a notion of current or fluid flow. The most critical component that enables the action of switching revolves around the gate, which controls the flow of electrons through the transistor. As transistors are scaled to smaller and smaller feature sizes, the thickness of the gates decreases as well.

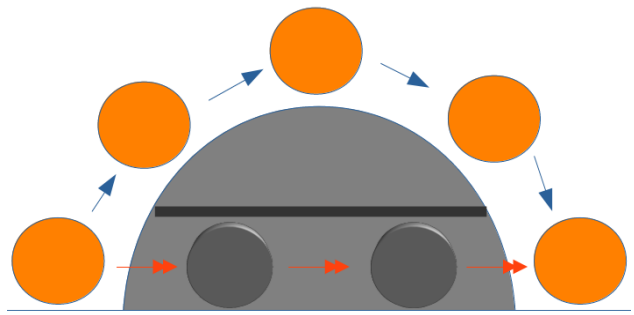

Figure 2.8: Quantum Tunneling

Thinner gates can lead to a phenomenon known as quantum mechanical tunneling wherein electrons can pass through the gate medium of a closed gate. In Classical Mechanics, if an electron has enough energy to overcome the potential energy at the top of the barrier, the electron will be able to traverse the barrier. Even if the electron does not have enough kinetic energy to overcome the barrier, however, quantum effects may allow the electron to tunnel through the barrier. Figure 2.8 shows an example of quantum tunneling with the blue arrows indicating the Classical Mechanics trajectory and the red arrows indicating the Quantum Mechanical trajectory.

Quantum tunneling effects can contribute to a significant gate leak current that increases exponentially as the gate thickness decreases. The break down in Dennard scaling, undesirable quantum effects, and limitations on clock frequencies led many researchers to assume

that Moore's Law was coming to an end. However, innovation would breathe new life into Moore's Law through the introduction of multi-core systems.

2.3 Multicore to Heterogeneous Architectures

Horizontal scaling, it would seem, has developed a lasting permanence in the computer industry. As noted previously, when computations required additional computing power, one could either scale vertically through the purchase of faster processors (if available) or scale horizontally with the acquisition of additional servers. The semiconductor industry took the horizontal approach and with transistor miniaturization, more processors could fill the same horizontal space on a microscopic level. With limitations curtailing transistor minimization, the semiconductor industry developed a new approach to the challenges which threatened to end Moore's Law. Instead of creating dense high frequency single core processors, they began to develop additional cores which ran at comparable frequencies as single core processors and the era of multi-core processing began. The increasing numbers of processor cores benefited many workloads but did little to help with sequential tasks.

Nevertheless, multi-core processing enabled the partitioning of workloads onto several processors and also introduced the notion of *dark silicon* [52]. The semiconductor industry designed the multi-core processors so that only a fraction of the processors was actually operating on a workload at any given time which greatly reduced the power constrains that were leading to thermal runaway and other complications. With many processors cores working on a subset of a given workload, some processors were running while other processors were idle or "dark". The dark silicon paradigm increased the industry's ability to improve power efficiency in computer systems. Yet, the growth of multi-core systems has lagged far behind architectural needs for massive computationally intensive workloads which in turn has opened the door to alternative architectures.

2.3.1 Alternative Accelerators

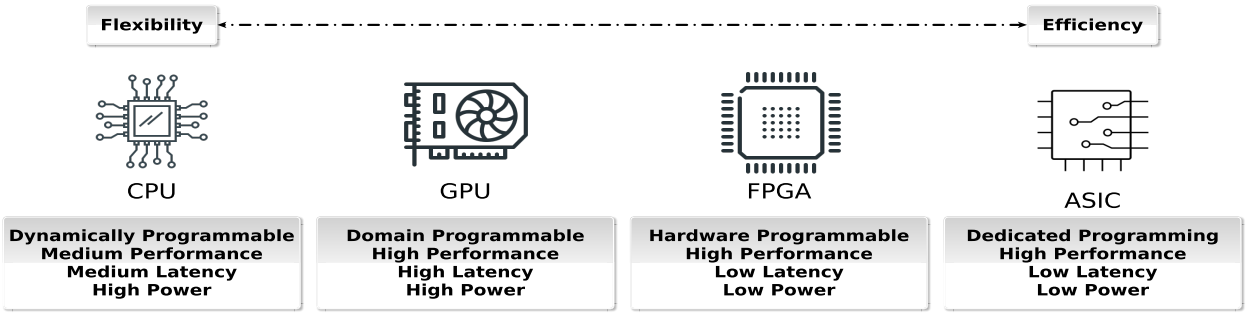

Figure 2.9: Modern Compute Devices

Given the demand for viable solutions to support computationally intensive workloads many industries have attempted to scale horizontally by adding additional CPUs in networked servers to increase processing power. However, scaling in this fashion is expensive and cost prohibitive for many industries. Not to mention, since about 2006, processor speeds have shifted from what appeared to be exponential growth to logarithmic growth and we find ourselves close to a plateau in processor frequency. Increases in the number of transistors per square inch for general purpose CPUs according to Moore's law predictions has declined in recent years. The International Technology Roadmap for Semiconductors (ITRS) has warned that transistor minimization may reach its limit in 2021 [52]. While there has been a steady increase in processing power of general-purpose CPUs, the doubling of performance according to Moore's law projections no longer holds. In light of this, there is a need for new technologies and programming paradigms to offset some of the deficiencies in performance scaling.

In recent years, many have turned towards heterogeneous computing to satisfy computational demands. The most prominent technologies in terms of hardware have been GPUs, FPGAs, and ASIC. Of the three, GPUs offer a readily available option. They are commonly found in most modern desktops and laptops, offering competitive raw processing power as compared to their CPU counterparts [26]. GPUs are capable of processing computationally intensive workloads that would typically be performed by the CPUs. The underlying hardware features of GPUs have been exposed by manufacturers to support parallelism in processing that can be orchestrated by the programmer. While traditional superscalar processors support

many hardware features such as branch prediction, instruction pipelining, and out-of-order execution, GPUs do not. However, what GPUs lack in terms of execution model versatility, they compensate for with exceptional performance.

For the past few decades, general purpose CPUs have been tailored almost exclusively towards serialized workloads. Examples of such workloads include compilation, network communication, and user interactions. There have been advances in recent years towards parallel workloads with the inclusion of multi-core architecture for such purposes. However, CPU cores continue to be optimized for single-threaded performance. Consider an arbitrary x86 CPU with 8 cores, hierarchical shared memory caches, with a max frequency of 4-5 GHz. If this example CPU is made with a 14 nm process, each chip will have approximately 2 billion+ transistors consuming around 35W of power. This example processor is shown in figure 2.10 (notice the portion of the die area used for the ALUs of the processor).

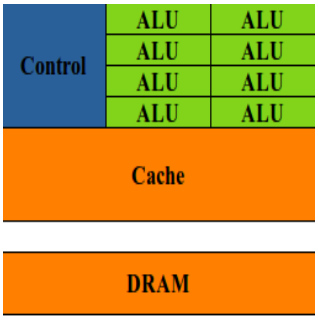

Figure 2.10: General Purpose CPU Architecture

With so little of the die area committed to arithmetic computations, it is evident why additional CPUs are required for intensive computations. The remaining space of the die and the main contribution to heat comes from multiple caches, decoders, and additional extensions while not visible on the die, contribute significantly to the increase in single thread performance. One of the primary contributors to increasing single thread performance is the concept of speculation. Speculation optimizations go far beyond speculative execution (the execution of instructions prior to determining if the resulting output will be needed for future operations) into such areas as data caches, branch predictions, and out-of-order processing. Speculation allows the system to anticipate with a given probability the need for data. This

in some sense could be thought of as a bet that a certain portion of the data will be in high demand in the near future.

These speculative optimizations may work well depending on the workload, particularly in cases where the data has high locality, significant branching, and a variety of operations. However, these optimizations come at the expense of precious die space which could otherwise be used for additional processing units. If the workloads tend to be of a scientific nature such as similar sequential operations, then these optimizations remain underutilized, merely hoarding precious die space and consuming countless watts of thermal design power.

Many companies elect to use advanced process technologies to achieve higher operating speeds, but increasing the clock rate comes at a price. Particularly, transistors operating at higher speeds lead to higher operating temperatures. Overclocking processors requires significant adjustments in heat dissipation techniques beyond air cooling. Water or nitrogen cooling is unfeasible for most users and companies for that matter. In addition to thermal issues, high speed processors also contribute to increased power leakage and are cost prohibitive to develop. In light of these issues, processor speeds have leveled off around 4 GHz to date but other methods have been employed to increase the performance of these processors up to a maximum of 5 GHz.

It should come as no surprise that micro-architecture optimizations can contribute significantly to performance of these chips. The typical methods used fall into two class: *dual-issue core* and *multi-issue core* [43]. The dual-issue core is a superscalar architecture which allows the processor to execute up to two instructions per clock cycle. In many cases, there is a set of specifications which define which instructions can be executed together. Those rules often define explicitly what operations can be performed simultaneously on particular data types. For instance, two integer operations or one integer operation and one floating-point but not two floating-points can be executed simultaneously. Most modern programs typically execute in sequential order and because of this, the dual-issue core can often exploit the intrinsic instruction-level parallelism (ILP) found in many programs for a significant performance gain. While the processor cannot always exploit ILP in every program, there is typical enough ILP in most programs to take advantage of this functionality. Multi-issue cores follow the same principles but have more execution units to utilize simultaneously.

Modern processors often utilize the multi-issue architecture and advanced features of an x86 processor such as multi-branch prediction, speculative execution, and simultaneous multi-threading among others. Most of these performance enhancements are used to optimize single-threaded performance of the device. New methods are being used to squeeze as much performance out of these processors as possible given that manufacturers have been hitting the wall or limitations for Von Neumann architecture. There are actually three particular types of "walls" that manufacturers encounter for this architecture:

1. Power
    - Increased frequency leads to increased power density.
    - Difficult to mitigate dynamic and static power dissipation.
    - Diminishing return on performance for higher power density.
2. Memory
    - Compute bandwidth continues to outpace memory bandwidth.
    - Data migration can become the limiting factor on performance.
    - Exacerbated by increasing data set sizes.
3. Instruction-level parallelism
    - Increasingly difficult to find parallelism in single instruction streams.
    - Diminishing returns on additional ILP hardware.
    - Functional Units remain idle waiting for memory access.

Many programs have sequential execution which allows the CPU to exploit the natural ILP of these programs but the performance is program specific. In terms of the *power wall*, adding more and more transistors to the processor and running them at high speeds has increased the power dissipation of the processors far beyond the capacity of inexpensive cooling techniques. This *memory wall* also has another name, the Von Neumann Bottleneck. In the past few decades, processor speed has increased dramatically but the performance of memory has not kept pace. Much of the improvements in memory have been attributed to increased density (i.e. higher memory capacity). With the advent of faster processors, the CPU spends an

increasing amount of time waiting for data to be fetched from memory. Irrespective of the processor, its speed is in effect limited by the rate of transfer from memory for most operations. A faster processor would simply mean increased idle time. Thus, memory has secured its place as the primary bottleneck in Von Neumann architectures.

Several solutions are used to alleviate the intensity of the bottleneck. The most prominent ones are Larger Caches, Hardware Prefetching, Software Prefetching, and Multi-Threading [12]. Increasing the cache size can be prohibitively slow and only efficient if data has both temporal locality and fits into the cache. Hardware Prefetching cannot be optimized for each application and its functionality is based on the behavior of the program execution at runtime. Software Prefetching typically excels for iterative loops with regular array access, but it requires source code and manual programmer intervention which is not often feasible, especially in the case of precompiled programs and closed source. Finally, Multi-threading solves the problem of throughput but ultimately does not contribute to decreases in memory latency.

The present solutions can only take this so far but in order to traverse these walls, many have chosen to look to a paradigm shift in terms of hardware architecture. The Von Neumann architecture has securely reserved itself a place in computing. However, as researchers look to move beyond the inherently sequential processing paradigm for demanding applications, they look to incorporate modern GPUs.

2.3.2 Modern GPU Architecture

The Graphics Processing Unit (GPU) has changed dramatically since its inception. It began as a specialized device used to accelerate the rendering of computer graphics for visual displays. To output images to the screen via the frame buffer, the GPU used fixed-function 3D graphics pipelines to quickly process pixel data independently and in parallel from vertex, texture, and lighting data. This data needed to be processed quickly but unlike CPUs, there was less necessity for minimal latency and more of an emphasis on high throughput, as the visual acuity of humans is less sensitive and can operate on longer time scales than other critical systems. The effort to have maximum throughput was necessary as the images required the GPU to process millions of pixels at a time and in the case of real-time rendering,

billions of pixels per second. The process of image rendering has a fundamental level of parallelism.

Data input to the GPU goes through a series of pipeline tasks. The output of each pipeline stage is used as an input to the next stage of the pipeline. Operating on data simultaneously in consecutive tasks, the pipeline reveals the task parallelism of the GPU architecture. As each stage of the pipeline operates on multiple data inputs simultaneously, this exposes the intrinsic data parallelism capability of the GPU. The first programs written for GPUs were geared towards graphics processing and utilized languages similar to assembly that mapped user specified input data to particular operations. Researchers began investigating alternative computational methods for their parallelized workloads and began to re-purpose GPUs for such endeavors. Using GPUs for such tasks required researchers to reorganize their programs into a graphics processing format [25, 28, 41]. Writing programs for scientific computations as a graphics processing tasks proved difficult. The resulting programs were riddled with bugs that were hard to isolate and the code was challenging to debug, optimize, and develop.

However, research began to indicate that GPUs offered better performance for certain algorithms compared to their CPU counterparts, and the adoption of GPUs for computations increased [6, 35, 9]. With this new found interest in GPUs came the development of high level languages which simplified programming tasks and decreased the burden of dependence on knowledge of the underlying graphics systems in order to create programs. While these graphics programming languages allowed many researchers to show performance improvements of GPUs over CPUs for particular workloads, they were ultimately deprecated when hardware vendors released their own implementations for their hardware. The GPUs first conceived as simple graphics processing devices have evolved into indispensable tools for deep learning, artificial intelligence, bioinformatics, and essentially any computationally intensive process which requires a high level of parallelism [40, 1, 27, 51, 38, 21].

Figure 2.11: CPU vs GPU Architecture

Before the advent of GPU computations, mathematical libraries were used to create performant computations on general purpose CPUs. Developers spent considerable time analyzing CPU architectures and memory hierarchies to develop optimizations for general purpose hardware [10]. However, CPUs lack the necessary computational units to process large data sets in parallel given their high frequency but low core counts. GPUs on the other hand have low frequency but high core counts that are specifically designed to perform floating point computations for 32 and 64 bit data types. A comparison of the two architectures can be seen in figure 2.11. Developers have analyzed GPUs to create specialized libraries that take advantage of GPU architecture [3, 11, 39]. The computational power and specialized architectures leveraged by GPUs over CPUs for parallel workloads explains the rapid adoption of GPUs for ML and Big Data workloads today [38].

2.3.3 GPUs as Accelerators

The modern GPU has moved far beyond the display of graphics for visual applications. While some GPUs continue to display graphics, specialized GPU cards have been developed (sans graphics port) to utilize the GPU solely for the purpose of intensive computations. Many manufacturing companies create GPU devices, with the top manufacturing companies being Intel, AMD, and NVIDIA. Though Intel is the largest manufacturer, their graphics focus has been toward integrated and low-performance cards found in laptops and economy workstations and servers. The remaining two suppliers, AMD and NVIDIA, are well known for their high performance cards. Of the two, NVIDIA appears to be the dominant supplier of cards for academic and industrial environments. NVIDIA also happens to be the maker of CUDA, a programming language developed for their line of GPUs.

Newer GPUs are composed of multiple GPU processing Clusters (GPCs), Texture Processing Clusters (TPCs), Streaming Multiprocessors (SMs), and memory controllers. Differing from standard server architecture, advanced GPUs utilize High Bandwidth Stacked DRAM memory which has significant advances over standard GDDR5 server memory. A GPC combines multiple TPCs into individual units. Each GPC contains the characteristics of a standalone GPU and each one can be dedicated to an individual workload or used in conjunction with one another. The TPC is a cluster of SMs with a texture unit and logic controls. Similar to the GPC, the TPC can be grouped into higher level configurations known as a *Streaming Processor Array*. The highlight of the TPC is the ability to utilize texture memory functions. Texture memory found in the texture unit is a cache memory, entirely separate from the global, shared, and register memory. It can be used to improve both latency and bandwidth for certain workloads. The texture memory cache is geared towards 2D graphics processing but for computational workloads the optimization of the cache allows for 2D spatial data locality. With the data bound to pitch linear memory, a running kernel can update this data allowing for increased performance in caching behavior and minimize superfluous data duplication computations, in calculations that require two-pass updating.

