CREATING A PORTABLE PROGRAMMING ABSTRACTION FOR WAVEFRONT PATTERNS TARGETING HPC SYSTEMS

by

Robert Searles

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Spring 2019

CREATING A PORTABLE PROGRAMMING ABSTRACTION FOR WAVEFRONT PATTERNS TARGETING HPC SYSTEMS

by

Robert Searles

Approved: ______________________________

Kathleen McCoy, Ph.D.
Chair of the Department of Computer & Information Sciences

Approved: ______________________________

Levi Thompson, Ph.D.
Dean of the College of Engineering

Approved: ______________________________

Douglas J. Doren, Ph.D.
Interim Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ______________________________

Sunita Chandrasekaran, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ______________________________

Michela Taufer, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ______________________________

Rui Zhang, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ______________________________

Wayne Joubert, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____

Oscar Hernandez, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the support of my family, mentors, colleagues, and friends. First and foremost I would like to thank my parents, David and Rosanna Searles, for raising me in a wonderful, loving home and for giving me the best foundation possible. Their love and support set me up for success since the day I was born. I could not ask for better parents, mentors, and role models. I would like to thank my siblings, Paul and Victoria Searles, for being the best and most supportive siblings anyone could ask for. Both of you have been there for me since the day you were born. Without you, I would not be the man I am today. I would also like to thank my partner, Kayla Sampson, for all that she does for me, even when times are stressful. I am excited to start our life and our family together.

I am very fortunate to have had such amazing advisors during my time at UD. I would like to thank my undergraduate advisor, Dr. Terrence Harvey, for his years of support that have gone beyond the scope of my undergraduate education. I would also like to express my undying gratitude for both of my graduate advisors, Dr. John Cavazos and Dr. Sunita Chandrasekaran. As his undergraduate research assistant, Dr. Cavazos's mentorship encouraged me to pursue a graduate degree, and he offered to be my advisor in the process. Whether it meant staying up all night with me to meet paper deadlines, coming out to see me DJ at a local bar on the weekends, or wishing me well when I decided to pursue a different research direction, he was always there to support me. He is loved and missed. Dr. Chandrasekaran's mentorship is something I will never take for granted. She cares about each and every one of her students, and I am proud to be one of them. No matter how busy she gets, she always makes time to communicate with me and give me feedback on my work. She is an incredible role model to me, and I really appreciate all the time and energy she has invested toward my success. I would also like to thank my committee members, Dr. Rui Zhang, Dr. Michela Taufer, Dr. Wayne Joubert, and Dr. Oscar Hernandez for their guidance and mentorship. My research would not be possible without them.

Finally, I would like to thank my labmates, colleagues, and friends. I was fortunate to work with some amazing people throughout my time as a graduate student. I thank you all for your collaboration and support: Marco Alvarez, EJ Park, Sameer Kulkarni, William Killian, Tristan Vanderbruggen, Wei Wang, Lifan Xu, Amir Ashouri, Arnov Sinha, Kshitij Srivastava, Jose Monsalve Diaz, Sanhu Li, Eric Wright, Mauricio Ferrato, Dylan Chapp, and Michael Wyatt. I would also like to thank my friends and roommates. Whether you helped me put a roof over my head, shared laughs with me while playing video games to decompress, talked to me on the phone when I was stressed, or simply spent time with me on my nights off, know that you also played an integral role in my success: Matthew Senseny, Joseph DiGregorio, Mark Rushmere, Jacob Gaines, Stephen Herbein, Michael Bruno, Dylan Ross, Brandon Morris, Victoria Black, Logan Mears, Keith and Paige Parlier, Charlie Pens, Adam Petruzziello, Gerald Clericuzio, Michael Gallo, Aaron Yalcin, Arman Yalcin, Scott Ehrmann, James Murray, Adam Wisniewski, Emily Corbett, Jon Wentworth, David Blaha, Jesse Melvin, and Stephen Sprofera. Thank you all from the bottom of my heart.

This material is based upon work supported by the National Science Foundation (NSF) under grant no. 1814609.

This material is based upon work supported by the U.S. Department of Energy, Office of science, and this research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

I would also like to thank NVIDIA Corporation for donating the TITAN Xp, TITAN V, Tesla P40, and Tesla V100 GPUs used to prototype code for this research and for providing access to their Professional Service Group (PSG) machine that I used for this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Processor architectures have been rapidly evolving for decades. From the introduction of the first multicore processor by IBM in 2001 [2] to the massively parallel supercomputers of today, the exploitation of parallelism has become increasingly important, as the clock rates of a single core have plateaued. Heterogeneity is also on the rise since the revelation that domain-specific pieces of hardware (GPUs) could be repurposed for generalized parallel computation [10]. This shift has prompted the need to rethink algorithms, languages, and programming models in order to increase parallelism from a programming standpoint and migrate large scale applications to today's massively powerful platforms. This is not a trivial task, as these architectures and systems are still undergoing constant evolution. More recently, supercomputing centers are transitioning toward utilizing fat-nodes (nodes with even more cores due to the presence of multiple accelerators) in order to reduce node count and the overhead associated with cross-node communication. For example, Oak Ridge National Laboratory's TITAN supercomputer (OLCF-3), which was built in 2011, was comprised of 18,688 nodes, each containing a single NVIDIA Tesla K20x GPU accelerator [60]. In 2018, less than a decade later, ORNL constructed the Summit supercomputer (OLCF-4), consisting of 4,608 nodes, each equipped with six NVIDIA Tesla V100 GPU accelerators [67].

The trend toward fat-node based systems illustrates the importance of on-node programming models. Low-level languages like CUDA and OpenCL offer direct control over GPU hardware, but they incur a learning curve and lack portability, which are concerns for application developers. It is a huge time sink to have to learn a hardware-specific low-level language, port your code using that language, and then reimplement that same code when a newer GPU (or non-GPU) architecture emerges.

The demand for portable solutions for programming parallel systems with minimal programmer overhead lead to the creation of directive-based programming. Directive-based programming models, such as OpenMP [106] and OpenACC [40, 24, 105], allow programmers to simply annotate their existing code with statements that describe the parallelism found within that code. A compiler then translates this into code that can run on a specified target architecture. This type of programming approach has become increasingly popular amongst industry scientists [9]. Although directive-based programming models allow programmers to worry less about programming and more about science, expressing complex parallel patterns in these models can be a daunting task, especially when the goal is to achieve the theoretical maximum performance that today's hardware platforms are ready to offer. One such parallel pattern commonly found in scientific applications is called wavefront.

This thesis examines existing state-of-the-art wavefront applications and parallelization strategies, which it uses to create a high-level abstraction of wavefront parallelism and a programming language extension that facilitates an easy adaptation of such applications in order to expose parallelism on existing and future HPC systems. This thesis presents an open-source tool called Wavebench, which uses wavefront algorithms commonly found in real-world scientific applications to model the performance impact of wavefront parallelism on HPC systems. This thesis also uses the insights gained during the creation of this tool to apply the developed, high-level abstraction to a real-world case study application called Minisweep: a mini-application representative of the main computational kernel in Oak Ridge National Laboratory's Denovo radiation transport code used for nuclear reactor modeling. The OpenACC implementation of this abstraction running on NVIDIA's Volta GPU (present in ORNL's *Summit* supercomputer) boasts an 85.06x speedup over serial code, which is larger than CUDAs 83.72x speedup over the same serial implementation. This serves as a proof of concept for the viability of the solutions presented by this thesis.

Chapter 1
MOTIVATION AND PROBLEM STATEMENT

1.1 Thesis Objective

The objective of this thesis is to explore and develop a high-level programming abstraction for a complex parallel pattern called wavefront that is commonly found in real-world scientific applications.

1.2 Contributions

1. Design and create a standardized high-level programming abstraction (language extension) to be effectively mapped onto heterogeneous parallel architectures for the wavefront parallel pattern

2. Evaluate this abstraction on multiple state-of-the-art platforms using a tool we created to model the performance impact of the developed abstraction

3. Apply this abstraction to a real-world scientific application of broad interest

1.3 Organization

This thesis is organized as follows. Chapter 2 will provide background information on the state-of-the-art high-level programming models, motivate the need for extensions, and introduce the wavefront parallel pattern we will examine in order to meet our goals. Chapter 3 will outline the related work in regards to the wavefront pattern in the scope of high performance computing. Chapter 4 will outline the author's HPC background and past work that led up to this thesis. Chapter 5 will define the wavefront parallel pattern, as well as outline the goals of this thesis in regards to extending existing programming models in order to support this pattern. Chapter 6 will lay out the strategy the author used to accomplish these goals, and it will also present a preprocessor that applies appropriate code transformations, which serves as a proof of concept for the discussed strategy. Chapter 7 presents a tool the author developed to help model the performance impact of wavefront parallelism using wavefront algorithms found in many real-world scientific applications. Chapter 8 shows the author's methodology at work as the developed abstraction is applied to a real-world scientific application that is in use today as part of an Exascale project and presents associated results. Finally, this thesis concludes in Chapter 9.

Chapter 2
INTRODUCTION

Hardware architectures are rapidly evolving. High performance computing nodes are becoming increasingly heterogeneous, including the current and anticipated exascale accelerated node architectures [14], which are expected to contain a mix of throughput and latency optimized cores [102]. Since throughput cores will be more numerous on these machines, an application can achieve high performance if its algorithm exhibits a greater degree of parallelism, thus keeping all the cores on the chip busy. Latency optimized cores, lesser in count, will prioritize serial performance and deal with expensive memory accesses. Such a balanced mixture of cores is expected to manage different types of parallelism available in an algorithm. Memory has advanced as well. 3D memory stacking with memory moving onto the socket provide increased bandwidth and faster communication.

2.1 On-Node High-Level Programming Models

Such diverse architectures require their own code optimization strategies. Application developers prefer a "write-once" code development strategy in which a single piece of code will execute efficiently and portably on all targeted architectures. However, it is required that a programming model not be designed with just the underlying hardware in mind. Its implementation is also expected to address requirements of applications and its algorithms. The programming language that implements the model should provide the right abstractions to improve the productivity of scientific developers. Programmers often resort to a trade-off between achieving portability and high performance. Why? The issue is two-fold. Adequate application parallelism will not be exposed on a particular hardware architecture if the algorithm is structured in a way that limits the level of concurrency that a programming model can exploit. Secondly, a performance-portable single code representation is only possible if the programming abstractions are carefully crafted in a way that allows the programming models to provide informative hints to the compilers in order to generate optimized code across platforms.

Some of the most widely used types of on-node programming models that provide developers with such abstractions are based on directives: annotations added to existing serial code used to instruct the compiler as to how the code is meant to run on parallel architectures. Directives allow us to abstract the rich feature set of hardware architectures, incrementally improve, port, and maintain the codebase across platforms. Two of the most widely used directive-based programming models are OpenMP and OpenACC. OpenMP is a multi-platform, shared memory multiprocessing API that has been around since 1997 [34]. In 2015, it started supporting offloading features, such as the ability to target multiple devices within a system [106]. OpenMP 4.5 is being deployed to applications, e.g., Pseudo-Spectral Direct Numerical Simulation-Combined Compact Difference (PSDNS-CCD3D) [32], a computational fluid dynamics code on turbulent flow simulation using GPUs on the ORNL Titan system. OpenACC started in 2012 as a directive-based model for general purpose GPU (GPGPU) programming, and it has since been generalized to support many kinds of heterogeneous systems, including multi-core CPUs, GPUs, and FPGAs [40, 24]. Since its inception, it has been widely used to port large-scale applications spanning several domains such as ANSYS [95], GAUSSIAN [43], and Icosahedral non-hydrostatic (ICON) [96] to massively parallel architectures.

2.2 Essential Features of High-Level Programming Models

Directive-based models, along with other types of high-level programming models, aim to provide developers with four key features: parallelism, performance, programmability, and portability.

2.2.1 Parallelism

The stagnation of processor clock rates due to power consumption and cooling challenges has ended the era of sequential computing. Instead, industry has shifted its focus to parallel architectures as a way to increase on-chip performance. At the node level, multi-core processors have become ubiquitous. In addition, enterprise-grade systems utilize multiple nodes connected via some type of network in order to provide an additional layer of parallelism and computational power. These “supercomputers” are representative of the way in which hardware trends are evolving. As a result, parallelism is the foundation of any modern programming model.

2.2.2 Performance

There are many different types of parallel architectures, such as multi-core CPUs, Graphics Processing Units (GPUs), and Field-Programmable Gate Arrays (FPGAs), all of which benefit from different types of optimizations. A high-level programming model should aim to provide a way for developers to tune their applications to be performant on the architecture they intend to run their application on. Ideally, this type of architecture-specific optimization can be handled automatically by the compiler, requiring only that the programmer specify the target architecture at compile time. This ensures that the code will perform as optimally as manually written, low-level code specific to that architecture [46].

2.2.3 Programmability

As mentioned in Section 2.2.2, there are many different types of parallel architectures. In addition, heterogeneous hardware (hardware that combines elements of multiple architectures) is becoming increasingly common. Multi-node systems add an additional layer of complexity to the equation. Programming one node is hard. Programming many nodes is harder. Programming models must provide developers with a concise way of representing how their code should run on a parallel system, so that they only need to write their application once. That code should be generalized enough that it can be compiled into machine code that will run on any supported parallel architecture, including heterogeneous ones. These models should also provide a method for distributing computational tasks within an application across multiple nodes, as multi-node systems are becoming more and more common.

2.2.4 Portability

Finally, a programming model should aim to be portable. It is not enough to simply represent code in a way that is conducive to compilation on existing architectures. Architectures are constantly evolving, and we cannot predict what future systems will look like. For example, ORNL's Titan supercomputer is a GPU-based, multi-node machine that presented a number of programming challenges when it was first built [60]. ORNL's newest machine, Summit [67], adds another layer of complexity in that each node will be equipped with 6 NVIDIA Volta GPUs, unlike Titan, which only contained one GPU per node. High-level programming models must be robust enough to handle these types of evolutionary challenges. An inability to due so motivates the need for changes and/or extensions in order to keep up with future hardware trends.

2.3 Directive Based Programming Model: OpenACC

The on-node programming model this thesis aims to extend is OpenACC. OpenACC is a performance-portable, directive-based parallel programming model that targets modern heterogeneous HPC hardware [105]. The compiler directives that it provides can be used to annotate loops and regions of code that exhibit parallelism. The programmer can also specify a target at compile time to let the compiler know what type of hardware is being targeted based on the annotated code regions. Supported hardware includes multicore CPUs, as well as various types of accelerators, including Intel's Xeon Phi and many generations of NVIDIA's GPUs.

Another advantage of OpenACC is that when targeting accelerators, programmers do not have to explicitly manage data movement between the host (CPU) and accelerator(s), nor do they have to manually initiate the startup and shutdown of the accelerator(s). Instead, this is handled by the OpenACC compiler and runtime environment. These features dramatically reduce the amount of programmer overhead that is necessary, preserve performance, and allow for the creation of a single, portable codebase that will run on many different types of HPC hardware and systems. This section will present key features of OpenACC's execution and memory models that are used throughout this thesis.

2.3.1 OpenACC Execution Model

OpenACC employs a host-directed execution model. What this means is that most of a programmer's application executes within a host thread, and compute intensive parallel regions (denoted by directive annotations) are offloaded either to multiple host cores (multicore CPU) or an accelerator (such as a GPU). The OpenACC specification refers to the host thread as the *host* and the parallel hardware, whether it be a multicore CPU or an accelerator, as the *device*. The job of the device is to execute portions of the code that are annotated with directives. These portions can be *parallel regions*, which are regions of code that contain parallel loops specified by the programmer, *kernels regions*, which contain loops to be executed as kernels that are determined by the compiler to be parallelizable, or *serial regions*, which contain blocks of sequential code.

Since many parallel regions contain multi-dimensional loop nests and modern accelerators and multicore CPUs support multiple levels of parallelism, OpenACC provides clauses the programmer can use within a directive to describe which portions of a loop nest should execute at what level. Accelerators and multicore CPUs both support coarse-grained parallelism: fully parallel execution across execution units with limited synchronization support. OpenACC exposes coarse-grained parallelism via the *gang* clause. Most accelerators and some multicore CPUs also support fine-grained parallelism: multiple execution threads within a single execution unit. OpenACC exposes fine-grained parallelism via the *worker* clause. Last, most accelerators and multicore CPUs support SIMD operations within execution units. OpenACC exposes these operations via the *vector* clause. It is the job of the programmer to understand the difference between these levels of parallelism and know which portions of their code can execute in what manner in order to properly annotate their source code for parallel execution.

Another useful feature of OpenACC is its *async* clause, which can be used to allow the host and device to execute portions of code asynchronously. This is useful in a couple of ways. First, it can be used to allow multiple parallel regions to execute asynchronously on a device (or across multiple devices). This is appropriate when a single parallel compute region is unable to utilize all available computational power a device has to offer or when there is so much work in a single compute region that the programmer decides it would be advantageous to split it up and run portions of that region on different devices, yielding additional parallelism. Second, it can be useful in overcoming some of the overhead associated with data movement between the host and device. By allowing the host thread to continue execution while a parallel region is offloaded to the device, the host thread can begin to transfer data to the device that will be needed for a compute region yet to run, rather than waiting for an unrelated compute region to complete execution before beginning the data transfer.

2.3.2 OpenACC Memory Model

One defining characteristic of a host/device execution model is that device memory may be separate from host memory. For example, GPUs have their own onboard memory, which is separate from the rest of the system's memory. In such a system, the host thread is unable to read or write device memory, since it is not part of the host thread's virtual memory space, and the device is similarly unable to read or write host memory in most cases. Certain devices do provide the ability to read and write host memory, but such operations typically incur a significant performance penalty. Instead, all data movement between host and device memory must be performed by the host thread through explicit system calls.

Fortunately, OpenACC provides directives for managing data movement between the host and device. The compiler takes care of generating the appropriate system calls, but the programmer needs to be aware of what is actually happening on the back end. Typically, data transfer overhead is where most parallel applications that utilize an accelerator suffer performance losses. If data movement is not efficiently managed, these losses can even be so big as to outweigh the computational benefit of using the accelerator, resulting in a slowdown of overall application performance. Device memory also tends to be much smaller than host memory, so there is a limit to how much data can be offloaded onto an accelerator, which in turn prohibits the amount of associated computation that can be offloaded via a single data transfer.

Some types of accelerators (such as GPUs) employ what is known as a weak memory model: they do not support memory coherence across different threads. This can create issues where results cannot be guaranteed for each execution of the same program. For example, if two threads within a parallel compute region write to the same location in memory, results may differ. Compilers usually can detect such cases, but it is still possible to write a compute region where this behavior exists, and thus, produces inaccurate results. Similarly, programmers need to be aware of whether data synchronization between the host and device is necessary. If a compute region offloads data to the device and that same data is read by the host during execution later on, it is important to ensure that the data has been synchronized back on the host-side before the host thread reaches that point in execution. This can become a problem when haphazardly utilizing OpenACC's *async* compute clause. OpenACC provides a number of clauses for use with its *data* directive in order to manage data transfer and synchronization between the host and device.

2.4 Complex Parallel Pattern: Wavefront

Despite the widespread impact that existing high-level programming models have had on HPC in recent years, there still remains a gap in the way that they do not adequately expose and parallelize some of the complex algorithms often found in scientific applications. One such case is a wavefront-based algorithm that is of critical importance to solving scientific problems in multiple science domains. They are useful for problems for which the result values that are computed have dependencies, requiring that results be computed in stages (wavefronts), for which each stage's results depends on results computed in previous stages. The result is a limited amount of parallelism; all the elements of each stage can be computed in parallel as long as their dependencies from the previous stage are satisfied. Some examples of applications that can expose and exploit this type of wavefront parallelism include linear systems solvers, such as the Gauss-Seidel method [61], genome sequencing algorithms, such as Smith-Waterman [104], characterisation of chromosomal rearrangement, such as the Juicer/HiC tool [38, 51], and radiation transport codes [100, 66, 8].

This thesis aims to develop a viable extension to a directive-based programming model, by creating a high-level representation of the wavefront parallel pattern. Although our developed extension uses OpenACC, it can be considered for OpenMP as well without needing to change its functionality. Currently, such an abstraction does not exist, despite the prevalence of wavefront parallelism found in real-world applications. We support our abstraction with a tool we developed, called Wavebench, that is used to model the performance of different types of wavefront applications on hardware architectures found in state-of-the-art HPC systems. Our Wavebench tool features a collection of wavefront algorithms that are representative of the different types of computation found in the aforementioned applications. Throughout, we maintain a single codebase that can be used to target different types of existing HPC systems, ranging from multicore CPU-based systems, to heterogeneous GPU-based systems.

We also present a real-world case study application that is illustrative of the complexities in a wavefront-based algorithm: the Minisweep proxy application [78]. Parallelizing this application using current directive-based APIs will help reveal the gaps in their expressivity and features. We address this issue by designing and envisioning an abstract parallelism model that represents these gaps with a combination of notations. Integrating these notations into a programming language are key to exposing and mapping wavefront-based parallelism across HPC hardware architectures.

Chapter 3
LITERATURE REVIEW/RELATED WORK

This chapter presents work related to the prior research leading up to this thesis, as well as work that helps motivate the goals of this thesis. The discussed topics include auto-tuning code targeting accelerators, accelerating malware detection algorithms, and leveraging heterogeneous hardware in order to solve Big Data problems.

3.1 Performance Optimization and Auto-Tuning of Code Targeting Accelerators

Performance-portable code generation has been a popular field of research for decades. There are a number of library generators that automatically produce high-performance kernels, including FFT [41, 88, 119], BLAS [115, 123, 27, 44, 49], Sparse Numerical Computation [56, 112, 77, 22, 73], and domain specific routines [21, 31, 48]. Other research expands automatic code generation to routines whose performance depends not only on architectural features, but also on input characteristics [74, 75, 50]. These systems are a step toward automatically generating performance-portable code for different architectures. However, these prior works have been largely focused on small domain-specific kernels.

