ACCELERATING SCIENCE WITH DIRECTIVE-BASED PROGRAMMING ON HETEROGENEOUS MACHINES AND FUTURE TECHNOLOGIES

by

JACOB B. LAMBERT

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Division of Graduate Studies of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

September 2021

DISSERTATION APPROVAL PAGE

Student: Jacob B. Lambert

Title: Accelerating Science with Directive-Based Programming on Heterogeneous Machines and Future Technologies

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science by:

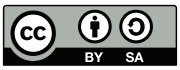
Allen Malony	Chair
Boyana R. Norris	Core Member
Hank R. Childs	Core Member
Seyong Lee	Core Member
Josef Dufek	Institutional Representative

and

Andy Karduna	Interim Vice Provost for Graduate Studies

Original approval signatures are on file with the University of Oregon Division of Graduate Studies.

Degree awarded September 2021



DISSERTATION ABSTRACT

Jacob B. Lambert

Doctor of Philosophy

Department of Computer and Information Science

September 2021

Title: Accelerating Science with Directive-Based Programming on Heterogeneous Machines and Future Technologies

Accelerator-based heterogeneous computing has become the de facto standard in contemporary high-performance machines, including upcoming exascale machines. These heterogeneous platforms have been instrumental to the development of computation-based science over the past several years. However, this specialization of hardware has also led to a specialization of associated heterogeneous programming models that are often intimidating to scientific programmers and that may not be portable or transferable between different platforms. Directive-based programming offers one high-level alternative to specialized code, but also introduces its own set of challenges. Many accelerators, like FPGAs, may not support a directive-based approach, and others like GPUs and CPUs may selectively support standards. In this dissertation we perform the necessary research required to further enable directive-based computing to consistently accelerate science on heterogeneous platforms. This research takes the form of three major projects: (1) an OpenACC-to-FPGA framework developed to bring FPGAs under the umbrella of directive-based computing, (2) an OpenACC and OpenMP interoperable framework designed to improve the portability and performance of directive-based standards across different platforms, and (3) an

exploration of exascale-intended platforms with directive-based applications. This dissertation includes previously published and co-authored material, as well as unpublished co-authored material.

CURRICULUM VITAE

NAME OF AUTHOR: Jacob B. Lambert

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
University of Tennessee, Knoxville, TN, USA
Technical University of Denmark, Kongens Lyngby, Denmark

DEGREES AWARDED:

Doctor of Philosophy, Computer and Information Science, 2021, University of Oregon
Bachelor of Arts, Computer Science, 2015, University of Tennessee

AREAS OF SPECIAL INTEREST:

High Performance Computing
Heterogeneous Programming Models
Computational Science

PROFESSIONAL EXPERIENCE:

Research Collaborator/Affiliate, Oak Ridge National Laboratory Future Technologies Group, 2016-present, Advisor: Seyong Lee, Jeffrey Vetter

Graduate Teaching Assistant, University of Oregon, Advisor: Allen Malony, 2015, 2016

Research Assistant, Oak Ridge National Lab Electrical and Electronics Systems Research Division, Advisor: Mark Buckner, 2013

Student Researcher, Eco-Informatics Summer Institute (EISI), 2013

Student Researcher, National Institute for Mathematical Biological Synthesis (NIMBioS), 2013

GRANTS, AWARDS AND HONORS:

Research Collaboration Grant, Oak Ridge National Laboratory, 2019

PUBLICATIONS:

Lambert, J., Lee, S., Vetter, J. S., and Malony, A. D. (2021). Optimization with the OpenACC-to-FPGA Framework on the Arria 10 and Stratix 10 FPGAs. *Journal of Parallel Computing (PARCO)*

Cabrera, A., Young, A., and Lambert, J. (2021). Towards Evaluating High-Level Synthesis Portability and Performance Between Intel and Xilinx FPGAs. *International Workshop on OpenCL and SYCL (IWOCL)*

Lambert, J., Lee, S., Vetter, J. S., and Malony, A. D. (2020). CCAMP: An Integrated Translation and Optimization Framework for OpenACC and OpenMP. *International Conference on Supercomputing (SC).*

Lambert, J., Lee, S., Vetter, J. S., and Malony, A. (2020). In-depth optimization with the OpenACC-to-FPGA framework on an Arria 10 FPGA. *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (pp. 460-470). IEEE.

Lambert, J., Lee, S., Vetter, J. S., and Malony, A. D. (2019). CCAMP: OpenMP and OpenACC Interoperable Framework. *European Conference on Parallel Processing (EuroPAR)*

Lambert, J., Lee, S., Kim, J., Vetter, J. S., and Malony, A. D. (2018). Directive-Based, High-Level Programming and Optimizations for High-Performance Computing with FPGAs. *International Conference on Supercomputing (ICS)* (pp. 160-171). ACM.

Lee, S., Lambert, J., Kim, J., Vetter, J. S., and Malony, A. D. (2018). OpenACC to FPGA: A Directive-Based High-Level Programming Framework for High-Performance Reconfigurable Computing. *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)* (Poster)

ACKNOWLEDGEMENTS

I want to thank Allen Malony for his unwavering support, for inspiring me to pursue high-performance computing, and for guiding me through this process of learning. I also want to thank Hank Childs and Boyana Norris for serving not only on my dissertation committee, but also for advising all of my graduate milestones. For their support throughout my graduate studies, I want to thank the ORNL Future Technologies Group, including Jeffrey Vetter, Joel Denny, and especially Seyong Lee. As the mastermind of OpenARC, Dr. Lee laid out the groundwork that made the research in this dissertation possible, and I value everything I have learned from him over the past five years. The research in this dissertation would have been impossible without Philip Roth, Steve Moulton, and Erik Keever's relentless support in maintaining the software and hardware environments I worked with. I also want to thank the program coordinators and administrative assistants that contributed to my success as a student and intern: Cheri Smith, Liz Herbert, Donna Wilkerson, Tara Hall, and Charlotte Wise, and Vickie Braga. Finally I want to thank my graduate student peers, including Jonathan Brophy, Mohammad Monil, Sam Pollard, Chad Wood, and so many others, whose support and companionship have made this graduate experience even more rewarding.

To my father, who taught me the value of integrity and commitment; my mother, who sparked my love for science; and my partner, for all of the love and support during this adventure

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

LIST OF SOURCE CODE LISTINGS

Listing Page

CHAPTER I

INTRODUCTION, BACKGROUND, AND MOTIVATION

Heterogeneous computing has undoubtedly become a permanent resident in the high-performance computing (HPC) landscape. The idea of using a diversity of hardware devices or systems together to solve a single problem is already a reality in today's leading supercomputer systems [1, 2]. The upcoming exascale systems, the largest and most powerful computing machines ever built, all depend innately on heterogeneous design [3, 4, 5]. As we approach the physical limitations of CPU-based fabrics, advancement in computational system design will require specialization not just in terms of processors and accelerators, but also memory hierarchies, storage, and more. While this era of *extreme heterogeneity* [6] will certainly give rise to interesting and powerful machines, it will also give rise to significant challenges. Below, we lay out the most significant and universal challenges in today's and tomorrow's heterogeneous programming and computing landscape, and describe how this dissertation's contributions move us one step closer to a solution.

A - Diversity of Hardware: In this dissertation, we discuss projects involving both GPU and FPGA accelerators. However, for upcoming and future systems, other types of accelerators besides GPUs and FPGAs are being explored as hardware accelerators. More exotic, customized, and specialized hardware accelerators are being explored as viable options in heterogeneous systems. Machine learning accelerators, neuromorphic chips, and quantum accelerators promise to bring incredible performance to science, but also incredible challenges. As we see in this dissertation, the introduction of GPU and FPGA accelerators has already created hurdles for efficient heterogeneous computing.

B - Diversity of Programming Models: The diversity of heterogeneous hardware has already led to a diversity of high-performance programming models. This will undoubtedly be exacerbated as we transition toward extreme heterogeneity. While this diversity of models may appear necessary to support the wide range of devices, it leaves classes of devices inaccessible to classes of application developers.

C - Abstraction Level for Scientific Computing: Another point of contention in contemporary heterogeneous programming is the appropriate abstraction level for programming models. While computer scientists and professional programmers may prefer lower-level models that provide opportunities to fine-tune applications to specific devices, domain scientists may prefer a higher-level model that allows for portability between ecosystems. This can again lead to divergent programming models. DSLs like Tensorflow [7] may provide an optimal high-level approach, and these DSLs can be built using generalized low-level approaches. However, this has stranded the programmer looking for a high-level heterogeneous programming approach for an application outside the popular DSL frameworks.

D - Balance Between Open-Source and Proprietary: Another significant trade-off is the balance between open-source and proprietary frameworks. Nvidia's CUDA Toolkit [8] has been extremely successful as a frontier of GPU-CPU heterogeneous computing. Because of the toolkit's proprietary nature, Nvidia has been financially motivated to maintain, update, document, and market its tools, which has led to widespread adoption and longevity. However, the successes of a proprietary framework are less likely to extend to an extremely heterogeneous environment. An alternative could be open source solutions and

Table 1. Summary of heterogeneous computing challenges addressed in each chapter.
A - *Diversity of Hardware*,
B - *Diversity of Programming Models*,
C - *Abstraction Level for Scientific Computing*,
D - *Balance Between Open-Source and Proprietary*

Chapter I	Introduction
Chapter II	A, C
Chapter III	A, B
Chapter IV	A, B, C, D
Chapter V	Conclusion

standards like OpenMP, but these solutions may experience less streamlined development. For example, the OpenMP offloading standard was first released in 2013, but the first fully functional implementations were very recently released.

Working Toward Solutions - This Dissertation: In this dissertation, we push directive-based programming forward as one solution to the challenges above. In Table 1, we list the specific challenges addressed in each chapter of this dissertation. Our main research question is as follows: **Can an open-source, high-level, directive-based programming approach deliver specialized performance on the diversity of contemporary heterogeneous accelerators and exascale hardware?**

This question is directly related to each of the challenges mentioned above. (A) Because of its high-level nature, a directive-based solution can more easily incorporate new heterogeneous device families without significant restructuring of a language or standard. In contrast, a lower-level approach may need to be significantly extended or specialized to support a new device. (B) A single directive-based approach with implementations across systems can circumvent the issue of branching programming models designed for specific devices. (C) The

abstraction level of directive-based models can be more palatable for scientists from different domains compared to lower-level languages. And finally, (D) an open-source directive-based standard allows for wide adoption across ecosystems, even if implementations on specific platforms internally rely on proprietary backends.

We now present an outline of this dissertation. In Chapter I, we introduce the history of heterogeneous computing, and the contemporary programming models and compiler frameworks most commonly featured in heterogeneous computing-related research and science. We also discuss the available benchmark suites designed to evaluate heterogeneous platforms. The material in this chapter is unpublished with no co-authorship, although revision suggestions were given by Seyong Lee and the dissertation committee (Allen Malony, Hank Childs, Boyana Norris) as part of an Area Exam submission.