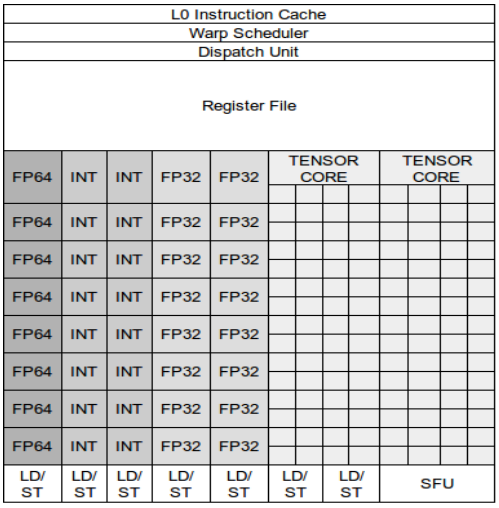

Figure 2.12: Streaming Multiprocessor Architecture

The primary workhorse of computations is the SM shown in figure 2.12. The SM contains several cores and Special Function Units (SFU). The collection of SMs are multithreaded

SIMD processors. At a high level, the GPU is a MIMD composed of a grouping of multithreaded SIMD processors. The SM are optimized for high throughput floating point operations. The SM is equivalent to the Processing Element as seen in CPUs. The floating point and integer cores allow for faster throughput in computations. The SFU supports high throughput intrinsic functions (sin,cos,etc). The Tensor cores are high throughput mixed-precision FP16/FP32 cores optimized for deep learning applications. Multiple Tensor cores per SM are capable of performing multiple floating point operations per clock cycle. Proactively placing these specialized Tensor cores on the GPU allows for specialized machine learning tasks and emerging applications.

To process data GPUs uses two different architectures. For global execution across the TPCs, the MIMD architecture is used. To execute across the SMs, the Single Instruction Multiple Thread (SIMT) architecture, a subset of the SIMD classification, is used. Unlike traditional Von Neumann processors, the SIMT architecture allows a substantial number of parallel computations to execute across thousands of hardware threads each with different data sets. SIMT advances the SIMD design pattern not only in performance but in ease of use for programmers. Given that SIMT is scalar, it has no predefined set vector width which allows SIMT to perform at maximum speed regardless of the vector width. By comparison, the SIMD architecture experiences a capacity reduction when the input size is smaller than the SIMD/MIMD width. SIMT guarantees that the processing cores operate at full capacity consistently. Demanding workloads with substantial parallelism can benefit from the massively parallel computational abilities of GPUs.

2.3.4 GPUs are not the solution

GPU adoption has increased over the past few decades and has become the go-to device for parallel workloads, but have we really solved the challenges faced by the modern CPU? GPU architecture with its own caches, memory hierarchy, and PCI bus communication exploits embarrassingly parallel workloads by design but the primary bottleneck for Von Neumann processors is inherent to GPUs as well and significant performance degradation in data migration continues to persist. We have seen that CPUs lacked sufficient compute units to process massively parallel workloads and researchers have scaled horizontally to increase computational power. As we look at the landscape of GPU computing are we not doing

the same thing? That is, densely packing servers with more and more GPUs and linking them over high bandwidth technologies such as RDMA and infiniband. Have we not simply shifted the problem to a different (albeit more capable) device? As new cards emerge with additional features, compute units, and extensions, we find ourselves refactoring code again. Another challenge that is faced by GPUs which is often overlooked is the notion of code branching. GPUs perform very badly, in fact worse than CPUs, when they encounter code with conditional branches. To circumvent this problem, many tools recommend executing branching code on the CPU and strictly parallel code with no branching on the GPU. This can bring other challenges as some data is on the CPU and in system memory while other portions are traveling across the bus to the internal GPU memory. Such synchronization, aggregation, and coherence can severely affect computational performance. The constant refactoring continues as new revisions of GPUs arrive to market and data migration challenges persist across device specific memory hierarchies. In a sense, we have not solved the problem but have simply shifted the issue to another architecture.

2.4 Alternative Hardware Accelerators

2.4.1 Application-Speciic Integrated Circuits

Industries with specific computational demands have diverged from standard CPU and GPU adoption in favor of Application Specific Integrated Circuits (ASIC). These devices differ from other hardware solutions in that they are not targeted for general purpose applications. These application specific integrated circuits, or chips, are targeted for very specific use cases. Such chips are often used in embedded device or single purpose applications in fields such as networking, bitcoin mining, and automotive industries just to name a few. As discussed previously, by adopting general purpose processing solutions, one has to tailor computations to the underlying hardware which may have drawbacks that limit computational performance. ASICs on the other hand, tailor the underlying hardware architecture to the computational demands. This makes for hardware with very precise architecture that ensure certifiable execution times and can take full advantage of the resources in order to complete the computation or function. However, developing such chips require a significant amount of time in terms of design, development, and fabrication. Traditionally, an industry

System Specifications	Identify the non-formal functionality specifications of the ASIC.
Architectural Design	Layout development taking into account area, power, and size considerations.
Functional and Logic Design	An electronic components description used to capture functional requirements of the integrated circuit logic.
Circuit Design	The physical description of ASIC circuitry that achieves the system specification using a hardware description language.
Physical Design	Partitioning, floor-planning, placement, Clock Tree Synthesis, Signal Routing, and Timing.
Physical Verification and Signoff	Formal circuit, timing, power testing, evaluation, and verification.
Fabrication	Physical device creation.
Packaging and Testing	Device packaging and final testing to ensure that ASIC operates within specified tolerances.
Completed Chip	Ready for use by the industry.

Table 2.1: ASIC Design Flow

with interest in using ASICs would need a large team including several designers that would come up with an architectural layout using hardware description languages (HDL). Once the design was completed, the developer would need to engage an ASIC manufacturer to fabricate the design. Initially, designers would normally be constrained to using the available design tools of a manufacturer based on Verilog or VHDL. Over time third parties began developing design tools to provide features comparable to those found in manufacturer-specific tools, and today there are many options to choose from. These modern tools assist with logic synthesis and are able to compile HDL design descriptions into gate-level netlist which gives a description of the connectivity of all electronic components to one another. Creating ASICs follows a very specific design flow as shown in Table 2.1.

One of the most intensive processes of ASIC development is Functional Verification. Such verification may include techniques such as logic simulation, emulation, and formal verification. Unlike FPGAs, ASICs cannot be reprogrammed once they complete the fabrication phase. If errors are found after the fabrication phase, the redesign and re-fabrication of the device can be cost prohibitive. To eliminate errors, designers may elect to use full coverage

testing wherein the device is tested through all of the possible permutations of its functionality. However, given the volume of potential permutations or test-cases, even for a simple design, functional verification tests could exceed a Vigintillion (10^{63}) test cases in order to verify a design. The functional verification process is often compared to program verification, both of which are NP-hard and may have zero possible solutions in all test cases. Solutions such as simulation, emulation, and intelligent verification can assist with the process in most cases to ensure the correctness of the design to a given margin of error.

Design and fabrication of ASICs can fall into two categories: Fully Custom and Semi-Custom designs. Many circuit designs have common components that fulfill basic logic or functional requirements. These common components are often packaged into cells and comprise a library of circuitry that may be offered by a vendor in order to speed up development and limit not only re-inventing common functionality but also required Functional Verification for a portion of the design. Designers are free to choose such libraries or create new ones for their applications. In the case of fully custom designs, the designer has the flexibility to create a complete layout of the circuitry on the device whereas semi-custom designs are slightly more constrained given that the design may use one or more of the pre-designed cell libraries provided by the manufacturer.

Clearly, ASICs provide the best performance given that the architecture is completely tailored to the computation, but this flexibility comes at a considerable price in terms of cost, price, and effort. Some of the time expense can be reclaimed by taking advantage of semi-custom designs but ultimately, one has to create a new hardware architecture which is certainly more time consuming than merely refactoring source code for the CPU or GPU. The multi-factor expense of ASIC development seems larger than the effort required to refactor code for other architectures. Have ASICs solved the refactoring problem or is there another solution that will enable professionals to take advantage of architectural flexibility without reinventing the wheel?

2.4.2 Field-Programmable Gate Array

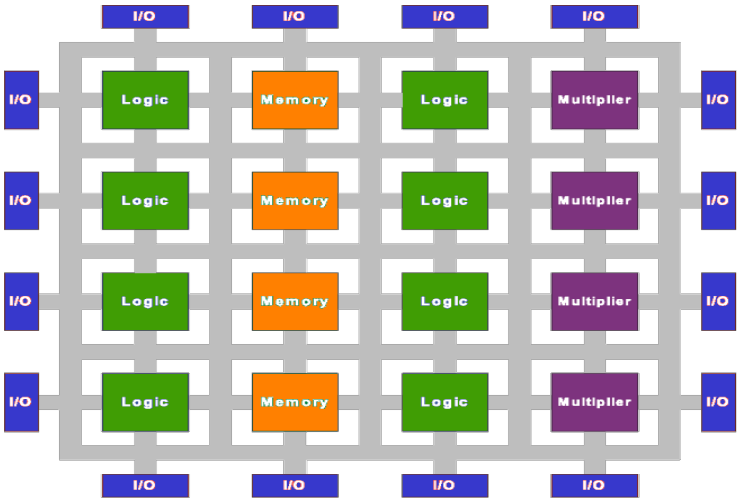

Figure 2.13: FPGA Architecture

While ASICs have a long development time, a solution that is seeing a lot of adoption lately is that of the Field Programmable Gate Array (FPGA) shown in Figure 2.13. These devices have a lot in common with ASICs and when compared to the ASICs fully customizable design abilities, FPGAs fall into the semi-customizable category. FPGAs are integrated circuits that provide re-programability which is something that ASICs lack. They also can be configured using the hardware description languages as used for ASIC development. FPGAs are comprised of arrays of programmable logic blocks and a nested hierarchy of reconfigurable interconnects that allow designers to connect available circuitry at will. The logic blocks support simple or complicated functions and most FPGAs include complete memory blocks. With an ever growing amount of resources at the disposal of the FPGA, designers can build ASIC like circuity with competitive I/O rates to that of modern computer systems and can be used in mission critical areas where timing constraints must be fulfilled.

FPGAs provide an unparalleled level of flexibility and enable designers to create dynamic architectures with considerably less effort and costs than ASIC development. Yet, why have FPGAs not been adopted in all industries with demanding computational needs? One major factor is the steep learning curve required to master hardware description languages such as

VHDL and Verilog. Even with the growth in programmers' workforce there has always been a limited number of professionals with an understanding of low-level architecture development.

Improvements in FPGA architectures make them attractive for real-time cyber-physical systems with timing, area, and power specific requirements for applications ranging from autonomous vehicles, to computer vision to robotics [24, 31, 32, 37, 47]. Unlike CPUs and GPUs with fixed datapaths, pipelines, and computational units, FPGAs allow users to adapt the hardware to critical features of computation. Of specific relevance to our problem of interest, Zhuo and Prasanna [56] deploy matrix multiplication on an FPGA using HDL, as have Thomas and Luk [53] in the context of random number generation. Instead of waiting for GPU manufacturers to develop new logic for a given computation, users can rapidly develop new architectures, augment existing designs, and iterate through revisions to find the best design for the task at hand. Rather than using low-level languages such as Verilog or VHDL, our focus is on exploring the utility of a higher-level language, OpenCL. FPGAs provide dynamic architecture which is needed to solve the matrix multiplication issue but given the barriers to languages, has not been adopted at large. If the learning curve was lessened would the FPGA be a viable solution? By incorporating the FPGA, one's computation would still suffer from the bottlenecks associated with the CPU and GPU wherein memory has to be shuttled from one device to another across the slow bus. An interesting synergy of solutions have developed which may make the FPGA a viable solution for matrix multiplication and other computational expensive tasks.

2.4.3 Hybrid CPU+FPGA

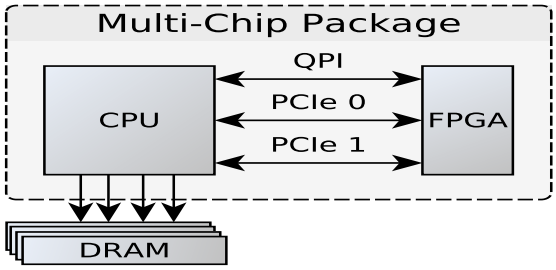

Figure 2.14: Intel HARP Version 2 CPU+FPGA Architecture

No single device excels at all computational tasks, and computations can alternate between serial and parallel execution leaving the performance improvements of accelerators diminished by data migration that limits computational performance (and may be further exacerbated by imperfect coordination of multiple devices). GPU makers have focused on high-bandwidth memory to reduce external memory transactions and device-to-device interconnects to speed up cooperation. Ultimately, most of these devices are limited by the speed of PCIe interface. One radical architecture, which may minimize data migration in the case of FPGAs, comes from a solution which combines CPU and FPGA architectures. Intel has introduced the Heterogeneous Architecture Research Platform version 2 (HARPv2), shown in figure 2.14 which consists of an Intel Broadwell Xeon CPU combined with an Intel Arria 10 GX1150 FPGA into a Multi-Chip Package (MCP) with shared DRAM memory through a low latency, high bandwidth, Intel QuickPath Interconnect (QPI) and two Peripheral Component Interconnect Express (PCIe) busses. This supports a common last-level cache and DDR memory.

General purpose communications can travel across the PCIe bus and the primary workload can be shared across distributed-memory between the CPU and FPGA caches. The Intel HARPv2 is the second-generation of the HARP platform. The first-generation HARP platform, released in 2015, consisted of an Intel Xeon CPU combined with an Altera Stratix V FPGA in a Discrete Configurable Platform (DCP). These two architectures were connected with a single QPI channel and shared DRAM memory. The HARPv2 moves beyond a discrete configuration for separate chips and combines both chips into a single MCP. In addition to unified DDR memory, the FPGA supports cache coherence and virtual-to-physical memory address translation. This provides a unique communications capability between the CPU and FPGA. This emerging technology gives users an opportunity to architect their own solutions without having to perform the arduous task of designing new circuitry from schematic to fabrication.

2.5 Hardware Agnostic Programming

2.5.1 OpenCL Overview

While architecture unification is emerging in platforms such as the HARPv2, one of the barriers to working with FPGAs has traditionally been the lack of knowledge and experience with Hardware Design Languages(HDLs) such as VHDL and Verilog which describe precise low-level structure and behavior of electronic circuits. Using these tools, one can define a formal description and perform automated analysis and simulation of an electronic circuit. Thereafter, the HDL semantics can be synthesized, using High-Level Synthesis (HLS), into a netlist that describes the physical component layout along with exact connections to each component. Because HDL-based languages target digital logic circuits, their effective use requires a deep knowledge of electronic circuit design, theory, and applications, which introduces a steep learning curve for some users who would like to delve into hardware design.

OpenCL is an industry standard language developed by the Khronos Group and is designed to take advantage of parallelism in multi-core processors such as CPUs, GPUs, and FPGAs. We live in an environment where there is a high demand for devices with superior computational performance in industries from science and engineering to video production and financial corporations. To satisfy this need, many industries look to transition to heterogeneous platforms which utilize co-processors and other accelerators. Over the years there have been multiple languages developed to take advantage of GPU hardware. In the early 2000s, the prominent tools were graphical APIs such as OpenGL and DirectX. There were various intermediate solutions such as BrookGPU and RapidMind. Thereafter, vendors developed their own implementations to support GPU computing. Currently, OpenCL is rapidly becoming the dominant language for GPUs given its open model and its ability to be used on cross-platform GPUs. To better understand this model, we will now look at the OpenCL architecture.

To enable broader experimentation with heterogeneous architectures and designs, the the OpenCL framework allows rapid development of programs for such platforms. The HARP platform supports HLS through OpenCL and allow users to implement FPGA computations using a C-based language without describing the hardware implementation of an algorithm.

However, the OpenCL framework has one drawback in that the developer does not have precise control over the hardware implementation. To have a fully optimized kernel may require an investigation into the low-level architecture. Yet, OpenCL is a step in the right direction and makes emerging architectures accessible to a wide range of programmers.

2.5.2 Platform Model

The OpenCL framework offers a high-level abstraction that removes requirements for low-level hardware configuration and enables orchestration of memory and execution models for parallel workloads across accelerators. Source code can be written, compiled, and executed on a range of OpenCL compatible devices. Every OpenCL program has three primary components: Compute Units, Kernels, and Data Buffers. OpenCL generalizes heterogeneous devices into an OpenCL Platform model. One "Host", typically a CPU, controls multiple "Compute Devices". These "Compute Devices" contain multiple "Compute Units" which have multiple "cores". Each "core" is typically an execution unit referred to as a "Processing Element" and each "Processing Element" can be used by one work-item. Work-items can be arranged into workgroups using an abstraction called an NDRange. OpenCL programs, called "Kernels", are executed on multiple "Processing Elements". The host sends kernels to the compute units and associates data buffers with compute unit(s) memory hierarchies. In many instances, when the hardware allows, the number of kernels sent to the compute device can be proportional to the dimensions of the data to be processed. These kernels, also known as work items, are then directed to individual processing elements. Work items are processed in groups and are executed in parallel to process all the data in the compute unit memory. Each work item, will operate on a specified region of memory.

OpenCL provides two abstractions for partitioning workloads: NDRange and Single-Work-Item (SWI). An NDRange describes a 1- to 3-dimensional space for work-items. Contrasting this is the SWI, which follows a sequential model similar to many programming languages. However, OpenCL can extract pipelined parallelism from code at compile time, based on dependency analysis, to replicate a deeply-pipelined worklflow that is common for FPGAs.