There is also related work on GPU code optimization that looked at a number of manual optimizations of CUDA kernels, including tiling, pre-fetching, and full loop unrolling within CUDA kernels [93]. Some researchers looked at varying thread block dimensions [76]. Others studied loop unrolling extensively as optimizations in CUDA kernels, but they were concerned with improving the performance of a single application [20]. A domain-specific auto-tuning framework for sparse matrix-vector multiplication on the GPU exists [30], and 3D-FFT of varying transform sizes on GPUs has been optimized using auto-tuning [82]. These two related works applied auto-tuning to specific applications, but not in a general sense. Another related work is the CUDA-CHiLL project, which can translate loop nests to high performance CUDA code [92]. The developers of this project provide a programming language interface that uses an embedded scripting language to express transformations or recipes. While this work supports CUDA transformations, it relies on the creation of an external script, which is not as convenient as using a directive-based approach like OpenMP or OpenACC.

The work presented here provides additional motivation behind the notion that code transformations can provide huge performance benefits to applications. However, these code transformations prove inconvient to implement most of the time. In order for high-level programming models to grow in popularity, they have to begin to tackle some of this more complex behavior without creating considerable additional programmer overhead.

3.2 Malware Detection Strategies

In this section, we examine previous works on malware detection strategies that make use of various compiler representations and techniques (particularly graphs), and graph kernel parallelization. The work presented here proves that while these strategies are viable, they are very computationally intensive. Utilizing accelerators can help overcome the computational overhead created by these strategies, but programming these accelerators proves to be challenging.

3.2.1 Graph-Based Malware Detection

Solving the problem of malware detection by examining structural and behavior qualities of applications can be seen as a compiler techniques issue. In most cases, source code is not available, so some type of decompiler is required. Additionally, compiler representations and techniques can be used to analyze an applications decompiled code. For example, DroidMiner uses static analysis to automatically mine malicious program logic from known Android malware [122]. Behavior graphs are constructed from malware in DroidMiner, and these graphs are flattened into feature vectors that are then fed into several machine learning classifiers including naive Bayes, SVM, decision trees, and random forests for malware detection. The best algorithm of DroidMiner can achieve a 95.3% detection rate on a dataset of 2466 malware. It can also reach 92% for classifying malware into its proper family.

Researchers at Los Alamos National Laboratory and the University of Technology in Iraq proposed algorithms for malware detection that make use of graph-based representations of instruction traces of binaries [13, 11]. Each graph represents a Markov Chain, where the vertices represent instructions. They use a combination of different graph kernels to construct a similarity matrix between these graphs. They then feed this resulting similarity matrix to an SVM to perform classification. These papers does not address the possibility of optimizing or parallelizing these algorithms, which exceeds $O(n^2)$.

Researchers at the University of Gttingen proposed a method for malware detection based on efficient embedding of Function Call Graphs (FCG), which are high level characteristics of the applications [42]. They extracted function call graphs using the Androguard framework [36]. The nodes in the graph were labeled according to the type of instructions contained in their respective functions. A neighborhood hash graph kernel was applied to evaluate the count of identical substructures in two graphs. Finally, an SVM algorithm was used for classification. In an evaluation of 12,158 malware samples, the proposed method detected 89% of the malware. The work discussed in Section 4.2 presents a similar framework that achieves higher accuracy and yields better performance due to a parallel implementation of the graph kernel used to construct the similarity matrix that is fed into the SVM [98], thus highlighting the importance of utilizing available accelerators.

3.2.2 Graph Kernel Parallelization

There is a limited amount of research available that focuses on parallel implementations of graph kernels. Researchers at Stanford developed a method for implementing a parallel version of breadth-first search (BFS), and they present results on both multicore CPU and GPU [55]. They also present a hybrid method which dynamically chooses which of their implementations will yield the best performance during each BFS iteration. Although the kernel itself is different, this work shows a viable hybrid parallel implementation of a graph traversal algorithm which scales well when operating on large graphs. As discussed in Section 4.2, the hybrid implementation of SPGK is similar in nature [120], and it can be leveraged for the purpose of malware detection [98].

3.3 Leveraging Heterogeneous HPC Hardware for Big Data Applications

Many applications take advantage of heterogeneous hardware using an approach known as MPI+X that leverages MPI for communication and an accelerator language (e.g., CUDA and OpenCL) or directive-based language (e.g., OpenMP and OpenACC) for computation. Codes that utilize MPI+OpenACC include: the electromagnetics code NekCEM [84], the Community Atmosphere Model - Spectral Element (CAM-SE) [81], and the combustion code S3D [71]. Codes that utilize MPI+OpenMP include computational fluid dynamics MFIX [109], Second-order Mller-Plesset perturbation theory (MP2) [62], and Molecular Dynamics [65].

Several prototype MapReduce frameworks have been specifically designed to take advantage of multi-core CPUs and GPUs: Mars [52], MapCG [54], and MATE-CG [58]. Unfortunately, they all have limitations which reduce their portability and incur a much higher programming overhead than our solution. All three prototypes are restricted to a single node or GPU, which greatly limits the size of problems that they can handle. In addition, all three prototypes use CUDA as their backend GPU language, which limits the supported hardware to only NVIDIA GPUs. Mars stores all the intermediate results in GPU memory, which requires the user to specify beforehand how much data will be emitted during the Map phase [52]. This step requires additional effort from the programmer and is highly error-prone. MapCG uses a C-like language for its Map and Reduce functions which is then converted to OpenMP and CUDA code for parallelism. This restricts the capabilities of the application to their C-like language, which doesnt support many of the advanced feature of CUDA. MATE-CG does not support a Map operation and limits the user to using only Reduce and Combine operations, which makes porting existing MapReduce applications much harder.

Researchers from Tokyo Tech present a method for scheduling Map tasks on either the CPU or GPU depending on a dynamic profile of the task [103]. Others from Ohio State created a MapReduce framework that is optimized specifically for AMDs Fusion APUs [29]. With the Fusion APU, the GPU shares the same memory space as the CPU, which enables their framework to do both pipelining and scheduling of MapReduce tasks across the CPU and GPU. Again, we see a solution that lacks portability.

3.4 Wavefront Algorithm Parallelization

Wavefront-based algorithms have been in discussion over the past approximately 30 years. The wavefront method was revisited in [117] using loop skewing, a procedure to derive the wavefront method of execution of nested loops. Considered as far back as 1974 by Lamport [69], wavefront computations have had applications to diverse areas including linear equation solvers [72, 87], gene sequence alignment [104] and radiation transport [64, 17], iterative solution methods [91], particle physics simulations [63], and parallel solution of triangular of systems of linear equations [53].

State-of-the-art research shows wavefront parallelization on GPUs, FPGAs and co-processors. Smith Waterman, a dynamic programming concept and a local sequence alignment algorithm, expresses wavefront-based computations and the algorithm has been mapped to NVIDIA GPUs [94], on Cell BE [116] and on reconfigurable computing platforms [124, 25]. ASCI Sweep3D wavefront application solves a 1-group time-independent discrete ordinates neutron transport problem on IBM Blue Gene/P machine [57] by using blocking techniques for better parallel efficiency as the application undergoes rapid succession of wavefronts. Preliminary studies to use TBB, Cilk, CnC, and OpenMP 3.0 for wavefront in [37] indicate that optimizations be wrapped in a higher level template to make it easier for less experienced users. AWE Chimaera [80], NAS-LU [16] use different implementations of Lamport's original parallel pipelined wavefront 'hyperplane' algorithm [69]. Acceleration of generalized pipeline wavefront applications on modern GPU is discussed in [86]. Geometric Multigrid (GMG) is a class of algorithm used to accelerate the convergence of iterative solvers for linear systems [114] using wavefront techniques. Proxy apps such as KRIPKE [5], SNAP [8] (mimicking communication patterns of PARTISN [19] transport code) wavefront codes investigate different data layout patterns and parallelism. A one-sided communication in Sweep3D using Coarray Fortran achieved comparable performance to that of the MPI version [33].

Classic compiler approaches for wavefront shows loop skewing followed by loop permutation, where skewing breaks a dependence that would otherwise prevent permute; this is implemented within CHiLL [28], a polyhedral compiler transformation framework. Other work includes IEGenLib [108], Codegen+ [26]. A recent paper [110] presents compiler and runtime framework within polyhedral framework to automatically generate wavefront parallelization of sparse matrix computations. Swift/T [118] has also been used to express wavefront computations [15]. Automatic generation of wavefront parallelization of sparse matrix computations was discussed in [110]. Wavefront parallelization of Gauss-Seidel and related algorithms employ manually written inspectors and executors as discussed in [107, 45]. High Productive Computing System (HPCS) languages: Chapel [23], X10 [79] and Fortress [12] also aim to provide users with better programmability and productivity but they do not offer enough abstractions or vocabulary for heterogeneous platforms. HTP [121] proposed a hierarchical tree place that maps to an architecture with the goal of scheduling tasks to different different nodes in the tree.

All these prior studies indicate that this is an important problem to solve and that there are different types of approaches to approach this problem. However, most of these strategies cannot be easily adopted for large scale applications since they are either not solving the wavefront problem itself but offering solutions to a specific problem type, requiring the user to incur a steep learning curve, or providing a solution that is confined to a particular compiler and/or hardware. Such gaps in the state-of-the-art work served as a motivation for us to rethink this problem and develop a suitable solution that will provide scientific developers with directives supported by appropriate loop transformation algorithms addressing the wavefront problem.

Chapter 4
MOTIVATION FOR USING ACCELERATORS

Prior to investigating high-level complex parallel patterns, the author studied related topics including auto-tuning high-level languages targeting heterogeneous systems, parallelizing machine learning algorithms used for malware detection, and combining MapReduce with on-node parallel programming frameworks in order to accelerate Big Data applications. This chapter dives into those topics, provides some insight into the authors related contributions in the field of high performance computing, and highlights the benefits of using accelerators in the associated domains.

4.1 Auto-Tuning GPU Accelerated Applications Using High-Level Languages

Determining the best set of optimizations to apply to a kernel to be executed on the graphics processing unit (GPU) is a challenging problem. There are large sets of possible optimization configurations that can be applied, and many applications have multiple kernels. Each kernel may require a specific configuration to achieve the best performance, and moving an application to new hardware often requires a new optimization configuration for each kernel.

This work applies optimizations to GPU code using HMPP and OpenACC, which are high-level directive-based languages backed by source-to-source compilers that can generate GPU code based on annotations in portions of the host code. However, programming with high-level languages was previously thought to mean a loss of performance compared to using low-level languages. This work shows that it is possible to improve the performance of a high-level language by using auto-tuning. We perform auto-tuning on a large optimization space on GPU kernels, focusing on loop permutation, loop unrolling, tiling, and specifying which loop(s) to parallelize, and show results on convolution kernels, codes in the PolyBench suite, and computationally expensive kernels extracted from the QuantLib library, which is widely used in the domain of computational finance. The results show that our auto-tuned implementations are significantly faster than the default HMPP and OpenACC implementations and can meet or exceed the performance of manually coded CUDA / OpenCL implementations [46, 47].

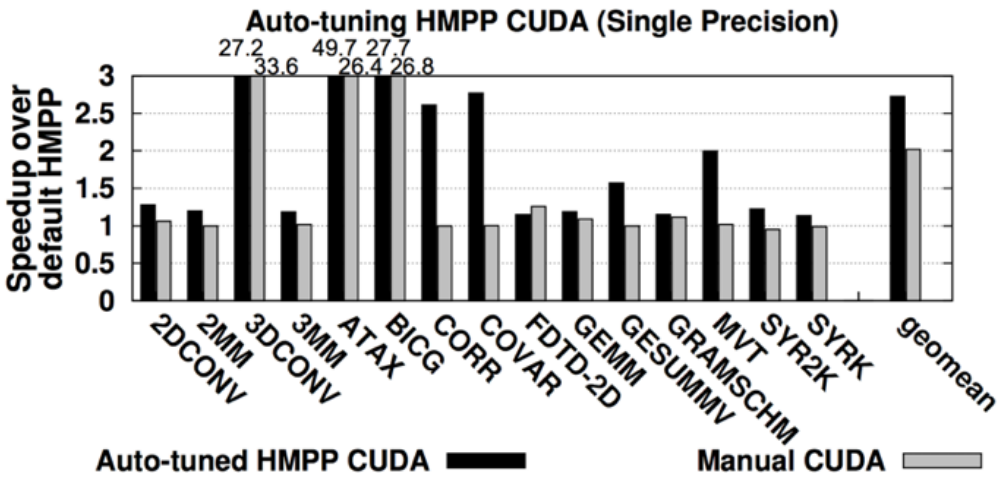
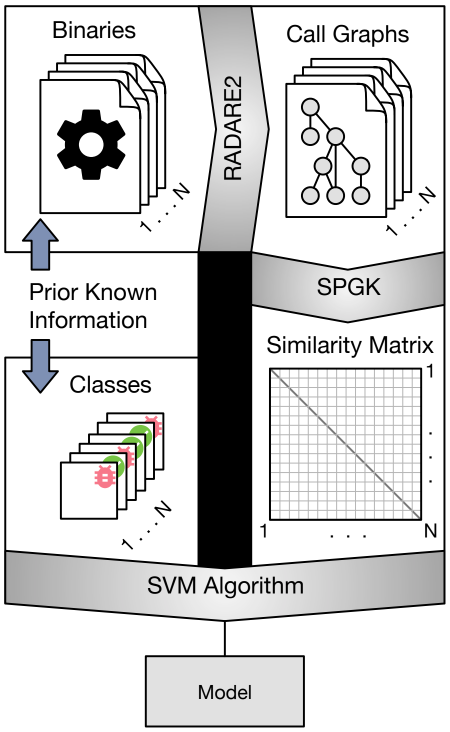


Figure 4.1: Single precision speedup graph comparing the performance of auto-tuned HMPP code against manually written CUDA code

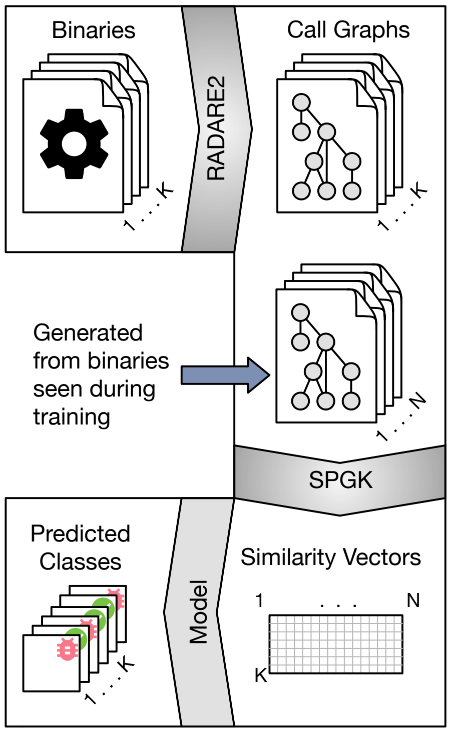
Figure 4.1 shows a sample of the results obtained using HMPP to auto-tune kernels from the Polybench benchmark suite [46]. We see that the HMPP auto-tuned code is at least on par with manually written and tuned CUDA code in all cases, and in some cases, it exceeds the performance of the CUDA code by a considerable margin. This demonstrates the viability of high-level languages from a performance standpoint, in addition to the benefits of their programmability and inherent portability.

4.2 Parallelization of Graph-Based Machine Learning for Malware Detection

Prior work in learning-based malware detection engines primarily focuses on dynamic trace analysis and byte-level n-grams. The approach in this work differs in that compiler intermediate representations are used, i.e., the call-graph representation of binaries. Using graph-based program representations for learning provides structure of the program, which can be used to learn more advanced patterns.



(a) Training Phase



(b) Testing Phase

Figure 4.2: Figure 4.2a shows the workflow toward construction of a machine learning model generated during this work's training phase. Figure 4.2b shows a flow diagram demonstrating the classification of an unseen binary application used during the evaluation phase.

This work uses a computationally expensive graph kernel to identify similarities between call graphs extracted from binaries [98]. The output similarity matrix is fed into a Support Vector Machine (SVM) algorithm to construct highly-accurate models to predict whether a binary is malicious or not, as shown in Figure 4.2a. Once a model has been constructed, similarity vectors for call graphs of unknown applications need to be computed in order to feed into the model for comparison against the training data set. This is known as the testing phase, as shown in Figure 4.2b.

Since this graph kernel is computationally expensive due to the size of the input graphs, different parallelization methods for CPUs and GPUs are evaluated to speed up this kernel, allowing continuous construction of up-to-date models in a timely manner. The hybrid implementation presented, which leverages both CPU and GPU, yields the best performance, achieving up to a 14.2x improvement over an already optimized OpenMP version, as shown in Table 4.1. The generated graph-based models are then compared to previously state-of-the-art feature vector 2-gram and 3-gram models on a dataset consisting of over 22,000 binaries. This work's classification accuracy using graphs is over 19% higher than either n-gram model and gives a false positive rate of less than 0.1%. It is also possible to consider large call graphs and dataset sizes because of the reduced execution time of the parallelized graph kernel implementation, which leads to the construction of more accurate prediction models.

Dataset Size	GPU 1D	GPU 1D Overlap	OpenMP Matrix	OpenMP Graph	Hybrid
6K	1.58 h 0.01x	40.52 m 0.03x	1.01 m 1.0x	3.19 m 0.35x	4.65 s 14.2x
12K	9.36 h 0.02x	5.61 h 0.03x	10.96 m 1.0x	20.48 m 0.53x	52.57 s 12.5x
21K	memory exhausted		4.55 h 1.0x	5.50 h 0.83x	50.37 m 5.42x
22K	memory exhausted		6.51 h 1.0x	12.24 h 0.53x	2.47 h 2.63x

Table 4.1: Comparing similarity matrix computation time for each dataset. Runtimes (h: hours, m: minutes, s: seconds) are presented for each implementation. Below each runtime we give speedup compared to the best OpenMP (OpenMP Matrix) implementation is shown. VRAM limitation was exceeded for our larger datasets when only running on the GPU.

Despite the excellent performance achieved utilizing this novel approach to machine learning applied to malware detection, the hybrid implementation presented is not portable due to the use of architecture-specific programming models (such as CUDA). At the time when this work was presented, there were no high-level tools available that were suitable for abstracting a complex graph kernel like the one used. This presented a significant implementation challenge because this approach needed to be rewritten in order to run on the GPU. However, it still offered insights as to how certain types of algorithms behave on different parallel architectures. We found that on average, smaller graphs were better compared using the CPU and larger graphs, whose computational cost drastically outweighed the overhead of data movement, were better suited for execution on the GPU.

4.3 A Portable, High-Level Graph Analytics Paradigm Targeting Distributed, Heterogeneous Systems

This paper presents a portable, high-level paradigm that can be used to run Big Data applications on existing and future HPC systems. More specifically, it targets graph analytics applications, since these types of applications are becoming increasingly more popular in the Big Data and Machine Learning communities. Using this paradigm, we accelerate three real-world, compute and data intensive, graph analytics applications: a function call graph similarity application (similar to the one discussed in Section 4.2), a triangle enumeration subroutine, and a graph assaying application.

Our paradigm utilizes the popular MapReduce framework, Apache Spark, in conjunction with an on-node computational framework (in our case CUDA) in order to simultaneously take advantage of automatic data distribution and specialized hardware present on each node of our HPC systems, as shown in Figure 4.3. We demonstrate scalability with regard to compute intensive portions of the code that are parallelizable, as well as an exploration of the parameter space for each application. We present results on a heterogeneous (hybrid) cluster with a variety of CPUs (from both Intel and AMD) and GPUs (from both AMD and NVIDIA), as well as NVIDIAs PSG cluster, which is homogeneous and utilizes NVIDIAs next-generation P100 GPUs. Figure 4.4 presents performance results for the graph assaying application when run on a heterogeneous cluster consisting of a variety of CPUs and GPUs. It exhibits favorable scalability when run across multiple nodes, and a considerable speedup when using the GPU instead of the CPU on those nodes for their portion of the computation. These results prove that

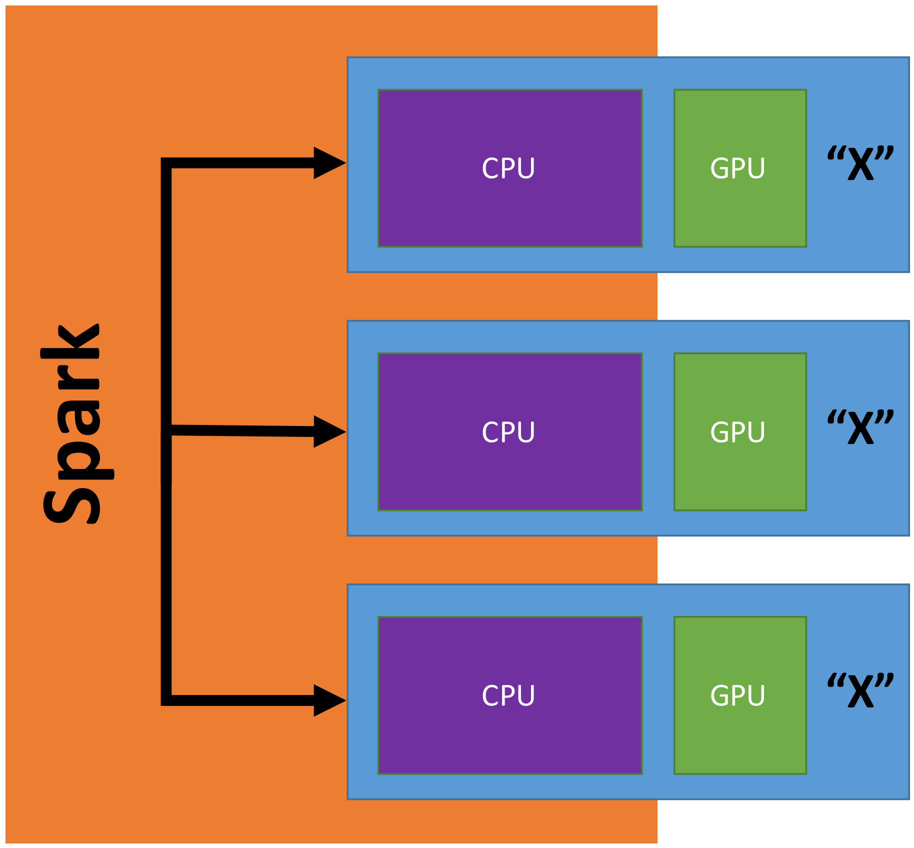
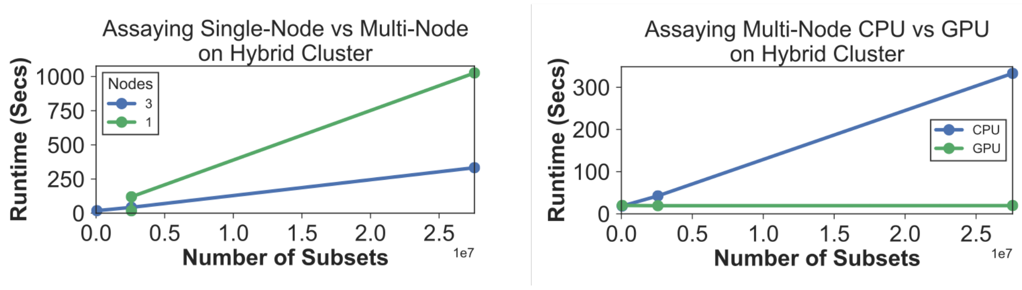


Figure 4.3: A high-level depiction of the proposed paradigm, which uses Spark for data/task distribution in conjunction with a local computational framework (X) on each node.

our method yields a portable solution that can be used to leverage almost any legacy, current, or next-generation HPC or cloud-based system [97, 101].