In Chapter II, we address the *diversity of hardware* by presenting an OpenACC-to-FGPA framework that encapsulates FPGAs under the umbrella of directive-based acceleration. Using this framework, a single application written using OpenACC can be run on a CPU, GPU, and FPGA. Within the framework, device-specific compiler optimizations produce low-level code specific to the targeted hardware. This framework is also one solution to the *abstraction level for scientific computing*, as it provides a palatable programming abstraction level for a very specialized device. Chapter II contains previously published material with co-authors from ICS 2018 [9], AsHES 2020 [10], PARCO 2021 [11], and IWOCL 2021 [12].

In Chapter III, we introduce an OpenACC and OpenMP interoperable framework that addresses the *diversity of programming models* between the two most widely used directive-based standards. This framework also addresses the

diversity of hardware by allowing a single application written in one standard to execute on any device supporting either standard. Chapter III contains previously published material with co-authors from HeteroPar 2019 [13] and SC 2020 [14].

In Chapter IV we present an exploration of exascale-intended platforms using applications written in a single programming model, OpenACC. Each OpenACC application is then source-to-source translated and compiled to several different platforms. This addresses all four problems above: (A and B - *diversity of hardware and programming models*) we target multiple different hardware accelerators using not only a single programming model, but a single source code without modification, (C - *abstraction level for scientific computing*) we assess a single high-level directive-based abstraction model for several specialized platforms, and (D - *balance between open-source and proprietary*) our evaluated applications are written using a single open-source standard, and source-to-source translated using an open-source compiler into several proprietary low-level programming models, allowing us to take advantage of both ownership approaches. Chapter IV contains unpublished material with co-authors.

Finally, in Chapter V we make a high-level assessment of contemporary heterogeneous computing, summarize the research in this dissertation, and discuss avenues for future research. Chapter IV contains unpublished material.

In Figure 1, we see a summary, albeit simplified, of the current state of heterogeneous computing. We see directive-based programming models, low-level models, and accelerators. The solid lines here means a mainstream implementation supports compilation of a model on an accelerator or device, while a dotted line indicates supports with extra steps. While there is some overlap between models on many platforms, even in these cases, the code written in the same standard

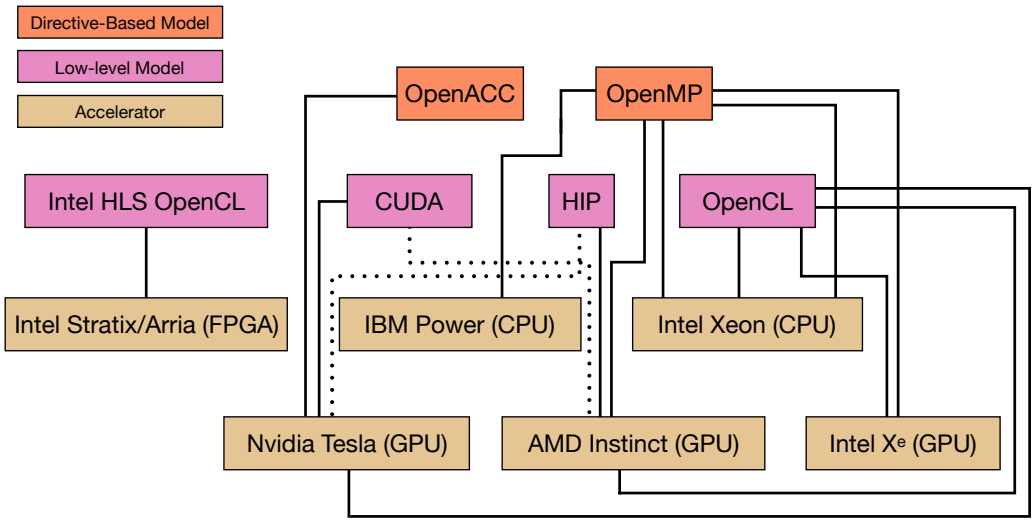


Figure 1. Summary of the state of heterogeneous programming and computing.

may not be directly portable between the platforms. Furthermore, for devices with support for multiple programming models, the associated implementations may be significantly more mature for a single model or subset of the technically supported models. Figure 1 succinctly exposes the challenges mentioned above.

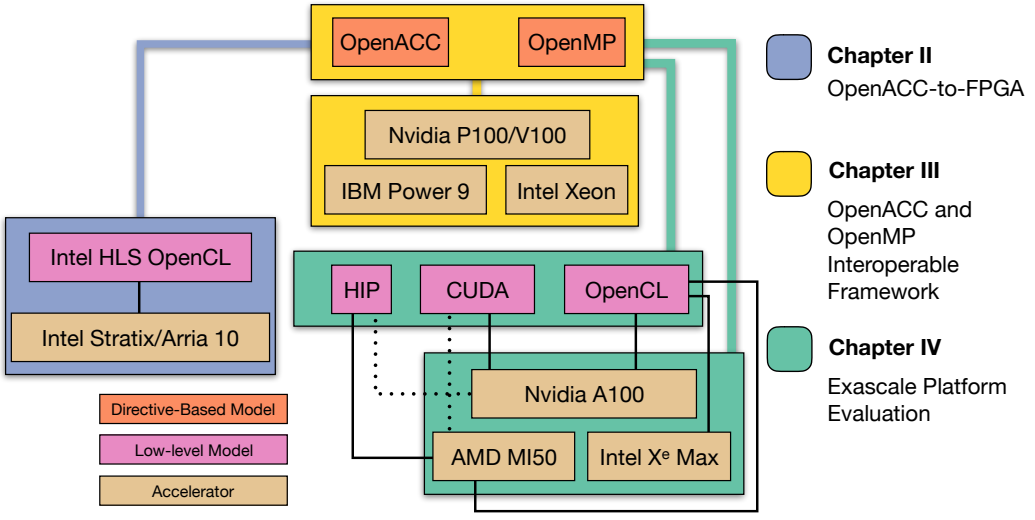


Figure 2. Re-evaluation of the state heterogeneous programming and computing after including this dissertation's contributions.

In Figure 2, we see a re-imagined landscape for heterogeneous computing as a result of the research in this dissertation. Because of the contributions in Chapter III, we can now encapsulate OpenACC and OpenMP as an interchangeable, high-level, front-end programming model. The contributions of Chapter II allow us to include Intel FPGAs into this encapsulation, instead of relying on Intel-specific OpenCL. Finally, due to the contributions of Chapter IV, we see the diversity of low-level models and devices that can be evaluated using a single directive-based frontend and a sufficiently optimized source-to-source compiler. Throughout the rest of this dissertation, we take an extended dive into these three projects, and examine how the performed research motivates Figure 2.

This dissertation includes prose, figures, and tables from previously published conference, workshop, and journal proceedings.

1.1 History of Heterogeneous Computing

Heterogeneous computing is paramount to today's high-performance systems. The top and next generation of supercomputers all employ heterogeneity, and even desktop workstations can be configured to utilize heterogeneous execution. The explosion of activity and interest in heterogeneous computing, as well as the exploration and development of heterogeneous programming approaches, may seem like a recent trend. However, heterogeneous programming has been a topic of research and discussion for nearly four decades. Many of the issues faced by contemporary heterogeneous programming approach designers have long histories, and have many connections with now antiquated projects, ideas, and technologies.

In this section, we explore the evolution and history of heterogeneous computing, with a focus on the development of heterogeneous programming approaches. In Section 1.1.1, we do a deep dive into the field of distributed

heterogeneous programming, the first major application of hardware heterogeneity in computing. We also briefly explore the phasing-out of distributed heterogeneous systems and approaches, and discuss the transitional period for the field of heterogeneous computing. In Section 1.1.2, we provide an exploration into contemporary accelerator-based heterogeneous computing, specifically analyzing the different programming approaches developed and employed across different accelerator architectures.

1.1.1 Distributed Heterogeneous Systems. Even 40 years ago, computer scientists realized heterogeneity was needed due to diminishing returns in the homogeneous systems. In the literature, the first references to the term "heterogeneous computing" referenced the distinction between single instruction, multiple data (SIMD) and multiple instruction, multiple data (MIMD) machines in a distributed computing environment.

Several machines dating back to the 1980s were created and advertised as heterogeneous computers. Although these machines were conceptually different than today's heterogeneous machines, they still were created to address the same challenges: using optimized hardware to execute specific algorithmic patterns.

The Partitionable SIMD/MIMD (PASM) [15] machine developed at Purdue University in 1981 was initially developed for image processing and pattern recognition applications. PASM was unique in that it could be dynamically reconfigured into either a SIMD or MIMD machine, or a combination thereof. The goal was to create a machine that could be optimized for different image processing and pattern recognition tasks, configuring either more SIMD or MIMD capabilities depending on the requirements of the application.

However, like many early heterogeneous computing systems, programmability was not the primary concern. The programming environment for PASM required the design of a new procedure-based structured language similar to TRANQUIL [16], the development of a custom compiler, and even the development of a custom operating system.

Another early heterogeneous system was TRAC, the Texas Reconfigurable Array Computer [17], built in 1980. Like PASM, TRAC could weave between SIMD and MIMD execution modes. But also like PASM, programmability was not a primary or common concern with the TRAC machine, as it relied on now-arcane Job Control Languages and APL source code [18].

The lack of focus on programming approaches for early heterogeneous systems is evident in some ways by the difficulty in finding information on how the machines were typically programmed. However, as the availability of heterogeneous computing environments increased throughout the 1990s, so did the research and development of programming environments.

Although the first heterogeneous machines consisted of mixed-mode machines like PASM and TRAC, mixed-machine heterogeneous systems became the more popular and accessible option throughout the 1990s. Instead of a single machine with the ability to switch between a synchronous SIMD mode and an asynchronous MIMD mode, mixed-machine systems contained a variety of different processing machines connected by a high-speed interconnect. Throughout the 80s and early 90s, this environment expanded to include vector processors, scalar processors, graphics machines, etc.

Examples of machines used in mixed-machine systems include graphics and rendering-specific machines like the Pixel Planes 5, Silicon Graphics 340 VGX,

SIMD and vector machines like the MasPar MP-series and the CM 200/2000, and coarse-grained MIMD machines like the CM-5, Vista, and Sequent machines.

It was well understood that different classes of machines (SIMD, MIMD, vector, graphics, sequential) excelled at different tasks (parallel computation, statistical analysis, rendering, display), and that these machines could be networked together in a single system. However, coordinating these distributed systems to execute a single application presented significant challenges.