2.5.3 Memory Model

Before we explore OpenCL Memory architecture, it is important to say a few words about consistency. Consistency models are seen across many aspects of computer science and in systems with distributed shared memory. A given system, is said to support a particular model if the memory operations follow a given set of rules. There are two categories that sequential consistency models fall into. Those being Issue and View. Issue defines rules that specify the ways in which a process can issue operations. View defines a set of rules as to how those operations can be ordered and their visibility to the affected processes. The consistency model can for instance specify that a given process is not allowed to issue operations until a previous set of issue operations have been completed. These categories are also referred to as program order and and write atomicity. If all of the criterion for a given consistency model are satisfied, it is considered stronger than another model which may not fulfill the entire criteria specified. By relaxing one or more of the sequential consistency model requirements we derive a model called the Relaxed Consistency Model.

This model offers no memory consistency at the hardware level. Under this model, the programmer is responsible for realizing consistency via synchronization techniques. Generally, a few methods are used to define a relaxed consistency model as shown in Table 2.2

Relaxation	Requirements (usually program order or write atomicity requirements) may be relaxed in the sequential consistency model .
Synchronizing	Assign variable restrictions to two group so that one group has weak consistency and the other defines a more restrictive model.
Non-Synchronizing	One consistency model for all memory access.

Table 2.2: OpenCL Consistency Model

Relaxation techniques can also define a relaxed write to read, relaxed write to read and write to write, or relaxing read and read to write program orders for example.

In the case of OpenCL, the memory model follows the relaxed consistency model and classifies memory into four groups: Private Memory, Local Memory, Global/Constant Memory, and Host Memory, shown in Table 2.3

Private memory	Private to individual work items executing within a processing element. No visibility to the host. Can be accessed by all work items but variables created by individual work items are not visible to other work items in the work group.
Local memory	Allocated exclusively to individual compute units. Not visible to the host. Allows read and write access by all of its processing elements within the compute unit. Typically used to store shared data that must be accessible to multiple work items. Synchronization and consistency may be achieved by utilizing fencing and barrier methods of OpenCL.
Global memory	Accessible by both the host and device. Allows for read and write access for host and all compute units.
Constant Global memory	Accessible by read and write access to the OpenCL host but allows only read access to the device.
Host memory	Only accessible by the OpenCL host. In order to move data to lower regions of the memory hierarchy, it must be copied sequentially. For instance, to move data to private memory, it must be moved from host memory to global memory to local memory and finally to private memory.

Table 2.3: OpenCL Memory Hierarchy

2.5.4 Execution Model

The OpenCL execution model acts on applications. Applications can be split into two parts, host side and device side. OpenCL applications created for the host call OpenCL

APIs, compile and submit kernels, allocates memory for devices, create command queues, and other administrative processes for device management. An OpenCL host will utilize the OpenCL API platform to orchestrate the computational task by identifying compute devices, submitting kernels to selected devices, and managing the workload across the devices. On the device side, OpenCL kernels, written in the OpenCL implementation of C are created and execute across work items, perform parallel tasks, and operate compute device processing elements. OpenCL provides granular data parallelism and thread parallelism within data parallelism and task parallelism. A typical host program manages the device kernel execution by creating queues for memory and kernel execution commands and synchronization. It also creates a context for the kernels which includes compute units, program and memory objects, along with the kernels themselves.

The sequence of Kernel execution is as follows:

1. The host defines a kernel.
2. The host submits the kernel to the compute unit for execution.
3. OpenCL generates an NDRange for work-items.
4. Based on the NDRange, an instance of the kernel is created for each element in it.

OpenCL work-groups have a number of properties and restrictions that should be elaborated on. OpenCL work-groups are independent of one another and multiple work-groups can be executed in parallel. Work-items within individual work-groups can communicate with each other by utilizing shared data buffers. However, these buffers must be synchronized in order to be accessed. At the lowest level, processing elements execute all instructions sequentially. Unlike their CPU counterparts, the processing elements do not have branch prediction nor speculative execution. Consequential, if conditional branch paths exist, execution of both paths are required. Code can be modified to perform branch logic on the host side to limit the superfluous branch execution.

By creating a context consisting of heterogeneous devices, multiple accelerators can work in tandem to solve problems for which they are well suited. However, this solution does not resolve the Von Nuemann bottleneck. For most architectures, transferring data through

memory (across bus channels) is still the weakest link in the computational chain. An alternate solution would be to combine the best of the architectures, perhaps in the form of Application Specific Integrated Circuits (ASICs) but these would only prove valuable for one application which would not accommodate general purpose computations. The current developments in the field of hybrid processors such as CPU+FPGA built into the same die space may become a promising solution. With both devices on the same die, memory transfer speeds increase significantly. Branching code can be executed by the CPU with parallelized workloads sent directly to the FPGA.

While FPGAs may present one part of the solution, CPUs are still a relevant part of heterogeneous computing. OpenCL enables the aggregation of heterogeneous computational systems into a well defined, manageable, and cohesive package. OpenCL offer a high level abstraction language which allows the creation of parallel algorithms that can execute efficiently on a variety of hardware architectures. OpenCL allows programmers to describe and manage parallelism in a hardware agnostic manner in contrast to hardware description languages (HDLs) such as Verilog and VHDL which require descriptions at a much lower level which limit the portability and constrain the description. Standard high-level synthesis tools have some measure of higher level abstractions but they have a fundamental limitation of converting sequential C based source code into a parallel HDL implementation. This methodology makes it problematic to accurately express maximum performance thread-level parallelism in FPGAs.

NDRange

FPGAs and GPUs contain similar but different execution models. A traditional GPU can take advantage of SIMD parallelism wherein a single instruction can be performed on multiple data inputs. The substantial number of compute units available on these devices can significantly increase performance for embarrassingly parallel operations. The parallelism comes from the fact that such operations partitioned across processing elements are independent and can all execute at the same time. This method uses a programming style called an NDRange. That is, an N-dimensional range of processing elements with $N \in \{1, 2, 3\}$.

Single Work-Item Instruction

Contrasting SIMD parallelism is the notion of pipeline parallelism that is readily suggested for use in FPGA based applications. Instead of having all the processing elements execute a single instruction on multiple datasets, pipelining allows each operation to move in lock step across the processing elements.

One interesting consequence of these parallelization methods is the way in which branching code is executed. When branching occurs in code of the SIMD processing elements, it is necessary for all of the processing elements to perform the same operation or stall until the particular operation is finished. This can introduce long idle times in the computation as all operations must be synchronized. In many tasks, the pipeline parallelism provided by the Single Work-Item instruction can be beneficial to many workloads.

The overall execution model for both methods is characterized in Figure 2.15. Computations involving branch statements or following a particular ordering of operations are characterized in Figure 2.16.

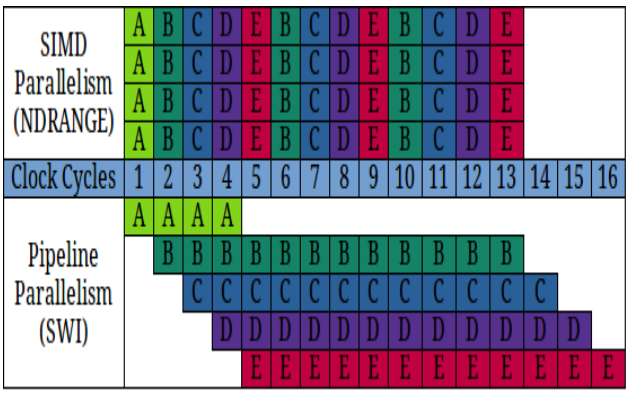

Figure 2.15: Execution Behavior

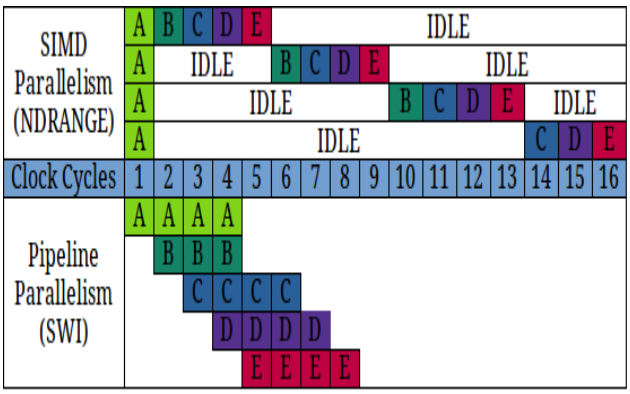

Figure 2.16: Branching Behavior

Chapter 3

Matrix Multiplication

3.1 Theory

In ages past, great monolithic machines occupied entire floors of academic halls, laboratories, and government institutions. With the progression of time, the computer evolved. Smaller, faster, and more powerful generations superseded their monolithic predecessors. Today the computer is pervasive. It can be found in our homes, offices, pockets, and even within us. Even as computers have become smaller, more powerful, and geographically distributed, we find them being aggregated together to form larger (sometimes nebulous) clusters. Today, as we explore a small facet of computer technology and its evolution, we will find this theme of expansion and contraction occurring over and over again in our investigation of emerging technology.

Underlying any calculation is an equation. However, calculations can come in many different forms and the most challenging problems typically involve multiple equations. These individual equations can be aggregated to become a system of equations wherein valuable answers are often derived from solutions that satisfy all equations simultaneously. The coefficients of such equations, placed into a matrix in row/column order, can be used to describe a system of linear equations.

Numerous operations can be performed on a system of equations. One of the most prominent and time consuming operations involves multiplication. Matrix multiplication has applications in many areas, ranging from modern physics and graph theory to Markov Chains and

computer science, establishing a great need for viable solutions to support these computationally intensive workloads.

Many calculations that we perform by hand only involve simple arithmetic, but more complicated calculations typically involve an equation, such as:

$$x_1' = a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n$$

Individual equations can be manipulated with little effort, but significantly more effort may be required when we move to systems of equations:

$$\begin{aligned} x_1' &= a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \\ x_2' &= a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \\ &\vdots \\ x_m' &= a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \end{aligned}$$

Solving these equations and keeping track of variables and subscripts by hand can become a very tedious process. In light of this, it may be better to place the equations into the standard matrix equation format:

$$\begin{bmatrix} x_1' \\ x_2' \\ \vdots \\ x_m' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{n2} & \ldots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Using concise notation, the matrix equation can be written as $x' = Ax$ where x' and x are vectors and A is an $m \times n$ matrix. The $m \times n$ matrix A consists of m rows and n columns and the set of $m \times n$ matrices with real coefficients may also be denoted $\mathbb{R}^{mn}$. This equation is known as a vector matrix product and has many applications in circuit and state equations. The vector matrix product is a special case of matrix-matrix multiplication. Given an $m \times k$

matrix A and an $k \times n$ matrix B, the $m \times n$ matrix C, is the product of AB that is defined by:

$$A_{mk}B_{kn} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1k} \\ a_{21} & a_{22} & \dots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{n2} & \dots & a_{mk} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \dots & b_{kn} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{n2} & \dots & c_{mn} \end{bmatrix}$$

The product AB can be obtained by multiplying each term of the ith row of A_{ik} and the jth column of B_{kj} over k and summing the products. Therefore, c_{ij} is the dot product of the ith row of A and the jth column of B and is defined as $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$.

When data are represented in matrix format, many techniques can find solutions to this system of equations, but more importantly placing the data into a row/column format makes the data more amenable to the row-major or column-major order used in modern computer systems. Notice that data placed in matrix format is uniquely tailored to the underlying architecture of computer hardware with its rows and columns of ordered data found in memory, pipelines, and files makes the computer a prime candidate to perform operations on such ordered data.

To perform matrix multiplication efficiently on computer systems, numerous matrix multiplication algorithms have been proposed with the most trivial implementation running in $\mathcal{O}(n^3)$ time. To illustrate this, we will briefly investigate some of the algorithms which lead to the necessity of improving matrix multiplication. The standard computer algorithm for performing the matrix multiplication requires three for loops. Each nested loop runs in exactly n, m, and k iterations, respectively, and with the assignment of sum occurring in constant time, this algorithm runs in $\mathcal{O}(mnk)$. When the dimensions of the matrices are equal (i.e. $m = n = k$), this algorithm runs in $\mathcal{O}(n^3)$ time. This running time can contribute to increasing computational wait times as the size of the matrices increase. The challenge of matrix multiplication has led to the development of various algorithms which have well-known behavior and performance. Improved running times for matrix mutiplication were discovered by researchers such as Strassen ($n^{2.807}$), Coppersmith & Winograd ($n^{2.376}$), Slothers ($n^{2.374}$) , Williams ($n^{2.3728642}$), and Le Gall ($n^{2.3728639}$).

Given the plethora of matrix multiplication algorithms available, one would imagine that there are many implementations available for complex workloads. However, in practice, very few of these algorithms are actually implemented. By using recursive techniques, commonly called *divide-and-conqueror* methods, several libraries utilize the Strassen algorithm for subdividing large matrices into smaller subsets that fit nicely into the processors cache. Notice that the Coppersmith & Winograd algorithm, which runs asymptotically faster than the Strassen method is rarely implemented and is typically utilized to prove theoretical time bounds. This is largely due to the fact that in order to benefit from the Coppersmith & Winograd algorithm would require matrices so large that they would exhaust the capacity of all modern computer hardware, making the algorithm, what some would call, a *galactic algorithm* [29].

3.2 Optimizations

For programs developed to run on an FPGA, our only limitation was the amount of resources available to the FPGA. Initially, we intended to increase the blocking and loop unrolling increments but found that we ran into issues wherein the HLS system was unable to place and route additional replicated units. In light of this, we standardized on a maximum of 64 units for blocking and loop unrolling. We set our increments to powers of 2 in order to create a thorough and well-defined analysis of the hardware. We subdivided our optimizations by levels to get a better feel for the impact that each optimization has on the computation. Our optimization levels are as follows:

- Level 0: Naïve Implementation - Standard unoptimized methods for matrix multiplication (i.e., 3 nested loops).
- Level 1: Transposition – Transpose one of the source matrices to enable more efficient row-major order access to benefit spatial locality.
- Level 2: Blocking 2, 4, 8, 16, 32, 64 – Operate on 2-dimensional sub-blocks of the matrices to benefit temporal locality.
- Level 3: Loop Unrolling 2, 4, 8, 16, 32, 64 – Unrolling inner loops to allow deeper computational pipelining.

3.2.1 Naïve

The naïve implementation consists of the textbook, 3 nested loop implementation shown in Algorithm 1. Even though our performance expectations are low for this design, it forms a baseline for comparison with what follows. We will consider this the *unoptimized* version.

Algorithm 1 Naïve_Matrix_Multiply $(A_{M\times K}, B_{K\times N}, M, N, K)$

```
C[M,N] = 0
for row = 1 to M do
    for  column = 1 to N do
        sum = 0
        for index = 1 to K do
            sum = sum + A[row,index] * B[index,column];
        end for
        C[row,column] = sum
    end for
end for
```

While a naïve implementation many not traditionally be classified as an optimization, it provides a good baseline to determine if future optimization choices are beneficial or detrimental to the performance of the computation. Referring to Algorithm 1 we see that the computation consists of three sequential for loops that range over the indices of the matrices.

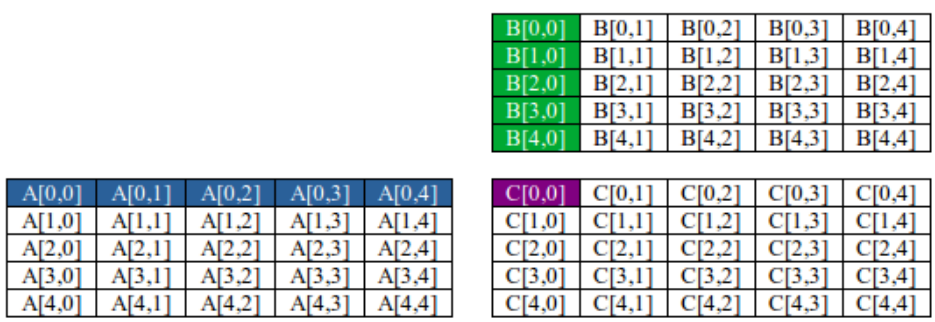

Figure 3.1: Naïve Algorithm - Matrix Multiplication

Looking at the matrix multiplication operations graphically, as shown in figure 3.1, each element of the resulting matrix is comprised of one row of matrix A and one column of matrix B. This sequential method has a significant impact on the cache behavior. Disregarding

cache line sizing and instead focusing on data arrangement within each cache line, we see that this algorithm makes poor use of the cache for columns of Matrix B as shown in figure 3.2.

Figure 3.2: Naïve Algorithm - Cache Behavior

Notice that in the case of Matrix A, all the elements required for the computation fit into an arbitrary length cache line. However, in the case of Matrix B, only one element of the B column is available in the cache line. This will cause significant cache misses and cache reloading which will negatively affect the performance of the overall computation. In the case of sequential matrix multiplication, notice that the row-major order allows us to retrieve elements of A in an efficient manner but given that we require column entries of B, we suffer from numerous cache misses equal to the dimension of the matrices themselves. To eliminate these misses, we will be performing a transposition of the B matrix which will streamline our element retrieval.