This work aimed to tackle the challenges of automating the process of task distribution across nodes in an HPC system, and it redefined the way we think about portability by running real-world graph analytics applications across a cluster composed of a range of different parallel architectures, while achieving optimal load balancing. To our knowledge, this is the first work to combine and efficiently utilize legacy, state-of-the-art, and next-generation hardware simultaneously in an abstract fashion.



(a) Single-node vs multi-node

(b) CPU vs GPU

Figure 4.4: Runtime results for Graph Assaying run on the hybrid cluster

4.4 Summary

In short, these projects all contributed to our understanding of HPC systems, in regards to both parallel hardware architectures and the software programming models used to harness the power of such hardware. The skills we have acquired in overcoming the obstacles faced in each work provides the foundation needed to tackle a multifaceted problem like the one described in this thesis. In order to accomplish the goal of creating a portable programming abstraction for the wavefront parallel pattern, we need to use these skills to understand how the pattern in question behaves on different types of hardware, identify the challenges in abstracting the pattern such that it will be performant across existing and future parallel architectures, and understand the needs of parallel application developers in order to develop extensions that are easy to use from a development perspective.

Chapter 5

EXPLORING A COMPLEX PARALLEL PATTERN: WAVEFRONT

This section introduces the complex parallel pattern investigated by this thesis, demonstrate it's real-world impact, discuss the obstacles faced when attempting to parallelize such a pattern, and outline our goals moving forward.

5.1 Wavefront Parallel Patterns

A wavefront parallel pattern is a type of complex parallel pattern that is not easily representable in existing high-level parallel programming models due to its irregular, multi-dimensional data dependencies. This pattern is used to examine a multi-dimensional space that is split into components called $gridcells$. Each gridcell contains elements that require some sort of in-gridcell computation, allowing for parallelism across these gridcells. However, the complexity in implementing this type of parallel computation arises due to the upstream data dependencies between gridcells. Wavefront patterns exhibit a directional behavior in that the result computed at each gridcell depends on the result computed at each of its neighboring gridcells along each axis in the multi-dimensional space. This dependency places a restriction on the order in which results can be computed. A parallel wavefront implementation sorts these in-gridcell computations into a series of ordered $wavefronts$ described by a sequence of planes of gridcells starting at a corner of the multi-dimensional grid and sweeping through the whole grid.

An example of the 3-dimensional wavefront sweep ordering used in Minisweep is shown in Figure 5.1. While other orderings are allowed insofar as the upstream data dependencies are satisfied, this is generally deemed to be the most efficient way

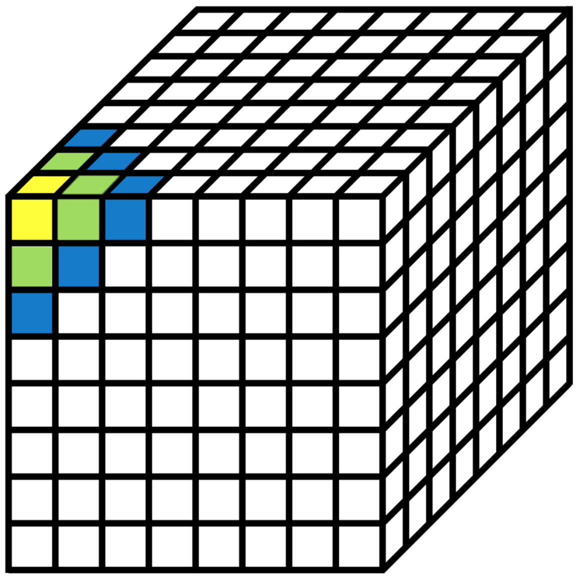


Figure 5.1: Minisweep's wavefront computational pattern

of parallelizing a wavefront-based pattern. It is worth noting that there are some variations in upstream data dependency behavior across different wavefront algorithms.

As shown in Figure 5.2, the Smith Waterman sequencing algorithm utilizes a parallel wavefront pattern across a 2-dimensional space, but unlike Minisweep, it contains upstream data dependencies for all neighboring gridcells, not just the ones along each individual axis. This means that the diagonal neighboring gridcells have been computed two wavefront iterations before the current iteration, which is unlike Minisweep's upstream dependencies that all reside along the previous wavefront iteration.

5.2 Real World Impact

Wavefront-based parallel patterns have been found in several applications including particle simulation, bioinformatics, plasma physics and linear solvers. Currently, scientists are either resorting to not exploiting this pattern due to the computational complexity, or they are restructuring their codebase manually, which can be quite time consuming and error prone. By creating a high-level parallel programming language extension, we will be enabling application scientists to express these patterns

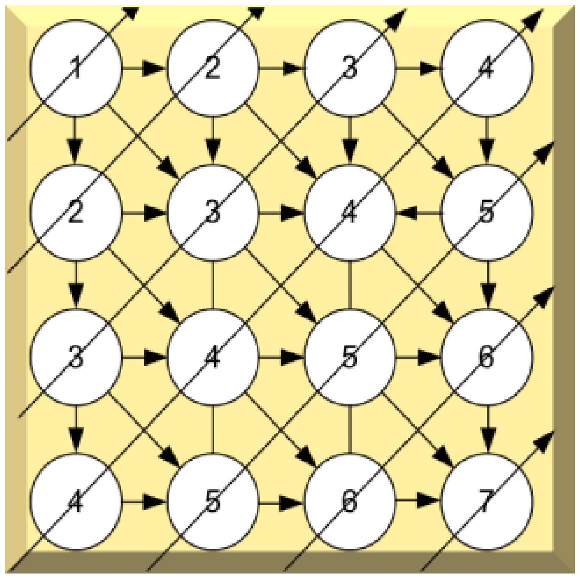


Figure 5.2: Smith Waterman's wavefront computational pattern

on current and future platforms, thus overcoming these obstacles. Our case study, Minisweep, represents a large scientific application, Denovo, as we mentioned earlier. This is the radiation transport application that has a direct impact on the accuracy of radiation shields built around nuclear reactors. By creating a high-level, directive-based port of this code, we are enabling easy adaptability of wavefront patterns on hardware platforms, allowing scientists to run several more configurations that can determine and impact the shield accuracy. Such wavefront-based patterns were also observed in miniapps KRIPKE and SNAP, which are also radiation transport codes.

Similarly, the Smith Waterman algorithm is a classic local sequence alignment algorithm that has been widely studied and adopted for several sequence alignment tools that are instrumental in studying cancer tumors [104]. Most of these tools are created using either low-level or proprietary programming languages, and thus, they are not portable across platforms. Therefore, the codebase is not contained in an easy-to-use framework that biologists can utilize. By creating a software abstraction and exploiting the wavefront pattern this alignment algorithm exposes, we will be able to create a high-level Smith Waterman code that can be used for sequence alignment purposes more easily. Similarly, the Hi-C application determines chromosome interactions that directly impacts tumor cell identification. The interactions currently have not been studied from a computational standpoint. Observing a small set of interactions with naked eye reveals a wavefront pattern of a different type compared to that of Minisweep and Smith Waterman. Further exploration of the data flow and dependency analysis of these interactions has lead us to examine the right usage of the developed high-level wavefront-based software abstraction.

In summary, we can see that wavefront-type parallel patterns are observed in several applications. They provide an opportunity to exploit parallelism, but due to the complexity of data flow and dependencies in these patterns, exploring such parallelism has remained a challenge. Creating a software abstraction for these patterns has been an even bigger challenge.

5.3 Implication of Wavefront's Parallel Pattern on Data Transformation

As mentioned in Section 5.1, a wavefront parallel pattern contains a number of upstream data dependencies that vary from application to application. Identifying a wavefront access pattern in code just tells us that backward dependencies are satisfied, but it doesnt tell us how many of those are needed for the current computation. From a computational standpoint, all we know during each wavefront iteration is that each cell in the current wavefront iteration can be calculated in parallel. This doesnt specify how far back the data dependencies go. For example, Smith Waterman looks at all neighboring cells, some of which were calculated two wavefronts behind the current wavefront. In Minisweep, we only use data from the preceding wavefront. Additionally, the contents of each gridcell will vary application to application, as the goals of the program could be totally different despite utilizing a wavefront sweep pattern. This means that the data representations of each application is going to follow a particular pattern, but they will not be uniform across codes.

5.4 Thesis Objective

The objective of this thesis is to create a high-level programming language extension for wavefront-based parallel patterns, such that applications representing such a pattern can easily adopt this high-level extension and be able to expose wavefront parallelism on hardware platforms without struggling through the phase of manually restructuring and reprogramming the given code. This thesis also uniquely explores the data and dependency flow in applications that expose wavefront computations in structured and unstructured meshes. A high-level language feature is created with compiler and runtime support that is evaluated on applications from different domains. In summary, this thesis creates a high-level portable solution for wavefront-based codes that the scientists can use to port their legacy applications to modern HPC systems without compromising performance or accuracy.

Chapter 6

DESIGNING SOFTWARE ABSTRACTIONS IN PROGRAMMING MODELS FOR WAVEFRONT PATTERN

6.1 Analyzing the Flow of Data and Computation in a Wavefront Model

Chapter 5 provides a clear definition of what a wavefront code is in terms of both functionality, as well as what the implementation generally entails. In order to construct an abstract parallelism model, we need to consider two main components: code transformation and data representation.

Wavefront codes typically follow a very similar pattern in terms of their loop nests. Parallelizing these types of codes usually involves transforming a serial loop nest into one with a wavefront-style pattern where we can parallelize the in-gridcell computations across gridcells within a given wavefront iteration. In practice, this requires some source-to-source loop translations. Generally, this consists of taking a serial loop nest, calculating the number of wavefront iterations required for the desired parallelism-friendly access pattern, and transforming the loop nest such that we iterate over each wavefront in serial, while parallelizing its inner loop(s). This is less than trivial, since we need to perform bounds checks to make sure that all the threads launched from the inner loops indeed lie within the bounds of the dimensions of the current wavefront iteration in order to preserve computational accuracy. It also requires that we have some way of storing and representing data such that each wavefronts dependencies are preserved.

Despite being unable to completely automate the representation of a parallel wavefronts data model, we can still construct a guided generalization that requires limited input from the programmer. Typically, codes that utilize a wavefront algorithm have uniform gridcells (all gridcells contain a fixed amount of data). Given that presumption, it is reasonable to assert that a generalized data representation for satisfying dependencies between wavefront iterations can be constructed with a little guidance from the programmer. There are two components to such a representation: gridcell size and dependency depth. If the programmer provides information about these components, we can automate the construction of an intermediary storage unit where data can be compiled and updated between wavefront iterations such that each wavefront iteration has access to all its necessary data dependencies, while still minimizing the memory footprint of such a data structure.

The reason that this programmer input is required is due to the lack of a concrete data model. Each wavefront application is a little different in terms of data dependencies and access patterns. We contend that automating the analysis of dependency information for wavefront applications is a separate research problem entirely, and it extends well beyond the scope of wavefront codes. There are currently no state-of-the-art tools that do this well for parallel applications. Existing tools prioritize accuracy, and therefore, any slight ambiguities are assumed to be dependencies. In the case of an application with an irregular access pattern (like wavefront codes), extensive analysis would have to be performed on the computational component of the code to determine all possible values at a given wavefront iteration to ensure that there are no data collisions, while also ensuring that all dependencies from previous iterations are met. An example of this can be shown by utilizing PGI's OpenACC compiler with the *kernels* directive, which is meant to automatically detect which portions of the code that are parallelizable. When this is used on our case study application of Minisweep, discussed later in Chapter 8, all the loops in the loop nest corresponding to the spatial decomposition of the simulation are considered to have potential data dependencies, and are therefore no considered to be parallelizable. Based on our analysis of this wavefront codes access pattern, we know this to be untrue. Within the bounds of a wavefront iteration, Minisweep does not contain any data collisions or unmet dependencies.

6.2 The Memory Model Abstraction

In addition to simply constructing a data structure to satisfy data dependencies between wavefront iterations, it is important to also consider the best ways to optimize the structure to minimize main memory accesses and maximize both spatial and temporal locality within the cache [113]. This is especially important in parallel systems because we have limited amounts of cache and multiple processing units will access that cache simultaneously. To that end, this thesis presents a general rule for optimizing data structures. Since a wavefront algorithms loop nest is structured such that the spatial decomposition exploits coarse-grained parallelism and the in-gridcell computations utilize fine-grained parallelism, it is important to make sure that the data structure used to store temporary data is structured in a way that is advantageous to the manner in which the data will generally be accessed. More specifically, the components of the in-gridcell computations (whatever they may be) should be the fastest-varying components of the storage array(s), and the spatial components should be the slowest-varying. The spatial components should also reflect the order in which they appear in the code (i.e. in a 3D loop nest accessed in the order $Z/Y/X$, Z should be the slowest varying dimension, followed by Y, and then X).

This generalization can be applied to any existing parallel architecture. Generally speaking, parallel systems have some sort of multi-unit design. Whether that unit is a core (CPU) or a multiprocessor (GPU), each unit typically has a certain amount of cache that its computational resource(s) have access to. In the case of a CPU, each core has its own L1 cache. In a GPU, such as NVIDIAs new V100 GPU, each SM (multiprocessor) has its own L1 cache that each core within the SM has access to. Our goal is to perform as much computation as possible on the data we can fit within these cache spaces before we discard it in favor of reading in new data from main (CPU) or global (GPU) memory.

Our OpenACC prototype of Minisweep accomplishes this on both architectures.

When compiled for multicore CPU architectures, the spatial decomposition is parallelized across CPU cores. Each core is responsible for a subset of the in-gridcell computations along each wavefront. For a given in-gridcell computation, we must perform a series of computations on all data that reside within that gridcell. If our data structure follows the generalization above, each CPU can exploit spatial and temporal locality because the in-gridcell data is the fastest-varying, and thus, the data resides in adjacent blocks of main memory. When this same code is compiled for execution on a GPU, a similar behavior is observed. The spatial decomposition is parallelized at the gang-level (per SM). This is equivalent to the CPUs parallelization. The GPU simply adds an additional layer of parallelism within each in-gridcell computation at the vector-level (per core within an SM). This allows the GPU to exploit additional temporal locality, since vector-level threads access spatially local pieces of memory simultaneously. In both cases, accesses to main (CPU) or global (GPU) memory are minimized. This generalization can be applied to any existing or future parallel architecture that has a multi-unit design where each unit has its own dedicated cache.

6.3 Translating the Wavefront Abstraction

Wavefront applications can be broken down into two main computational components: spatial decomposition and in-gridcell computation. Spatial decomposition refers to the challenge of decomposing a multidimensional space (array, matrix, 3-dimensional space, etc) in a way that parallelism can be exploited. Such a decomposition in these types of applications involves overcoming data dependencies in an organized fashion so as to exploit as much cross-gridcell parallelism as possible. Most wavefront algorithms also have an in-gridcell portion of computation, which may or may not involve additional data dependencies.

It is important for us to consider how these computational components can be mapped onto hardware. From a basic multicore CPU standpoint, this analysis is relatively straightforward: we have a number of threads available to us, and we need to determine whether the spatial component or the in-gridcell component will utilize a higher percentage of available threads. Typically, computational science applications will contain enough in-gridcell parallelism to utilize all available threads on a single CPU, leaving our wavefront dependency at the spatial level insignificant and/or unconsidered. However, modern HPC systems are becoming increasingly heterogeneous, dense (more computational resources on a single node), and distributed (multi-node). In particular, accelerators such as GPUs have become a central component of some of the world's fastest supercomputers, including ORNL's Titan [60] and Summit [67] machines. GPUs contain thousands of cores, have a hierarchy of parallelism within them, and are abundant on machines like Summit and Sierra, where a single node contains multiple GPUs. Failure to exploit all available layers of parallelism within an application leaves a lot of performance on the table in machines with this much computational power.

While wavefront applications don't share a common type of in-gridcell computation or even an identical data dependency, they do have some commonalities at the spatial level. In particular, wavefront dependencies exist exclusively in the upstream direction. As we mentioned in Section 6.1, we cannot know how far in the upstream direction a wavefront algorithm will reach for data, so it is the job of the programmer to set up their data structures in a way that this can be satisfied and kept up-to-date throughout the application's execution. To study further, we referred to the famous Koch-Baker-Alcouffe (KBA) [63] and Pautz [85] algorithms in order to explore restructuring the order of a wavefront application's execution to exploit the available parallelism at the spatial level, while satisfying the required dependencies. The KBA algorithm allows us to extract parallelism sweeping through the cartesian coordinates without running into data dependency issues and Pautz defines special heuristics to relax KBA for unstructured meshes.

6.3.1 What is the Koch-Baker-Alcouffe (KBA) Method?

The KBA method is a method of transforming a serial loop nest into one with a wavefront-style pattern, which allows us to parallelize the in-gridcell computations across gridcells within a given wavefront iteration. An implementation of the KBA method consists of two major components:

1. The transformation of a serial loop nest into a wavefront-based sweep over the multi-dimensional space being examined
2. The parallelization of in-gridcell computations within each wavefront iteration

The translation of a wavefront-dependent loop nest into an iteration scheme that is friendly toward upstream data dependencies contains two major code transformations: the creation of a wavefront hyperplane iteration and an inner bounds check across parallel threads to ensure that we are computing cells that lie along the created hyperplane. We can calculate the loop bounds for the desired hyperplane iteration scheme as a factor of the dimensions of the original loop nest by summing the bounds of all the dimensions in the multi-dimensional space and subtracting $N - 1$ from the result, where N is equal to the number of dimensions being examined. In a traditional 3-dimensional space, the calculation of the total number of wavefront iterations is as follows:

```
num_wavefronts = (dimX + dimY + dimZ) - (3-1)
```

We can generalize this calculation using the following equation:

num_wavefronts $= \sum_{X=1}^{N} dimX$ $-$ (N$-$1)

Listing 6.1: Calculating the number of wavefronts

Once we have performed the wavefront calculation, we have to make some modifications to the spatial components of our loop nest. First, we replace our outermost spatial loop with a sequential wavefront loop, which iterates from 0 to the $num_wavefronts$ value we just calculated using the formula above. The inner spatial dimension loops do not need to be modified, except that they should now execute in parallel. This will result in the creation of a number of threads equal to the product of these inner dimensions. Each thread then must calculate the missing dimension (previously the outermost dimension of the multi-dimensional loop nest), as a function of the wavefront number and the values of the other N dimensions, and perform a bounds check to make sure that the calculated value lies along the current wavefront. The dimension calculation and bounds check of a 3-dimensional wavefront is shown below:

```
int iz = wavefront - (ix + iy);
if (iz >= 0 && iz <= wavefront && iz < dimZ)
{ // Perform computation }
```

We can generalize this calculation and bounds check using the following abstraction:

```
iN = current_wavefront_number - $\sum_{X=1}^{N-1} iX$
if (iN >= 0 && iN <= current_wavefront_number && iN < dimN)
{ // Perform computation }
```

Listing 6.2: Wavefront bounds calculation

If this check passes, the thread will continue on and perform its portion of the computation involving the in-gridcell computations of the cell in question. If it fails, the thread will become idle. The result of this is an expansion of saturation across whatever parallel architecture the code is running on. As the wavefront iteration progresses through the multi-dimensional space, the size of each wavefront (in terms of number of gridcells) gets larger, yielding more working threads and better saturation of the parallel hardware. This behavior is illustrated in Figure 5.1. Pseudocode for this code transformation is shown in Algorithm 1.

Algorithm 1: Wavefront loop transformation algorithm

```
Input: An N-dimensional loop nest
Result: A transformed N-dimensional parallelizable wavefront loop nest
num_wavefronts = $\sum_{X=1}^{N} dimX$ - (N-1);
for $wavefront \leftarrow 0$ to $num\_wavefronts$ do
    #pragma parallel
    for $i_{n-1} \leftarrow 0$ to $dim_{n-1}$ do
        ...
        #pragma parallel
        for $i_1 \leftarrow 0$ to $dim_1$ do
            $i_N$ = wavefront - $\sum_{X=1}^{N-1} iX$;
            if $i_N >= 0$ && $i_N < dim_N$ then
                computation;
        end
    end
end
```

6.3.2 High-Level Extension for Wavefront Codes

This thesis develops an extension of existing high-level programming frameworks that automates the process of parallelizing wavefront codes by utilizing the strategies discussed in Section 6.1 and employing the implementation methodology presented in Section 6.3.1. We contend that this has far-reaching implications on existing and future applications, as wavefront algorithms are very common and widely used in the domain of computational science. Currently, such applications require manual code refactoring/restructuring to implement a parallel wavefront sweep. As a result, developers are constantly reinventing the wheel, instead of focusing their efforts on the science their wavefront code is meant to simulate or analyze.

6.3.3 Exploring Asynchronous Execution to Saturate Massively Parallel Processors

In addition to the high-level extension presented, this thesis explores different types of asynchronous execution models to help further optimize wavefront codes. Specifically, there are two main types of execution models investigated. The first involves asynchronously executing portions of wavefront sweeps across multiple parallel devices. Since future high performance computing (HPC) machines are built on a multi-node/multi-device platform, this could be beneficial. For example, ORNLs next-generation supercomputer Summit will feature approximately 4,600 nodes, each equipped with 6 NVIDIA Tesla V100 GPUs. It would be beneficial if a high-level wavefront extension allowed a developer to utilize all the available devices in a node in order to fully utilize the hardware resources that machines like Summit have to offer.