The 1988 work by Ercegovac [19], *Heterogeneity in Supercomputer Architectures*, represents one of the first published works specifically surveying the state of high performance heterogeneous computing. They define heterogeneity as the combination of different architectures and system design styles into one system or machine, and their motivation for heterogeneous systems is summed up well by the following direct quote:

> Heterogeneity in the design (of supercomputers) needs to be considered when a point of diminishing returns in a homogeneous architecture is reached.

As we see throughout this work, this drive for specialization to counter diminishing returns from existing hardware repeatedly resurfaces, and this motivation for heterogeneous systems is very much relevant today.

At the time of Ercegovac's work, there existed three primary homogeneous processing approaches in high-performance computing: (1) vector pipeline and array processors, (2) multiprocessors and multi-computers following the MIMD model, and (3) attached SIMD processors. These approaches were ubiquitous across all the early surveyed works related to distributed heterogeneous computing,

and they heavily influenced the construction of heterogeneous systems and the development of heterogeneous software and programming approaches.

A later survey was published in 1995: *Goals of and Open Problems in High-Performance Heterogeneous Computing* by Siegel et al. [20]. Siegel was very involved in the early development of distributed heterogeneous computing, including the outline of the PASM system mentioned above. The authors presented the following goal for heterogeneous computing:

> To support computationally intensive applications with diverse computing requirements. Ideally presented to the user in an invisible way.

Looking to the future, this survey by Siegel et al. introduced a conceptual model for an end-to-end heterogeneous programming and computing approach, recreated for this dissertation in Figure 3. Although the model is conceptual, as no complete implementation existed at the time, the model and derivations of it appear frequently in the subsequent heterogeneous computing literature. The concepts of 1) automated machine and algorithm classification, 2) automated task profiling and analytical benchmarking, and 3) automated scheduling and assignment of sub-tasks to heterogeneous components were open questions at the time, and largely remain open questions today.

While most parallel computing research at the time focused on computational models, algorithms, or machine architectures, the PVM project [21], started at Oak Ridge National Laboratory, was an early attempt to provide a unified programming model for both homogeneous and heterogeneous distributed environments. The overarching goal of PVM was to allow a diverse and scalable set of heterogeneous computer systems to be programmed as a single parallel

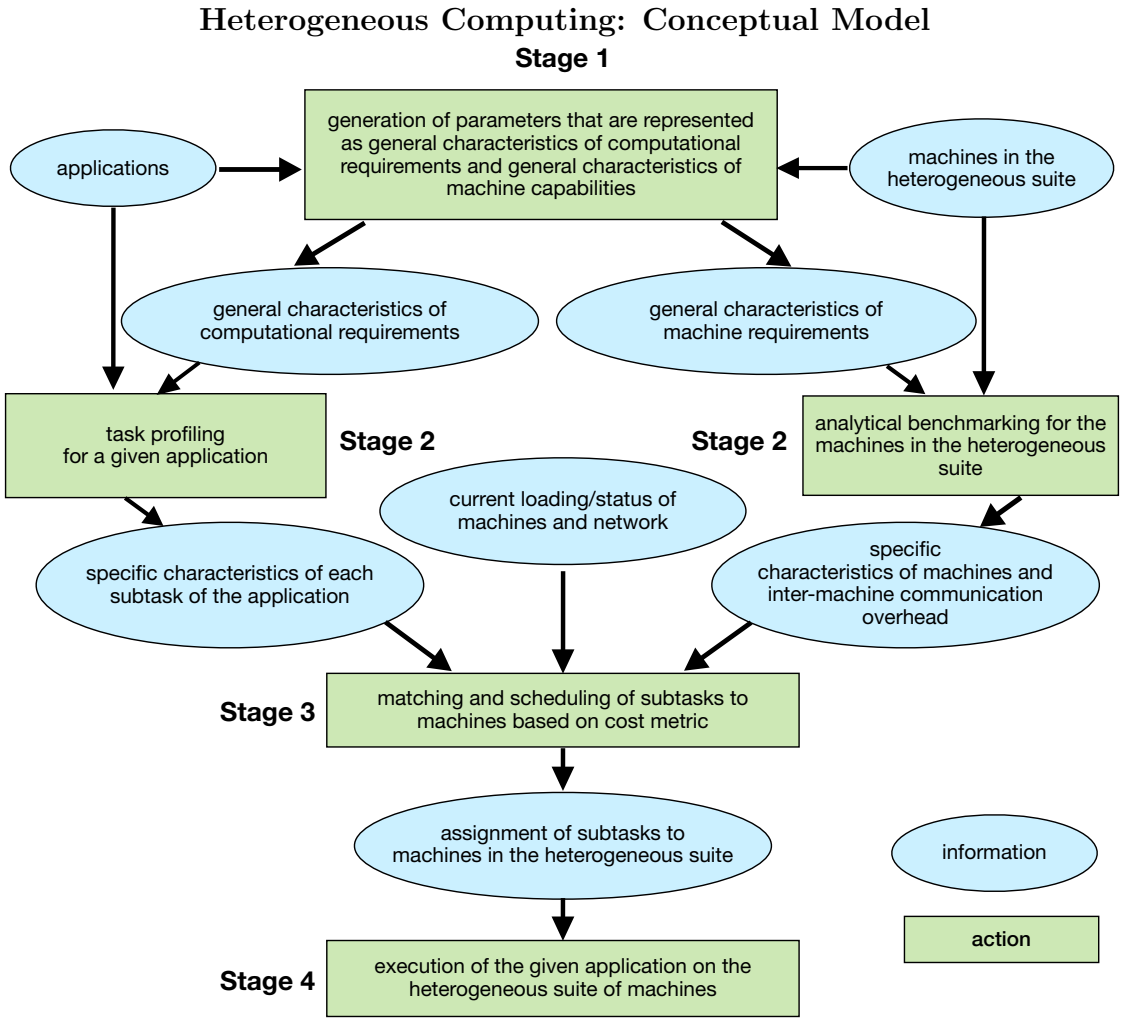


Figure 3. Re-creation of conceptual model of heterogeneous computing by Siegel et al. [20]

virtual machine. Essentially, PVM was designed as a programming environment for interacting but independent components. Other early heterogeneous programming languages included HeNCE [22, 23, 24], an extension to PVM, the p4 project [25] from Argonne National Laboratory, and the Mentat Language [26, 27], developed as extensions to C++.

Thirty years later, many of the visions of the developers of early distributed heterogeneous systems are still just that—visions. As we see in the later chapters and sections of this dissertation, most modern heterogeneous programming approaches still require some manual management of data transfers, communication, and synchronization, although typically with more user-friendly programming approaches than those of early systems like Mentat. Much of the research and discussion today in heterogeneous computing revolves around finding the appropriate abstraction level, as previously mentioned.

The diversity of processors in these early heterogeneous distributed systems seems small relative to today's array of co-processors (GPUs, FPGAs, TPUs, etc.). These early processors would all typically fall into the "traditional CPU" in today's categorization.

However, the diversity in supporting hardware and software was far greater in early heterogeneous ecosystems than today's typical cluster and supercomputer environments. Because the sub-components were typically completely separate machines, they experienced heterogeneity in the network architecture, the connection latencies, and the different communication bandwidths for different machines. On the software side, different machines had different operating systems, different process support and inter-process communications, varied compiler and language support, and multiple file systems. Unlike today's cluster and

supercomputing environments with mostly homogeneous software environments, early distributed heterogeneous system approaches required masking these network and software diversities. However, as we transition into an era of *extreme heterogeneity*, many of these early considerations are likely to resurface.

Around the turn of the century, the keywords and terminology surrounding heterogeneous distributed systems research began to shift. The next realization of heterogeneous computing systems began to be referred to as Metasystems, or referenced in the context of Metacomputing, and Grid Computing. This shift in perspective reflected a more universal or global outlook on heterogeneous computing. Distributed heterogeneous computing, coincident with the rapid and impressive growth of the internet and web-based computing, expanded into Metacomputing, Grid Computing, and eventually set up the backbone for the monolith that is today's cloud-based computing.

The goals of Meta and Grid computing were to create infinitely scaling systems by harnessing the power of remotely connected heterogeneous systems. While some projects tackled this, these ideas were ultimately re-purposed for commercial success under the umbrella of cloud computing. Additionally, with respect to scientific endeavors, the construction of large-scale homogeneous clusters and supercomputers beckoned a shift from distributed heterogeneous machines. At the same time, the growth of MPI, without a major focus on heterogeneous interoperability, overshadowed projects like PVM and p4 that targeted heterogeneous systems.

Finally, the very things that made early machines heterogeneous began to be integrated into single homogeneous processors. Unlike mixed-mode machines like PASM with distinct SIMD and MIMD processing, many new multi-core vectorizing

processors seamlessly integrate both SIMD and MIMD capabilities, which forgoes the need for a heterogeneous programming environment. Similarly, as we previously discussed, early distributed heterogeneous systems contained separate processors for visualization, statistics, and data processing. However, with the expansion of x86 and inclusion of specialized and vector instructions on general purpose CPU processors, the problems these early heterogeneous systems tackled could now be solved by homogeneous systems.

The shift into cloud computing, the ubiquity of MPI, and the continuous consolidation into x86 CPUs in many ways signaled the end of heterogeneous computing as it was originally imagined. However, as we see in the next section (Section 1.1.2), the rebirth of heterogeneous computing, and reinvention of many of the ideas previously mentioned, was sparked by the introduction of accelerator-based heterogeneous systems.

1.1.2 Multicore, Manycore, and Accelerator-based Heterogeneous Systems.

Hardware processing chips evolved from a single core, to multi-core and manycore chips, which then developed into hardware accelerators. These developments revolutionized the architectures of nearly all high-performance machines, and effectively re-birthed the field of heterogeneous computing.

The construction of large homogeneous machines marked the end of the 2000s decade and the end of heterogeneous distributed systems like we saw in the 1980s and 1990s. Jaguar [28], built around 2009 at Oak Ridge National Laboratory, was a Cray XT5 system, consisting of 224,256 x86-based AMD CPU cores, and was listed as the world's fastest machine in 2009 and 2010. Kraken [29], another Cray Xt5 system built in 2009, was listed as the world's fastest academic machine at the time. These homogeneous machines dominated the domain of HPC for

several years. Likewise, HPC software support, programming approaches, and compiler infrastructure developed during this time was also largely homogeneous. However, at the same time, scientific programmers began experimenting with programming using Graphics Processing Units, or GPUs, a trend that would eventually revolutionize the HPC field.

In 2000, Toshiba, Sony, and IBM collaborated on the Cell Project [30]. This project culminated in the release of the Cell Processor in 2006. While not strictly a GPU, the Cell Processor was one of the first architectures to apply accelerator-based heterogeneity to multi-media and general purpose applications. The Cell Processor's first major commercial application was inside the Sony PlayStation 3 gaming console. In 2008, IBM and Los Alamos National Laboratory (LANL) released the Roadrunner supercomputer, which consisted of a hybrid design with 12,960 IBM PowerXCell and 6,480 AMD Opteron dual-core processors [31]. The IBM PowerXCell processors absorbed the original Cell processor design.