3.2.2 Transposition

The first optimization that we employ is to transpose the B source matrix. In OpenCL (which is based on C/C++) matrices are stored in row-major order. As a result, when the B matrix is accessed down a column, there are significant inefficiencies in the cache usage. Algorithm 2 shows the resulting implementation, which benefits the temporal locality of the accesses to B.

To get a better feel for how transposition effects the computation, the rearrangement of the data will occur on the host system and not the accelerator. The transposition optimization concerns the arrangement of the data, and the available ordering methods are row-major order and column-major order. These orderings are the traditional methods used for storing multidimensional array data in linear storage systems. The orderings are often language dependent and in the case of languages such as Fortran, MATLAB, R, and GNU Octave, column-major order is used. The common row-major order languages are C, C++, and SAS. It is important to note that the CBLAS library which was used to perform computations

Algorithm 2 Transposition_Matrix_Multiply $(A_{M\times K}, B_{K\times N}, M, N, K)$

```
C[M,N] = 0
for row = 1 to M do
    for  column = 1 to N do
        sum = 0
        for index = 1 to K do
            sum = sum + A[row,index] * B[row,index];
        end for
        C[row,column] = sum
    end for
end for
```

on the CPU is written in Fortran and has adopted the column-major ordering scheme. Nevertheless, data layout is a critical component in parsing arrays, especially those written in any number of languages.

B[0,0]	B[1,0]	B[2,0]	B[3,0]	B[4,0]
B[0,1]	B[1,1]	B[2,1]	B[3,1]	B[4,1]
B[0,2]	B[1,2]	B[2,2]	B[3,2]	B[4,2]
B[0,3]	B[1,3]	B[2,3]	B[3,3]	B[4,3]
B[0,4]	B[1,4]	B[2,4]	B[3,4]	B[4,4]

A[0,0]	A[0,1]	A[0,2]	A[0,3]	A[0,4]
A[1,0]	A[1,1]	A[1,2]	A[1,3]	A[1,4]
A[2,0]	A[2,1]	A[2,2]	A[2,3]	A[2,4]
A[3,0]	A[3,1]	A[3,2]	A[3,3]	A[3,4]
A[4,0]	A[4,1]	A[4,2]	A[4,3]	A[4,4]

C[0,0]	C[0,1]	C[0,2]	C[0,3]	C[0,4]
C[1,0]	C[1,1]	C[1,2]	C[1,3]	C[1,4]
C[2,0]	C[2,1]	C[2,2]	C[2,3]	C[2,4]
C[3,0]	C[3,1]	C[3,2]	C[3,3]	C[3,4]
C[4,0]	C[4,1]	C[4,2]	C[4,3]	C[4,4]

Figure 3.3: Transposition Algorithm - Matrix Multiplication

Notice that when the data is arranged in row-major order, the computation now operates on row elements of matrices as seen in figure 3.3. The CPU can retrieve the data in contiguous chunks given that the data has been arranged in a manner that benefits both spatial and temporal locality. In the case of column-major order, the entire line that is transferred to the cache contains only one of the needed elements. When the CPU is ready to process additional elements, it must evict the current cache line and retrieve another line to obtain the data in the next row given the column-major ordering. With the transposition of the Matrix B we see an efficient use of the cache behavior as shown in figure 3.4.

Figure 3.4: Transposition Algorithm - Cache Behavior

One interesting caveat about transposition is that it is not a cure-all for cache misses. Even with matrices being loaded into the cache in row-major order, we can still suffer from cache misses when the rows of the matrices are larger than the length of the cache lines. This is a hardware limitation that can significantly affect the performance of our computation.

3.2.3 2 Dimensional Block

The blocking of data is a method that is beneficial to the computation irrespective of whether the data undergoes transposition or not. The key idea is to split the data set into smaller partitions to be worked on independently. This is shown in Algorithm 3. Blocking benefits both temporal and spatial locality. We implement blocking with $TILE\_SIZE \in \{2, 4, 8, 16, 32, 64\}$.

Algorithm 3 Blocking_Matrix_Multiply $(A_{M\times K}, B_{K\times N}, M, N, K, TILE\_SIZE)$

```
1: C[M, N] = Asub[TILE_SIZE] = Bsub[TILE_SIZE] = 0
2: tile1 = tile2 = TILE_SIZE
3: for k2 = 0 to N by tile2 do
4:     for j2 = 0 to N by tile2 do
5:         for i2 = 0 to N by tile2 do
6:             for k1 = k2 to k2 + tile2 by tile1 do
7:                 for j1 = j2 to j2 + tile2 by tile1 do
8:                     for i1 = i2 to i2 + tile2 by tile1 do
9:                         for i = i1 to i1 + tile1 do
10:                            for j = j1 to j1 + tile1 do
11:                                index = 0
12:                                for k = k1 to k1 + tile1 do
13:                                    Asub[index] = A[i * K + k]
14:                                    Bsub[index] = B[j * K + k]
15:                                    index++
16:                                end for
17:                                for k = k1 to k1 + tile1 do
18:                                    C[i * N + j] += Asub[index] * Bsub[index]
19:                                    index−−
20:                                end for
21:                            end for
22:                        end for
23:                    end for
24:                end for
25:            end for
26:        end for
27:    end for
28: end for
```

In the case of transposition, we wanted to reduce cache misses by reusing elements and taking advantage of the spatial and temporal locality found in the data. If we subdivide our computation into sub-blocks we can identify valuable decomposition information.

Consider matrices A and B:

$$A = \begin{bmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \\ a_6 & a_7 & a_8 \end{bmatrix}, B = \begin{bmatrix} b_0 & b_1 & b_2 \\ b_3 & b_4 & b_5 \\ b_6 & b_7 & b_8 \end{bmatrix}$$

We know that the product AB is as follows:

$$AB = \begin{bmatrix} (a_0b_0 + a_1b_3 + a_2b_6) & (a_0b_1 + a_1b_4 + a_2b_7) & (a_0b_2 + a_1b_5 + a_2b_8) \\ (a_3b_0 + a_4b_3 + a_5b_6) & (a_3b_1 + a_4b_4 + a_5b_7) & (a_3b_2 + a_4b_5 + a_5b_8) \\ (a_6b_0 + a_7b_3 + a_8b_6) & (a_6b_1 + a_7b_4 + a_8b_7) & (a_6b_2 + a_7b_5 + a_8b_8) \end{bmatrix}$$

We determine that the rows C are comprised of the following elements of A and B

$$\begin{aligned} C_1 &= [c_0 \quad c_1 \quad c_2] = [(a_0b_0 + a_1b_3 + a_2b_6) \quad (a_0b_1 + a_1b_4 + a_2b_7) \quad (a_0b_2 + a_1b_5 + a_2b_8)] \\ C_2 &= [c_3 \quad c_4 \quad c_5] = [(a_3b_0 + a_4b_3 + a_5b_6) \quad (a_3b_1 + a_4b_4 + a_5b_7) \quad (a_3b_2 + a_4b_5 + a_5b_8)] \\ C_3 &= [c_6 \quad c_7 \quad c_8] = [(a_6b_0 + a_7b_3 + a_8b_6) \quad (a_6b_1 + a_7b_4 + a_8b_7) \quad (a_6b_2 + a_7b_5 + a_8b_8)] \end{aligned}$$

The elements of A have good spatial locality for every row of C and each element has a stride of 1. Similarly, the elements of B have good temporal locality across the columns of C. By reading sub-blocks of data into our cache we hope to take advantage of the data reuse that is inherent in the matrix multiplication calculation. Investigating the rows of C we can see that it is comprised of the following elements of A and B:

$$C = \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix} = \begin{bmatrix} A_{1,j}B_{i,1} \\ A_{2,j}B_{i,2} \\ A_{3,j}B_{i,3} \end{bmatrix}$$

We will get good cache behavior from row reads of A, but given that the calculation requires columns of B, we may experience a cache miss on every element of B. Subdividing the elements into blocks for Algorithm 1 is shown in figure 3.5.

B[0,0]	B[0,1]	B[0,2]	B[0,3]
B[1,0]	B[1,1]	B[1,2]	B[1,3]
B[2,0]	B[2,1]	B[2,2]	B[2,3]
B[3,0]	B[3,1]	B[3,2]	B[3,3]

A[0,0]	A[0,1]	A[0,2]	A[0,3]
A[1,0]	A[1,1]	A[1,2]	A[1,3]
A[2,0]	A[2,1]	A[2,2]	A[2,3]
A[3,0]	A[3,1]	A[3,2]	A[3,3]

C[0,0]	C[0,1]	C[0,2]	C[0,3]
C[1,0]	C[1,1]	C[1,2]	C[1,3]
C[2,0]	C[2,1]	C[2,2]	C[2,3]
C[3,0]	C[3,1]	C[3,2]	C[3,3]

Figure 3.5: Naïve Algorithm - Blocking

Now after transposing matrix B, the rows of C would become comprised of the following elements:

$$C = \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix} = \begin{bmatrix} A_{1,j}B_{1,j} \\ A_{2,j}B_{2,j} \\ A_{3,j}B_{3,j} \end{bmatrix}$$

By working with the rows of A & B, we can make better usage of the cache as we can infer from the transposition blocking behavior shown in figure 3.6.

B[0,0]	B[1,0]	B[2,0]	B[3,0]
B[0,1]	B[1,1]	B[2,1]	B[3,1]
B[0,2]	B[1,2]	B[2,2]	B[3,2]
B[0,3]	B[1,3]	B[2,3]	B[3,3]

A[0,0]	A[0,1]	A[0,2]	A[0,3]
A[1,0]	A[1,1]	A[1,2]	A[1,3]
A[2,0]	A[2,1]	A[2,2]	A[2,3]
A[3,0]	A[3,1]	A[3,2]	A[3,3]

C[0,0]	C[0,1]	C[0,2]	C[0,3]
C[1,0]	C[1,1]	C[1,2]	C[1,3]
C[2,0]	C[2,1]	C[2,2]	C[2,3]
C[3,0]	C[3,1]	C[3,2]	C[3,3]

Figure 3.6: Transposition Algorithm - Blocking

One of the other factors which we will investigate involves the block sizes themselves. Is there one block size that works in every case? Can variations in the block size affect performance as the dimensions of the matrices increase? We will be looking at block sizes of 2x2, 4x4, 8x8, 16x16, 32x32 and 64x64 sub-blocks to investigate this question.

3.2.4 Loop Unrolling

The final optimization we perform is loop unrolling. This is supported in the development toolchain via a `#pragma` statement. The unroll level is specified as one of 2, 4, 8, 16, 32, or 64. As with all of the other optimization levels, this is implemented in both the SWI and NDRange implementations.

So as to not suffer a combinatorial explosion of experimental configurations, we apply the above levels of optimization cumulatively. As such, level 2 optimizations are all applied on code that has already been optimized at level 1. In addition, the loop unroll factor in level 3 is tied to the blocking factor used in level 2. So if we are performing a blocking size of 2, then we will unroll the computation by the same amount.

Algorithm 4 Traditional For Loop

```
for index = 0 to 1000 do
    purge(index);
    ...
end for
```

Algorithm 5 Unrolled For Loop

```
for index = 0 to 1000 by 2 do
    purge(index);
    purge(index + 1);
end for
```

There is often a trade off, known as the space-time or time-memory trade-off, wherein increasing program size may decrease execution time as we will see in the following example. The normal loop case in Algorithm 4 must make 1000 iterations as compared to only 500 iterations for Algorithm 5. Algorithm 5 can be thought of as using an unroll factor of 1. By selecting an unrolling factor of K, our loop body will be repeated K number of times. This can increase our algorithmic efficiency, reduce loop overhead, and independent statements can be executed in parallel. Given that we using this statement for FPGA execution, we will essentially create additional pipelines in our architecture that can support K operations per cycle.

With a slight modification to the code, we have reduced the number of iterations by 50%. In the case of FPGAs, we are only limited by the available resources in terms of lowering the number of iterations. This notion will become important in our matrix multiplication calculations as we must perform computations using the traditional method. Yet, the question may arise, what is a good choice for the number of loop unrolls? In our investigation, we

based the loop unroll factor on the previous blocking optimization. So if we are performing a blocking size of 2 where we load two elements of A and B respectively into sub-tiles, then we will unroll the computation by the same amount.

Chapter 4

Experimental Results

4.1 Experimental Setup

All experiments are conducted on the Intel HARPv2 system at the Texas Advanced Compute Center (TACC). The HARPv2 system consists of a 14 core (28 thread) Broadwell Class Xeon CPU paired with an Intel Arria 10 GX1150 FPGA. We were given access to this device in 4 hr increments. Consequentially, time became a limiting factor on the length of our experiments. We elected to not use any additional performance features such as OpenCL's Shared Virtual Memory (SVM) or any other architecture specific features. We considered that while such features will increase the performance of our computation, given that we are exploring the design space for a completely new hardware architecture, it was important to get the baseline performance for the most popular techniques without any additional platform specific optimizations.

The programs are coded in OpenCL, conforming to version 2.0 of the specification [17]. A number of commonly used practices for matrix multiplication on multicore devices are applied to dense matrices that range in size from 1024×1024 to 8192×8192. The common advice for FPGA programming recommends writing code in the SWI format, allowing the compiler to identify elements that could be pipelined to take advantage of parallelism on the FPGA [19]. This is in contrast to the approach on GPUs which are naturally well suited to the NDRange methodology [40].

We explore both approaches. For NDRange implementations, the literature encourages users to set up a workgroup size that partitions the workload across processing elements in a

uniform manner [40]. We utilize a method wherein the workgroup sizes are representative of the blocking and loop unrolling sizes. For instance, in kernel L2_B16_U16_ndrange, the kernel divides the matrices into 16×16 blocks and processes 16 elements simultaneously. This configuration will have a workgroup size of 16 by 16 processing elements. Each of the 256 processing elements takes a 16×16 block of the matrix and processes 16 elements each clock cycle. To ensure correctness, each computation performed by the FPGA is compared against the same computation performed on the host processor using the `cblas_sgemm` function of the well-known CBLAS library.

In designing new architectures on the HARP system our only constraint was the area of the FPGA. In selecting algorithms for matrix multiplication, we found that the proper choice is often dependent on not only the size but the sparsity of the matrix. Given that a general matrix multiplication computation may come in any number of sizes or densities, we decided to investigate some of the general methods for optimizing dense matrix multiplications. After reviewing the literature and common practices in industry, we identified many solutions that were highly architecture specific. That is, there were many custom functions and specialized operations that would allow one to take advantage of new features built into emerging devices but we wanted to look at relevant optimizations that work across the myriad of devices in the architectural topology.

4.2 Experimental Levels

Given the many observations and the unique benefits of each optimizations along with the execution methods, how does one pick the right combinations for matrix multiplication or any other complex operation? We decided to use different experimentation levels with increasing optimizations to get a rough topography of the optimization space for the new Intel HARPv2 accelerator. Those optimizations are delineated as follows:

- Level 0: Naïve Implementation - At this level we use standard unoptimized methods for matrix multiplication (i.e., 3 nested loops).
- Level 1: Transposition – At this stage, we will simply transpose one of the source matrices to enable more efficient row-major order access, benefiting spatial locality.

- Level 2: Blocking 2, 4, 8, 16, 32, 64 – Building on the previous optimizations, we operate on 2-dimensional subblocks of the matrices, benefiting temporal locality.
- Level 3: Loop Unrolling 2, 4, 8, 16, 32, 64 – Unrolling the inner loops will allow deeper pipelining in the implementation.

The full set of experiments is as follows (along with their labels):

1. Level 0 – naïve, both NDRange and SWI
   (indicated with L0_ndrange, L0_swi)
2. Level 1 – transpose matrix B, both NDRange and SWI
   (L1_ndrange, L1_swi)
3. Level 2 – blocking size in $\{2, 4, 8, 16, 32, 64\}$, both NDRange and SWI
   (L2_B2_ndrange, L2_B4_ndrange, L2_B8_ndrange, L2_B16_ndrange, L2_B32_ndrange, L2_B64_ndrange, L2_B2_swi, L2_B4_swi, L2_B8_swi, L2_B16_swi, L2_B32_swi, L2_B64_swi)
4. Level 3 – loop unrolling factor in $\{2, 4, 8, 16, 32, 64\}$, both NDRange and SWI
   (L3_B2_U2_ndrange, L3_B4_U4_ndrange, L3_B8_U8_ndrange, L3_B16_U16_ndrange, L3_B32_U32_ndrange, L3_B64_U64_ndrange, L3_B2_U2_swi, L3_B4_U4_swi, L3_B8_U8_swi, L3_B16_U16_swi, L3_B32_U32_swi, L3_B64_U64_swi)

The labels encode the relevant information to identify each experiment, the number after the L indicates the optimization level, the number after the B indicates the block size, and the number after the U indicates the unrolling factor.