The second type of execution model explored is asynchronous execution of sweeps within a single device. In some wavefront codes, such as Minisweep, multiple wavefront sweeps are performed within each timestep iteration. Depending on the configuration of a given experiment, a single sweep might not be enough to fully saturate a single device. Asynchronous execution could help ensure that the device is fully utilized by running these sweeps asynchronously on a single device. For example, Minisweep models a 3-dimensional space (3D sweep), and it performs a sweep from each of the 8 corners of that space. So, for each timestep, 8 total sweeps are performed. In the case of a smaller spatial configuration, a single sweep might not be able to fully saturate a device, so asynchronous execution on a single device could be advantageous. However, if we run this on a machine like Summit with a large spatial configuration, each sweep could potentially saturate one of Summits GPUs. In this case, it would be more advantageous to distribute each of Minisweeps 8 sweeps across the 6 GPUs available in a node on Summit in order to better utilize the available hardware.

As with the general high-level extension, adding both types of asynchronous execution as an additional abstraction layer on top of the parallel wavefront abstraction layer would be greatly beneficial to developers from both a implementation and portability standpoint.

6.3.4 Prototyping Software Abstraction

Due to the repetitive nature of such a modification and the prevalence of wavefront algorithms within the domain of computational science, this thesis creates an

```
#pragma acc wavefront(3)
{
 for (iz=0; iz<dim_z; ++iz)
   for (iy=0; iy<dim_y; ++iy)
     for (ix=0; ix<dim_x; ++ix)
       {
         /*--- In-Gridcell Computation ---*/
       }
}
```

(a) An example of an annotated wavefront loop nest prior to transformation.

```
 int num_wavefronts = (dim_z + dim_y + dim_x) - 2;
 for (int wavefront=0; wavefront < num_wavefronts; wavefront++)
{
#pragma acc loop independent gang, collapse(2)
   for (iy=0; iy<dim_y; ++iy)
     for (ix=0; ix<dim_x; ++ix)
       {
 iz = wavefront - (iy + ix); /*--- Solve for outer dim ---*/
 if (iz >= 0 && iz < dim_z) /*--- Bounds check ---*/
{ /*--- In-Gridcell Computation ---*/ }
       }
}
```

(b) The result of transforming the serial loopnest shown in Figure 6.1a.

Figure 6.1: An example of our extension to the OpenACC standard. Note that the dimensionality of the wavefront component is specified in the directive so that the preprocessor knows how many loops within the loop nest to consider when performing the requested transformation of the loop nest.

extension to be considered by programing model standards that would support this type of a code transformation. In addition to formalizing the representation of this code transformation, we have created a preprocessor that can be used to automate this transformation and will serve as a proof-of-concept of our developed extension to the existing OpenACC standard. Figure 6.1 shows an example loop nest before and after the wavefront transformation is performed. Note that all of the required loop transformations are performed using the original loop bounds while also preserving the in-gridcell computation, denoted by the placeholder for in-gridcell code. In a real application, the full body of the in-gridcell computation would be preserved. The number following the $wavefront$ clause tells the compiler (or preprocessor in our case) how many loops to consider as part of the wavefront. Due to the complexity of many computational science applications, additional loops could be present as part of the in-gridcell computation, and we would not want to consider these as part of the wavefront, since they have their own application-specific behaviors and the behavior varies from one scientific domain to the other.

We also notice that the resulting spatial loops are parallelized using two clauses from the OpenACC standard: *gang* and *collapse* clauses. By default, PGI's OpenACC compiler will parallelize loops annotated with the *gang* clause across CPU threads if targeting a CPU, and it will parallelize these same loops across thread blocks when targeting GPUs. Loops annotated with the *vector* clause are ignored when targeting multicore CPUs. Second, using *collapse* ensures that both of these loops are parallelized at the gang level. This leaves OpenACC's *vector* clause free for use within the in-gridcell component. In real-world applications (like the algorithms discussed in Sections 7.1.2 and 7.1.3), there is a fair bit of computation to be done inside of each gridcell. When working with state-of-the-art hardware like GPUs, we can expose this parallelism at the thread level in order to leverage vector threads within thread blocks and ensure that we are utilizing the full capability of the device.

We want to specifically point out that, in a more general sense, these strategies are not confined to use within the OpenACC standard. While our extension is targeting the OpenACC standard, a similar strategy can be used to create abstractions for other programming models and frameworks. In the case of OpenMP 4.5, for example, we can use the *teams* and *simd* directives where we would use OpenACC's *gang* and *vector* directives, respectively. This would yield a similar accelerator offloading strategy. Wavefront algorithms usually feature (at minimum) the two layers of parallelism we have discussed: spatial (wavefront) parallelism, and in-gridcell parallelism. The challenge when proposing an abstraction and/or an extension to an existing model or framework lies in the efficient mapping of these layers of parallelism to concepts within the model in question that map appropriately to the targeted hardware architecture(s).

Chapter 7

WAVEBENCH: A TOOL TO MODEL THE PERFORMANCE IMPACT OF WAVEFRONT PARALLELISM

7.1 Introducing Wavebench

In addition to our formal representation of the wavefront transformation, we have developed a tool, called Wavebench that can be used to model the performance of wavefront-based applications on modern HPC systems. Wavebench is comprised of algorithms that represent the core computational components of the types of applications discussed in Section 2.4. These algorithms have one thing in common: limited parallelism due to upstream data dependencies that can be overcome using our wavefront abstraction detailed in Section 6.3.4. Wavebench is written in C++ and features a modular design. Each algorithm is contained within its own C++ class, which inherits from a common *Simulation* parent class. At runtime, the user can select which algorithm they would like to run, as well as specify the problem size dimensions for the simulation in question. The modularity of Wavebench's code structure allows for it to be extended in the future with new wavefront-based algorithms. Currently, we feature three algorithms: Local Sequence Alignment (LSA), Gauss-Seidel, and Radiation Transport.

7.1.1 Local Sequence Alignment

Wavebench's Local Sequence Alignment (LSA) algorithm is representative of computation found in gene sequencing applications like Smith-Waterman [104] and HiC [38]. The computations populate a 2D matrix. As shown in Figure 7.1, each cell is dependent on its three neighbors in the upstream direction on the x-axis, y-axis, and along the diagonal (both x and y axes). Note that all nodes in a diagonal can be computed in parallel, as all their dependencies are satisfied from the previous diagonal, as illustrated by Figure 7.2. This pattern is the main commonality between wavefront algorithms.

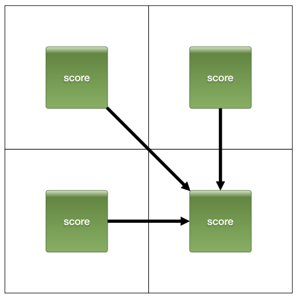


Figure 7.1: Local Sequence Alignment gridcell dependency and in-gridcell computation

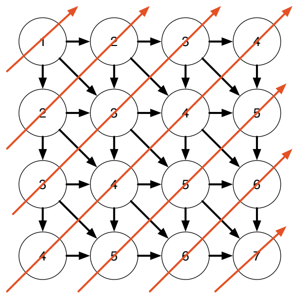


Figure 7.2: 2D wavefront dependency exhibited by the Local Sequence Alignment algorithm

Figure 7.3 shows a pseudocode snippet representative of the LSA algorithm's wavefront computation. A score is calculated using the value of each neighboring cell, and the max is taken as the result stored in the cell currently being computed. This is the only item that is computed for each cell, so the only parallelism that can be exploited in this case is dependent on overcoming the wavefront data dependency.

```
for (int x=1; x<rows; x++)
  for (int y=1; y<cols; y++)
    {
      /*--- Calculate scores ---*/
      int diag_score = matrix[(x - 1) * cols + (y - 1)] + sim;
      int up_score   = matrix[(x - 1) * cols + y] + gap;
      int left_score = matrix[x * cols + (y - 1)] + gap;

      matrix[x][y] = max(0, diag_score, up_score, left_score);
    }
```

Figure 7.3: Local Sequence Alignment loop nest

7.1.2 Gauss-Seidel

The Gauss-Seidel method is an iterative method used to solve a linear system of equations [61]. It is used for a variety of applications, from calculating market demand and price fluctuations to modeling physical 3-dimensional space [35]. Computationally, its structure is quite similar to the LSA algorithm discussed in Section 7.1.1. It solves a 2D matrix with the same data dependencies, less the diagonal upstream dependency. However, it has an additional layer of parallelism within each gridcell, as shown in Figure 7.4. At each gridcell within the computed matrix, a summation is performed for a group of data points within the given gridcell (denoted by the *process_cell*() method shown in Figure 7.5). This inner component is trivial to parallelize, as these data points can be computed independent of one another. However, there are too few of these in-gridcell components to yield peak performance on the majority of today's massively parallel hardware architectures. This justifies the need for overcoming the cross-gridcell wavefront data dependency.

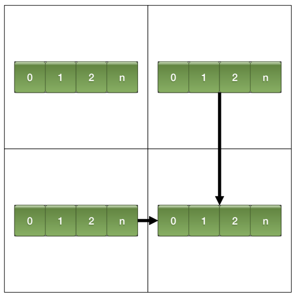


Figure 7.4: Gauss-Seidel gridcell dependency and in-gridcell computation

```
/*---Loop over gridcells---*/
for (int iy=1; iy<ncelly-1; ++iy)
  for (int ix=1; ix<ncellx-1; ++ix)
    {
      process_cell(ix, iy);
    }
```

Figure 7.5: Gauss-Seidel loop nest

7.1.3 Radiation Transport

Radiation Transport applications are widely used in the realm of computational science for simulating flow of neutrons within nuclear reactors and predicting weather. National laboratories have implemented various impactful radiation transport applications, such as Oak Ridge National Laboratory's Denovo [18, 39, 59, 78], Lawrence

Livermore National Laboratory's KRIPKE [66], The United States Nuclear Regulatory Commission's SNAP [8], and The National Center for Atmospheric Research's MuRAM [111, 90, 89]. As illustrated by Figure 7.6, these codes are very complex, requiring the programmer to consider a series of in-gridcell computations in addition to the cross-gridcell wavefront data dependency they exhibit. These codes provide us with insight regarding how complex and computationally intensive wavefront applications in the real world can be.

Much like the previously discussed algorithms, this radiation transport algorithm exhibits a wavefront dependency. However, since it is used to model 3-dimensional space, the wavefront dependency is also 3D. This is actually quite similar to the upstream data dependency of the Gauss-Seidel algorithm discussed in Section 7.1.2 in that each cell requires information from each cell in the upstream direction along each axis. Since there are three axes to consider in a 3D case, this means that each cell will depend on three neighboring cells. Figure 7.7 shows an example of this 3D wavefront behavior by highlighting the first few wavefronts in different colors. We begin by computing the yellow cell, and then we are able to compute the green cells in parallel. Next, we can compute the blue cells in parallel because they rely on the information contained by the green cells, and so on.

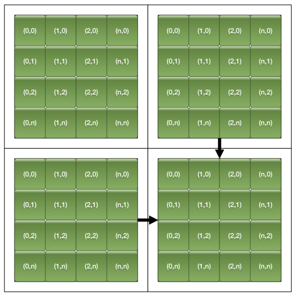


Figure 7.6: Radiation Transport gridcell dependency and in-gridcell computation

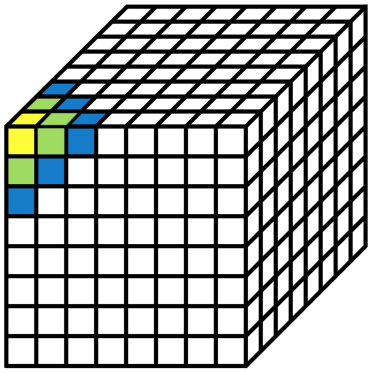


Figure 7.7: 3D wavefront dependency exhibited by the Radiation Transport algorithm

```
for (iz=0; iz<dim_z; ++iz)
  for (iy=0; iy<dim_y; ++iy)
    for (ix=0; ix<dim_x; ++ix)
      {
        /*--- Transform ---*/
        for (ie=0; ie<dim_ne; ++ie)
          for(iu=0; iu<dim_nu; ++iu)
            for(ia=0; ia<dim_na; ++ia)
              {
                for (im=0; im<dim_nm; ++im)
                  { /*--- moments to angles conversion ---*/ }
              }
        /*--- Solve ---*/
        for(ie=0; ie<dim_ne; ++ie)
          for(ia=0; ia<dim_na; ++ia)
            {
              compute(ix, iy, iz, ie, ia, octant);
            }
        /*--- Transform and write ---*/
        for (ie=0; ie<dim_ne; ++ie)
          for(iu=0; iu<dim_nu; ++iu)
            for (im=0; im<dim_nm; ++im)
              {
                for(ia=0; ia<dim_na; ++ia)
                  { /*--- angles to moments conversion ---*/ }
                output[z][y][x][e][u][m] += result;
              }
      }
```

Figure 7.8: Radiation Transport loop nest

The pseudocode representative of the main sweep kernel of ORNL's Denovo radiation transport application is shown in Figure 7.8. The computations expose much more complexity than the previous two types, and it models 3D space, unlike the 2D algorithms discussed in Sections 7.1.1 and 7.1.2. Inside of each gridcell in this 3D space lies a series of multi-dimensional computations. While the data examined by each of the three pieces of in-gridcell computation contain no data dependency with respect to the other components of the given gridcell, it is worth noting that each piece of computation must be completed before the next piece of computation can begin. This creates the need for synchronization points between pieces of parallel computation within a gridcell.

7.2 Evaluation & Results

We use Wavebench to evaluate the efficacy of our extension. We use NVIDIA's Professional Service Group (PSG) cluster, as well as one of our own lab machines in order to examine the performance impact of our solution on many state-of-the-art hardware architectures, including multicore CPUs and multiple generations of NVIDIA GPUs. While we target both multicore and accelerators, we have maintained a single codebase for both types using OpenACC. Table 8.2 shows the configurations of the different nodes that we used within the cluster. Performance of each algorithm is measured using a speedup graph in order to show the performance impact of wavefront parallelism compared to serial code.

Machine	CPU	NVIDIA GPU
UDel Skywalker (V100)	AMD Threadripper 1950x (16 cores)	V100 (16GB HBM2)
NVIDIA PSG (V100)	Intel Xeon E5-2698 v3 (16 cores)	V100 (32GB HBM2)
NVIDIA PSG (P100)	Intel Xeon E5-2698 v3 (16 cores)	P100 (16GB HBM2)
NVIDIA PSG (K40)	Intel Xeon E5-2690 v2 (10 cores)	K40 (12GB GDDR5)

Table 7.1: Specifications of the nodes in the systems we used to test different configurations of Wavebench.

As discussed in Section 7.1.1, Wavebench's Local Sequence Alignment (LSA) algorithm is dependent on overcoming the wavefront data dependency in order to exploit any parallelism, since only a single value is computed within each gridcell. As shown in Figure 7.9, this algorithm runs up to 10x faster using state-of-the-art GPUs in conjunction with our solution. We also notice that the 16-core CPUs outperform the 10-core CPU by a considerable margin, due to the increased parallelism. Since each cell computes only a single value, parallelism is critical to this algorithm's performance. Single-core clock rates do not do much to contribute to the speed of the computation here. While this use-case is relatively simple from a computational standpoint, this algorithm is widely used in applications within the domain of genomics.

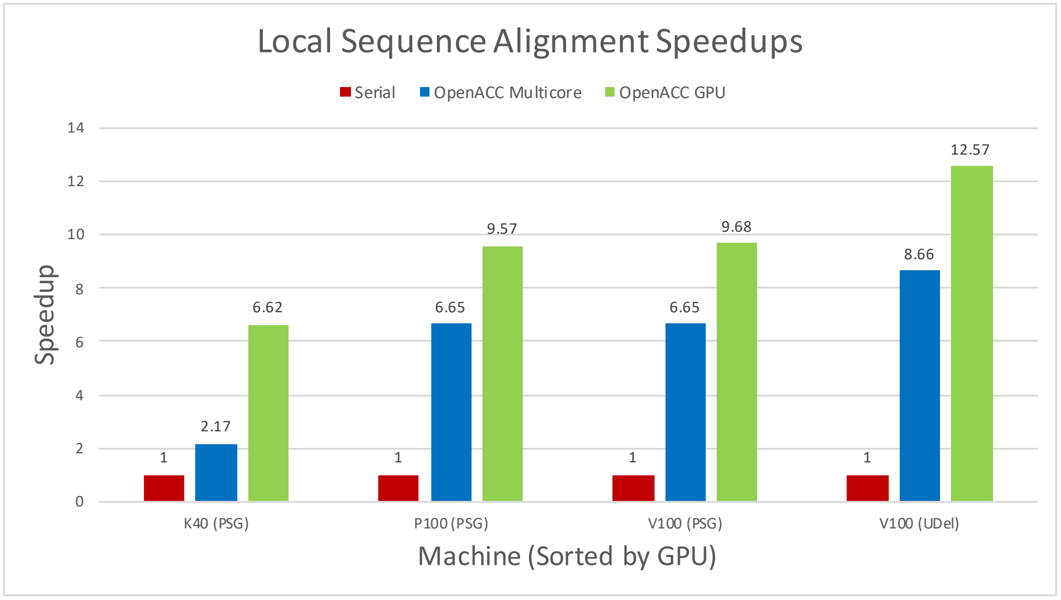


Figure 7.9: Local Sequence Alignment speedups

The complexity of Wavebench's Gauss-Seidel and radiation transport algorithms created an issue in measuring performance because both algorithms are memory-intensive. To that end, we adjusted the simulation's problem size on each node to be as large as possible, while still fitting within each GPU's onboard memory. Our goal is to measure the performance impact of our solution on a single device. Tables 7.2 and 7.3 specify the problem sizes used to test Wavebench's Gauss-Seidel and radiation transport algorithms, respectively. *ncell_x*, *ncell_y*, and *ncell_z* denote the sizes of the spatial dimensions being explored, while *ncomp* refers to the number of elements present within each gridcell.

Machine	ncell_x	ncell_y	ncomp
UDel Skywalker (V100)	500	500	32
NVIDIA PSG (V100)	750	750	32
NVIDIA PSG (P100)	500	500	32
NVIDIA PSG (K40)	500	500	32

Table 7.2: Problem sizes used to test Wavebench's Gauss-Seidel algorithm.

Much like the LSA algorithm's runtime, the Gauss-Seidel algorithm runs up to 10x faster on NVIDIA's V100 GPU, as shown by Figure 7.10. Despite the similarity

Machine	ncell_x	ncell_y	ncell_z	ncomp
UDel Skywalker (V100)	64	64	64	64
NVIDIA PSG (V100)	128	64	64	64
NVIDIA PSG (P100)	64	64	64	64
NVIDIA PSG (K40)	64	64	64	32

Table 7.3: Problem sizes used to test Wavebench's radiation transport algorithm.

in runtime, this algorithm presented several challenges. First, it is a memory bound application, meaning that there is a significant amount of overhead when transferring data from the host to the GPU. It also performs a considerable amount of reads and writes to data stored in the GPU's global memory during execution. This helps explain the poor runtime on the K40 GPU. The K40 only offers 12GB of onboard GDDR5 memory. This may seem like a lot, but in the age of Big Data where more and more applications are data-driven, this becomes a bottleneck. GDDR5 is also an old memory architecture, so we sacrifice global read/write speeds on the device. There is simply not enough compute on this GPU to outweigh the cost of data transfer and onboard memory access.

However, the P100 and V100 GPUs tell a different story. In addition to providing a lot more computing power than the K40 offers, they feature HBM2 High-Bandwidth Memory, making onboard global memory accesses much faster. They also have a lot more onboard memory, 16GB and 32GB respectively, which allows us to examine a larger problem size on a single device. The combination of these factors is where we see the GPU's performance surpass that of a multicore CPU. In the case of the V100 on the PSG system, we have 32GB of onboard memory, allowing us to explore a huge problem size. This is where we see the best performance because with more data comes more parallelism. As described by Table 7.2, we explore a problem size that is 2.25x larger on the V100 than on the P100 PSG machine, and the GPU's performance scales accordingly. We notice that these nodes share the same 16-core CPU, but the performance seems to level off, as it does not scale as well as the GPU's performance. This algorithm is also more complex than the aforementioned LSA algorithm in that it has an in-gridcell component that requires computation for a group of local elements within each gridcell, instead of simply computing a single value for each cell. This is advantageous for a GPU because we can exploit hierarchical parallelism and run these computations at the thread level in addition to parallelizing the wavefront iterations at the thread block level. It is worth noting that simply parallelizing these in-gridcell computations does not result in enough parallelism to fully utilize the GPU and offset the cost of data transfer and memory accesses, as illustrated by the GPU results marked "no wavefront" in Figure 7.10. In fact, there is so little in-gridcell computation that we get a significant slowdown even compared to executing the original serial CPU code.

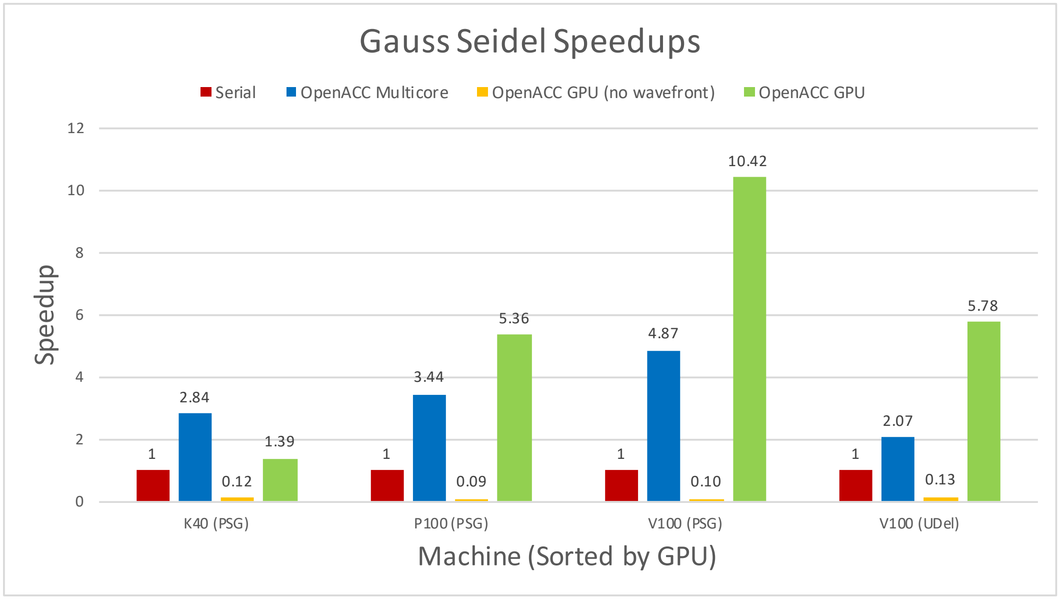


Figure 7.10: Gauss-Seidel speedups

Wavebench's radiation transport algorithm provides insight into the performance impact of our solution applied to a real-world application, not simply a generalized, widely-used algorithm. Like the Gauss-Seidel method, this radiation transport simulation is memory-bound, which presents a lot of the same challenges. However, there is a significant amount of additional complexity within each gridcell, as discussed in Section 7.1.3. The in-gridcell computation is multi-dimensional, presenting the opportunity for a lot more vector-level parallelism than found in the Gauss-Seidel method. In addition to the dimensional complexity, there is a series of not one, but three in-gridcell computations that are performed on the data found within a gridcell. This helps outweigh a lot of the memory overhead with computational demand and increased parallelism.