While the Cell processor generated excitement and a new interest in a different type of heterogeneous computing, it was only efficient for certain computations, and the overhead of manually transferring memory to and from the device was a performance bottleneck due to the small memory size of the Cell architecture. Although GPUs and other heterogeneous accelerators suffer from these same issues, they evolved and developed to meet the demand of scientific computing.

The scientific community began evaluating GPUs for general purpose processing well before their use became mainstream. In 2001, researchers evaluated general purpose matrix multiplication, and in 2005 LU decomposition on a GPU was shown to outperform a CPU implementation [32]. Interest in utilizing GPUs

in scientific computing continued to grow, but was inhibited by the complex programming approaches for GPUs, which typically required a low-level graphics interface and dealing with shaders and graphics-related APIs data structures. However, with the release and development of programming models and frameworks mentioned in subsequent sections, GPU programming, and the whole field of scientific heterogeneous programming including other types of accelerators, became the norm in high-performance computing. Throughout the rest of this dissertation, references to "heterogeneous computing" typically imply the contemporary accelerator-based flavor.

Since the initial release of CUDA in 2006, GPGPUs have been the dominant driving force for accelerator-based heterogeneous computing. The concept of offloading computationally intense regions of code to a heterogeneous hardware accelerator has become commonplace in scientific computing, and for the past decade, heterogeneous computing has almost exclusively referred to GPGPU offloading. However, FPGAs have recently emerged as a potential competitor to GPU accelerators, both in terms of computing power and power efficiency.

Field Programmable Gate Arrays (FPGAs) have been designed and developed for nearly 40 years. Altera, a major FPGA manufacturer, was founded in 1983, and released the first FPGA in 1984. Xilinx, the main competitor to Altera for several decades, was founded in 1984 and released their first FPGA in 1985. These devices have been promoted as potential architectures for high performance computing for decades, but until very recently, have not seen much adoption. The real revolution for FPGAs, and their adoption as a heterogeneous accelerator, has stemmed from the introduction of new FPGA programming approaches. Creating

a high-level programming approach for high-performance FPGA accelerators is the main motivation for Chapter II.

1.2 Heterogeneous Programming Models

In this section, we first discuss the accelerator-based heterogeneous programming models most heavily featured in this dissertation's research results. We then discuss several other relevant contemporary heterogeneous programming models.

1.2.1 CUDA.

Nvidia was formed in 1993, but first gained major recognition by winning the contract to develop the graphics hardware for the Microsoft Xbox gaming console in 2000. Nvidia continued to grow and increase its claim in the GPU market with the release of the GeForce line, in direct competition with AMD's Radeon line. However, these devices were still targeted toward graphics processing.

As the interest in scientific computing using GPUs continued to grow, Nvidia first recognized the potential financial advantages of supporting this community. In 2007, Nvidia launched the Tesla GPU, aimed at supporting general purpose computing, and the CUDA (Compute Unified Device Architecture) API and programming platform [8].

The CUDA programming platform abstracted programming GPU hardware into an API that was more consumable by scientific programmers and other programmers without extensive graphics programming experience. The CUDA programming model essentially presents a hierarchical multi-threading layout, where threads are executed as a 32- or 64-thread warp, warps are mapped onto thread-blocks, and thread-blocks are mapped onto a grid and grid blocks. These abstractions fit quite naturally with the nested loop structure of most scientific

Listing 1.1 Example CUDA C Application

```
1  #include <stdio.h>
2
3  __global__
4  void saxpy(int n, float a, float *x, float *y)
5  {
6    int i = blockIdx.x*blockDim.x + threadIdx.x;
7    if (i < n) y[i] = a*x[i] + y[i];
8  }
9
10 int main(void)
11 {
12   int N = 1<<20;
13   float *x, *y, *d_x, *d_y;
14   x = (float*)malloc(N*sizeof(float));
15   y = (float*)malloc(N*sizeof(float));
16
17   cudaMalloc(&d_x, N*sizeof(float));
18   cudaMalloc(&d_y, N*sizeof(float));
19
20   for (int i = 0; i < N; i++) {
21     x[i] = 1.0f;
22     y[i] = 2.0f;
23   }
24
25   cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
26   cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
27
28   // Perform SAXPY on 1M elements
29   saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
30
31   cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
32
33   cudaFree(d_x);
34   cudaFree(d_y);
35   free(x);
36   free(y);
37 }
```

software. Listing 1.1 shows an example CUDA application, sourced from Nvidia's website [33].

As the popularity of CUDA and GPGPU programming grew, several large supercomputers began including both host CPUs and GPU accelerators. In 2010, China's Tianhe-1A machine launched, containing 14,336 Xeon X5670 processors and 7,168 Nvidia Tesla M2050 general purpose GPUs [34]. This heterogeneous machine overtook the previously mentioned Jaguar machine from Oak Ridge National Laboratory (ORNL) as the "world's fastest supercomputer". ORNL's Titan supercomputer, a successor Jaguar, launched in 2013 and consisted of 18,688

AMD Opteron CPUs, each with an attached Nvidia Tesla (K20x) GPU [35]. This machine also secured the top spot as the world's fastest machine.

More recently, Nvidia GPUs and CUDA programming were employed in ORNL's Summit Supercomputer [1], another machine that briefly held the title as the world's fastest. Summit was launched in 2018 and contains 4,608 nodes each with 6 Nvidia Tesla V100 GPUs. Similarly, Lawrence Livermore National Laboratory (LLNL) launched the Sierra Supercomputer [2] in 2018, containing 4,320 nodes each with 4 Nvidia Tesla V100 GPUs.

Much of CUDA's success in scientific programming can be attributed to Nvidia's continued investment in and focus on CUDA training. Online and in-person training workshops, and a surplus of available training materials, made Nvidia and CUDA an attractive GPGPU option compared to other vendors. This focus on training and CUDA's success should provide a model for future heterogeneous programming approaches. Some newer approaches like OpenACC (also supported by Nvidia, and discussed in detail in Section 1.2.4) have also adopted this strategy, frequently hosting learning-focused hackathons and generating significant training materials [36].

In this dissertation, CUDA is employed as a backend programming model in Chapter IV.

1.2.2 OpenCL. CUDA's dependence on Nvidia devices spawned efforts to create an open-source alternative. OpenCL was developed as one alternative[37]. As we see in the remainder of this dissertation, OpenCL has become a staple of accelerator-based heterogeneous programming approaches, both as a stand-alone approach and as an intermediate representation or backend for higher-level approaches.

OpenCL (Open Computing Language) was originally developed by Apple as a GPGPU option under the OSX umbrella. In early 2008, Apple submitted a proposal to the Khronos Group for creation and management of an OpenCL standard [37]. On November 18, 2008 the OpenCL 1.0 technical specification was released. By the end of 2008, AMD, Nvidia, and IBM had all incorporated OpenCL support into their vendor toolchains.

Like CUDA, the OpenCL programming approach separates an application into host code and device code. The abstraction level for the OpenCL device code is very similar to CUDA, but the host code abstractions are arguably more verbose. Like CUDA, GPU cores are abstracted into a tiered parallelism. In OpenCL, work-items are executed as part of a work-group, and work-groups are organized inside an ND range (Table 2). Listing 1.2 demonstrates and example vector addition application in OpenCL. From the line count alone, we can see that OpenCL requires a significant amount of low-level and boilerplate code, although this functionality is typically encapsulated in routines and libraries by frequent OpenCL programmers. However, each programmer creating a personalized set of routines to abstract OpenCL API calls creates issues with code portability and interpretability.

Although OpenCL does provide an open-source alternative to CUDA that is supported across several different device vendors (Nvidia, Intel, IBM, AMD), it has not become the de facto standard for heterogeneous GPGPU computing. First, the widespread success of CUDA and Nvidia's dominance in the GPGPU market has allowed scientific programmers to safely choose a non-portable option. Second, the abstraction level, especially the verbosity of the host code, has led many GPGPU developers to seek higher-level abstractions, as we see in the following section.

Table 2. Comparison of CUDA and OpenCL GPGPU abstractions

CUDA	OpenCL
Grid	NDRange
Thread Block	Work group
Thread	Work item
Thread ID	Global ID
Block index	Block ID
Thread index	Local ID

However, as we discuss later, although OpenCL has not seen widespread adoption as a programming approach, many frameworks and compilers target OpenCL as a backend API (OneAPI [38], OpenARC [39], TVM [40], etc.)

The OpenCL programming model is a critical component of this dissertation, and is featured as a backend programming target in Chapters II and IV.

1.2.3 HIP. Nvidia's main competitor in the GPU market, traditionally in the consumer market but more recently also in the high-performance and scientific community, is AMD. Unlike Nvidia, AMD has not developed a proprietary programming approach and vendor compiler for heterogeneous computing. Instead, to support its GPU architectures AMD has developed the open-source ROCm (Radeon Open Compute) suite [41]. ROCm is a collection of APIs, drivers, and development tools that support heterogeneous execution on both AMD GPUs, but also other architectures like Nvidia GPUs. The actual programming model developed as part of ROCm is HIP, another low-level approach with a similar abstraction level to CUDA and OpenCL. However, the ROCm toolkit and associated compilers also support OpenMP and OpenCL applications. The compilers, libraries, and debuggers for ROCm are available from the open-source github [42]. Although the current generation of top

supercomputers like Sierra and Summit employ Nvidia GPUs, future systems like ORNL's Frontier [3], expected to launch in 2021, will employ AMD GPUs. This transition could herald a shift away from CUDA, and increase the use of ROCm and HIP across all of scientific computing.

In this dissertation, HIP is employed as a backend programming model in Chapter IV.

1.2.4 OpenACC. OpenACC (originally short for Open Accelerators) is one of this first high-level (as opposed to low-level approaches like HIP, OpenCL, and CUDA) GPGPU programming approaches that still supports a significant user base today (as of 2021). OpenACC was first released in 2012 as a collaboration between Cray, NVIDIA, and the Portland Group in order to support the users of ORNL's Titan, one of the first large heterogeneous supercomputers. As previously mentioned, Titan was a Cray machine with Nvidia devices. The Portland Group was involved because OpenACC was inspired by the high-level directive approach used in in the PGI-Accelerator model, and the first OpenACC compiler provided by PGI was developed as an extension to the PGI-Accelerator compiler [43, 44].

The dream of OpenACC was to create an open, directive-based standard for GPU-computing as an analog and counterpart to the then de facto standard for parallel processing on multi-core CPUs, OpenMP. In the same way that a small number of OpenMP pragmas can be used to parallelize an existing application, OpenACC intended to provide a minimal set of directives that application developers could apply to accelerate an existing CPU-based scientific application on a GPU. This contrasted with the existing lower-level programming approaches like CUDA and OpenCL, which required a significant amount of code restructuring and rewriting for GPU acceleration.