4.3 Performance

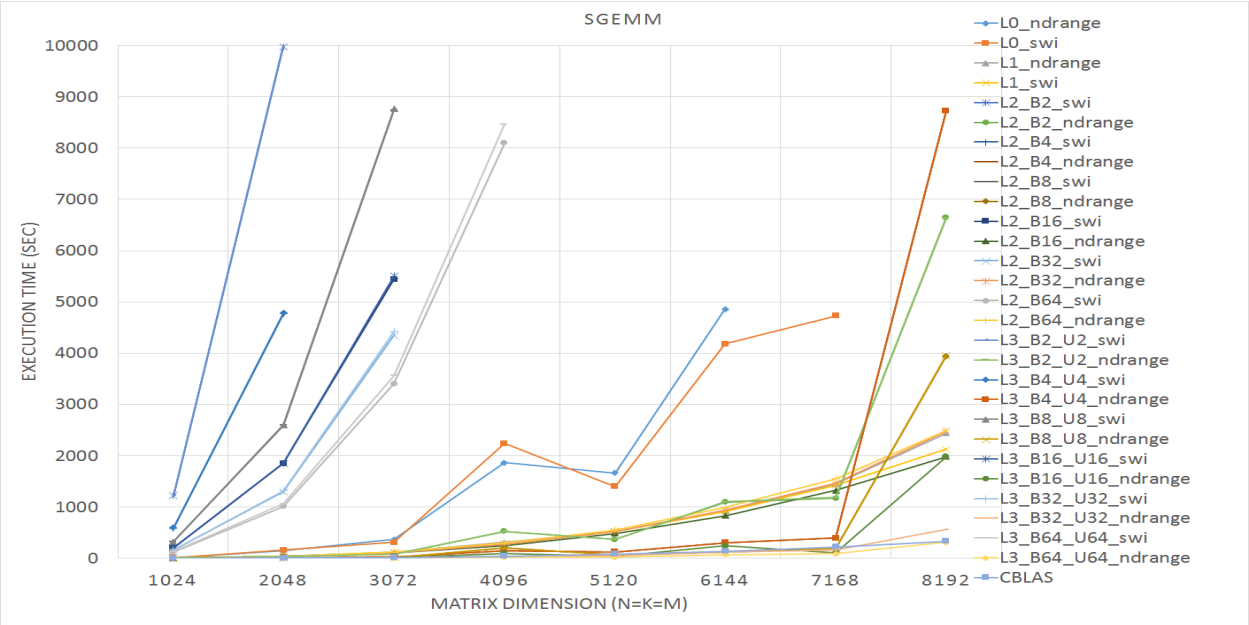

Figure 4.1: Performance Results – Execution Time vs Matrix Size

To provide an appreciation of the breadth of performance results, Figure 4.1 plots the execution time for the matrix multiply operation as a function of matrix size (for square matrices) in every case we consider in this work (including the CBLAS result). At first glance it is clear that there is significant variability among the different kernels. To investigate this variability, we will separately address subsets of the kernels to help us elucidate and characterize this behavior, starting with the unoptimized naïve kernel.

An important thing to note in this plot is that the software-only CBLAS performance is in the highest performing group. This implies that a large number of the kernels do not provide performance that is competitive with well-tuned library code executed on traditional processor cores.

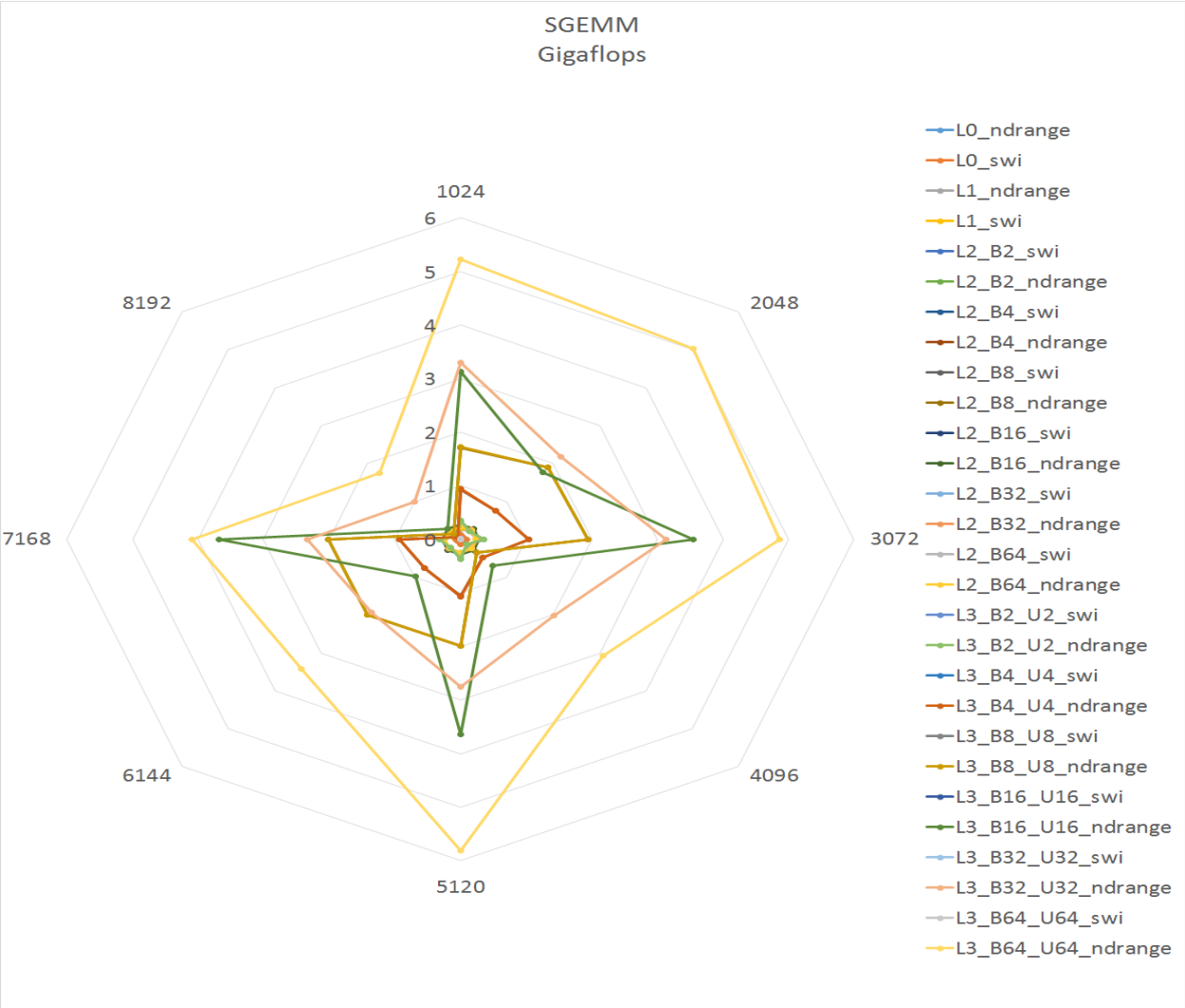

Figure 4.2: Gigaflop Performance - Overall

To help understand the performance of these kernels, we will contrast kernel execution time with an investigation of the kernel gigaflop performance as shown in Figure 4.2. All kernels are depicted in terms of their gigaflop performance on the inner axis with matrix dimensions shown radially on the outer edges.

As we look at the Level 0 kernels for both execution methods, shown in Figure 4.3 and 4.4, we see that for the kernel dimensions that they could complete, they completed their workloads

with only fractions of a Gigaflop and never exceed a 10th of a Gigaflop in performance. This somewhat explains the poor execution time of these kernels.

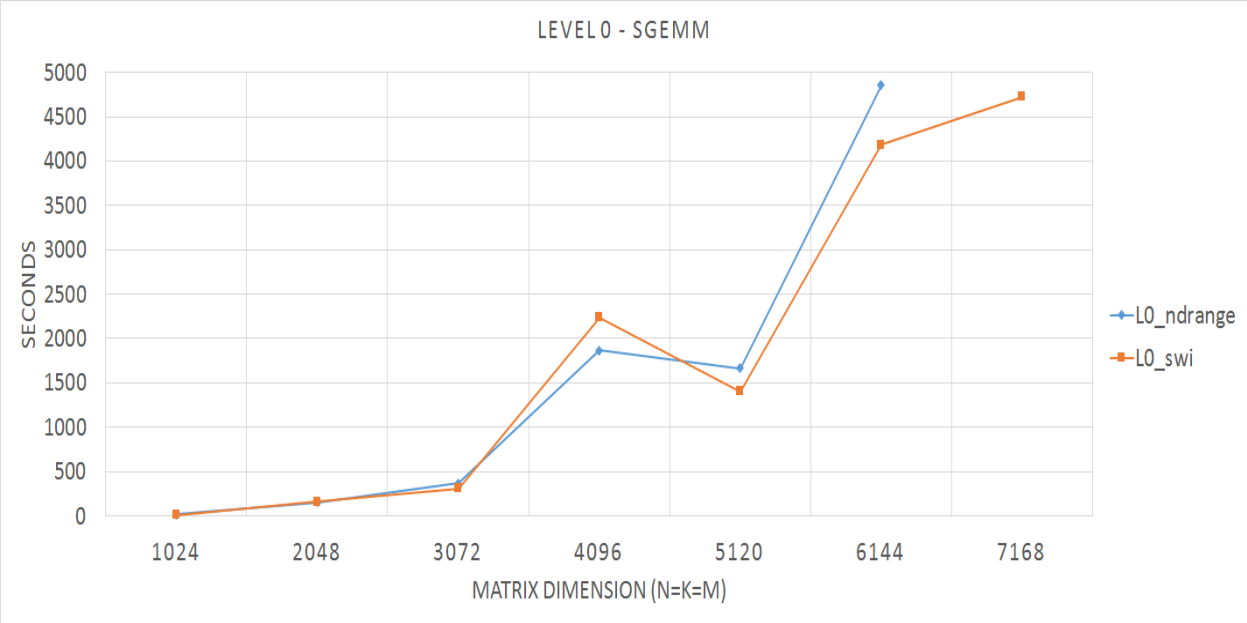

Figure 4.3: Performance results – Level 0 (Naïve) Implementation.

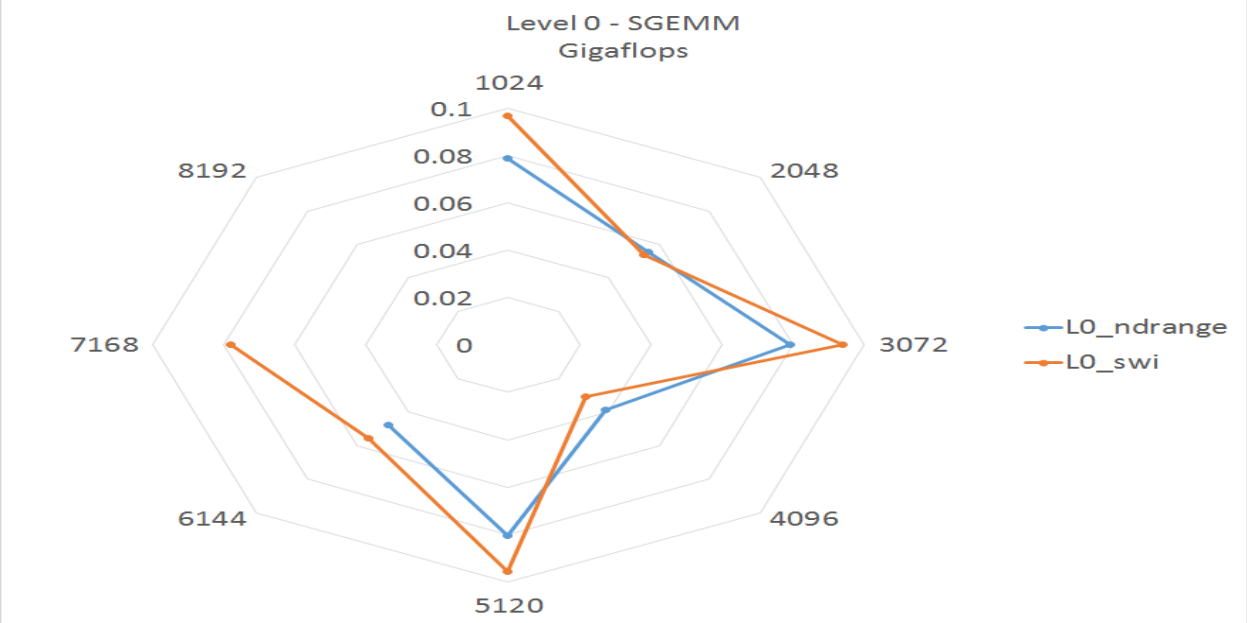

Figure 4.4: Gigaflop Performance - Level 0

The level 0 (naïve) implementation performance results occupy the middle ground of the overall performance graph (kernels L0_ndrange and L0_swi). Not surprisingly, this unoptimized kernel does not provide performance that is competitive with other kernels. For matrices smaller than 4096, the NDRange methods performs slightly better than the SWI kernel implementations. However, for larger matrices we see that the SWI kernel takes the lead in performance. Unfortunately, neither were fast enough to complete the 8192 dimension in the allotted experiment time.

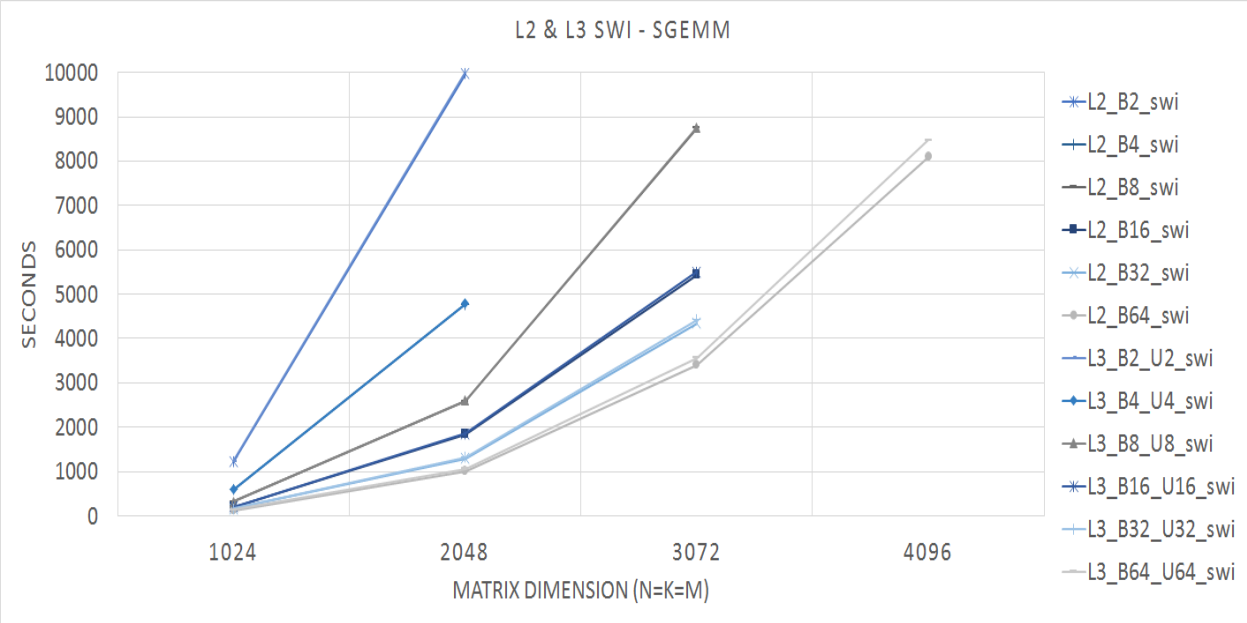

Figure 4.5: Performance Results - Optimized SWI Kernels

We next turn our attention to the SWI kernels shown in Figure 4.5. All but one of this set is bunched in the upper left corner of the initial graph shown in Figure 4.1, indicating that they performed the worst of all those considered. The single exception is the L1_swi kernel which had performance that was competitive with the L1_ndrange kernel. It is worth pointing out here that the SWI approach is the one most recommended for initial implementation by the manufacture's Best Practices Guide [19]. For the highly parallel task of dense matrix multiplication, this approach is clearly not the best one to pursue.