Figure 7.11 shows our radiation transport simulation's performance on the PSG systems. As described by Table 7.3, we explore a different problem size for each system, since we face the same challenge of limited onboard GPU memory. We double our problem size on the P100 GPU compared to the K40 GPU and again on the V100 GPU compared to the P100 GPU. Since the problem size we are able to explore on the K40 is relatively small, we see a modest 18x speedup. We also notice that we see the fastest CPU speedup on this configuration. The 10-core CPU on this node is actually clocked at a significantly faster speed (3.0 GHz) compared to the 16-core CPU (2.3 GHz) found in the P100 and V100 machines. Since this configuration also yields the least amount of parallelism, increasing the single-core performance with a faster clock yields some additional performance benefit.

However, the P100 and V100 configurations tell a different story yet again. In these cases, we explore a much larger problem size, and we are equipped with devices that have onboard HBM2 memory that is much faster. This, in conjunction with increased in-gridcell computational demands, outweighs the memory overhead of this algorithm by a significant margin. Here, we boast 68x and 97x speedups, respectively. It is worth noting that combining wavefront parallelism with in-gridcell parallelism still yields approximately 2x performance over simply parallelizing the in-gridcell component, despite the immense amount of computation done inside of each gridcell. This helps further demonstrate the importance of wavefront parallelism from a performance standpoint in these types of applications. 97x in a real-world setting is the difference between a simulation completing in an hour instead of 100 (over 4 days). The impact of this performance improvement is huge. It allows domain scientists to examine more simulation configurations in a shorter amount of time, meaning they can do more science overall. The benefit of our extension to a directive-based programming model, such as OpenACC, is that it also helps them cut down on their programming overhead, as well as minimizing the learning curve for scientists whose primary focus is not computer science.

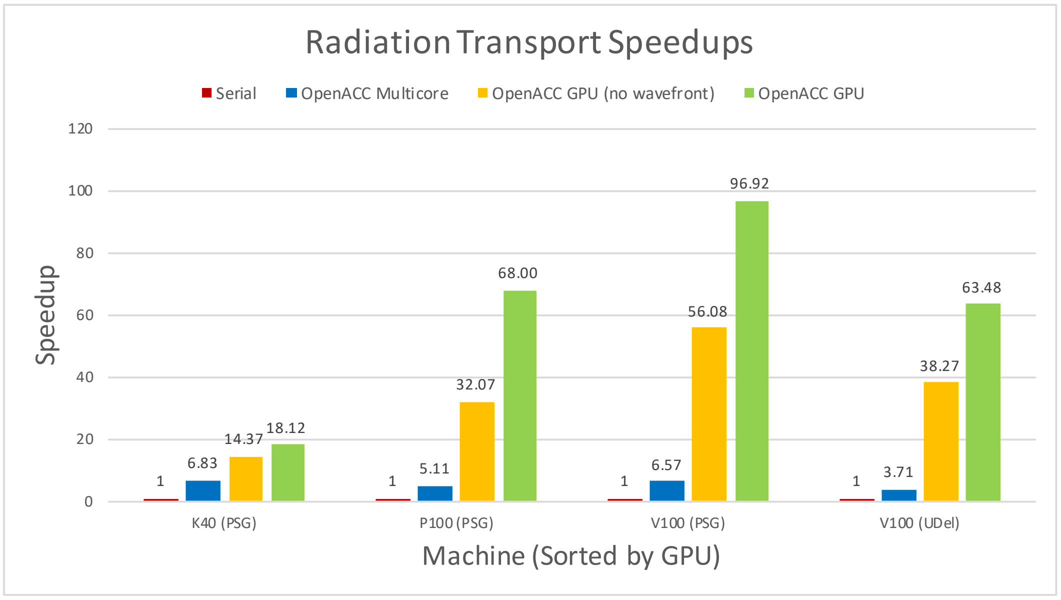


Figure 7.11: Radiation Transport speedups

7.3 Path Forward

Having applied the wavefront transformation in a real-world application in the author's prior work [100] while creating a formal representation, its supportive extension, and our Wavebench tool, the author plans to closely work with the standard organizations of both OpenMP and OpenACC and seek input on the standardization of a potential new *wavefront* directive. This will be a concrete contribution of the author's on-going research on wavefront parallel patterns.

Chapter 8

REAL-WORLD CASE STUDY: MINISWEEP

This chapter examines the real-world mini-application Minisweep and the impact that wavefront parallelism has on its performance. Sections 8.1 through 8.4.2 and 8.5.1 through 8.5.3 were contributed by Oak Ridge National Laboratory scientists Dr. Wayne Joubert and Dr. Oscar Hernandez in order to provide background on the application itself and the techniques that were previously used by ORNL to parallelize its wavefront sweep on ORNL's supercomputers. We show how OpenACC is used to parallelize and accelerate Minisweep on state-of-the-art HPC systems while maintaining a single codebase. This is all part of collaborative work published at The Platform for Advanced Scientific Computing (PASC) 2018 conference and the Computer Physics Communications (CPC) 2018 journal [99, 100].

The Minisweep proxy application [78] is part of the Profugus radiation transport miniapp project [6] that reproduces the computational pattern of the sweep kernel of the Denovo S_n radiation transport code [39]. The sweep kernel is responsible for most of the computational expense (80-99%) of Denovo. Denovo, a production code for nuclear reactor neutronics modeling, is in use by a current DOE INCITE project to model the International Thermonuclear Experimental Reactor (ITER) fusion reactor [7]. The many runs of this code required to perform reactor simulations at high node counts makes it an important target for efficient mapping to accelerated architectures.

This study involves S_n radiation transport algorithms for solving the linear Boltzmann equation [70]. Here, a continuum model is used to simulate the density of particles of a given energy and direction of motion within a 3-D volume. The approach yields a six dimensional problem (3-D in space, 2-D in angular particle direction and 1-D in particle energy) that is appropriately discretized in each dimension. Applications include neutronics calculations for nuclear reactor [4] and fusion reactor [7] design, radiation shielding, nuclear forensics and radiation detection. The large number of problem dimensions available in the S_n transport algorithm affords significant opportunities for parallelism on manycore parallel systems. However, the recursive nature of the wavefront calculation in the spatial dimensions is a challenge to efficient parallelization.

Denovo was one of six applications selected for early application readiness on ORNL's Titan system under the Center for Accelerated Application Readiness (CAAR) project [18] and is part of the Exnihilo code suite which received an R&D 100 award for modeling the Westinghouse AP1000 reactor [3]. Minisweep can be considered a successor to the well-known Sweep3D benchmark [1] and is similar to other S_n wavefront codes including KRIPKE [5], SN (Discrete Ordinates) Application Proxy (SNAP) [8] and PARTISN [19]. KRIPKE was developed at Lawrence Livermore National Laboratory (LLNL), while SNAP and PARTISN were developed at Los Alamos National Lab (LANL). PARTISN is an older (pre-2000) benchmark from LANL that also solves the Sn transport problem, but has a slightly different problem formulation compared to Minisweep which makes it less suitable for running on accelerators. Mapping this application to a GPU (or other type of accelerator) would most likely require a full rewrite.

In this thesis, Minisweep is used as a vehicle to examine parallelization of wavefront algorithms in general. However, it has multiple computational motifs (dense and sparse linear algebra, structured grids) and parallelism requirements (halo communications, hierarchical synchronizations, atomic updates) which make the study of this algorithm relevant to a much broader spectrum of codes. In addition to ORNL's existing parallel implementations of Minisweep (OpenMP and CUDA), this thesis presents a working OpenACC prototype that serves as a proof of concept for the strategy discussed in Chapters 5 and 6.

8.1 Overview of Sweep Algorithm

The S_n transport sweep algorithm possesses features common to wavefront algorithms in general yet has structure specific to the requirements of S_n transport. It can be considered in two parts: first, a wavefront algorithm relating the computations between gridcells of a 3-D grid, and second the computations performed on a single gridcell within this wavefront sweep across a grid.

8.1.1 Grid-level computations

We consider here a 3-D structured grid, with locally connected gridcells; see Figure 5.1. Importantly, the result computed at a gridcell is dependent on the results computed at the three neighbor gridcells in the upstream x, y and z directions; thus the computation is described by a four-point stencil. This dependency puts a restriction on the order in which results can be computed. One possible ordering is a series of wavefronts described by a sequence of planes of gridcells starting at a corner of the grid and sweeping through the whole grid (Figure 5.1). Other orderings are allowed as well, as long as the dependencies are satisfied.

To model the physical problem, requires modeling particle flux in all directions. To accomplish this, an execution instance of the algorithm performs a total of eight sweeps, one starting at each corner of the domain. These directions are referred to as "octants." The results of all eight octant sweeps are added together form the final result.

8.1.2 Gridcell-level computations

To further describe the algorithm, we define array v, which is dimensioned $v(n_x, n_y, n_z, n_u, n_e, n_a, n_o)$. The n_x, n_y, n_z dimensions refer to the spatial grid size. The dimension $n_o = 8$ is the octants axis, across which results are summed for the final result. The value n_a is a set of angular directions, and n_e is the number of energy groups, each representing a decoupled instance of the problem. Finally, n_u represents a set of unknowns for each gridcell based on the spatial discretization, e.g., finite element coefficients for a gridcell. Though v is relevant to key computations of the algorithm, the arrays that actually hold the input and output of the algorithm have the form $v'(n_x, n_y, n_z, n_u, n_e, n_m)$. Here v and v' are related by the fact that the n_a, n_o axes are compressed into n_m moments to form v' from v. The computation at a gridcell then is composed of the following steps:

1. **Moments-to-angles conversion** - For a given octant and energy group the input $v'_{in}(i_x, i_y, i_z, *, i_e, *)$ is transformed into array $v_{in}(i_x, i_y, i_z, *, i_e, *, i_o)$ by a small matrix-vector product that relates the n_m moments to the n_a angles. The matrix depends on the octant but is independent of spatial location, energy group and unknown.

2. **Face contribution** - The upstream components from the sweep are added to $v_{in}(i_x, i_y, i_z, *, i_e, *, i_o)$.

3. **Solve** - An operation is performed on the array values $v_{in}(i_x, i_y, i_z, *, i_e, *, i_o)$ which is coupled between the n_u unknowns but decoupled in all other dimensions. The result is $v_{out}(i_x, i_y, i_z, *, i_e, *, i_o)$.

4. **Face update** - The values of $v_{out}(i_x, i_y, i_z, *, i_e, *, i_o)$ are stored for use downstream by the sweep.

5. **Angles-to-moments conversion** - Array $v_{out}(i_x, i_y, i_z, *, i_e, *, i_o)$ is transformed and added to $v'_{out}(i_x, i_y, i_z, *, i_e, *)$ with another matrix-vector product with matrix depending on octant only.

8.1.3 Summary of problem axes

The problem dimensions and their respective couplings are thus summarized as follows:

- Space: in the x, y and z dimensions, each gridcell depends on results from three upstream cells, based on octant direction, resulting in a wavefront problem.

- Octant: results for different octants can be calculated independently, the only dependency being that results from different octants at the same entry of v'_{out} are added together and thus are a race hazard, depending on how the computation is done.

- Energy: computations for different energy groups have no coupling and can be considered separate problem instances, enabling flexible parallelization.

- Moment, angle: for fixed energy, octant and spatial location, the moments and angles are interrelated by small (dense) matrix-vector products.

- Unknown: when the problem is represented as angles, a computation is performed which may couple (only) the unknowns within the gridcell; for all other parts of the computation, elements on this axis are fully independent.

8.2 Parallelizing the Sweep Algorithm

To map the sweep algorithm to a parallel system, it is of paramount importance to minimize data motion as well as maximize parallelism, these being increasingly critical for high performance on exascale systems. As a result, the general guiding principle is that spatial dimensions must be the outermost loops, due to their sparse coupling, whereas moment, angle and unknown loops must be innermost due to the strong all-to-all couplings. The specific approach to parallelizing each axis is as follows:

Space: Spatial parallelism is based on the Koch-Baker-Alcouffe (KBA) algorithm [64]. Here the 3-D structured grid is decomposed to processors with a 2-D tiling in x and y (Figure 8.1). Each processor's part of the grid is decomposed into blocks along the z axis. Then a block wavefront process is applied starting at the corner block of the domain. Processors proceed in a series of parallel steps, with one block wavefront computed at each step and block face information communicated between consecutive steps. For a GPU or other accelerated processor, the KBA block described above is further decomposed into subblocks, and the computation is arranged into a series of subblock wavefronts, which are then mapped to parallel threads.

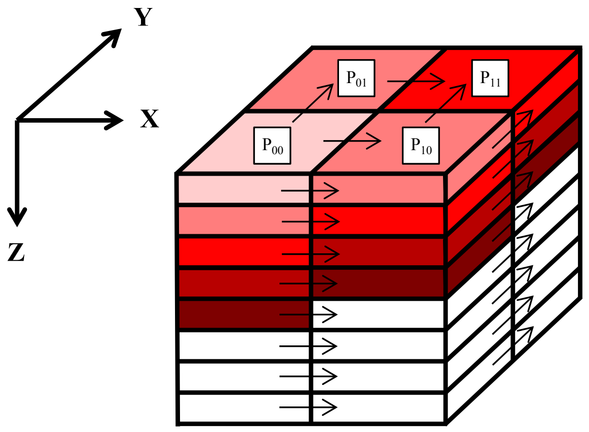


Figure 8.1: KBA parallel wavefront algorithm

Octant: Minisweep assigns compute threads to the eight wavefronts corresponding to the eight octant directions. The wavefronts are independent; however there is a potential race condition when two or more threads are updating the same KBA block on the same KBA block wavefront step. The solution used in Minisweep is a grid coloring approach. The KBA block is split in half along each dimension resulting in eight "semiblocks." Eight semiblock steps are taken, and for each step every one of the (up to) eight active wavefronts is assigned to a different semiblock. This is arranged so that the wavefront dependencies are satisfied. A synchronization and thread fence are required between consecutive semiblock steps.

Energy: Since the algorithm is embarrassingly parallel along the energy axis, this problem axis can be decomposed in any way: across nodes, across threads in a node, in a core or vector unit, or any combination.

Moment, Angle: The moment and angle axes are coupled to each other in an all-to-all fashion via two small matrix-vector products. The moment, angle, and unknown dimensions of the relevant arrays are ordered to be most rapidly varying, enabling efficient stride-1 memory access. When possible, the matrix-vector products are arranged to fit entirely within a vector unit. If n_a or n_m exceed the vector unit size, a blocking strategy is used with the computation fitting within the vector unit.

Importantly, the moments-to-angles transform is threaded in angle and the angles-to-moments transform is threaded in moment (otherwise a reduction across threads and/or vector lanes is required). Because threads must be reassigned between moment and angle threads, a synchronization and memory fence is required between these operations.

Unknown: No couplings exist along the unknowns axis when the small matrix-vector products are performed, therefore this axis permits some opportunity for threading here. However, for the inner solve computation in angle, each unknown may require different kinds of computations. To prevent poor use of vector units, these computations are kept serial.

8.3 Abstract Parallelism Model

Minisweep defines and implements a set of high level abstractions to describe the parallelism of its algorithm such that these abstractions can be used by any similar application implementing the sweep motif. The goal of these abstractions is to achieve productivity and performance portability across different architectures (GPUs, Xeon Phi, CPUs, etc). These abstractions can be instantiated using general purpose parallel programming languages like CUDA, OpenMP, and OpenACC. The need for these abstractions is also suggestive of possible shortcomings in parallel programming languages and suggests the need for extensions to support applications of this type.

The large number of problem dimensions inherent in S_n transport solves makes the need for managing thread parallelism axes and hierarchical memory places via use of abstraction layers acute. However, the techniques described here are applicable to many other problems requiring multidimensional parallelism, for example, batched dense linear algebra, block sparse linear solvers, and others.

Abstract machine model: Modern compute node hardware has an execution hierarchy. For example, a compute node may be composed of multiple GPUs, each with multiple cores possessing hardware threads and employing vector units composed of vector lanes. Some of these have co-located memories, for example node main memory, GPU high bandwidth memory or GPU shared memory associated with a streaming multiprocessor (SM) core. Execution threads are also associated with each level: for NVIDIA GPUs, in-warp threads execute in lock-step within a warp, in-threadblock threads are associated with an SM, and the thread grid is associated with the GPU. One can thus view a node as a hierarchy of execution units, memory places and compute threads, and in particular hardware threads can be thought of as indexed as a tuple depending on the location in the hierarchy. Threads also have characteristics based on location, e.g., thread synchronization across different cores of a node may be impossible or much slower compared to on-core synchronization. Likewise memories at different levels have different speeds, and thread access to memories may have NUMA effects depending on the level. Note that these concepts readily apply to heterogeneous node as well as homogeneous systems.

To abstract the characteristics of heterogeneous / homogeneous architectures, we define "place." A place consists of abstract executions units with local memories where threads can possibly synchronize with each other (e.g., barriers, memory fences) and access the memory. An architecture can be described as a set of "hierarchical places," where places can be nested in order to abstract the memory hierarchy of an architecture and its local execution units. In our abstract machine model, a nested place can access the memory of its parent place, but sometimes cannot synchronize with sibling places. We use the term "place threads" to refer to execution threads associated with each of the specific places. In Minisweep, place threads are created during execution via an adaptor function which instantiates or destroys the threads for the requested places in the underlying architecture; in practice, this is implemented for example by launching a CUDA kernel or entering an OpenMP parallel region, depending on programming language implementation on the given architecture.

Abstract arrays: In Minisweep, an "abstract (multidimensional) array" is defined as an object that consists of a list of dimensions and a base pointer associated with a memory place in the hierarchy. Array elements are accessed using a multiindex through an indexing function. In this way the memory layout is controlled by an abstraction layer that can be easily modified based on the architecture. An abstract array thus has a local view within the place it is allocated.

Abstract threads: Each independent variable of the science problem is assigned an abstract "threading axis" of abstract thread indices assigned to the corresponding problem axis. For example, the axis of n_e energy group values is assigned a set of (n_e or fewer) abstract compute thread indices used to compute those values. The collection of these abstract thread indices (which can be used to describe threads applied to the energy, octant, y location, z location, etc. problem dimensions) form a tuple or thread multiindex, which will be later bound to a place thread.

Instantiation of the abstract threaded region: A fundamental operation for multithreaded or accelerated codes is entry into a fork-join parallel region. In Minisweep this is abstracted as a multi-threaded region that instantiates the abstract threads. These regions can be nested and mapped to the same or different places.

Binding of abstract threads to places: A mapping of the abstract thread multiindex to a place thread multiindex is made based on the type of parallelism required. For example, since spatial wavefronts of the sweep algorithm require barriers between wavefronts, the spatial y and z dimensions must necessarily be mapped to threads within a place (e.g., threadblock on a GPU) that allows synchronization. By comparison, energy groups are fully decoupled, thus no restrictions are placed on where the abstract energy thread axis is mapped.

Parallel worksharing construct: This construct schedules work along problem axes to a set of abstract threads. and executes the work in parallel. The array index values for a given problem axis are distributed to the abstract thread indices via a block decomposition. For example, the full set of n_e energy groups is partitioned into blocks which are in turn assigned to abstract energy threads.

Table 8.1 describes the mapping of problem axes and associated abstract threads to the place thread hierarchy for the algorithm. It is evident that the ability to synchronize a subset of threads, akin to a barrier within an MPI sub-communicator, would be of benefit, since synchronization does not scale well to large thread counts.

Figure 8.2 is a simplified version S_n sweep parallelization using the abstract parallelism model. The pseudocode shows the allocation of the required arrays and definition of the hierarchical thread regions, followed by nested parallel loop over energy groups, serial loop over wavefronts, parallel loop over gridcells in the wavefront, and then the three threaded operations of moment-to-angles, solve and angles-to-moments.

problem dimension	dependency type	GPU threading	Intel Phi threading
energy	(none)	grid	OpenMP thread
octant	coloring	threadblock	CPU thread
spatial y	wavefront	threadblock	CPU thread
spatial z	wavefront	threadblock	CPU thread
moment	all-to-all	warp, serial	vector, serial
angle	all-to-all	warp, serial	vector, serial
unknown	all-to-all	warp, serial	vector, serial

Table 8.1: Problem dimensions mapping to thread hierarchy.

8.4 Translation of Abstract Parallelism Model

This section shows flavors of how different models, CUDA, OpenMP and OpenACC parallelize Minisweep. The narrative also discusses what we need (referring to Section 8.3) and what the models lack (Section 8.5).