The ideology of OpenACC is to allow users to expose and identify parallelism in an application using descriptive directives, and to leave the more complicated task of mapping parallelism to GPU devices in the hands of the OpenACC compiler. This deviates from the OpenMP model, which traditionally employed a very moderated and prescriptive application of directives.

This high burden of effort tasked to OpenACC compilers in some ways has prevented OpenACC from reaching the popularity and monopoly status of its OpenMP analog. Although OpenACC is intended for general-purpose GPU computing across different vendors, for most of its history, the PGI OpenACC compiler has been the only available production-level option, and was restricted to Nvidia devices. Now, nearly a decade later, other implementations have more fully adopted the OpenACC standard and implemented more functional support. We discuss these compilers in more detail in Section 1.3.

An OpenACC annotated application typically contains a combination of data and compute directives centered around a computationally intense region of code or loop nest. In Listing 1.3, we see a small C program annotated with two OpenACC directives, a data directive (line 16) and a compute directive (line 19). Replicating this high-level programming approach in a low-level approach like CUDA or OpenCL would require significantly more code, several source files, and multiple compilations.

OpenACC is featured heavily in this dissertation's research results, most often as the primary source code language for evaluations in Chapters II, III, and IV.

1.2.5 OpenMP. OpenMP reigned as the de facto standard for directive-based homogeneous multi-core CPU computing throughout the early

2000s, at least in the scientific computing domain. As the demand for high-level programming approaches for GPGPU computing increased in the early 2010s, there was a push for OpenMP to support accelerator-based heterogeneous computing in addition to the homogeneous multi-core computing. Although the previously-mentioned OpenACC was developed to address this demand, motivation for OpenMP prevailed for several reasons:

1. OpenACC and OpenACC compilers have been too-tightly bundled to Nvidia devices, especially since PGI (the primary OpenACC compiler) was acquired by Nvidia in 2013.

2. Most high-performance-oriented scientific programmers were already familiar with basic OpenMP directives and OpenMP programming styles.

3. Many scientific applications already employed OpenMP for homogeneous CPU-based computing, lightening the burden of developing a new accelerator-based implementation.

As a result, in 2013, a year after the launch of OpenACC, the OpenMP standards committee released OpenMP 4.0, which included new directives for offloading to GPU accelerators. In 2018, the standards committee released OpenMP 5.0, which expanded support for accelerators and included additional directives for tasking and auto-parallelism. Even before the official inclusion of offloading directives in OpenMP, several research-oriented compilers had been prototyping support for GPU offloading for OpenMP [45, 46].

Initially in their development, OpenACC and OpenMP differed in their programming approach philosophy. As mentioned, OpenACC employed a more descriptive approach, where users expose parallelism and compilers map that

parallelism to devices. In OpenMP, the directives supplied by users are taken more literally and prescriptively, in that the user directly controls how the parallelism is mapped to a device. However, the two standards have recently become more aligned due to the *loop* directive introduced in OpenMP 5.0, which mimics the behavior of the descriptive OpenACC directives. The relationship between OpenMP and OpenACC has been somewhat contentious at times. However, both standards are still currently being maintained as a high-level programming approach for heterogeneous computing.

Although OpenACC has been limited due to its ties to Nvidia devices, the availability of the production-level PGI OpenACC compiler throughout its history has certainly been an advantage. In contrast, although OpenMP 4.0 originally was approved in 2013, compilers fully supporting the standard have been slow in coming. Only very recently have mature compilers successfully supported the entire standard, and many mainstream compilers are still under development for the OpenMP 4.0 standard and especially the OpenMP 5.0 updates. We discuss this further in Section 1.3.

In Listing 1.4, we show the same application as the previous listing, now annotated with OpenMP directives. Although this short example trivially highlights the use of OpenMP, the example still demonstrates how OpenMP greatly simplifies heterogeneous computing compared to CUDA and OpenCL.

1.2.6 Other Modern Programming Models. Although the previous sections describe the programming models targeted in the research results of this dissertation, for the sake of completeness we briefly describe several other contemporary heterogeneous programming models.

1.2.6.1 *Kokkos.*

In 2012, around the same time as the release of OpenACC and OpenMP 4.0, H.C. Edwards and a team at Sandia National Laboratory developed the Kokkos portability layer [47, 48, 49].

Kokkos is implemented as a performance portability layer. Unlike OpenACC and OpenMP that rely on directives, Kokkos is implemented as a C++ template library on top of OpenMP, CUDA, HPX [50] (discussed below), or Pthreads [51]. Essentially, the goal is to allow programmers to implement the Kokkos abstraction layer once in their application, which can then be executed across a diversity of hardware architectures. The C++ templating abstraction is an attractive model for heterogeneous programming, as it allows the same API calls to have multiple backend implementations.

Kokkos has been a popular option within the scientific community, and is supported by several national labs, including Sandia and Argonne National Laboratories. Compared to OpenMP and OpenACC, the Kokkos abstractions do require more in-depth knowledge of C++ including concepts like templates and functors, compared to the directive-based approaches. However, the integration with C++ also provides a powerful programming abstraction compared to the directive-based approaches that require kernels to use minimal C++ features.

1.2.6.2 *Raja.*

Like Kokkos, Raja is a C++-based GPGPU programming approached developed by a major US national laboratory, Lawrence Livermore National Lab (LLNL) [52, 53]. Raja was first released in 2014, shortly after Kokkos, OpenACC, and OpenMP 4.0. Raja is essentially another collection of C++ abstractions intended to provide architecture portability for HPC systems, specifically those with GPGPU architectures.

A 2015 Supercomputing poster compared Raja and Kokkos using the TeaLeaf application [54]. While Kokkos relied on the C++ template metaprogramming approach, Raja instead relies on the C++11 lambda features. They also found that porting an application to Raja was relatively intuitive, on a similar level to an OpenMP port. Conversely, porting the application to Kokkos required extensive architectural changes. Like Kokkos, Raja relies on OpenMP and CUDA internally to target CPUs and GPUs, respectively.

1.2.6.3 SYCL, DPC++, and OneAPI.

The SYCL standard is yet another C++-based heterogeneous programming approach [55]. First released in 2014, SYCL originally aimed to be a programmer-productivity oriented abstraction layer on top of OpenCL. However, later implementations targeted other intermediate representations, like AMD HIP and CUDA. We discuss this further in Section 1.3. Although SYCL is several years old, it has seen limited uptake in the scientific community, until its recent involvement with DPC++ and Intel's OneAPI initiative.

DPC++ [56], launched in 2019, is a SYCL implementation developed and managed by Intel, that integrates the SYCL and OpenCL standards with additional extensions. These extensions are often championed for inclusion in the SYCL standard itself, analogous to how several of the heterogeneous and parallelism features of SYCL are then pushed for inclusion into the C++ standard. Examples of features in SYCL that originated in DPC++ include unified shared memory, group algorithms, and sub-groups.

Intel's OneAPI Library [38, 57] attempts to encapsulate several of the technologies and programming approaches discussed in the section under a single

umbrella. OneAPI consists of several APIs based on DPC++, SYCL, C++ Parallel STL, and Boost.Compute, including:

- oneAPI DPC++ Library
- oneAPI Math Kernel Library
- oneAPI Data Analytics Library
- oneAPI Threading Building Blocks
- oneAPI Video Processing Library
- Collective Communications Library
- oneAPI DNN Library
- Integrated Performance Primitives

1.2.6.4 Legion. The Legion Project [58, 59, 60] originates from Stanford University, and was first published in 2012. Legion, a portmanteau of logical regions, is unique from many of the other high-level approaches in this section in that it aims to support both distributed and accelerator-based heterogeneous computing.

Like many of the other frameworks, a main goal of Legion is to abstract or decouple the algorithm design from the mapping or execution on heterogeneous architectures. For Legion, this concept extends to distributed heterogeneous machines. Legion specifically focuses on data movement and management abstractions, primarily by introducing the abstraction of logical regions. By partitioning data into logical regions and sub-regions, programmers can indicate

data locality and independence, which can be used by the underlying framework components to facilitate communication and parallelism.

Legion remains relevant today, and regular software releases address bugs, performance issues, features and extensions, and additional system support. Furthermore, the Legion project is supported and funded by the DOE Exascale Computing project [61].

1.2.6.5 HPX. HPX, short for High Performance ParalleX, is another distributed computing focused framework, developed by Louisiana State University and first published in 2014 [50, 62, 63]. Like Legion, HPX aims to provide a unified programming approach, allowing both single-node and distributed parallelism from a single API. HPX is strongly connected to C++, and depends heavily on the Boost C++ libraries. Although HPX has traditionally focused on CPU-based distributed and single-node parallelization, more recently, efforts have been made to support heterogeneous computation with HPX, either through integration with OpenCL (HPXCL [64]), development of a SYCL backend [65], or other approaches.

1.2.6.6 C++. While Raja and Kokkos are two of the most popular C++-based high-level GPGPU programming approaches, especially in scientific computing, several other C++ libraries and extensions have been developed to support heterogeneous computation. AMP [66], Boost.Compute [67, 68], Thrust [69], Bolt [70], and VexCL [71] are all either extensions to C++ or C++-based libraries that aim to enable heterogeneous computing.

All of the other programming approaches in this section refer to libraries and extensions not incorporated in the C++ standard. However, newer versions of C++ have begun to incorporate different types of CPU parallelism directly into the standard. For example, C++17 has increased SIMD support for parallel

loops. Furthermore, there is a push with the C++ community to add support for heterogeneous computing in future releases. The major drawback is the slow timeline for C++ releases and the significant burden of defending inclusions into the already massive C++ standard.

1.2.6.7 Domain Specific Languages. Both the high-level and low-level general-purpose GPU programming approaches allow developers to create heterogeneous applications for a huge diversity of application domains. However, many domain and computational scientists spend the entirety of their programming efforts within a very specific field or area. To combat the issues with the general purpose approaches, such as the complexity of the low-level approaches and inconsistency and performance issues with the high-level approaches, a multitude of domain-specific GPU programming approaches have been developed. More specifically, libraries or domain-specific languages (DSLs) targeting a single application space or area were developed to meet the very specific needs of a smaller user-base.

Linear Algebra: Linear and matrix algebra algorithms have consistently been some of the most important but also most computationally demanding components of scientific computing. It is no surprise then, that several heterogeneous libraries and frameworks have been developed specifically for this domain. Several linear algebra libraries have been developed by Nvidia as part of the CUDA Toolkit, including cuBLAS [72], cuSparse [73], and cuFFT [74]. Some open-source counterparts have also been built, including clBLAS [75], MAGMA [76, 77], Eigen [78], Odient [79, 80], and SPIRAL [81, 82]. Interestingly, Odient has been used as recently as 2020 to model the spread of the Covid-19 virus [83, 84, 85].