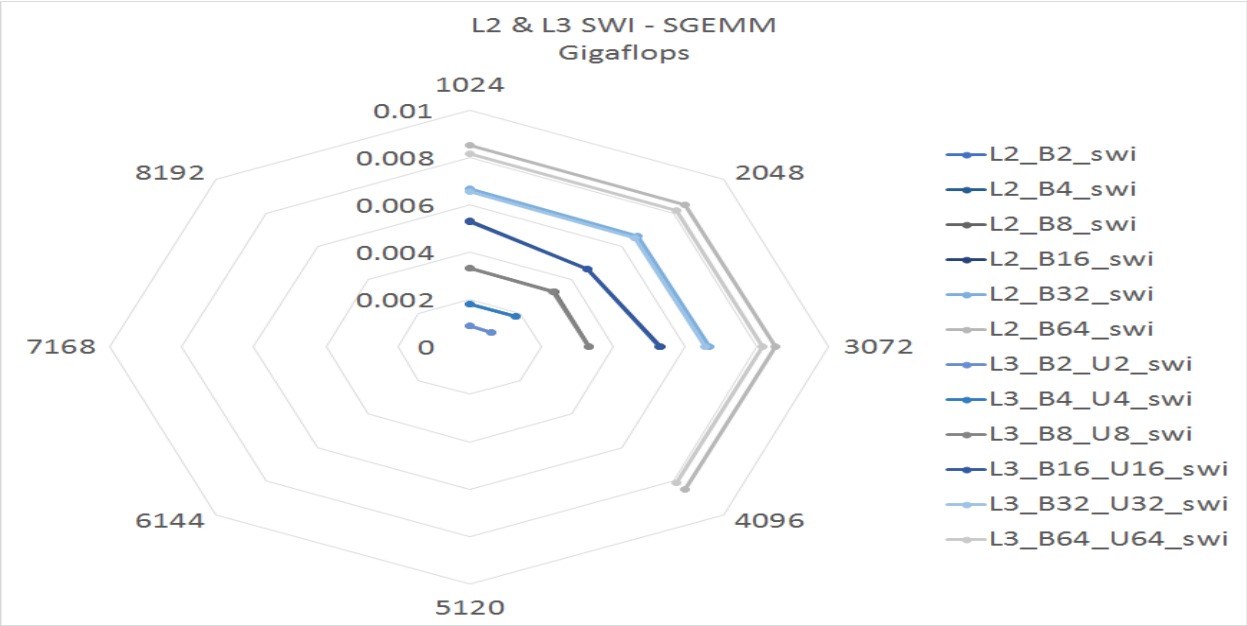

Figure 4.6: Gigaflop Performance - Optimized SWI Kernels

Looking at a subset of the optimized SWI kernel, particularly those with Level 2 or greater optimizations, we see a clear delineation in Gigaflops performance as shown in Figure 4.6. Clearly, kernels with a higher loop unroll factor increased the throughput noticeably. However, overall, the performance for these kernels, in terms of throughput, is less than the un-optimized versions.

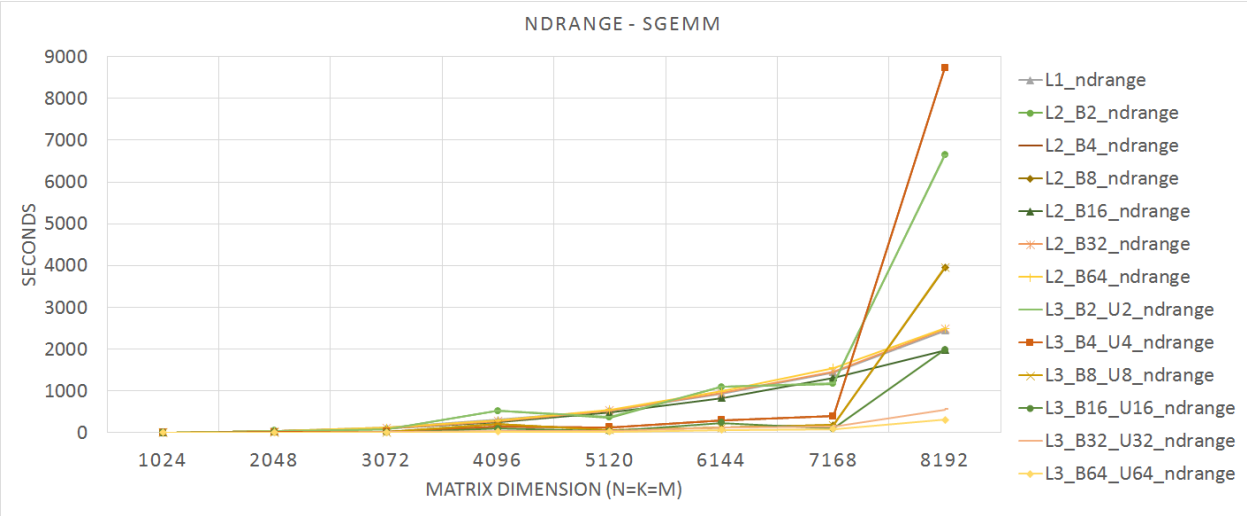

Figure 4.7: Performance Results – Optimized NDRange Kernels

The best performing kernels are the NDRange kernels shown in Figure 4.7. The performance for many of the kernels bifurcates into two comparable groupings for the majority of matrix sizes. Figure 4.9 zooms in on the smaller matrix dimensions (6144×6144 and smaller) and includes only NDRange kernels.

As we look towards the level 2 NDRange kernels, shown in Figure 4.8 we see a great deal of expansion and contraction as we move through the matrix sizes. Because of the differences in performance between the Level 2 and Level 3 optimizations for the NDRanges, we will observe them separately to get a better understanding of their performance profiles.

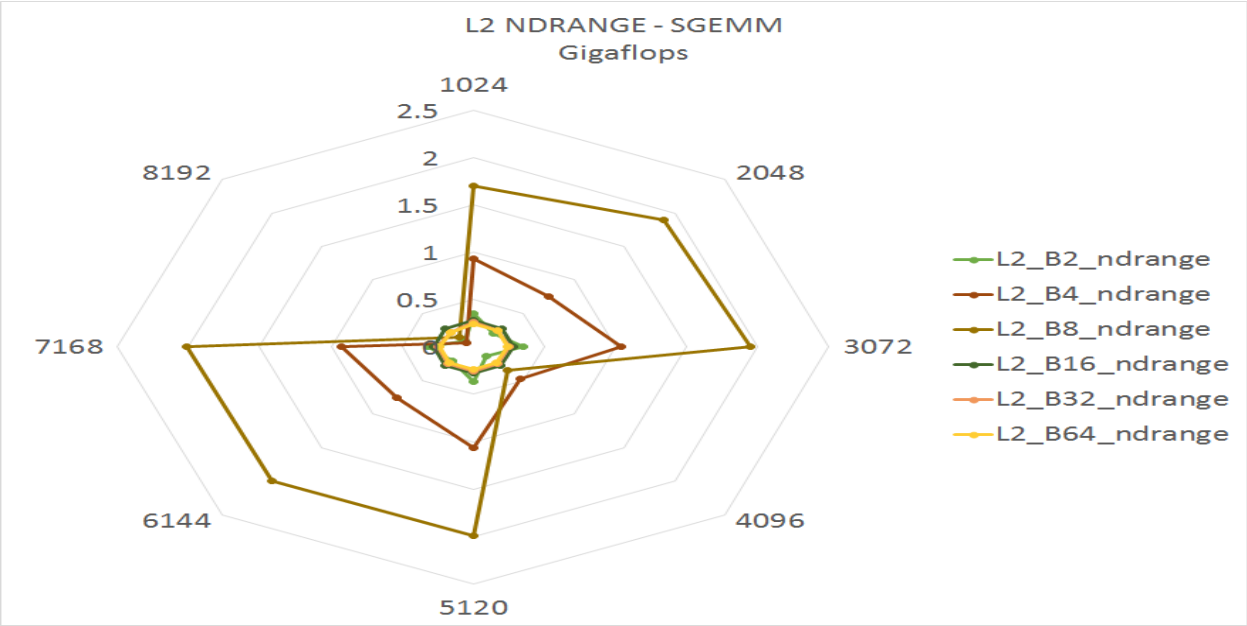

Figure 4.8: Gigaflop Performance – NDRange Level 2

For the Level 2 NDRange kernels, we see some very unexpected behavior. Kernels with a blocking size of 4 and 8 have significantly higher throughput than the rest of the block sizes in this figure. Referring back to Figure 4.9, kernels with blocking sizes of 4 and 8 also experience performance inflections at matrix sizes of 4096 and 8192. Notice that the kernel with a blocking size of 64 has not only a longer execution time but also has a smoother performance profile and a smaller but relatively uniform Gigaflop performance across the matrix sizes.

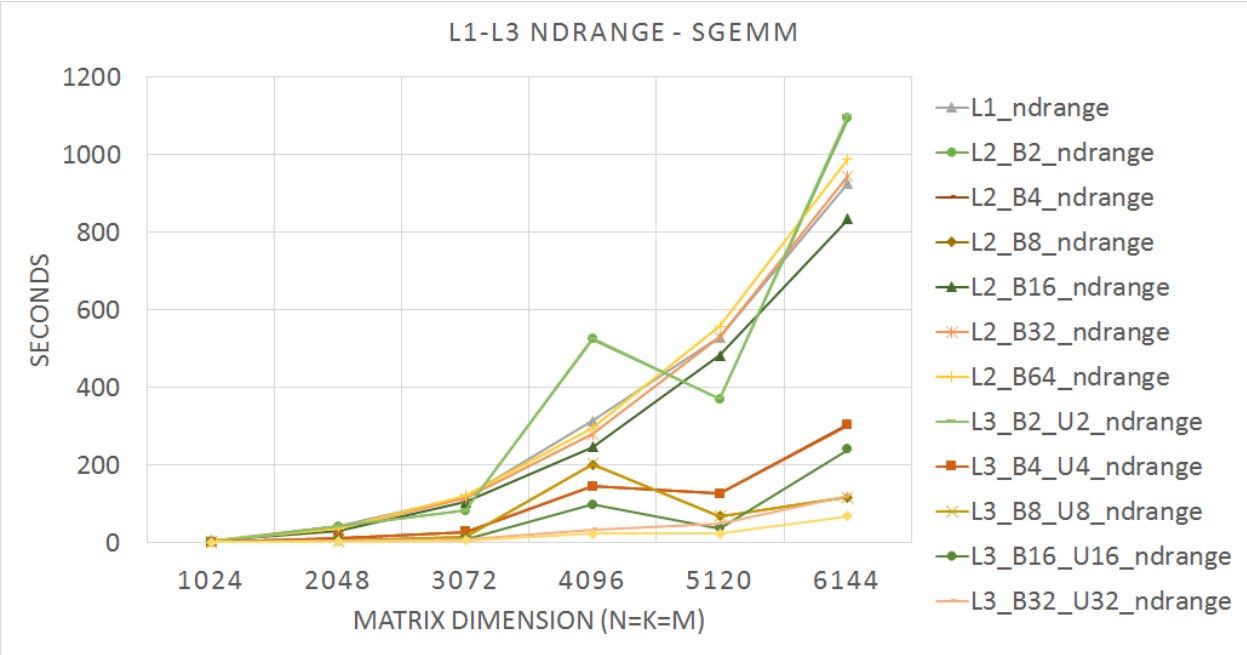

Figure 4.9: Performance Results – NDRange Small Matrices (6144 × 6144 and smaller)

Looking at these kernels, we see that the bifurcation starts almost immediately. All kernels in the upper diverging path, except for L3_B2_U2_ndrange, do not have the level 3 optimization (i.e., loop unrolling). Interestingly, L3_B2_U2_ndrange has the same performance profile as L3_B2_ndrange, and their execution times differ by a fraction of a second. We assume that is because the L3_B2_U2_ndrange kernel has a loop unroll factor of only 2. We conclude that greater loop unrolling is critical for this application.

As we look at NDRange performance between sizes 3072×3072 and 6144×6144, we can see the general trend of bifurcation with the exception of the aforementioned L3_B2_U2_ndrange kernel. Notice that kernels without the loop unroll optimization continue along smooth gradations towards higher execution times but all kernels with the level 3 optimization have a spike in execution time at matrix dimensions of 4096×4096 and 6144×6144 while decreasing for the 5120×5120 dimension. This is an illustration of a pattern that happens frequently, in which we realize large swings in performance for unexpected reasons.

As we move forward to the larger matrix sizes, we have some interesting behavior starting after 7168×7168. Figure 4.10 shows the results zoomed in to these matrix sizes. For kernels

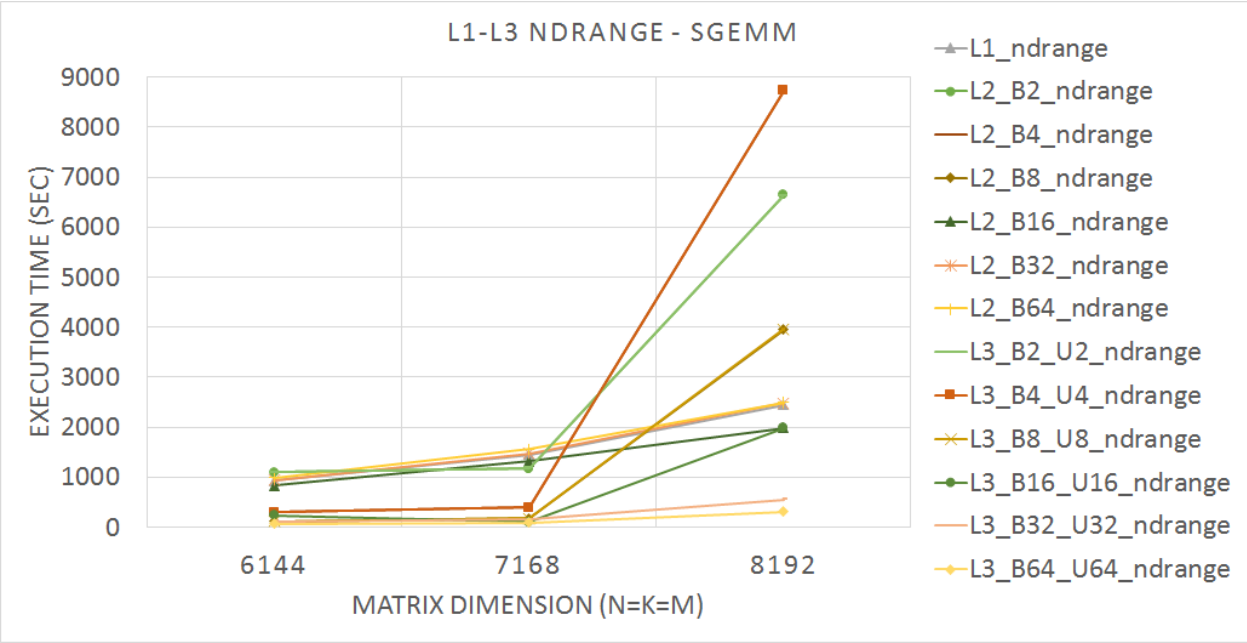

Figure 4.10: Performance Results – NDRange Matrices (larger than 6144 × 6144 inclusive)

with level 3 optimizations, those with a loop unroll factor of 2, 4, 8, and 16, have yet another spike which has a profile that degrades their performance even over kernels with only level 2 (i.e. blocking) optimizations. However, this behavior does not seem to impact level 3 kernels with unrolling factors of 32 or 64. Notice for 8192 × 8192 matrices, the performance of L3_B16_U16_ndrange is comparable to L2_B16_ndrange and we see a similar trend to the one that we saw in the L3_B2_U2_ndrange kernel.

The Level 3 NDRange kernels, as shown in Figure 4.11, take advantage of loop unrolling to increase throughput, and have significantly more performance inflections across matrix dimensions. A point of interest lies in a comparison of the last three kernels in the figure. Particularly, inspecting the throughput of kernels with a loop unroll factor of 16, 32, and 64 respectively, we see that kernels with an unroll factor of 32 and 64 have similar performance profiles as well as a better execution time over the kernel with a loop unroll factor of 16. Those kernels of interest also have a relatively similar shape. Interestingly, the kernel with a loop unroll of 16 has more pronounced throughput inflection points and also experiences performance degradation for matrix size of 8192.

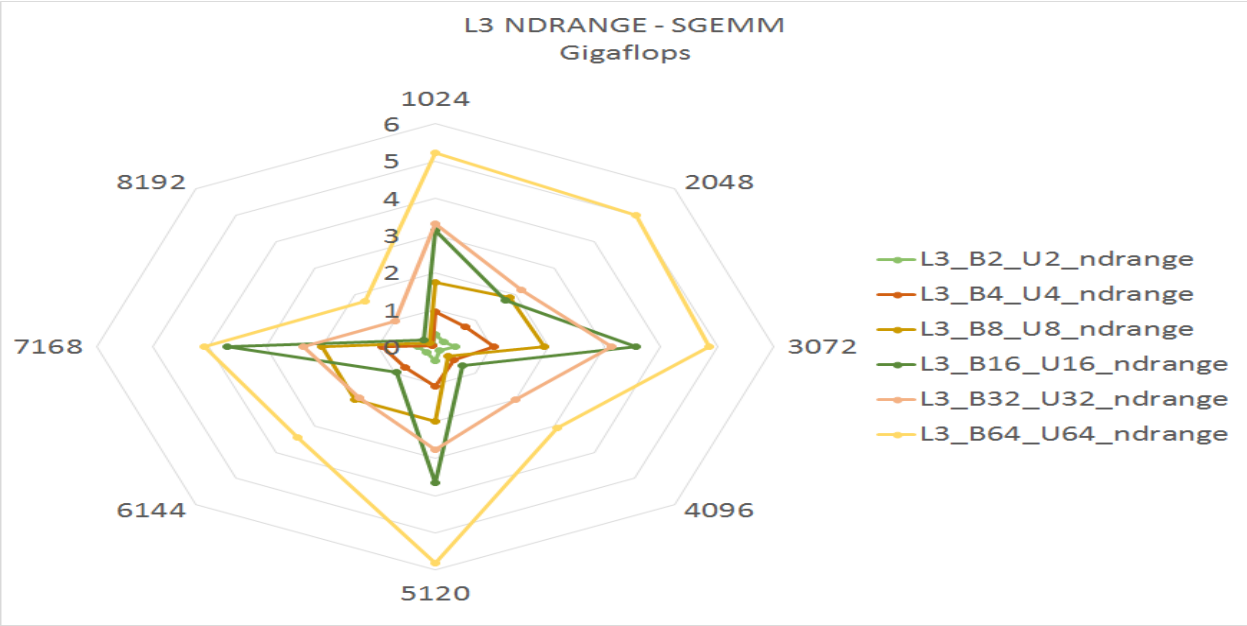

Figure 4.11: Gigaflop Performance – NDRange Level 3

We see some interesting results in terms of execution time and gigaflop throughput. Based on these results, we would like to merge the two metrics to gain some further insight into the heterogeneous architecture behavior.