8.4.1 CUDA

The main sweep function, `Sweeper_sweep()`, of Minisweep has a KBA pipeline loop to support the KBA block sweep calculations and related asynchronous face communication between nodes using MPI. For the CUDA case, faces and KBA blocks are also transferred to and from the GPU asynchronously; for systems not requiring offload, these calls do nothing. A mirrored array datatype, resembling the underlying mechanisms of OpenACC, maintains copies of an array on CPU and GPU and manages transfers; an accessor function returns the CPU or GPU pointer depending on where the computation takes place. For details, see [59].

```
// Abstract Arrays Allocation
AbstractArrayAlloc(vi(nx,ny,nz,ne,nm,nu): place_main)
AbstractArrayAlloc(vo(nx,ny,nz,ne,nm,nu): place_main)
AbstractArrayAlloc(neighbors(num_neighbors,ne,na,nu): place_main)
// Multithreaded Regions for Abstract Threads
Abstract_Threaded_Region(abstract_threads_e: place_main) {
Abstract_Threaded_Region(abstract_threads_a,
                    abstracts_thread_xy: place_local) {
// Do_All Parallel Worksharing
 Do_All(e in range(0,ne); abstract_thread_e) {
  AbstractArrayAlloc(vs(na,nu): place_local)
  do(w in range(0,w_max)) {
   // Do_All Parallel Worksharing
   Do_All((x,y) in wavefront(w); abstract_thread_xy) {
    z = z_coord(x,y,w)
    //Do_All Parallel Worksharing Matrix-Vector Product
    Do_All(a in range(0,na); abstract_thread_a) {
     do(u in range(0,nu)) { vs(a,u) = 0
      do(m in range(0,nm)) {
       vs(a,u) += a_from_m(a,m) * vi(x,y,z,e,m,u)}}
    } // end of Do_All(a)
    // Do_All Parallel Worksharing
    Do_All(a in range(0,na); abstract_thread_a) {
     // Apply upstream wavefront dependencies
     do(i neighbor of (x,y,z) in wavefront(w-1)) {
      do(u in range(0,nu)) { vs(a,u) -= neighbors(i,e,a,u) }}
     solve(vs,a) // Computation based on unknowns
     // Save downstream wavefront dependencies
     do(i neighbor of (x,y,z) in wavefront(w+1)) {
      do(u in range(0,nu)) { neighbors(i,e,a,u) = vs(a,u) }}
    } // end of Do_All(a)
    //Do_All Parallel Worksharing Matrix-Vector Product
    Do_All(m in range(0,nm); abstract_thread_m) {
      do(u in range(0,nu)) { vo(x,y,z,e,m,u) = 0
       do(a in range(0,na)) {
        vo(x,y,z,e,m,u) += m_from_a(a,m) * vs(a,u) }}}
   } // end of wavefront loop w
  AbstractArrayFree(vs)
  } // end of Do_All(e)
}} // end of Abstract_Threaded_Regions
AbstractArrayFree(vi, vo, neighbors)
```

Figure 8.2: Abstract representation of sweep algorithm

The key kernel operation of Minisweep is the block sweep operation, in function `Sweeper_sweep_block()`. This is launched as a CUDA kernel or alternatively is initiated as a parallel region in OpenMP, as described above in the abstract model. Since energy groups are independent, the energy thread axis is mapped off-threadblock into the GPU thread grid; all other axes are mapped in-threadblock due to coupling requirements as described earlier.

8.4.2 OpenMP

OpenMP is used to run Minisweep natively on a multicore or manycore processor with OpenMP 3.1 parallel directives and the OpenMP 4.0 `simd` directive. The model also implements KBA. OpenMP runs with a single thread of execution until the block sweep function is encountered, at which point threads are spawned in energy, octant and the y and z spatial dimensions. The temporary arrays placed in GPU shared memory for the CUDA case are now CPU arrays, with one part of the array reserved for each compute thread.

Since the OpenMP model uses a SIMD loop rather than thread numbers to access vector lanes, loops are placed in the code for the angle, moment and unknown dimensions, and each of these dimensions is assigned only a single thread; for the CUDA case, however, these axes receive multiple threads and the SIMD for loop is removed. The OpenMP port models the particle flux in all directions, i.e. performs a total of eight sweeps using the tasks concept. Note: We have not ported the code to OpenMP4.5 as part of this work. We believe that OpenMP4.5 could be an alternate solution to explore minisweep on GPUs.

8.4.3 OpenACC

Our OpenACC implementation effort consists of two parts: parallelizing the initialization of faces at the beginning of the sweep and parallelizing the in-gridcell computations.

```
/*--- Loop over wavefronts ---*/
for (wavefront = 0; wavefront < num_wavefronts; wavefront+=1) {

   /*---KBA threading---*/
#pragma acc loop independent gang, collapse(2)
     for( iy=0; iy<dim_y; ++iy )
     for( ix=0; ix<dim_x; ++ix ) {

       int iz = wavefront - (ix + iy);
       if (iz >= 0 && iz <= wavefront && iz < dim_z) {

                /*---moments to angles---*/
#pragma acc loop independent vector, collapse(3)
         for( ie=0; ie<dim_ne; ++ie )
                for( iu=0; iu<NU; ++iu )
                for( ia=0; ia<dim_na; ++ia ) {
                   P result = (P)0;
#pragma acc loop seq
                   for( im=0; im < dim_nm; ++im )
                    { /*---moments to angles conversion---*/ }
                }

         /*---solve---*/
         #pragma acc loop independent vector, collapse(2)
         for( ie=0; ie<dim_ne; ++ie )
                for( ia=0; ia<dim_na; ++ia )
                 { /*---solve calculation---*/ }

                /*---angles to moments---*/
#pragma acc loop independent vector, collapse(3)
         for( ie=0; ie<dim_ne; ++ie )
                for( iu=0; iu<NU; ++iu )
                for( im=0; im<dim_nm; ++im ) {
#pragma acc loop seq
                   P result = (P)0;
                   for( ia=0; ia<dim_na; ++ia )
                    { /*---angles to moments conversion---*/ }
         }
       }
     }
  }
```

Figure 8.3: Sweep loop nest with OpenACC annotations

Parallelization of the face initializations involves five nested loops (spatial decomposition of gridcells, unknowns within gridcell, energy groups and angles). However, it is worth noting that PGI's OpenACC compiler only provides us with two levels of parallelism currently: gang (block) and vector (thread). The compiler is yet to thoroughly exploit the worker level of parallelism. So, in order to map a loop nest of five loops onto the accelerator to achieve full parallelization, we utilized OpenACC's `collapse` clause to collapse a specified number of nested loops into one large loop, which we can then map at either the gang or vector level. For Minisweep's face initializations, we collapse the outer three loops (corresponding to the unknowns and two spatial dimensions of the gridcells) and execute at the gang level. We also collapse the inner two loops (corresponding to the energy groups and angles) and execute at the vector level.

Parallelization of the in-gridcell computations in Minisweep is not as trivial, as there are data dependencies between gridcells, as mentioned in Section 8.1.1. To that end, we utilize the KBA parallel sweep algorithm (discussed in Section 8.2) in order to exploit *gang*-level parallelism across the x, y, and z gridcell loops. Since there is currently no existing high-level language that provides functionality for implementing this type of parallel sweep, the programmer must modify the loop nest manually in order to achieve the desired behavior. This involves creating an outer *wavefront* loop that iterates over the wavefront decomposition, as discussed in Section 8.1.1 and shown in Figure 5.1. The computations within these wavefronts can be parallelized, albeit not trivially. First we must parallelize across the inner two dimensions: y and x. This spawns a number of threads on the GPU. Within each of these threads, we calculate our z value based off of the thread's y and x values and the wavefront iteration number. Then, we can perform a bounds check to determine whether that z value is within the bounds of the wavefront being examined (denoted by the current wavefront iteration number). This allows us to exploit parallelism across gridcells, while still accounting for data-dependencies between wavefront iterations.

The embarrassingly parallel in-gridcell computations are performed for each energy group within each gridcell. Within each of these groups, there lies a series of in-gridcell computations that must be performed for the given energy group in the gridcell in question, all of which are embarassingly parallel. We mark these computations for execution at the *vector* level. A representation of the result is shown in Figure 8.3. Note that this code snippet is also the serial code if one were to simply remove all the directives.

Transport algorithm sweeps are particularly challenging. After parallelizing this sweep kernel using only two levels of parallelism, we faced an additional obstacle. Minisweep doesn't run just a single sweep each timestep; it runs eight. Each of these sweeps represents a different *octant*, which refers to the direction the sweep iterates through the 3-dimensional simulation space. Each octant starts at a different corner of the space and iterates diagonally to the opposing corner. This complicates our calculation of the z dimension value, since we have to consider the direction that a given sweep is moving in along each axis, as well as the bounds check to make sure that the calculated z value lies along the current wavefront. If that isn't already a daunting enough task, these *octant* sweeps are also parallelizable, but OpenACC doesn't provide us another explicit layer of parallelism. Our solution to this predicament is to use asynchronous parallel regions within our *octant* loop. This effectively will launch kernels on the GPU asynchronously, which allows us to overlap computation across octants. This asynchronous behavior provides us with a third level of parallelism that we can use to saturate the GPU for a longer period of time, yielding optimal performance.

8.4.4 MPI Domain Decomposition

The goal of the domain scientists who use Minisweep is to explore the largest 3-dimensional space possible. This poses a challenge because we are limited by the amount of memory present on a single node or device. We can overcome this limitation by decomposing our simulation's spatial component across nodes and/or devices using MPI. Details about the data synchronization required to resolve dependencies between spatial blocks are presented in Section 8.4.1. In short, we decompose Minisweep's simulation domain across spatial dimensions (corresponding to gridcell-level computation). A sub-component of the problem space is allocated on each device, which is bound to a single MPI rank. This includes the collection of gridcells that a given MPI rank is responsible for computing, as well as the few neighbors that it may need to read data from (despite not computing). After each wavefront iteration, a synchronization is used to update the neighbors that lie along the edge of the rank's portion of the simulation space. This incurs a small amount of overhead that is outweighed by the computational benefit of utilizing many accelerators.

Minisweep allows the user to control the behavior of this decomposition by providing two command line flags: *nproc_x* and *nproc_y*. The product of the values passed using these flags should equal the number of MPI ranks used. By using these additional arguments, we are able to decompose our problem across two spatial axes instead of one, and the user is given the ability to control to what extent the simulation is decomposed along each axis. For example, if we use 4 MPI ranks, we have the option to decompose either the x or y axis across 4 ranks, or we can set both *nproc_x* and *nproc_y* to 2, which decomposes the simulation across both axes simultaneously. When using larger simulation configurations and a larger number of MPI ranks, we can play with the *nproc_x* and *nproc_y* values to decompose the simulation in other ways and observe the impact on Minisweep's performance. An example of this would be using 16 MPI ranks and setting *nproc_x* to 8 and *nproc_y* to 2, or vice-versa. For the purposes of this thesis, we stick to an even decomposition across both axes in our experimental setup.

8.5 Programming Model Limitations

This section dives into the obstacles created by limitations in the programming models that were explored. Section 8.5.1 presents general challenges that were faced due to the structure and design of Minisweep's code. Sections 8.5.2, 8.5.3, and 8.5.4 then explore model-specific challenges faced when using CUDA, OpenMP, and OpenACC, respectively. These obstacles serve as additional motivation regarding the need for extensions to existing programming models.

8.5.1 General

In all cases, inadequacies of current compilers required that some code be rewritten in an unnecessarily low-level fashion to obtain correctness and/or performance. This seems to be a systemic challenge, insofar as it is difficult for compiler teams to develop mature and performant compilers for frequently changing complex processor hardware. Programming models support vectorization in different ways, leading to portability challenges. CUDA treats vector lanes as threads, whereas OpenMP uses SIMD loops and OpenACC has a `vector` clause for parallel loops. Such differences can lead to increased use of undesirable `ifdef`s if it is required to support these multiple programming models. Developers would prefer a single highly performant programming model with a high level of abstraction targeting all architectures rather than the need to use multiple programming models.

The Minisweep code requires in several places a thread synchronization or barrier over only a subset of threads. A barrier across fewer threads could potentially run much faster in current hardware. This feature is not currently supplied by any of the programming models, though in principle a barrier across a subset of OpenMP threads could be written, and the new CUDA 9 Cooperative Groups feature may be useful here.

The Minisweep design makes it easy to change the mapping of machine threads to abstract problem threads and problem dimensions. A more challenging goal is to allow easy modification of the execution hierarchy. Such a design would allow easy loop order permutation and other loop restructuring operations, loop blocking to optimize cache use or reduce loop overheads, and on-demand reassignment of loop axes either to parallel threads or alternatively serial execution. Such changes generally require motion of significant portions of code, e.g., to optimize for loop invariant quantities. Presently this must be done by hand, and is not directly supported by programming models or imperative programming languages as currently conceived. Likewise, the use of accessor functions in Minisweep permits easy modification of memory locale and layout for an array. One must still however schedule memory transfers across the memory hierarchy manually for peak performance. Automatic transfers via paging/caching such as the CUDA Unified Memory feature and similar functionality for Intel Phi on-package memory will simplify programming for this, however past experience has shown that manual prefetching of data across the hierarchy is sometimes necessary to attain high performance. As memory layers proliferate, e.g., with inclusion of NVRAM, managing this will become more challenging.

8.5.2 CUDA

CUDA by nature provides a lower level programming model compared to directives-based methods. Though the CUDA runtime API provides a slightly higher abstraction level than the CUDA driver API, both cases require `ifdef`s to make a code portable between CUDA for GPUs and standard C/C++ for conventional architectures. CUDA has the advantage that vector lanes are addressed explicitly as threads, resulting in reliable vectorization. However, certain coding constructs can lead to losses in performance in unexpected ways. For example, in the course of developing the Denovo sweeper and Minisweep, it was observed that when loop bounds were passed into a CUDA kernel within a `struct`, performance was noticeably degraded compared to when passed in as scalars. Furthermore, in some cases a `for` loop that was provably one-trip at compile time ran slower than when the loop was altogether removed, necessitating use of an `ifdef` to make a single CUDA / OpenMP-SIMD code. CUDA additionally has limitations with repect to deep copy of structs and classes—since pointers in a host struct are invalid on the device—though this is improving with the support of GPU Unified Memory. In short, limitations of this nature can make it challenging to raise the abstraction level in CUDA codes and maintain performance portability with other platforms.

8.5.3 OpenMP

Intel Phi performance typically depends on the effective vectorization of loops, using the native `simd` directive or alternatively the OpenMP `simd` directive. In the process of porting Minisweep to the Intel Phi using the Intel compiler, challenges to loop vectorization were encountered. In one case an array accessor function needed to be flattened by removing its use of a `struct` in order to enable the loop to vectorize. In another case the compiler failed to remove a provably loop invariant quantity from a loop, inhibiting vectorization. Also, the compiler would not vectorize the outermost loop of a deep loop nest, though CUDA had no problem threading this loop. The differing treatment of vector lanes as threads by CUDA and by SIMD loops in OpenMP required the undesirable use of special case code to handle the differences. Also, CUDA generally favors larger kernels to minimize kernel launch overhead and maximize data reuse, whereas with the Intel compiler it is difficult or impossible to vectorize large, complex loops in one piece. These differences made it challenging to support the different platforms without special case programming. Overall, the difficulty of predicting a priori when a complex loop would vectorize and the need at times to rewrite code at a lower abstraction level was detrimental to writing maintainable, performance portable code.

We also explored converting current the OpenMP 3.1 code to 4.5 (only within the scope of the ideas presented and not with respect to porting the full code to 4.5 to use GPUs). Adapting *doacross* for this type of wavefront problem would have been a potential direction to take. However, *doacross* assumes a flat memory hierarchy (shared memory), but instead, what we need for our type of case study is to map data objects to a memory hierarchy (e.g. place and child place) that would allow the wavefront computations to be more data-centric and be scheduled where the data is.

8.5.4 OpenACC

Similarly to OpenMP, we faced a number of challenges when implementing our parallelization strategy discussed in Section 8.4.3 in OpenACC. We used PGI's

18.4 community edition compiler, and bugs were reported to PGI's team. The first issue was handling array accesses in the original Minisweep implementation. Accessor functions are used to calculate the address of the flattened array accesses that occur throughout the `Sweeper_sweep` function, as described in Section 8.3. These functions returned the address of the array access in question, which was then dereferenced by the `Sweeper_sweep` function in order to perform the manipulation on the array element. OpenACC requires that we use the `routine` directive to convert these function calls to routines. However, the compiler was unable to properly generate routine code for functions utilizing external variables like these array accesses do. The solution to this was to simply eliminate the use of these functions and inline the calculation of the array address into the array accesses within the given loops, resulting in a more traditional array access. Unfortunately this is detrimental to efforts to raise the abstraction level of the code.

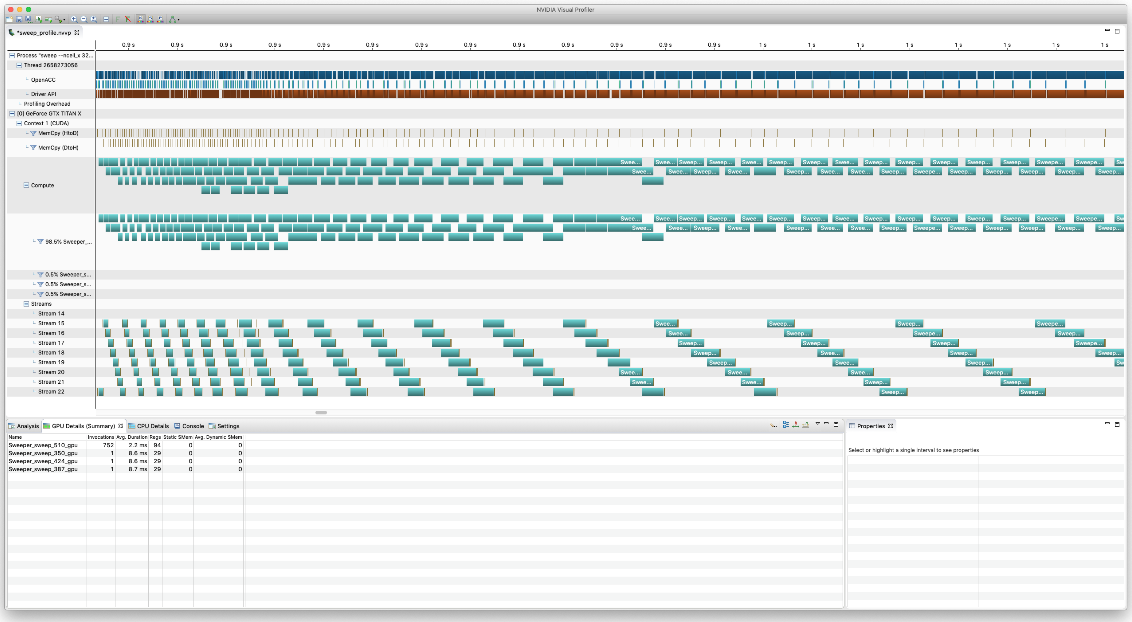
Another issue was related to loop bounds. In Minisweep, input parameters are stored in a globally defined struct. Since these values are representing the sizes of each dimension of the application, they are used later as loop bounds. However, while parallelizing, OpenACC does not assume that no aliasing is being done since this struct is defined globally (out of scope). There are two simple solutions to this issue. First, the compiler flag `-Msafeptr` can be used to specify that there is no aliasing. However, this would not be the best option for this application as there is some sort of pointer aliasing present elsewhere. Instead, we simply extract the value of the dimension being used for a given loop bound and store it in an integer variable prior to the start of the accelerated loop.

The final issue we faced was identified as a compiler bug in PGI OpenACC 18.4. OpenACC can use `kernels` or `parallel` to generate code from an accelerated region. With `kernels`, the onus is on the compiler to check for dependencies and generate code, whereaas with `parallel`, the onus is on the programmer; failure to do so will result in inaccurate results. In our case, we observed that even though we used the `parallel` directive and marked a loop nest for collapsing and parallelizing, the compiler still performed dependency checks as if we had used `kernels` directive. We confirmed this behavior by parallelizing a loop with a known dependency. The compiler generated parallel code but showed incorrect results. We then added a collapse clause to the end of this loop directive, and we specified that it should be collapsed with the next loop in the loop nest, which contained no such dependencies. The result here was that the compiler reported that it parallelized this collapsed loop nest, but the results of the computation were accurate. Due to the increased runtime, we were able to conclude that the inner loop was indeed executing in parallel, but the outer loop was executing in serial despite what the compiler had reported. All of these issues were easily overcome in practice, but identifying them presented significant challenges along the way.

8.6 Profiling Minisweep

In order to validate the strategies discussed in Section 8.4.3 that we employed to accelerate Minisweep using OpenACC, we analyzed the application's execution using PGI's profiling tool *pgprof*. Figures 8.4 and 8.5 show different stages of Minisweep's execution. Our attention is mainly focused on the behavior of the compute regions (light blue bars) and the two rows preceding the compute region breakdown, which show the associated data transfer times from host to device and device to host, respectively. Fortunately, these are barely visible, as we are keeping the vast majority of our data on the GPU throughout Minisweep's entire execution.

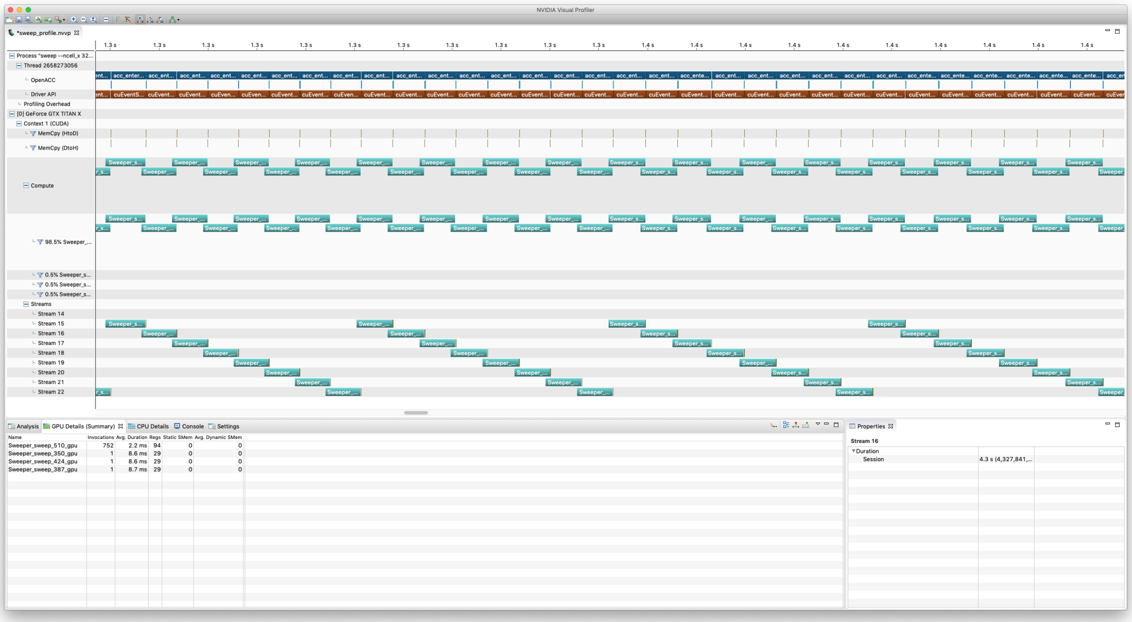
Figure 8.4 shows the beginning of Minisweep's execution. As discussed throughout Chapter 8, we are iterating over the diagonal wavefronts of a three-dimensional space in this radiation transport simulation. As the wavefronts progress from a single corner of the space toward the center, they grow in size. Since more and more gridcells are being examined, the computational cost grows accordingly. This is reflected in the profiler's visualization, as we see the compute regions taking more and more time to execute as the execution of the application progresses. The execution time of a single compute region eventually plateaus once the GPU is being fully utilized, as shown in Figure 8.5.



Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem
Sweeper_sweep_510_gpu	752	2.2 ms	94	0	0
Sweeper_sweep_350_gpu	1	8.6 ms	29	0	0
Sweeper_sweep_424_gpu	1	8.6 ms	29	0	0
Sweeper_sweep_387_gpu	1	8.7 ms	29	0	0

Select or highlight a single interval to see properties

Figure 8.4: Examining Minisweep's execution using *pgprof*. The beginning of execution is shown here.



Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem
Sweeper_sweep_510_gpu	752	2.2 ms	94	0	0
Sweeper_sweep_350_gpu	1	8.6 ms	29	0	0
Sweeper_sweep_424_gpu	1	8.6 ms	29	0	0
Sweeper_sweep_387_gpu	1	8.7 ms	29	0	0

Stream 16

▼Duration	
Session	4.3 s (4,327,841,...

Figure 8.5: Examining Minisweep's execution using *pgprof*. The most compute intensive portion of execution is shown here.

We are also running eight sweeps in parallel using OpenACC's *async* directive, as discussed in Section 8.4.3. We can see the eight corresponding asynchronous CUDA streams at the bottom of the profiler's visual output. As shown in Figure 8.4, there is a lot of overlap between these streams at the beginning of execution. Since a single sweep isn't examining enough gridcells to sature the GPU, we are able to launch all of our octant sweeps asynchronously. As each sweep progresses through the three-dimensional space being examined, the size of its wavefront grows, thus increasing its computational requirement. To that end, each sweep utilizes more and more of the GPU, resulting in less and less asynchronous overlap between streams. Eventually, we reach the most computationally intensive portion of the space: the middle. Figure 8.5 shows each stream utilizing enough of the GPU that there is almost no asynchronous overlap. Eventually, each stream's sweep will progress to the opposite end of the space, resulting in a decrease of its active wavefront size. We then see the compute regions begin to shrink, and the asynchronous CUDA streams begin to overlap quite a bit again, mirroring the beginning of Minisweep's execution shown in Figure 8.4.

8.7 Evaluation & Results

Machine	CPU	NVIDIA GPU
NVIDIA PSG (V100)	2x Intel Xeon E5-2698 v3 (16 cores)	4x V100 (16GB HBM2)
NVIDIA PSG (P100)	2x Intel Xeon E5-2698 v3 (16 cores)	4x P100 (16GB HBM2)
NVIDIA PSG (K40)	2x Intel Xeon E5-2690 v2 (10 cores)	K40 (12GB GDDR5)
ORNL Titan	AMD Opteron 6274 (16 cores)	K20X (6GB GDDR5)
ORNL Summitdev	2x IBM POWER8 (10 cores)	4x P100 (16GB HBM2)
ORNL Summit	2x IBM POWER9 (21 cores)	6x V100 (16GB HBM2)
ORNL Percival	Intel KNL 7230 (64 cores)	N/A

Table 8.2: Specifications of the nodes in the systems we used to test different configurations of Minisweep.

As a validation of portability, Table 8.3 shows Minisweep results for one GPU of the Titan Cray XK7 system (CUDA), one GPU of the Summitdev IBM Minsky system (CUDA) and one node of the Percival Cray XC40 KNL system (self-hosted OpenMP 4.0). The problem solved has $n_e = 64$, $n_a = 32$, and $n_u = 4$, with $n_x, n_y, n_z =$

32 The codes are not fully optimized, in particular one of the inner loops for the OpenMP-KNL case did not vectorize. However, all cases across different hardware and software environments attained a similar 4-5% of peak flop rate, a typical figure for this algorithm which has significant memory accesses, register usage and integer index calculations. This result suggests that the code is in fact performance portable, since reasonable performance is reached for all systems.

System	Cores (SMs)	GF/s peak	GF/s	% peak GF/s
Titan(K20X)	14	1311	55.9	4.26
Summitdev(P100)	56	5312	244.8	4.61
Percival(Phi7230)	64	2662	124.9	4.69

Table 8.3: Comparative performance on several platforms.

We evaluate the effectiveness of our abstract wavefront parallelism model by comparing the runtimes of our parallel implementations of Minisweep (described in Section 8.4) to the runtime of a serial version of the code on multiple HPC systems. Table 8.2 describes the hardware available on nodes of each system. Note that the NVIDIA Professional Service Group (PSG) machines and the ORNL Titan machine are existing state-of-the-art HPC systems, while ORNL Summitdev is a development cluster representative of the hardware that is now present on nodes in ORNL's next-gen supercomputer *Summit* [67]. We also utilized the PSG cluster's V100 nodes, which house NVIDIA's next-generation GPU that are present on nodes in *Summit*. We used PGI's 18.4 compiler to compile our OpenACC and OpenMP. We have also used GCC 6.3.0 and ICC 17.0 for OpenMP codes. Compiling the code using Intel's OpenMP compiler was not successful and required code restructuring to take advantage of SIMD in minisweep.

Our experimental configuration is a representative example of what a real run of Minisweep within the Denovo radiation transport code looks like. Our problem dimensions on a single node are designed to be as large as we can fit on a single GPU: $n_e = 64$, $n_a = 32$, and $n_u = 4$, with $n_x, n_y, n_z = 32$, on K20x/K40 and $n_x, n_y, n_z = 64$

on P100/V100. For MPI runs, we ran across 4 nodes on NVIDIA's PSG cluster, which are each equipped with 4 GPUs. To that end, our in-gridcell sizes for n_e, n_a, and n_u remain unchanged, but we explore a larger 3-dimensional space using $n_x, n_y, n_z = 128$ on NVIDIA's PSG system.

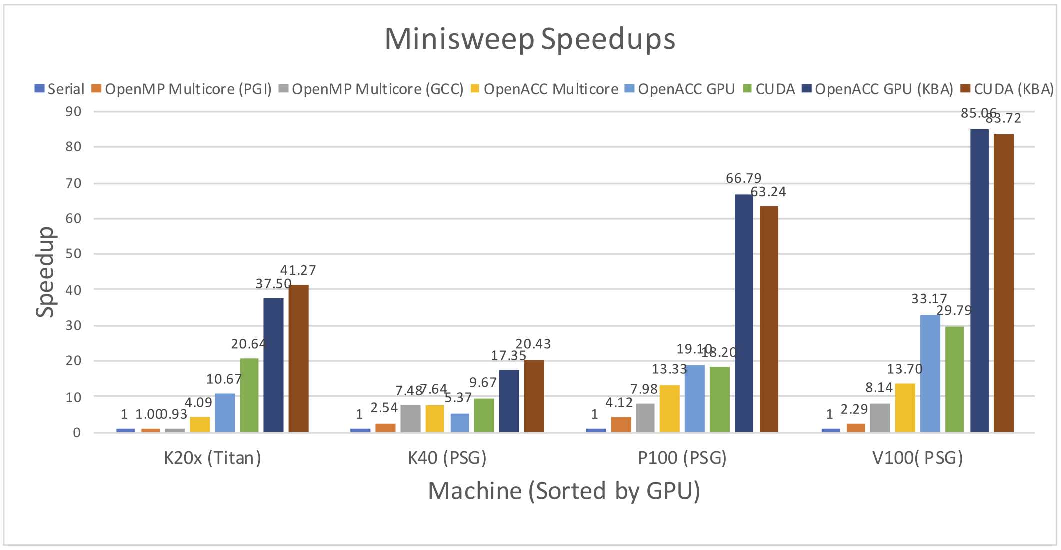


Figure 8.6: Minisweep's speedups over serial using different runtime configurations sorted according to each machine's GPU. Note that the CUDA version is parallelized along the same dimensions as the OpenACC GPU configuration. The corresponding KBA configurations utilize the KBA blocking method for additional parallelism across spatial dimensions.

Figure 8.6 presents the results when running different implementations of Minisweep using our single node configuration in the form of speedups over the baseline serial implementation on existing HPC systems. Note that the speedup results presented were obtained by calculating the average of a series of runs for each implementation. There are a few notable results. First, our multicore CPU GCC's OpenMP (3.1) and OpenACC implementations yield favorable speedups. Note that GCC's OpenMP performed better than PGI's OpenMP. As mentioned in Section 8.4.3, we have currently parallelized the in-gridcell computations, as well as the spatial decomposition utilizing the KBA parallel sweep algorithm to resolve data dependencies, as discussed in Section 8.1.1. This implementation boasts a larger speedup than our OpenMP GCC version, as well as our CUDA configuration when parallelized over the same problem dimensions. Our OpenACC KBA configurations yield an addition layer of parallelism across spatial dimensions and show a much larger speedup compared to configurations which only execute in-gridcell computations in parallel. This leads us to conclude that there is additional performance to be gained, albeit not trivial to implement. It is also worth noting that our OpenACC implementation running on NVIDIA's next-generation Volta GPU boasts an 85.06x speedup over serial code, which is larger than the 83.72x speedup over the same serial implementation achieved by CUDA. This supports our claim that our extension to existing high-level programming models is worthwhile, both from a performance standpoint, as well as a programming productivity standpoint. Currently, without major code modification, this challenge cannot be overcome.

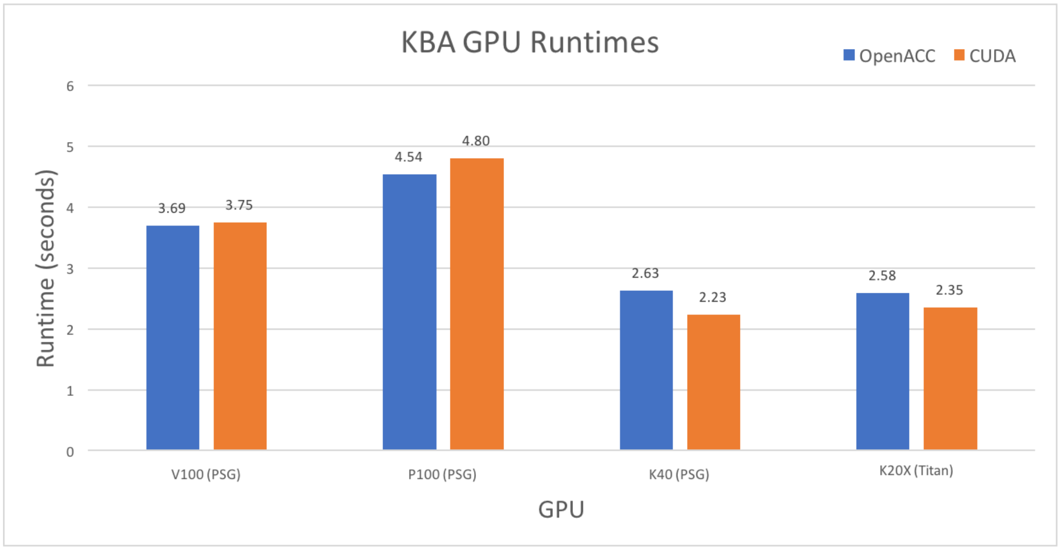


Figure 8.7: Absolute runtimes (measured in seconds) of OpenACC and CUDA experiments on all GPUs used. Note that the V100/P100 problem size is an order of magnitude larger than the K40/K20x configuration, as mentioned earlier.

Absolute runtimes for GPU configurations utilizing the KBA parallel sweep algorithm are presented in Figure 8.7. As shown, our OpenACC GPU implementation performs well compared to its CUDA counterpart in all cases. In addition to its excellent GPU performance, it is worth noting that this same OpenACC implementation was used to obtain results on our multicore CPU platforms by simply recompiling and specifying a different target. No additional code modifications were necessary to achieve this demonstration of portability. We contend that this provides additional evidence for the importance of an extension that would allow us to parallelize the outer spatial dimensions, yielding additional parallelism across gridcells without requiring a major coding effort on the part of the programmer. As stated in Section 8.3, this type of abstraction will benefit any wavefront-type code that performs some type of spatial dimension sweep. Our OpenACC proof-of-concept of such an abstraction demonstrates this on a real-world wavefront-type application used at a major national laboratory. Since these types of codes are very common in computational scientific applications, we contend that this contribution has far-reaching implications for modern-day HPC applications.

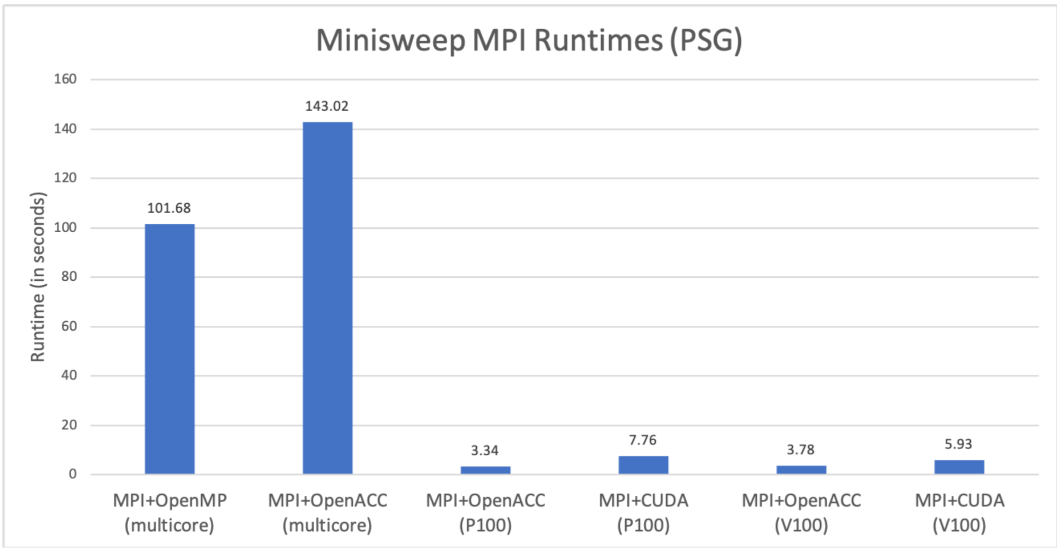


Figure 8.8: Minisweep's runtimes when running on 4 nodes (each with 4 GPUs) using 16 MPI ranks (1 rank per GPU). Lower is better. Note that the runtimes of OpenACC and CUDA are comparable even when run on multiple nodes. This reinforces our conclusions drawn from Figure 8.6.

Figure 8.8 takes the developed strategy a step further by introducing the additional layer of MPI communication into the KBA sweep component of the code. Using MPI, we can decompose the spatial domain across nodes, and in the case where a GPU architecture is targeted, across devices within a node. Here, we observe similar behavior to the results shown in Figure 8.6, as it relates to runtimes of our different configurations. It is worth noting that since we changed over from a single-directional sweep to Minisweep's true multi-directional octant sweeping method using asynchronous parallel regions, we lose a bit of performance when compiling for a multicore CPU target. This is due to the fact that PGI's OpenACC compiler will parallelize *gang* loops by default when compiled with a multicore target. The optimal configuration for Minisweep would be to actually parallelize our in-gridcell loops (currently annotated as *vector* parallel) because we have enough parallelism inside our gridcells to fully saturate most CPUs. Since we are unable to do this currently, we see OpenMP outperforming our OpenACC configuration. However, our GPU configurations yield very fruitful results when run across all 16 GPUs across 4 PSG nodes using 16 MPI ranks. Each rank is bound to a different GPU using the *acc_set_device_num* function. Here, we see OpenACC continuing to outperform CUDA, which serves as evidence that our OpenACC configuration will scale very well on larger, modern HPC systems.

We ran our GPU configurations of Minisweep on Summit itself. Constraints in accessing the system have limited the ability to collect complete results, hence we have not added our preliminary findings.

Scientifically, neutron flow simulation can take a considerable amount of computing time. If not for speeding up these simulation runs using accelerators, scientists will have to resort to simulating a model that is potentially less accurate leading to questionable results. This can be quite a risky task to rely on for scientists working in nuclear reactor facilities.

Chapter 9

CONCLUSION & FUTURE WORK

From single-core processors to the massively parallel supercomputers of today, the evolution of computing hardware has necessitated the development of new programming models in order to exploit all the computing power that state-of-the-art hardware has to offer. The rise of parallelism and heterogeneity has fundamentally reshaped the way we think about programming. We don't know exactly what the machines of tomorrow will look like hardware-wise, but we do know from a programming standpoint that parallelism will only increase. Although we don't know what hardware will be present in it, Oak Ridge National Laboratory is expected to have its first exascale machine, dubbed *Frontier*, operational just around the corner in 2021 [68]. The uncertainty of the future is a motivating factor behind the need for portable, abstract programming models that allow developers to maintain a single codebase across legacy, existing, and future hardware architectures.

This thesis presents motivation for the continued development of such models in Chapter 4. Section 4.1 presents results that dispel the myth that high-level programming models cannot compete with hardware-specific, low-level languages from a performance standpoint. Sections 4.2 and 4.3 provide insight from the applications side of computing. More specifically, the applications examined in the associated projects greatly benefit from utilizing state-of-the-art HPC hardware, but they faced obstacles during development due to the lack of support for complex parallel behaviors in high-level programming models that were available at the time.

Chapter 5 presents a complex parallel pattern, called wavefront, that is commonly found in real-world scientific applications across domains including (but not limited to) linear systems solvers, genetic sequencing, and nuclear physics. Chapter 6 presents an abstraction and designs an extension to existing high-level programming models, such as OpenACC, that can be used to represent wavefront behavior in order to achieve performance portability across HPC systems. Chapters 7 and 8 then evaluate the developed abstraction using different types of wavefront algorithms, as well as a real-world application, Minisweep, containing wavefront parallelism in its core compute region. The results presented in both chapters support the efficacy of the developed abstraction by showing favorable performance improvements on a variety of state-of-the-art HPC hardware, while maintaining a single codebase.

The techniques presented by this thesis are not unique to wavefront parallelism. Complex parallel patterns are present in almost all large applications, and they will have to be dealt with by application developers if those applications are to be ported to existing and next-generation HPC systems. Up until this point, many high-level programming models were playing catch-up with the evolution of state-of-the-art hardware architectures. For example, OpenMP was created in 1997 as a shared memory multiprocessing API [106], but it has since evolved to support accelerator offloading since 2015, starting with its 4.5 specification [83]. The on-node, high-level programming model of focus in this thesis, OpenACC, was introduced in 2012 with accelerator offloading and heterogeneous systems in mind from the start [105]. However, the OpenACC standard still does not support certain types of increasingly popular parallel architectures, such as ARM processors and FPGAs, and it lacks support for more complex parallel behavior, such as wavefront.

Moving forward, it is important to develop programming models that facilitate parallelism, while remaining platform-agnostic. Existing high-level models like OpenMP and OpenACC make certain assumptions about hardware. For example, OpenACC only provides three clauses with which to express parallelism in a piece of code: *gang*, *worker*, and *vector*. Given that most GPUs already support two or three levels of parallelism due to their hardware layout, it is not unreasonable to imagine a future parallel architecture that contains additional levels of parallelism. We already see an example of this in the form of ORNL's *Summit* supercomputer, which contains six NVIDIA GPUs per node. This can be thought of as a fourth layer of on-node parallelism. To that end, high-level programming models need to become more abstract in the way they describe parallelism within a piece of code. It is also important that the programming models of the future support more and more complex behavior. As the wavefront parallel pattern shows us, parallel applications are becoming increasingly complex. Simply providing methods of parallelizing traditional-style loop nests and embarassingly-parallel computational algorithms is not going to be good enough in the long term. The best way to extend existing programming models and facilitate the development of robust, high-level programming models in the future is to begin looking at more complex parallel patterns found across a range of scientific applications. This is an ongoing, and potentially neverending challenge.

BIBLIOGRAPHY

[1] The ASCI SWEEP3D README file. 1995. http://www.ccs3.lanl.gov/PAL/software.shtml [Online; accessed 24-June-2014].

[2] Power 4: The first multi-core, 1ghz processor. https://www.ibm.com/ibm/history/ibm100/us/en/icons/power4/, 2001.

[3] Scientists successfully test code that models neutrons in reactor core. *R&D Magazine*, 2014. https://www.rdmag.com/news/2014/02/scientists-successfully-test-code-models-neutrons-reactor-core.

[4] Supercomputer team wins award for core work. *World Nuclear News*, 2014. . http://www.world-nuclear-news.org/NN-Supercomputer-team-wins-award-for-core-work-0707147.html [Online; accessed 15-August-2017].

[5] Kripke. 2017. . https://codesign.llnl.gov/kripke.php [Online; accessed 15-August-2017].

[6] Ornl-cees. https://github.com/ORNL-CEES/Profugus, 2017.

[7] Quadrillions of calculations per second for fusion. *ITER Newsline*, 2017. https://www.iter.org/ajax/www/pop/wd_700/lang_/urldepth_0/id_newsline_ofinterest-670 [Online; accessed 15-August-2017].

[8] Snap: Sn (discrete ordinates) application proxy. 2017. . https://github.com/lanl/SNAP [Online; accessed 15-August-2017].

[9] Boosting industry with openacc. https://www.olcf.ornl.gov/2018/05/29/boosting-industry-with-openacc/, 2018.

[10] top500: November 2018. https://www.top500.org/lists/2018/11/, 2018.

[11] A. M. Mustafa Al-Bakri and Hussein Hussein. Static analysis based behavioral api for malware detection using markov chain. 2014.