Graph Processing: The Halide programming language was developed in 2013 as a collaboration between MIT's CSAIL laboratory and Adobe [86, 87, 88]. Halide is a DSL targeted for image processing and graph algorithms. Like many of the other programming approaches in this section, Halide is embedded in C++, with a dedicated Halide C++ API. More recently, Halide has also developed python bindings. Halide supports a wide array of architectures, including x86, ARM, PowerPC, and other CPU architectures and CUDA, OpenCL, OpenGL, and DirectX enabled GPUs. Halide is used internally in Adobe Photoshop, and in projects related to Google's Tensorflow.

Machine Learning: The explosion of machine learning, undoubtedly the fastest-growing field in computer science, has led to the development of several heterogeneous programming approaches targeted specifically toward the machine learning domain. Nvidia has contributed to the machine learning domain with the development of their cuDNN library [89]. Although the cuDNN (CUDA Deep Neural Network) library can be programmed directly, similarly to cuBLAS, more typically cuDNN is used as a backend to one of the widely used deep learning front end frameworks, including MxNet [90], Tensorflow [7], Keras [91], Pytorch [92], Chainer [93], and Caffe [94]. AMD has also developed an OpenCL-based analog to cuDNN, named MIOpen [95, 96]. Recently released in 2019, MIOpen is provided as part of the ROCm suite, and based on a software stack including both OpenCL and HIP. Although MIOpen is currently not as popular as cuDNN, and lacks integration into the major front-end frameworks and tools, it could become popular in the near future with new AMD systems like ORNL's Frontier supercomputer [3], projected release in 2021 with four AMD GPUs on each node.

Scientific Visualization: A very natural domain for heterogeneous programming is scientific visualization. Visualization applications typically already heavily rely on GPU architectures for image and video rendering and display, typically through low-level APIs like OpenGL or OpenCV. Development of domain-specific heterogeneous programming approaches for scientific computing is a natural extension. One approach involves in-situ visualization, where the computation and visualization are tightly coupled, without requiring offloading to the host device. VTK-m [97] is an example of a heterogeneous scientific visualization approach. Like many other approaches, VTK-m relies on C++ template metaprogramming. The VTK-m programming abstraction is based on "data-parallel primitives", high-level algorithmic API calls that are then executed on the accelerator device. Another example is the Alpine framework [98], which builds on the VTK-m framework and ideas. Alpine is focused on supporting modern supercomputing architectures, a flyweight infrastructure, and interoperability with software like R and VTK-m. Alpine was designed to accelerate scientific visualization codes using Nvidia GPUs and Intel Xeon Phis.

Climate and Weather: Due to the high number of computational resources required to model climate and weather at scale, climate and weather simulations represent a large fraction of most HPC system workloads. Unsurprisingly, DSLs have also been created to ease the creation of climate-based HPC applications. One example, the CLAW project [99, 100] developed in 2018 at ETH Zurich, is a FOTRAN-based DSL that aims to provide performance portability for column- and point-wise weather and climate computations.

1.3 Heterogeneous Compiler Frameworks

Development of new heterogeneous programming approaches, APIs, libraries, and frameworks is important for advancing the field of heterogeneous computing. However, even the world's best-designed programming approach is rendered useless without an effective implementation, typically in the form of a compiler. Much of the success of different programming approaches hinges on the availability and usability of compilers for said approaches. In this section, we discuss different tiers of compilers, from vendor supported production-level to academic, each of which plays a crucial role in the life cycle of heterogeneous programming approaches.

1.3.1 Vendor-supported Compilers. We first discuss vendor compilers. These typically refer to a language implementation, in the form of a compiler, developed by a major accelerator manufacturer, such as Nvidia, AMD, IBM, Intel, etc. We briefly highlight some major advantages and disadvantages of the vendor compiler model for heterogeneous programming approaches, and then discuss several vendor compilers in detail.

Advantages: Compilers developed and maintained by hardware accelerator vendors are typically very consistent and reliable for a small set of supported devices. The documentation and user guides are often detailed, thorough, and updated. These companies are financially motivated for success with their devices, which results in many of the advantages listed. These compilers also have somewhat of a guarantee of longevity, at least compared to the independent and open-source projects.

Disadvantages: Vendor compilers, for obvious reasons, are limited to only compile code for devices produced by the vendor. This leads to replication of efforts for each different manufacturer. Furthermore, vendor compilers often

introduce extensions to otherwise portable programming approaches that optimize the performance for their specific devices. These extensions break the original language intentions, and result in code that is no longer portable across an array of different accelerators. The vendor compilers also typically have a slower release cycle, are slower to incorporate updates to programming approaches, and are more conservative for the implementation of new language features and the release of updated language versions.

1.3.1.1 NVCC. Arguably the most popular, and dominant, vendor compiler in all of heterogeneous computing is *nvcc*, Nvidia's core CUDA compiler [101]. Released in 2006 along with Nvidia's CUDA toolkit, *nvcc* is based on the LLVM compiler toolchain [102], which is discussed later in this section. The *nvcc* compiler is implemented as a compiler driver; *nvcc* invokes the needed tools to perform a given compilation. Typically in a C CUDA application, the host code is compiled with *gcc*, and the device code is compiled using *cudacc*. In this case, *nvcc* would invoke *gcc* and *cudacc*, generating a C-code host binary and PTX device code respectively. PTX, or NVPTX, is a low-level instruction set architecture used by CUDA-enabled GPUs.

The *nvcc* compiler is used in every subsequent chapter in this dissertation: in Chapter II to compare FPGA and GPU performance, in Chapter III as part of OpenARC's OpenACC compilation, and in IV as a backend programming model.

1.3.1.2 PGI. The PGI OpenACC compilers, *pgcc* and *pgft*, have been the de facto standard for OpenACC compilation since its inception in 2012 [44, 44]. The PGI (Portland Group Inc.) company was founded in 1989, and originally developed parallel computing compilers for x86 architectures. PGI especially specialized in high-performance FORTRAN compilers. Because of this

specialization, in 2009 PGI was contracted by Nvidia for the development of the first FORTRAN-based CUDA compiler.

PGI also worked with Nvidia to develop the PGI-Accelerator programming model, which we briefly mentioned in Section 1.2.4. As mentioned, the PGI-Accelerator compiler was eventually extended to develop the first OpenACC compiler. In 2013 PGI was acquired by Nvidia, redefining it as a "vendor compiler", at least for the purposes of this dissertation. Interestingly, in 2013 PGI also developed an OpenCL compiler for ARM cores [103], but this was removed after the Nvidia acquisition.

Since then, PGI has continued to develop compilers for Nvidia devices for OpenACC C and OpenACC FORTRAN, and has been very involved in the promotion and development of OpenACC itself. Although PGI compilers have existed independently from the CUDA toolkit in the past, as of August 2020 *pgcc* and *pgft* have now been fully absorbed into Nvidia, and are now re-branded as part of the Nvidia HPC SDK (NVHPC) [104, 105].

The PGI compiler and its predecessor, NVHPC, are featured in Chapters III and IV.

1.3.1.3 AMD. The other major GPU manufacturer after Nvidia, at least in the context of scientific computing, is AMD. Unlike Nvidia, AMD has not developed a proprietary programming approach and vendor compiler for heterogeneous computing. AMD has developed a C/C++ optimizing vendor compiler, *aocc*, for its CPU Ryzen devices [106], but for their Radeon GPU devices, AMD has opted for an open-source solution.

In order to support its GPU architectures, AMD has developed the open-source ROCm (Radeon Open Compute) suite [41]. ROCm is a collection of APIs,

drivers, and development tools that support heterogeneous execution on both AMD GPUs, but also other architectures like CUDA GPUs. ROCm supports the AMD HIP representation, but can also process OpenMP and OpenCL applications. The compilers, libraries, and debuggers for ROCm are available from the open-source GitHub [42].

AMD's compilers are featured and discussed in more detail in Chapter IV.

1.3.1.4 Intel. Intel has long been at the frontier of high-performance compilers for their optimizing and parallelizing CPU compilers, enabling SIMD and multi-threaded parallelism for their homogeneous Intel Xeon CPU devices. Intel's first foray into heterogeneous compilation came in 2010 with the introduction of the Intel Xeon Phi coprocessor chip [107]. These chips followed a similar offload model and architecture as the contemporary GPU models.

Intel's acquisition of the FPGA-manufacturer Altera has also resulted in the release of a vendor-specific Intel-based OpenCL compiler for FPGAs [108]. However, this compiler framework suffers from many of the vendor-specific extensions and optimizations mentioned in the above "disadvantages" discussion, rendering the resulting OpenCL not portable to other devices. We discuss this further in Chapter II.

Finally, with the release of the X^e GPGPU, Intel has also released an Intel-based GPU-specific vendor compiler [57]. This compiler currently supports OpenMP, OpenCL, SYCL, and DPC++ for GPU compilation, and is discussed in more detail in Chapter IV.

1.3.2 Open-source Compilers. The main alternative to heterogeneous proprietary vendor compilers are production-level open-source heterogeneous compilers. These compilers and compiler toolchains are typically

maintained by steering committees, which can consist of representatives from accelerator vendors, scientific institutions, and independent companies. We discuss the advantages, disadvantages, and some examples of open-source heterogeneous compilers.

Advantages Unlike the vendor compilers, open-source compilers are often more community driven. That is, the direction and implementation of the compiler is not completely motivated and driven by device manufactures, although device manufactures are often involved. Also, most of the open-source compiler frameworks support a variety of accelerators and architectures. More generally, open-source compilers benefit from all of the same advantages of open-source software as a whole, including transparency, flexibility, and independence. Specific to heterogeneous programming approaches, open-source compilers can more quickly adapt new standards and features and the rapidly evolving array of architectures. Also, because the same compiler can be used across several architectures, the input programming approach used is inherently more portable. Most open-source projects are managed through git or subversion, and hosted on a popular git repository hosting site like GitHub.

Disadvantages open-source compiler projects, especially the smaller ones, may not have the financial security of the vendor compilers. They also may not have the secured longevity. For example, if the main contributors to an open-source compiler projects change positions or careers, continued maintenance on the project may terminate. Also, the open-source compilers may not have access to low-level architecture details that the vendor compilers use to get increased performance on their specific devices. However, the large open-source compiler projects, like LLVM

and GCC, typically have no issues with longevity and closely tail vendor compilers in terms of performance.

1.3.2.1 LLVM, Clang, and MLIR.

LLVM, originally an abbreviation for Low-Level Virtual Machine, has become one of the most important compiler toolchains, not just in heterogeneous compilation, but in all of computing [102, 109, 110]. As previously mentioned, the LLVM backend intermediate representation and compilation tools form the backbone of many of the other compilers, including the vendor compilers like nvcc.