4.4 Insights

We would now like to review the three highest performing kernels across all of our experiments, comparing them to those conducted on the CPU. In all of our experiments, the highest performing kernel is L3_B64_U64_ndrange. However, we noticed that the runner-up kernels vary based on matrix dimension. In some cases, such as for 8192×8192 matrix dimension, the standard CBLAS computation has a faster execution time than the other optimized kernels.

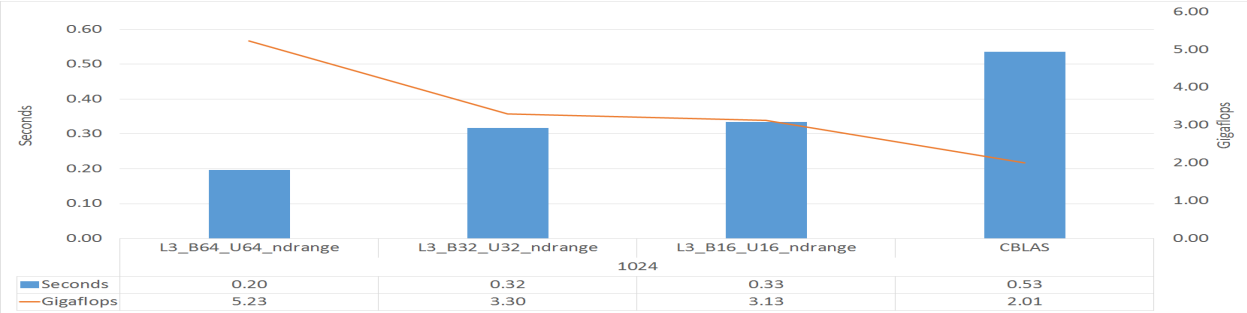

Figure 4.12: Top 3 Highest Performance – 1024 x 1024

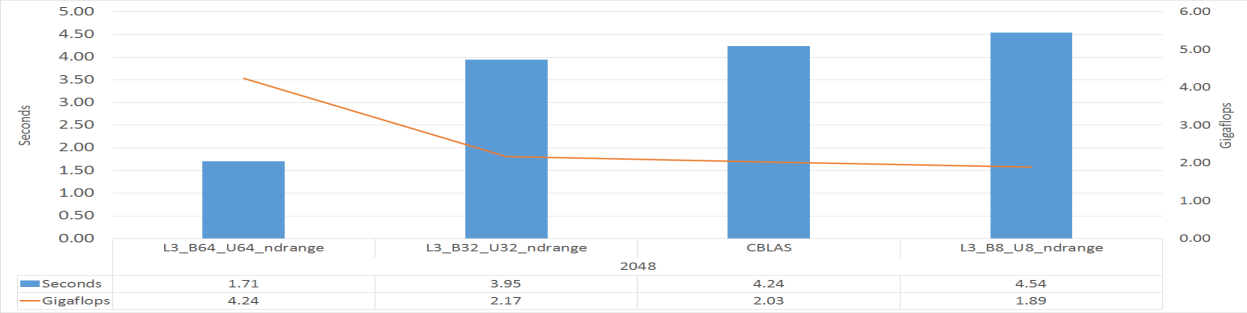

Figure 4.13: Top 3 Highest Performance – 2048 x 2048

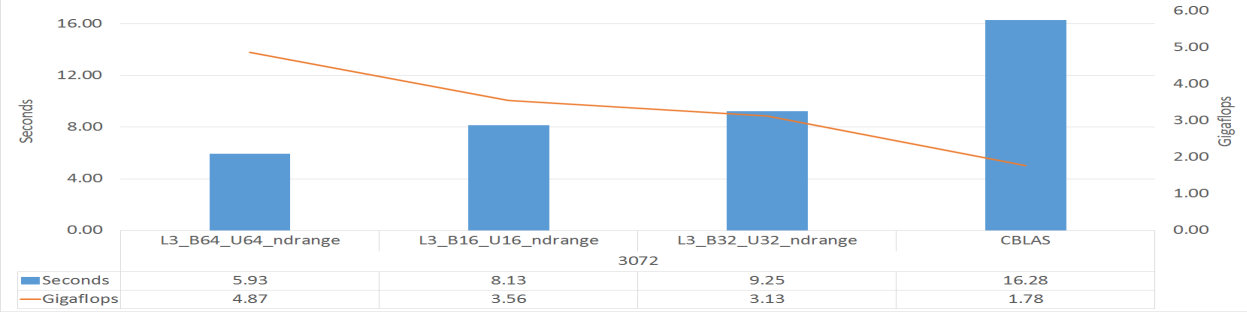

Figure 4.14: Top 3 Highest Performance – 3072 x 3072

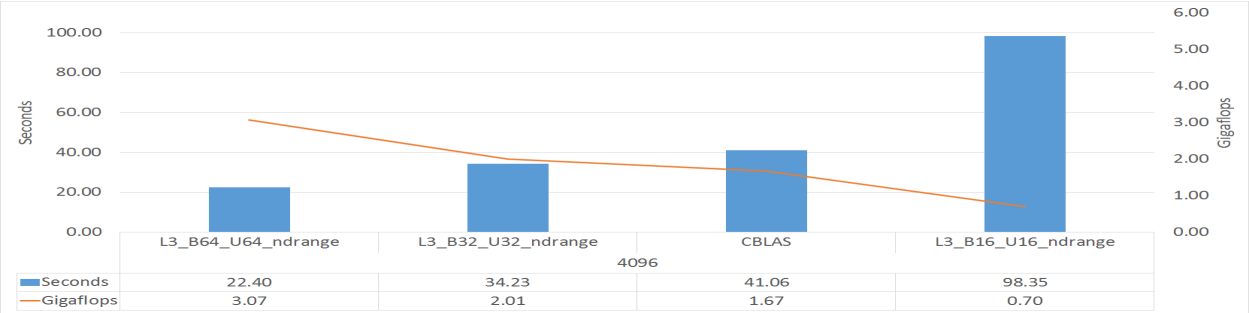

Figure 4.15: Top 3 Highest Performance – 4096 x 4096

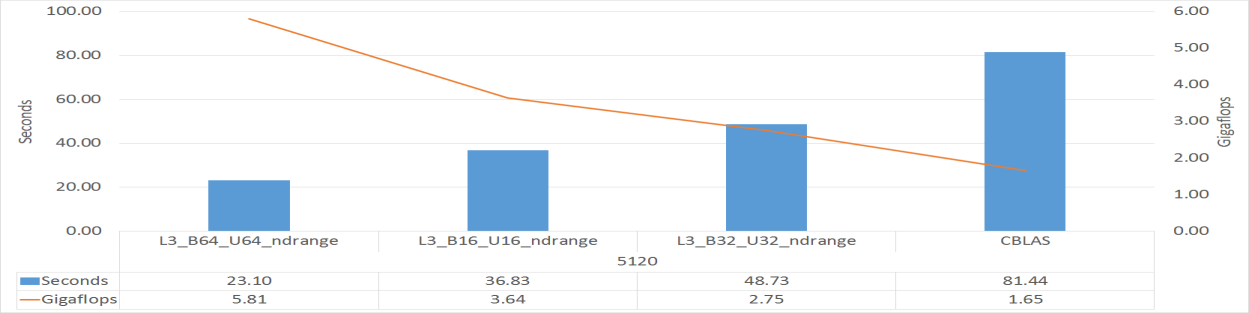

Figure 4.16: Top 3 Highest Performance – 5120 x 5120

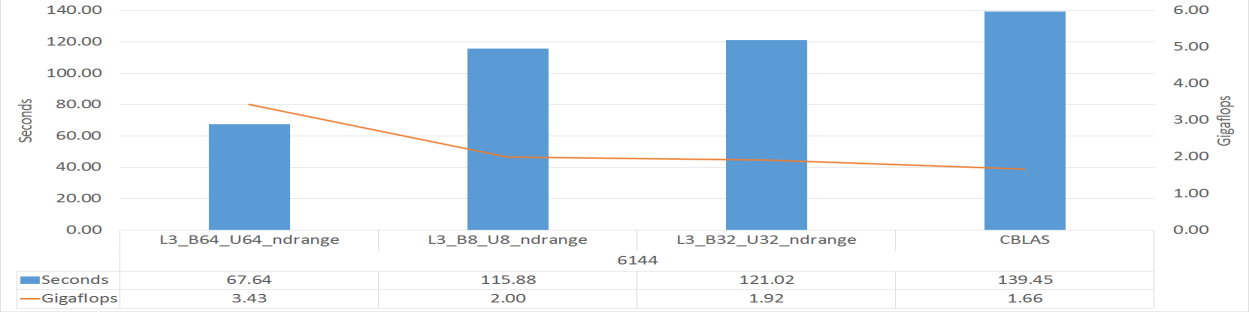

Figure 4.17: Top 3 Highest Performance – 6144 x 6144

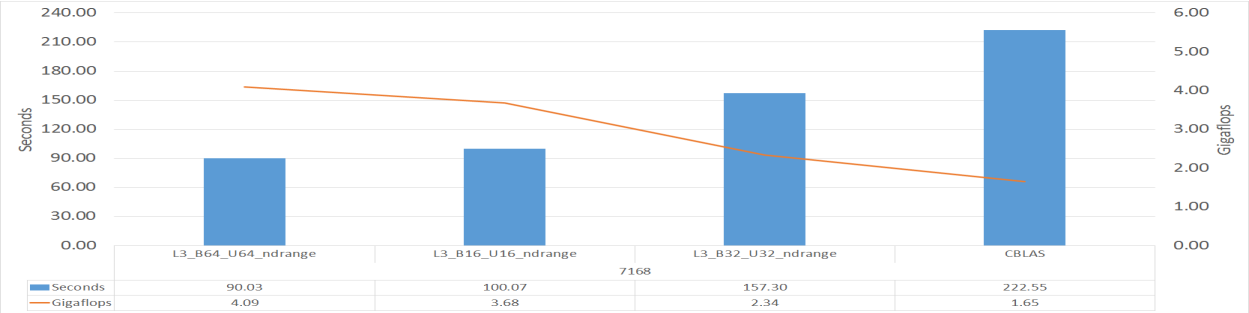

Figure 4.18: Top 3 Highest Performance – 7168 x 7168

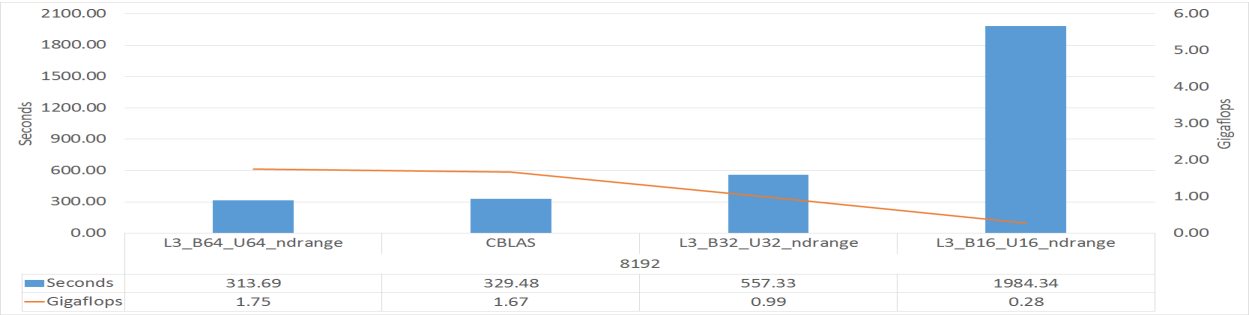

Figure 4.19: Top 3 Highest Performance – 8192 x 8192

While the particular runner-up kernel varies across matrix dimensions, what is consistent throughout is that the full suite of optimizations is needed in order for the FPGA deployment to be competitive with the CPU implementation. The top 3 in every case were level 3 optimizations that include both blocking and loop unrolling of an NDRange kernel.

To better understand this behavior, we plotted the top 4 highest performing kernels by Gigaflops as shown in Figure 4.20.

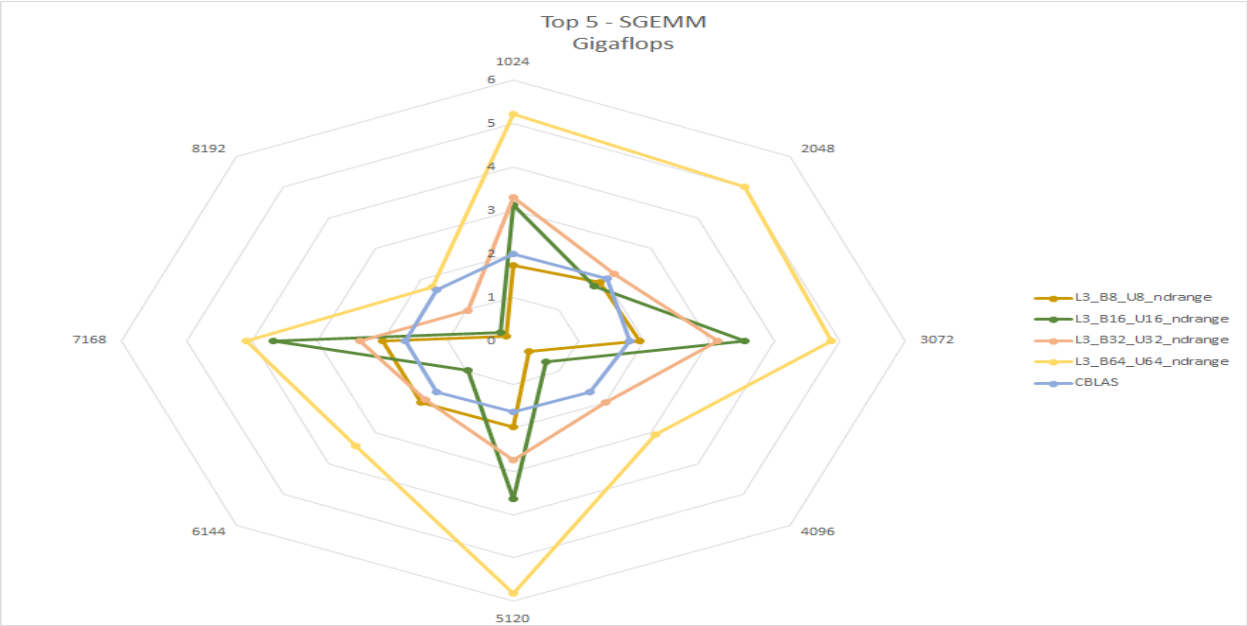

Figure 4.20: Top 5 Highest Performance – Gigaflops

As a final comparison, Figure 4.21 shows the execution time and Figure 4.22 shows the gigaflop performance of the best-performing kernel, L3_B64_U64_ndrange, and the software CBLAS implementation.

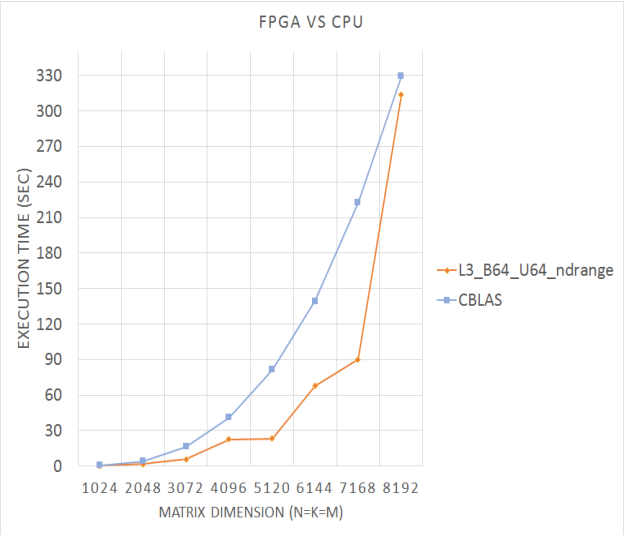

Figure 4.21: FPGA vs CPU Execution Time

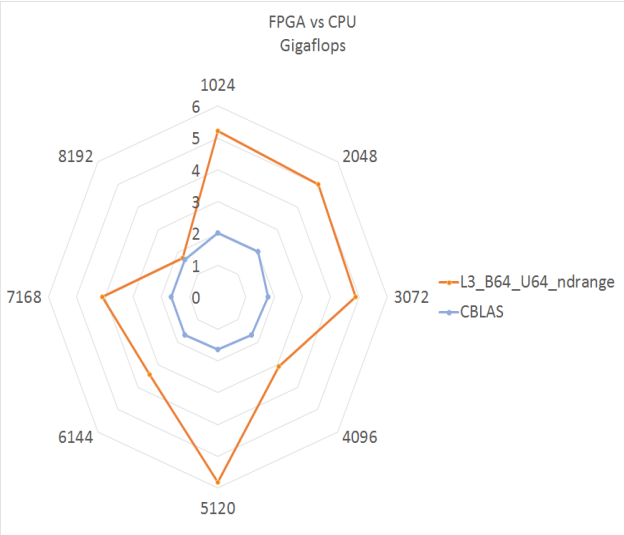

Figure 4.22: FPGA vs CPU Gigaflops

As one would expect, the execution time is growing $O(N^3)$ with the dimension N for both the software and hardware implementations. The software dependence upon matrix size, however, is fairly smooth, while there is considerable variability for the FPGA design. The FPGA design outperforms the software implementation at every matrix size, however, the performance gain is highly variable, ranging from 1.05× to 3.53×. The average performance across dimensions is shown in Figure 4.23. The average speed up is 1.59 with an average execution time of 65.59 on the FPGA and 104.38 on the CPU.