[12] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L Steele Jr, Sam Tobin-Hochstadt, Joao Dias, Carl Eastlund, et al. The fortress language specification. *Sun Microsystems*, 139(140):116, 2005.

[13] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. Graph-based malware detection using dynamic analysis. *J. Comput. Virol.*, 7(4):247–258, November 2011.

[14] James A Ang, Richard F Barrett, Robert E Benner, D Burke, C Chan, J Cook, David Donofrio, Simon D Hammond, Karl S Hemmert, SM Kelly, et al. Abstract machine models and proxy architectures for exascale computing. In *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014*, pages 25–32. IEEE, 2014.

[15] Timothy G Armstrong, Justin M Wozniak, Michael Wilde, and Ian T Foster. Compiler techniques for massively scalable implicit task parallelism. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 299–310. IEEE, 2014.

[16] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[17] Christopher Baker, Gregory Davidson, Thomas M Evans, Steven Hamilton, Joshua Jarrell, and Wayne Joubert. High performance radiation transport simulations: preparing for titan. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–10. IEEE, 2012.

[18] Christopher G. Baker, Gregory G. Davidson, Thomas M. Evans, Steven P. Hamilton, Joshua J. Jarrell, and Wayne Joubert. High performance radiation transport simulations: Preparing for TITAN. In *Proceedings of Supercomputing Conference SC12*, 2012.

[19] Randal S. Baker. Partisn on advanced/heterogeneous processing systems. 2014. `http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-13-20948` [Online; accessed 24-June-2014].

[20] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 225–234, New York, NY, USA, 2008. ACM.

[21] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, , R. J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, , , M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2):276–292, Feb 2005.

[22] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 917–924, New York, NY, USA, 2003. ACM.

[23] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[24] Sunita Chandrasekaran and Guido Juckeland. *OpenACC for Programmers: Concepts and Strategies.* Addison-Wesley Professional; 1 edition, 2017.

[25] Sunita Chandrasekaran, Shilpa Shanbagh, Ramkumar Jayaraman, Douglas L Maskell, and Hui Yan Cheah. C2fpga?a dependency-timing graph design methodology. *Journal of Parallel and Distributed Computing*, 73(11):1417–1429, 2013.

[26] Chun Chen. Polyhedra scanning revisited. *ACM SIGPLAN Notices*, 47(6):499–508, 2012.

[27] Chun Chen, Jacqueline Chame, and Mary Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 111–122, Washington, DC, USA, 2005. IEEE Computer Society.

[28] Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. Technical report, Technical Report 08-897, U. of Southern California, 2008.

[29] L. Chen, X. Huo, and G. Agrawal. Accelerating mapreduce on a coupled cpu-gpu architecture. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov 2012.

[30] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 115–126, New York, NY, USA, 2010. ACM.

[31] Ron Choy and Alan Edelman. Parallel matlab: Doing it right. *Proceedings of the IEEE*, 93:331–341, 2005.

[32] M. P. Clay, D. Buaria, and P. K. Yeung. Improving scalability and accelerating petascale turbulence simulations using openmp. http://openmpcon.org/conf2017/program/, 2017. To Appear.

[33] Cristian Coarfa, Yuri Dotsenko, and John Mellor-Crummey. Experiences with sweep3d implementations in co-array fortran. *The Journal of Supercomputing*, 36(2):101–121, 2006.

[34] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[35] Eline De Cuypere, Koen De Turck, Herwig Bruneel, and Dieter Fiems. Optimal inventory management in a fluctuating market. pages 158–170, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[36] Anthony Desnos. Androguard - reverse engineering, malware and goodware analysis of android applications. https://github.com/androguard/androguard, 2011.

[37] Antonio J Dios, Angeles G Navarro, Rafael Asenjo, Francisco Corbera, and Emilio L Zapata. A case study of the task-based parallel wavefront pattern. In *PARCO*, pages 65–72, 2011.

[38] Neva Durand, Saad Shamim, Ido Machol, Suhas Rao, MiriamH. Huntley, Eric Steven Lander, and ErezLieberman Aiden. Juicer provides a one-click system for analyzing loop-resolution hi-c experiments. *Cell Systems*, 3:95–98, 07 2016.

[39] T. M. Evans, W. Joubert, S. P. Hamilton, S. R. Johnson, J. A Turner, G. G. Davidson, and T. M Pandya. Three-Dimensional Discrete Ordinates Reactor Assembly Calculations on GPUs. In *ANS MC2015Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method, Nashville, TN, American Nuclear Society, LaGrange Park, 2015*, 2015.

[40] Rob Farber. *Parallel Programming with OpenACC*. Newnes, 2016.

[41] Matteo Frigo. A fast fourier transform compiler. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 169–180, New York, NY, USA, 1999. ACM.

[42] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, AISec '13, pages 45–54, New York, NY, USA, 2013. ACM.

[43] Roberto Gomperts. Quantum Chemistry on GPUs. http://images.nvidia.com/content/tesla/pdf/quantum-chemistry-may-2016-mb-slides.pdf, 2016.

[44] Kazushige Goto and Robert van de Geijn. On reducing tlb misses in matrix multiplication, 2002.

[45] R Govindarajan and Jayvant Anantpur. Runtime dependence computation and execution of loops on heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10. IEEE Computer Society, 2013.

[46] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Autotuning a high-level language targeted to gpu codes. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10, May 2012.

[47] Scott Grauer-Gray, William Killian, Robert Searles, and John Cavazos. Accelerating financial applications on the gpu. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 127–136, New York, NY, USA, 2013. ACM.

[48] S. Z. Guyer and C. Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE*, 93(2):342–357, Feb 2005.

[49] Jesse D. Hall, Nathan A. Carr, and John C. Hart. Cache and bandwidth aware matrix multiplication on the gpu, 2003.

[50] S. Han, F. Franchetti, and M. Pschel. Program generation for the all-pairs shortest path problem. In *2006 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 222–232, Sep. 2006.

[51] Louise Harewood, Kamal Kishore, Matthew D Eldridge, Steven Wingett, Danita Pearson, Stefan Schoenfelder, V Peter Collins, and Peter Fraser. Hi-c as a tool for precise detection and characterisation of chromosomal rearrangements and copy number variation in human tumours. *Genome biology*, 18(1):125, 2017.

[52] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A mapreduce framework on graphics processors. In *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 260–269, Oct 2008.

[53] Michael T Heath and Charles H Romine. Parallel solution of triangular systems on distributed-memory multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 9(3):558–588, 1988.

[54] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. Mapcg: Writing parallel program portable between cpu and gpu. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 217–226, New York, NY, USA, 2010. ACM.

[55] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 78–88, Oct 2011.

[56] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, February 2004.

[57] Accelerated Strategic Computing Initiative. The asci sweep3d benchmark code, 1995.

[58] W. Jiang and G. Agrawal. Mate-cg: A map reduce-like framework for accelerating data-intensive computations on heterogeneous clusters. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 644–655, May 2012.

[59] Wayne Joubert. Minisweep. https://github.com/wdj/minisweep, 2017.

[60] Wayne Joubert, Richard K. Archibald, Mark A. Berrill, W. Michael Brown, Markus Eisenbach, Ray Grout, Jeff Larkin, John Levesque, Bronson Messer, Matthew R. Norman, and et al. Accelerated application development: The ornl titan experience. *Computers and Electrical Engineering*, 46, May 2015.

[61] W. Kahan. Gauss-seidel methods of solving large systems of linear equations. PhD Thesis, University of Toronto, 1958.

[62] Michio Katouda and Takahito Nakajima. Mpi/openmp hybrid parallel algorithm of resolution of identity second-order mllerplesset perturbation calculation for massively parallel multicore supercomputers. *Journal of Chemical Theory and Computation*, 9(12):5373–5380, 2013. PMID: 26592275.

[63] Kenneth R Koch, Randal S Baker, and Raymond E Alcouffe. Solution of the first-order form of the 3-d discrete ordinates equation on a massively parallel processor. *Transactions of the American Nuclear Society*, 65(108):198–199, 1992.

[64] K.R. Koch, R.S. Baker, and R.E. Alcouffe. Solution of the first-order form of the 3-D discrete ordinates equation on a massively parallel processor. *Transactions of the American Nuclear Society*, 65:198–199, 1992.

[65] Manaschai Kunaseth, David Richards, James Glosli, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta. Analysis of scalable data-privatization threading algorithms for hybrid mpi/openmp parallelization of molecular dynamics. *The Journal of Supercomputing*, 66:406–430, 10 2013.

[66] Adam J Kunen, Teresa S Bailey, and Peter N Brown. Kripke-a massively parallel transport mini-app. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2015.

[67] Oak Ridge National Lab. Summit. https://www.olcf.ornl.gov/summit/, 2017.

[68] Oak Ridge National Lab. Frontier: Olcfs exascale future. https://www.olcf.ornl.gov/2018/02/13/frontier-olcfs-exascale-future/, 2018.

[69] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, 1974.

[70] Edward W. Larsen and Jim E. Morel. Advances in discrete-ordinates methodology. In Yousry Azmy and Enrico Sartori, editors, *Nuclear Computational Science: A Century in Review*, chapter 1, pages 1–84. Springer, New York, 2010.

[71] John M. Levesque, Ramanan Sankaran, and Ray Grout. Hybridizing s3d into an exascale application using openacc: An approach for moving to multi-petaflops and beyond. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 15:1–15:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[72] Ruipeng Li and Yousef Saad. GPU-accelerated preconditioned iterative linear solvers. *Journal of Supercomputing*, 63(2):443–466, February 2013. http://link.springer.com/article/10.1007%2Fs11227-012-0825-3 [Online; accessed 24-June-2014].

[73] Sherry Li and James W. Demmel. Superlu_dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Math. Softw.*, 29:110–140, 06 2003.

[74] Xiaoming Li, María Jesús Garzarán, and David Padua. A dynamically tuned sorting library. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 111–, Washington, DC, USA, 2004. IEEE Computer Society.

[75] Xiaoming Li, Maria Jesus Garzaran, and David Padua. Optimizing sorting with genetic algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 99–110, Washington, DC, USA, 2005. IEEE Computer Society.

[76] Yixun Liu, Eddy Z. Zhang, and Xipeng Shen. A cross-input adaptive framework for gpu program optimizations. *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10, 2009.

[77] John Mellor-Crummey and John Garvin. Optimizing sparse matrix-vector product computations using unroll and jam. *Int. J. High Perform. Comput. Appl.*, 18(2):225–236, May 2004.

[78] Bronson Messer, Ed D'Azevedo, Judy Hill, Wayne Joubert, Mark Berrill, and Christopher Zimmer. Miniapps derived from production hpc applications using multiple programing models. *The International Journal of High Performance Computing Applications*, 0(0):1094342016668241, 2016.

[79] Josh Milthorpe, V Ganesh, Alistair P Rendell, and David Grove. X10 as a parallel language for scientific computation: Practice and experience. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1080–1088. IEEE, 2011.

[80] Gihan R Mudalige, Mary K Vernon, and Stephen A Jarvis. A plug-and-play model for evaluating wavefront computations on parallel architectures. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–14. IEEE, 2008.

[81] Matthew Norman, Jeffrey Larkin, Aaron Vose, and Katherine Evans. A case study of cuda fortran and openacc for an atmospheric climate kernel. *Journal of Computational Science*, 9:1 – 6, 2015. Computational Science at the Gates of Nature.

[82] Akira Nukada and Satoshi Matsuoka. Auto-tuning 3-d fft library for cuda gpus. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 30:1–30:10, New York, NY, USA, 2009. ACM.

[83] OpenMP. Openmp 4.5 specs. https://www.openmp.org/uncategorized/openmp-45-specs-released/, 2015.

[84] Matthew Otten, Jing Gong, Azamat Mametjanov, Aaron Vose, John M. Levesque, Paul F. Fischer, and Misun Min. An mpi/openacc implementation of a high-order electromagnetics solver with gpudirect communication. *IJHPCA*, 30:320–334, 2016.

[85] Shawn D Pautz. An algorithm for parallel sn sweeps on unstructured meshes. *Nuclear Science and Engineering*, 140(2):111–136, 2002.

[86] Simon J Pennycook, Simon D Hammond, Gihan R Mudalige, Steven A Wright, and Stephen A Jarvis. On the acceleration of wavefront applications using distributed many-core architectures. *The Computer Journal*, 55(2):138–153, 2011.

[87] S.J. Pennycook, S.D. Hammond, G.R. Mudalige, S.A. Wright, and S.A. Jarvis. On the acceleration of wavefront applications using distributed many-core architectures. *The Computer Journal*, 55(2):138–153, 2012. http://comjnl.oxfordjournals.org/content/55/2/138 [Online; accessed 24-June-2014].

[88] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, , F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, Feb 2005.

[89] M. Rempel. Extension of the muram radiative mhd code for coronal simulations. *apj*, 834:10, January 2017.

[90] M. Rempel, M. Schüssler, and M. Knölker. Radiative magnetohydrodynamic simulation of sunspot structure. *apj*, 691:640–649, January 2009.

[91] François Roddier and Claude Roddier. Wavefront reconstruction using iterative fourier transforms. *Applied Optics*, 30(11):1325–1327, 1991.

[92] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. A programming language interface to describe transformations and code generation. In Keith Cooper, John Mellor-Crummey, and Vivek Sarkar, editors, *Languages and Compilers for Parallel Computing*, pages 136–150, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[93] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[94] Edans Flavius de O Sandes and Alba Cristina MA de Melo. Smith-waterman alignment of huge sequences with gpu in linear space. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1199–1211. IEEE, 2011.

[95] Sunil Sathe. Accelerating the ANSYS fluent r18.0 radiation solver with openacc. http://on-demand.gputechconf.com/supercomputing/2016/presentation/sc6126-sunil-sathe-accelerating-ansys-fluent-radiation-solver-openacc.pdf, 2016.

[96] Will Sawyer, Guenther Zaengl, and Leonidas Linardakis. Towards a multi-node openacc implementation of the icon model. In *EGU General Assembly Conference Abstracts*, volume 16, 2014.

[97] R. Searles, S. Herbein, and S. Chandrasekaran. A portable, high-level graph analytics framework targeting distributed, heterogeneous systems. In *2016 Third Workshop on Accelerator Programming Using Directives (WACCPD)*, pages 79–88, Nov 2016.

[98] R. Searles, L. Xu, W. Killian, T. Vanderbruggen, T. Forren, J. Howe, Z. Pearson, C. Shannon, J. Simmons, and J. Cavazos. Parallelization of machine learning applied to call graphs of binaries for malware detection. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 69–77, March 2017.

[99] Robert Searles, Sunita Chandrasekaran, Wayne Joubert, and Oscar Hernandez. Abstractions and directives for adapting wavefront algorithms to future architectures. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '18, pages 4:1–4:10, New York, NY, USA, 2018. ACM.

[100] Robert Searles, Sunita Chandrasekaran, Wayne Joubert, and Oscar Hernandez. Mpi+ openacc: Accelerating radiation transport mini-application, minisweep, on heterogeneous systems. *Computer Physics Communications*, 2018.

[101] Robert Searles, Stephen Herbein, Travis Johnston, Michela Taufer, and Sunita Chandrasekaran. Creating a portable, high-level graph analytics paradigm for compute and data-intensive applications. In *International Journal of High Performance Computing and Networking*, volume 10, Jan 2017.

[102] John Shalf. Computer architecture for the next decade: Adjusting to the new normal for computing. *EEHPC Workshop*, 2013.

[103] K. Shirahata, H. Sato, and S. Matsuoka. Hybrid map task scheduling for gpu-based heterogeneous clusters. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 733–740, Nov 2010.

[104] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[105] OpenACC Specification. https://www.openacc.org/specification. Accessed: 2019-02-19.

[106] OpenMP Specification. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf. Accessed: 2017-06-30.

[107] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, Jonathan Freeman, and Barbara Kreaseck. Combining performance aspects of irregular gauss-seidel via sparse tiling. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 90–110. Springer, 2002.

[108] Michelle Mills Strout, Geri Georg, and Catherine Olschanowsky. Set and relation manipulation for the sparse polyhedral framework. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 61–75. Springer, 2012.

[109] Madhava Syamlal, Chris Guenther, Aytekin Gel, and Sreekanth Pannala. High performance computing: clean coal gasifier designs using hybrid parallelization. 01 2011.

[110] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. Automating wavefront parallelization for sparse matrix computations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 41. IEEE Press, 2016.

[111] A. Vögler, S. Shelyag, M. Schüssler, et al. Simulations of magneto-convection in the solar photosphere. equations, methods, and results of the muram code. *aap*, 429:335–351, January 2005.

[112] Richard Vuduc, James W Demmel, and Katherine A Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16:521–530, jan 2005.

[113] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 1, pages 579–586, July 2009.

[114] Gerhard Wellein, Georg Hager, Thomas Zeiser, Markus Wittmann, and Holger Fehske. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In *Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International*, volume 1, pages 579–586. IEEE, 2009.

[115] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1):3 – 35, 2001. New Trends in High Performance Computing.

[116] Adrianto Wirawan, Kwoh Chee Keong, and Bertil Schmidt. Parallel dna sequence alignment on the cell broadband engine. In *International Conference on Parallel Processing and Applied Mathematics*, pages 1249–1256. Springer, 2007.

[117] Michael Wolfe. Loops skewing: The wavefront method revisited. 15:279–293, 1986.

[118] Justin M Wozniak, Timothy G Armstrong, Michael Wilde, Daniel S Katz, Ewing Lusk, and Ian T Foster. Swift/t: Large-scale application composition via distributed-memory dataflow processing. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 95–102. IEEE, 2013.

[119] Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. Spl: A language and compiler for dsp algorithms. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 298–308, New York, NY, USA, 2001. ACM.

[120] Lifan Xu, Wei Wang, Marco A. Alvarez, John Cavazos, and Dongping Zhang. Parallelization of shortest path graph kernels on multi-core cpus and gpus. 2013.

[121] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *LCPC*, volume 10, pages 172–187. Springer, 2009.

[122] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In Mirosław Kutyłowski and Jaideep

Vaidya, editors, *Computer Security - ESORICS 2014*, pages 163–182, Cham, 2014. Springer International Publishing.

[123] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 63–76, New York, NY, USA, 2003. ACM.

[124] Peiheng Zhang, Guangming Tan, and Guang R Gao. Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform. In *Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications: held in conjunction with SC07*, pages 39–48. ACM, 2007.

Appendix
RELATED RESOURCES

This appendix provides additional resources related to the applications discussed in this thesis. The Wavebench tool's source code, discussed in Chapter 7, can be found on GitHub at `https://github.com/rsearles35/Wavebench`. The Minisweep mini-application, discussed in Chapter 8, can also be found on GitHub. The original code exists in Dr. Wayne Joubert's GitHub page at `https://github.com/wdj/minisweep`, and the OpenACC version of the code can be found on the author's GitHub page at `https://github.com/rsearles35/minisweep`. The OpenACC code will be merged into the original repository.

Appendix
PERMISSIONS

Chapters 4 and 8 are based off of several papers I have authored, which are published under four different publishers: ACM, IEEE, Inderscience, and Elsevier. Below is a list of the associated publications, the organizations that published them, and the Chapters of this thesis containing information from them:

- (Chapter 4, IEEE) S. Grauer-Gray, L. Xu, **R. Searles**, S. Ayalasomayajula, and J. Cavazos, "*Auto-tuning a High-Level Language Targeted to GPU Codes,*" at Innovative Parallel Computing (InPar) 2012, (San Jose, CA, USA) [46]

- (Chapter 4, ACM) S. Grauer-Gray, W. Killian, **R. Searles**, and J. Cavazos, "*Accelerating Financial Applications on the GPU,*" in Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, GPGPU-6, (New York, NY, USA), pp. 127136, 2013 [47]

- (Chapter 4, IEEE) **R. Searles***, S. Herbein*, and S. Chandrasekaran, "*A Portable, High-Level Graph Analytics Framework Targeting Distributed, Heterogeneous Systems,*" in Proceedings of the Third Workshop on Accelerator Programming Using Directives (WACCPD '16). IEEE Press, Piscataway, NJ, USA, 79-88 [97]

- (Chapter 4, Inderscience) **R. Searles**, S. Herbein, T. Johnston, M. Taufer, S. Chandrasekaran, "*Creating a Portable, High-Level Graph Analytics Paradigm For Compute and Data-Intensive Applications,*" at International Journal of High Performance Computing and Networking, IJHPCN 2017 Vol. 10. DOI: 10.1504/IJHPCN.2017.10007922 [101]

- (Chapter 4), IEEE) **R. Searles**, L. Xu, W. Killian, T. Vanderbruggen, T. Forren, J. Howe, Z. Pearson, C. Shannon, J. Simmons, and J. Cavazos , "*Parallelization of Machine Learning Applied to Call Graphs of Binaries for Malware Detection,*" at 25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2017, St. Petersburg, Russia, 2017 [98]

- (Chapter 8, ACM) **R. Searles**, S. Chandrasekaran, W. Joubert, O. Hernandez, "*Abstractions and Directives for Adapting Wavefront Algorithms to Future Architectures,*" at The Platform for Advanced Scientific Computing, PASC 2018. DOI: 10.1145/3218176.3218228 [99]

- (Chapter 8, Elsevier) **R. Searles**, S. Chandrasekaran, W. Joubert, O. Hernandez, "*MPI + OpenACC: Accelerating Radiation Transport Mini-Application, Minisweep, on Heterogeneous Systems,*" in Computer Physics Communications, CPC 2018. DOI: 10.1016/j.cpc.2018.10.007 [100]

All of these publishers allow authors to reuse their own work. Their policies are as follows:

- IEEE: "The IEEE shall grant authors and their employers permission to make copies and otherwise reuse the material." [1]

- ACM: Authors have the right to "reuse of any portion of the Work, without fee, in any future works written or edited by the Author, including books, lectures and presentations in any and all media." [2]

- Inderscience: Authors can use "the article in further research and in courses that the Author is teaching and Incorporating the article content in other works by the Author." [3]

[1] https://www.ieee.org/publications/rights/copyright-policy.html

[2] https://www.acm.org/publications/policies/copyright-policy#permanent%20rights

[3] https://www.inderscience.com/mobile/inauthors/index.php?pid=74

- Elsevier: “Authors can include their articles in full or in part in a thesis or dissertation for non-commercial purposes.” [4]

[4] https://www.elsevier.com/about/policies/copyright/permissions