First developed in 2000 by Chris Lattner at the University of Illinois at Urbana Champaign, LLVM has grow significantly from its initial role as a virtual machine processor. Originally designed for C/C++, LLVM now provides an internal representation and compile time, runtime, and idle time optimization for a multitude other languages. In 2005, Apple began to manage and maintain LLVM for use in their internal projects, but LLVM was later re-licensed under Apache.

LLVM exists as a main project, LLVM-core, and a number of sub-projects, including three specifically relevant to heterogeneous programming approaches, Clang, OpenMP, and MLIR.

First released in 2008, clang is LLVM's own front end compiler for C and C++ [109, 111]. The clang compiler processes C and C++ code and generates LLVM IR, which is then optimized and processed by LLVM. LLVM's OpenMP sub-project implements OpenMP functionality into the LLVM clang compiler. Through the clang and OpenMP sub-projects, LLVM supports heterogeneous computing by compiling C and C++ applications with OpenMP offloading directives.

Though not yet an official LLVM sub-project, OpenACC support is also being developed for LLVM as part of the Clacc (Clang OpenACC) project [112].

Clacc builds on the LLVM OpenMP infrastructure. Clacc accepts C-based OpenACC as input, internally translates to OpenMP, and then generates LLVM intermediate representation using the existing LLVM OpenMP infrastructure.

MLIR (multi-level intermediate representation) is another LLVM project with significant implications for heterogeneous programming [113, 114]. The MLIR project adopts a layered compilation and optimization model, with different MLIR layers, or dialects, that have distinct abstraction levels and areas of focused. These layers can be combined and lowered, from higher abstraction dialects to lower abstraction dialects. Essentially, MLIR offers a reusable abstraction toolbox. A main goal of MLIR is to prevent software fragmentation and improve support for heterogeneous hardware, as the concept of dialects maps well to the ideas of different accelerators. MLIR also aims to provide support for the development of domain-specific programming approaches, which has a straightforward mapping to MLIR dialects and the progressive conversion and lowering structure of MLIR. The previously discussed Tensorflow framework relies on MLIR, and has been a major motivation for the development of the project [115]. Additionally, the Flang project (a FORTRAN-based front-end for LLVM) and Flang's OpenACC support rely on MLIR [116].

LLVM and the clang compiler are featured heavily in this project's research results. In Chapter II, we indirectly rely on LLVM, as the Intel OpenCL SDK for FPGAs uses LLVM internally. In Chapter III, we use clang directly to compile OpenMP applications for several different GPU targets. Finally, in Chapter IV, clang is used to evaluate both OpenMP and OpenCL backends across several exascale-intended platforms, and LLVM is used indirectly during the evaluation of the Intel and AMD OpenMP compilers.

1.3.2.2 GNU C/C++.

The GNU Compiler Collection, commonly referred to as just GCC, is undoubtedly the longest-living and most widespread open-source compiler framework [117] (although Perl is a close second on longevity). It is no surprise then that GCC also plays a role in heterogeneous compilation.

GCC was first released in 1987 as the GNU C Compiler, but has since expanded to incorporate other languages such as C++ and FORTRAN. More recently GCC has worked to develop support for OpenACC [118] and OpenMP offloading models [119]. However, GCC's implementations are not as mature as PGI's OpenACC implementation and LLVM's OpenMP implementation.

1.3.3 Academic, Research, and Custom Compilers.

The last category of heterogeneous compilers we cover are academic project compilers. These projects are typically source-to-source translation compilers, or pre-compilers, that build on or extend existing production-level compiler projects. However, they play a crucial role in the development cycle of heterogeneous programming approaches. We briefly discuss the advantages and disadvantages of research-based compilers, and list a few notable examples.

Advantages Academic compilers are great for prototyping and experimentation of new language features. A production level compiler, either vendor or open source, may take months to push through new features and require several stages of approval. Conversely, an academic compiler is usually owned by a small group of researchers, and new features can be implemented and launched in a few days. Often, new language features are first evaluated in academic compiler settings, and only later re-implemented, or trickled down, into more production-

setting compilers. Most academic compilers also host open-source code on major code repositories.

Disadvantages Academic compilers often struggle with adoption and longevity. Because the projects are owned by a small number of people, small shifts in personnel can have disastrous effects on maintenance of a framework. Also, the compiler frameworks are typically funded by larger projects and grants, and therefore may be dependent on renewal of funding. Finally, because these compilers may be targeting a specific problem area for the research group, they often implement only a subset of the target programming language or approach.

1.3.3.1 ROSE. The ROSE compiler framework is an open-source, research-based, source-to-source transformation compiler developed at LNLL [120, 121]. First published in 1999, ROSE has not suffered from longevity issues, and is still cited frequently in 2020. In 2013, ROSE was used in one of the first initial implementations and evaluations of the OpenMP offloading model, OpenMP specification 4.0 [122].

1.3.3.2 OpenUH. The OpenUH project was managed by the HPCTools group at the University of Houston [123, 124]. OpenUH was based on the Open64 compiler framework [125], and was originally developed as an OpenMP and FORTRAN Coarray compiler. OpenUH did begin support for OpenMP offloading directives for heterogeneous programming, and experimental support for OpenACC on Nvidia and AMD GPUs, but as of 2020 the compiler framework does not seem to be under active development.

1.3.3.3 Omni. The Omni compiler project is maintained and developed by researches at the University of Tsukuba and the RIKEN Center for Computational Science, both in Japan [126, 127]. First released in 1999, the Omni

OpenMP compiler represented one of the first research-oriented implementations of the OpenMP standard. Over time, Omni has shifted to focus on cluster-based OpenMP computing. In 2010, an extension to the Omni project, XacalabeMP [45] integrated a PGAS-model distributed memory approach to OpenMP compilation. Also in 2010, the OMPCUDA project extended the Omni compiler to support compilation of OpenMP code for CUDA GPUs. Later in 2013, initial OpenACC support was added, shortly after the release of the OpenACC standard [128]. The next year, 2014, the XaclableMP and OpenACC extensions were combined to create the XalableACC extension [129], a PGAS-based heterogeneous distributed framework based on OpenACC.

The Omni compiler and its extensions are still under active development. In 2019 and 2020, extensions were made to include FPGA support [130, 131], although this support is still a work in progress.

Although we do not use the Omni compiler in this dissertation, the extensions to support FPGAs are very closely related to the work presented in Chapter II.

1.3.3.4 OmpSs. The OmpSs project, first published in 2011, aimed to support CUDA- and OpenCL-enabled GPUs with OpenMP input[46, 132]. OmpSs is developed and maintained by the Barcelona Supercomputing Center, BCS.

Because OmpSs pre-dated the OpenMP offloading directives, the developers created custom extensions to OpenMP for handling data, based on the StarSs framework [133]. OmpSs was evaluated and extended by a multitude of other works and projects [134, 135, 136], including one comparing OmpSs, OpenMPC, OpenACC, and OpenMP. OmpSs has also been explored for FPGA-based heterogeneous computing [137, 138, 139].

As is obvious from the numerous publications, OmpSs is still undergoing active development and still being used as part of the toolchain for a number of other projects.

1.3.3.5 OpenARC. The OpenARC compiler framework, first published in 2014, is maintained and developed by Oak Ridge National Laboratory [39]. OpenARC is an extension of the OpenMPC framework [140], and like OpenMPC, is built on the Cetus compiler toolchain [141]. OpenARC was originally designed to be the first open-source option for OpenACC compilation, acting as an source-to-source translator that consumes OpenACC C input and generates C and CUDA output. More recently, OpenARC has evolved to accept OpenMP offloading directives as additional inputs, and can generate OpenCL and AMD HIP as output sources, in addition to CUDA.

OpenARC also acts as the core framework for other heterogeneous programming projects. The Compass framework [142] relies on OpenARC to generate ASPEN performance models [143] of heterogeneous applications driven by user annotations and directives. The Iris runtime library, also integrated into OpenARC, is a work in progress that aims to allow multiple accelerators, even with different architectures, to collaborate together to execute a single application.

Figure 4 highlights the various components of the OpenARC compiler.

OpenARC is a core component of every subsequent chapter in this dissertation. In Chapter II, we discuss how OpenARC was extended to support OpenACC-to-FPGA compilation [144, 9, 10]. In Chapter III, we discuss the OpenARC extension CCAMP, developed to provide an interoperable optimization environment for OpenMP and OpenACC compilation. Finally, in Chapter IV

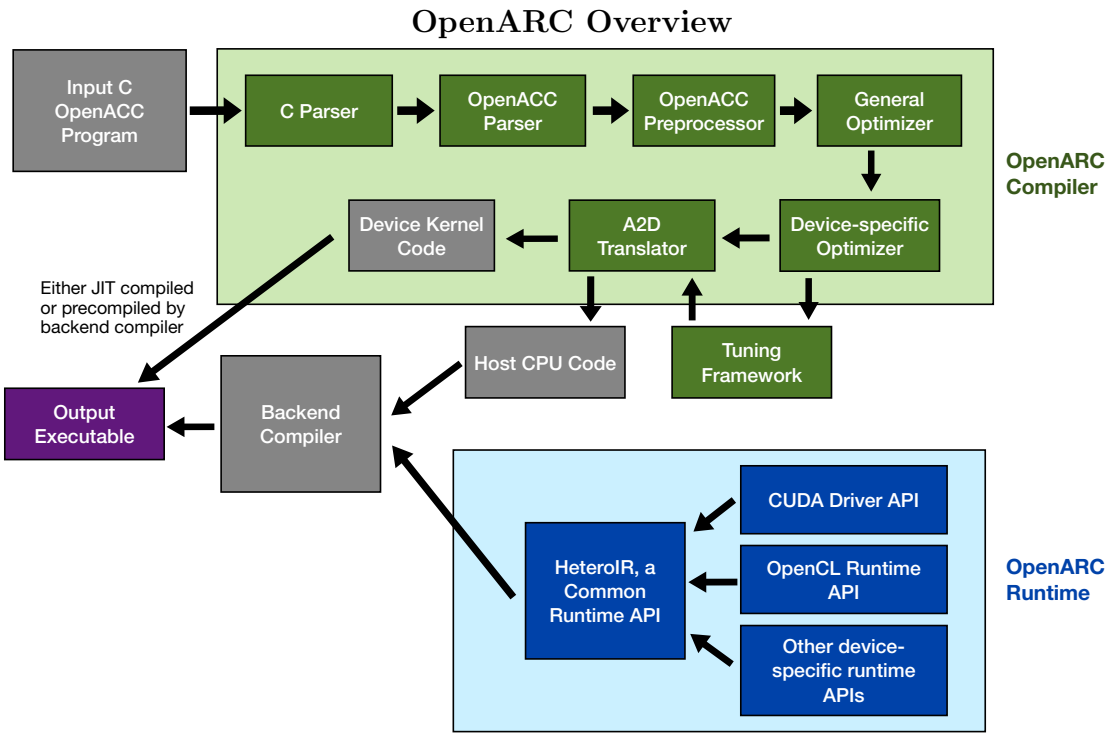


Figure 4. Overview of OpenARC compiler framework

we discuss how OpenARC was used to perform an exploration and evaluation of exascale-intended hardware and software platforms.