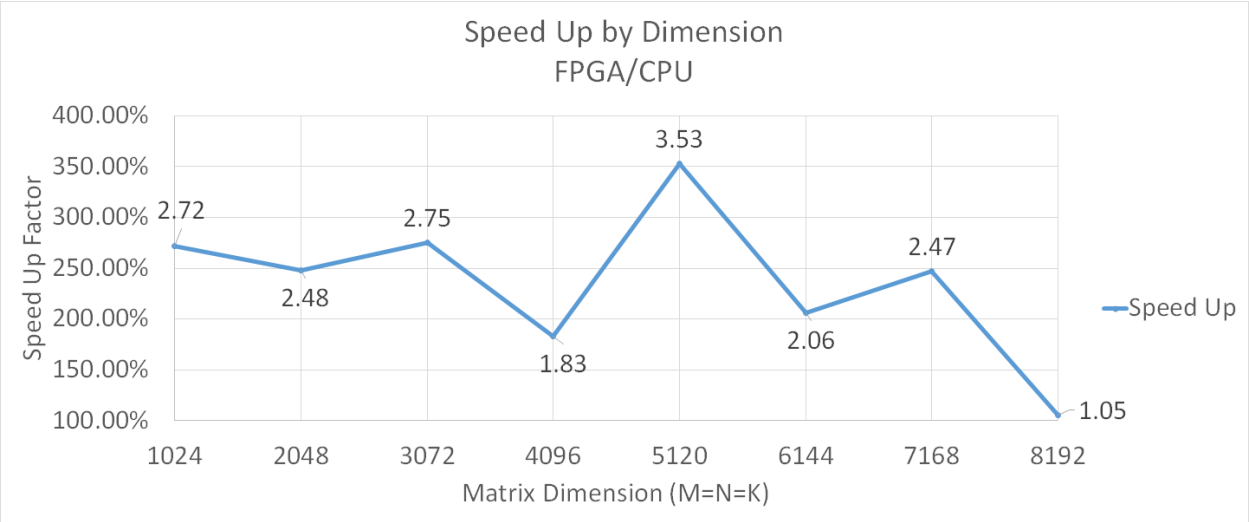

Figure 4.23: FPGA/CPU Speed Up

4.5 Discussion

The wealth of experiments conducted made it challenging to identify the natural trend of the data and what if any optimizations may have been causing performance degradation. To better understand the behavior, we took a look at the shortest overall execution times across all of the experiments which included the CBLAS computations that were performed on the CPU as well. We noticed a very interesting trend.

The SWI implementations, as seen in Figure 4.1, all performed worse than the standard CBLAS computations. The SWI execution model is recommended for FPGA implementations given that its architecture benefits pipelining, but after investigating Algorithm 3,

we conclude that the compiler was unable to determine the exact loop iterations needed to pipeline the for-loop stages, due to our dynamic tiling, and executed many of the for-loops sequentially, leading to considerable serial execution stages. Given that the execution window for an experiment on the HARP system was limited to 4 hours, many of the SWI computations where unable to complete the full range of experiments in the allocated time period. This time constraint allowed us to have a better view of real world scenarios wherein time may be a constraint on the computation. The NDRange implementations performed considerably better than the SWI implementations but as shown in Figure 4.9, they follow a scattered trajectory to each dimension.

The NDRange performance, as shown in Figures 4.7 & 4.9 had both interesting bifurcation patterns and oscillating performance spikes across dimensions. We speculate that this is caused by caches, memory subsystems, or underlying microarchitectural features, in effect hitting size boundaries of the various physical structures involved. A general rule in optimization is to design your algorithms to make optimal usage of architectural features such as cache behavior and memory coalescing [10]. Given the dynamic design of our implementation using HLS it is difficult to determine how to perform this behavior a priori. We should not assume a particular cache size or method to coalesce memory reads. We would argue that our results clearly show performance sensitivity to this class of optimizations. Some block sizes actually degraded performance which we speculate was caused either by imbalanced memory access or inefficient cache usage. We would argue that HLS introduces the need for new design methods that may differ from our assumptions of traditional cache and memory hierarchies.

The charts in Figure 4.12 - 4.19 show the top 3 fastest execution times by dimension. The line running through each graph is the amount of gigaflops achieved for each computation. Surprisingly, our most optimized kernel always finished first out of all of the computations but an interesting phenomenon occurred as we increased the dimensions of the matrices. At first, glance, given that kernel L3_B64_U64_ndrange always finished first, it may be natural to assume that the less optimized kernels would occupy 2nd, 3rd, and 4th place respectively. For the 1024x1024, we saw this behavior for the top 3 kernels. The last place was actually taken by the standard CBLAS computation. As we moved to larger kernels, we encountered situations wherein the less optimized kernel, for instance, in the 5120x5120 dimension, the L3_B16_U16_ndrange kernel outperformed the L3_B32_U32_ndrange kernel. This became more

poignant for the 6144x6144 dimension where the L3_B8_U8_ndrange kernel outperformed the L3_B32_U32_ndrange kernel. When we reached the final matrix dimension of 8192x8192 only our most optimized L3_B64_U64_ndrange kernel made it to the top. All other kernels were outperformed by the standard CBLAS kernel which was very surprising.

There are several things that we can conclude from these experiments:

1. In a system such as the HARP, in which the FPGA is tied in to the cache hierarchy, classic optimizations targeting cache behaviour are beneficial to the FPGA as well as the CPU.

2. In order to take advantage of the accelerator in this environment, all of the optimizations we consider are needed to achieve performance competitive with mature, optimized software.

3. The standard CBLAS library was able to outperform all but one optimized kernel in spite of the fact that these kernels are executing on an FPGA. even though the other kernels had advantages like transposed data, blocking, and loop enrolling enhancements.

4. Many optimized kernels have degraded performance for a range of workloads. Whether or not a particular optimization ends up being performant is not all clear prior to implementation and measurement.

5. In order to optimize emerging accelerators, it is important to consider a spectrum of metrics.

6. These experiments confirm that FPGA performance can exceed CPUs such as the Intel Xeon class processor when coding in OpenCL, but realizing that performance benefit is not necessarily a simple porting exercise.

When we approach performance, are we writing optimal code for the problem or simply cobbling together prebuilt components to suit our needs? With limitations on processor speeds, cost constraints, and power limitations, we have to rethink if we really want to use the tools that are already built for us or do we want to use new accelerators which give us the flexibility to truly design the components to solve the problem.

In order to make the most of accelerators, we are going to have to invest time, energy, and research into understanding our components. We can no longer simply rely on our systems to perform as we anticipate especially when underlying design decisions made in the past may actually be curtailing the performance of future calculations. The report should give pause to really think about what it means to have an optimized implementation not only interms of the execution time but the data types, memory structures, and computational units that such an implementation will execute on.

The performance of the FPGA kernels vaired considerably across both optimization levels as well as matrix dimensions. This is in contrast with CBLAS, giving performance uniform and competitive across matrix dimensions. The experiments reveal the possibility of developing performant kernels for the Intel HARPv2 system that are not only comparable but in some cases higher performing than their CPU counterparts. Yet, the results have shown conclusively that there are many considerations that must be taken into account in order to successfully develop high-performance kernels on reconfigurable hardware.

We have to rethink whether general purpose tools give us the flexibility to truly design, tailor, and reconfigure components to our particular computation. In order to make the most of accelerators, we are going to have to understand not only the algorithms but how they interact with data, workflows, and other cooperative components.

References

[1] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174, April 2009.

[2] O. Beaumont, B. A. Becker, A. DeFlumere, L. Eyraud-Dubois, T. Lambert, and A. Lastovetsky. Recent advances in matrix partitioning for parallel computing on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems*, 30(1):218–229, Jan 2019.

[3] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A CPU and GPU math compiler in Python. In *Proc. of 9th Python in Science Conf.*, pages 3–10, 2010.

[4] Dario Bini and Grazia Lotti. Stability of fast algorithms for matrix multiplication. *Numerische Mathematik*, 36(1):63–72, 1980.

[5] F. Busato, O. Green, N. Bombieri, and D. A. Bader. Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7, Sep. 2018.

[6] Daniel Castaño-Díez, Dominik Moser, Andreas Schoenegger, Sabine Pruggnaller, and Achilleas S Frangakis. Performance evaluation of image processing algorithms on the gpu. *Journal of structural biology*, 164(1):153–160, 2008.

[7] Y. Chen, K. Li, W. Yang, G. Xiao, X. Xie, and T. Li. Performance-aware model for sparse matrix-matrix multiplication on the sunway taihulight supercomputer. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):923–938, April 2019.

[8] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6, 1987.

[9] Daniel Castaño Díez, Hannes Mueller, and Achilleas S Frangakis. Implementation and performance evaluation of reconstruction algorithms on graphics processors. *Journal of Structural Biology*, 157(1):288–295, 2007.

[10] Jack J Dongarra, Iain S Duff, Danny C Sorensen, and Henk A Van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM, Philadelphia, PA, USA, 1998.

[11] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391–407, 2012.

[12] Danijela Efnusheva, Ana Cholakoska, and Aristotel Tentov. A survey of different approaches for overcoming the processor-memory bottleneck. *International Journal of Computer Science and Information Technology*, 9(2):151–163, 2017.

[13] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, Dec 1966.

[14] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with cuda. *IEEE Micro*, 28(4):13–27, July 2008.

[15] Gokul Govindu, L. Zhuo, S. Choi, and V. Prasanna. Analysis of high-performance floating-point arithmetic on FPGAs. In *Proc. of 18th International Parallel and Distributed Processing Symposium*, April 2004.

[16] P. Guo, L. Wang, and P. Chen. A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 25(5):1112–1123, May 2014.

[17] Lee Howes and Aaftab Munshi. The OpenCL specification, version 2.0. *Khronos Group*, 2015.

[18] Steven Huss-Lederman, Elaine M Jacobson, Jeremy R Johnson, Anna Tsao, and Thomas Turnbull. Implementation of strassen's algorithm for matrix multiplication. In *Supercomputing'96: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, pages 32–32. IEEE, 1996.

[19] Intel-Altera. Altera SDK for OpenCL: Best Practices Guide, 2016.

[20] Jian Liang, R. Tessier, and O. Mencer. Floating point unit generation and evaluation for FPGAs. In *Proc. of 11th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 185–194, April 2003.

[21] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. Gpus and the future of parallel computing. *IEEE Micro*, 31(5):7–17, Sep. 2011.

[22] S. Kestur, J. D. Davis, and O. Williams. Blas comparison on fpga, cpu and gpu. In *2010 IEEE Computer Society Annual Symposium on VLSI*, pages 288–293, July 2010.

[23] J. Kim, A. Sukumaran-Rajam, V. Thumma, S. Krishnamoorthy, A. Panyala, L. Pouchet, A. Rountev, and P. Sadayappan. A code generator for high-performance tensor contractions on gpus. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 85–95, Feb 2019.

[24] J. H. Kim, B. Grady, R. Lian, J. Brothers, and J. H. Anderson. FPGA-based CNN inference accelerator synthesized from multi-threaded C software. In *Proc. of 30th IEEE International System-on-Chip Conference (SOCC)*, pages 268–273, September 2017.

[25] E Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 55–55, 2001.

[26] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 451–460, 2010.

[27] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[28] Erik Lindholm, Mark J Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 149–158, 2001.

[29] Richard J Lipton and Kenneth W Regan. David johnson: Galactic algorithms. In *People, Problems, and Proofs*, pages 109–112. Springer, 2013.

[30] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E. Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017.

[31] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In *Proc. of 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, September 2017.

[32] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo. Optimizing the convolution operation to accelerate deep neural networks on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(7):1354–1367, July 2018.

[33] M. Maggioni and T. Berger-Wolf. Coadell: Adaptivity and compression for improving sparse matrix-vector multiplication on gpus. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 933–940, May 2014.

[34] A. C. I. Malossi, Y. Ineichen, C. Bekas, A. Curioni, and E. S. Quintana-Ortí. Performance and energy-aware characterization of the sparse matrix-vector multiplication on multithreaded architectures. In *2014 43rd International Conference on Parallel Processing Workshops*, pages 139–148, Sep. 2014.

[35] Michael D McCool, Kevin Wadleigh, Brent Henderson, and Hsin-Ying Lin. Performance evaluation of gpus using the rapidmind development platform. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pages 181–es, 2006.

[36] A. Morad, L. Yavits, and R. Ginosar. Efficient dense and sparse matrix multiplication on gp-simd. In *2014 24th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, Sep. 2014.

[37] D. J. M. Moss, E. Nurvitadhi, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. W. Leong. High performance binary neural networks on the Xeon+FPGA™ platform. In *Proc. of 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, September 2017.

[38] J. Nickolls and W. J. Dally. The GPU computing era. *IEEE Micro*, 30(2):56–69, March 2010.

[39] Cedric Nugteren. CLBlast: A tuned OpenCL BLAS library. In *Proc. of International Workshop on OpenCL*, pages 1–10, 2018.

[40] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[41] Matt Pharr and Randima Fernando. *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional, 2005.

[42] K. S. Rani, M. Kumari, V. B. Singh, and M. Sharma. Deep learning with big data: An emerging trend. In *Proc. of 19th International Conference on Computational Science and Its Applications (ICCSA)*, pages 93–101, July 2019.

[43] B Ramakrishna Rau and Joseph A Fisher. Instruction-level parallel processing: history, overview, and perspective. In *Instruction-Level Parallelism*, pages 9–50. Springer, 1993.

[44] Daniel A Reed and Jack Dongarra. Exascale computing and big data. *Communications of the ACM*, 58(7):56–68, 2015.

[45] S. Ristov, M. Gusev, and G. Velkoski. Optimal block size for matrix multiplication using blocking. In *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 295–300, May 2014.

[46] ITRS Roadmap. International technology roadmap for semiconductors 2.0 (itrs2. 0). *Semiconductor Industry Association*, 2015.

[47] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. From high-level deep neural models to FPGAs. In *Proc. of 49th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, October 2016.

[48] N. Shirazi, A. Walters, and P. Athanas. Quantitative analysis of floating point arithmetic on FPGA based custom computing machines. In *Proc. of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 155–162, April 1995.

[49] Dilpreet Singh and Chandan K Reddy. A survey on platforms for big data analytics. *Journal of Big Data*, 2(1):8, 2015.

[50] M. Son and K. Lee. Distributed matrix multiplication performance estimator for machine learning jobs in cloud computing. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 638–645, July 2018.

[51] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, May 2010.

[52] M. B. Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *DAC Design Automation Conference 2012*, pages 1131–1136, June 2012.

[53] David B Thomas and Wayne Luk. Multivariate Gaussian random number generation targeting reconfigurable hardware. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 1(2):1–29, 2008.

[54] Keith Underwood. FPGAs vs. CPUs: trends in peak floating-point performance. In *Proc. of ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, pages 171–180, 2004.

[55] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 887–898, 2012.

[56] Ling Zhuo and Viktor K Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on FPGAs. In *Proc. of 18th Int'l Parallel and Distributed Processing Symposium*. IEEE, 2004.

Vita

Steven D. Harris

Degrees	B.A. Mathematics, Computer Science, May 2018 Washington University - Saint Louis, MO M.S. Computer Science, August 2020 Washington University - Saint Louis, MO
Professional Societies	Institute of Electrical and Electronics Engineers
Publications	Li, Z, Harris, S. & Egan, T.M. 2007 P2X receptors. IN: Encyclopedia of Neuroscience. 4th Edition. L.R. Squire, Ed., Elsevier, Boston. Gill, Chris, James Orr, and Steven Harris. "Supporting Graceful Degradation through Elasticity in Mixed-Criticality Federated Scheduling." Proc. 6th Workshop on Mixed Criticality Systems (WMC), RTSS. 2018.

May 2020