1.3.3.6 HPVM. HPVM (Heterogeneous Parallel Virtual Machine) [145, 146, 146] is a research project first published in 2018 originating from the University of Illinois at Urbana-Champaign. On the surface, HPVM is an extension to LLVM with direct support for heterogeneous computation, simplifying the intermediate representation that many of the LLVM-dependent heterogeneous programming approaches rely on.

The HPVM project aims to develop a uniform representation that can capture an array of different heterogeneous architectures, including GPUs, multi-core CPUs, FPGAs, and more. The main components of HPVM include: (1) a dataflow graph-based parallel program representation to capture task and data

parallelism, (2) a heterogeneous compiler intermediate representation that supports optimizations commonly employed on GPU devices, like tiling and loop fusion, and (3) a heterogeneous virtual ISA supporting GPUs, SIMD vectorization, and multicore CPUs.

HPVM is implemented on top of the LLVM project, and aims to provide a valuable new asset, a heterogeneity-focused extension, to the LLVM community.

1.4 Heterogeneous Benchmark Suites

When evaluating heterogeneous programming approaches, typically performance is king. However, measurements of performance are relative, and difficult to compare across different projects, frameworks, or standards. The one control that makes performance comparisons possible are standard benchmarks. In this section, we review several different benchmark suites designed specifically for heterogeneous programming approaches.

1.4.1 Rodinia. First released in 2009, the Rodinia benchmark suite [147] is the oldest among the benchmark sets discussed in this section. Rodinia first released with CUDA and OpenMP versions of computational kernels from several different scientific domains. OpenCL kernels were added next, and after the release of the OpenACC standard, OpenACC versions of several of the kernels were included. The OpenMP kernels were updated to use some of the offloading directives, although they only annotated using directives specific to the Intel Xeon Phi devices, not general GPUs.

The Rodinia benchmarks form the basis of the evaluations in Chapter II.

1.4.2 SPEC Accel. The SPEC Accel [148] benchmark suite was released in 2014. SPEC (Standard Performance Evaluation Corporation) is a non-profit specifically focused on developing and maintaining high-quality

benchmarks. As a result, the SPEC Accel benchmarks are very well organized and documented, and have a robust set of scripts for executing and recording application information. However, the SPEC benchmarks are not open source, and require either a paid commercial license or a free academic licence.

The SPEC Accel benchmark suite is prominently featured in this dissertation's research results, specifically in Chapters III and IV.

1.4.3 Other Heterogeneous Benchmark Suites. In 2010, ORNL released the SHOC (Salable Heterogeneous Computing) benchmark suite [149]. The SHOC benchmarks released with both CUDA and OpenCL versions of several kernels. Unlike Rodinia, SHOC was designed to test applications at scale, not just on a single node.

The Parboil benchmarks [150] were developed by the University of Illinois at Urbana-Champaign and released in 2012. Like the other benchmark suites, Pairboil contains both CUDA and OpenCL code versions. One unique aspect with Parboil is that several different versions of each application are provided with different levels of optimizations. These versions can be used to measure the effectiveness of an automated optimizing compiler.

Also released in 2012, the OpenCL 13 Dwarfs benchmark suite [151] is a realization of Berkely's 13 computational dwarfs in OpenCL [152], where a dwarf is essentially a core computational or communication method or action.

In 2013, EPCC, the Edinburgh Parallel Computing Center, a supercomputing center associated with the University of Edinburgh, released a suite of OpenACC benchmarks [153, 154]. The suite contains low-level operations intended to test and measure the performance of hardware and compilers. The suite also contains a set of software kernels intended to replicate operations most

commonly seen in scientific applications. Although the EPCC Benchmarks also contain OpenMP implementations, these versions are based on non-offloading OpenMP standards, 3.0 and earlier.

Interestingly, the oldest benchmark suite, Rodinia, seems to be the most popular, with nearly an order of magnitude more citations than any of the other benchmark suites. This could be just an artifact of being released first, or from the Rodinia kernels more closely resembling desired scientific applications. However, the Rodinia benchmarks themselves are infrequently updated and fail to capture many of the new language features. This often requires each research project using Rodinia to develop their own updates to the benchmarks. The other benchmark suites face a similar challenge. Several newer benchmark suites have been presented, but all have faced issues with adoption. Moving forward, development, adoption, and maintenance of high-quality benchmark suites could significantly improve the productivity of developers.

Listing 1.2 Example OpenCL C Application

```
#include <stdlib.h>
#include <CL/cl.h>

const char* programSource =
"__kernel␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
"void␣vecadd(__global␣int␣*A,␣__global␣int␣*B,␣__global␣int␣*C)␣\n"
"{␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
"␣␣␣int␣idx␣=␣get_global_id(0);␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
"␣␣␣C[idx]␣=␣A[idx]␣+␣B[idx];␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
"}␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
;

int main() {
    int *A = NULL; int *B = NULL; int *C = NULL;

    const int elements = 2048;
    size_t datasize = sizeof(int)*elements;
    A = (int*)malloc(datasize); B = (int*)malloc(datasize); C = (int*)malloc(datasize);
    B = (int*)malloc(datasize);
    C = (int*)malloc(datasize);
    for(int i = 0; i < elements; i++) {
        A[i] = i; B[i] = i;
    }

    cl_uint numPlatforms = 0;
    cl_int status = clGetPlatformIDs(0, NULL, &numPlatforms);
    cl_platform_id *platforms =
        (cl_platform_id*)malloc(numPlatforms*sizeof(cl_platform_id));
    status = clGetPlatformIDs(numPlatforms, platforms, NULL);

    cl_uint numDevices = 0;
    cl_device_id *devices = NULL;
    status = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, 0, NULL, &numDevices);
    devices = (cl_device_id*)malloc(numDevices*sizeof(cl_device_id));
    status = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, numDevices, devices, NULL);

    cl_context context = clCreateContext(NULL, numDevices, devices, NULL, NULL, &status);
    cl_command_queue cmdQueue = clCreateCommandQueue(context, devices[0], 0, &status);

    cl_mem bufferA = clCreateBuffer(context, CL_MEM_READ_ONLY, datasize, NULL, &status);
    cl_mem bufferB = clCreateBuffer(context, CL_MEM_READ_ONLY, datasize, NULL, &status);
    cl_mem bufferC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, datasize, NULL, &status);
    status = clEnqueueWriteBuffer(cmdQueue, bufferA, CL_FALSE, 0, datasize, A, 0, NULL, NULL);
    status = clEnqueueWriteBuffer(cmdQueue, bufferB, CL_FALSE, 0, datasize, B, 0, NULL, NULL);

    cl_program program = clCreateProgramWithSource(context, 1, (const char**)&programSource, NULL, &status);
    status = clBuildProgram(program, numDevices, devices, NULL, NULL, NULL);
    cl_kernel kernel = NULL;
    status  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufferA);
    status |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufferB);
    status |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufferC);

    size_t globalWorkSize[1];
    globalWorkSize[0] = elements;
    status = clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL, globalWorkSize, NULL, 0, NULL, NULL);
    clEnqueueReadBuffer(cmdQueue, bufferC, CL_TRUE, 0, datasize, C, 0, NULL, NULL);

    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(cmdQueue);
    clReleaseMemObject(bufferA);
    clReleaseMemObject(bufferB);
    clReleaseMemObject(bufferC);
    clReleaseContext(context);

    free(A); free(B); free(C); free(platforms);free(devices);
}
```

Listing 1.3 Example OpenACC C Application

```
1  int main() {
2
3    int SIZE = 1024;
4
5    float *a, *b;
6    a = malloc(sizeof(float) * SIZE);
7    b = malloc(sizeof(float) * SIZE);
8
9    for (int i = 0; i = < SIZE; ++i) {
10     a[i] = 0;
11     b[i] = // some initial value
12   }
13
14   // Data Directives
15   #pragma acc data copyin(b[0:SIZE]) copyout(a[0:SIZE])
16
17   // Compute Directive
18   #pragma acc parallel loop collapse(2)
19   for (int i = 1; i <= SIZE; i++)
20     for (int j = 1; j <= SIZE; j++)
21       a[i][j] = (b[i - 1][j] + b[i + 1][j] + b[i][j - 1] + b[i][j + 1]) / 4.0f;
22 }
```

Listing 1.4 Example OpenMP C Application

```
1
2  int main() {
3
4    int SIZE = 1024;
5
6    float *a, *b;
7    a = malloc(sizeof(float) * SIZE);
8    b = malloc(sizeof(float) * SIZE);
9
10   for (int i = 0; i = < SIZE; ++i) {
11     a[i] = 0;
12     b[i] = // some initial value
13   }
14
15   // Data Directives
16   #pragma omp target data map(to:b[0:SIZE], from:a[0:SIZE])
17
18   // Compute Directive
19   #pragma omp teams parallel for collapse(2)
20   for (int i = 1; i <= SIZE; i++)
21     for (int j = 1; j <= SIZE; j++)
22       a[i][j] = (b[i - 1][j] + b[i + 1][j] + b[i][j - 1] + b[i][j + 1]) / 4.0f;
23 }
```

CHAPTER II

DIRECTIVE-BASED PROGRAMMING AND OPTIMIZATIONS FOR HIGH-PERFORMANCE COMPUTING WITH FPGAS

This chapter contains previously published material with co-authorship. All of the presented research in this chapter was conducted as a collaboration between the University of Oregon and Oak Ridge National Laboratory. Sections 2.1- 2.5 describe work related to the OpenACC-to-FPGA framework that was presented at ICS 2018 [9], AsHES 2020 [10], and in PARCO 2021 [11]. The material from these publications was reorganized in this dissertation for a more fluid presentation. For all three publications, Seyong Lee was instrumental in the conceptualization of the projects and provided continued support, suggestions, and advice throughout the projects with weekly meetings. Dr. Lee also assisted with revisions to the documents, and sometimes portions of the writing, typically in the introductions and conclusions. Allen Malony and Jeffrey Vetter both provided high-level guidance and advice during all three projects. They both also assisted with revisions, and contributed information for the introduction and conclusions sections. Jungwon Kim assisted with the related works section in the ICS 2018 [9] publication.
I researched, designed, and implemented the optimizations for the ICS 2018 submission. I also collected all data, performed all experiments, and did the bulk of writing for all three publications.

Section 2.5 describes an FPGA portability study presented at IWOCL 2021 [12]. I was a secondary author on this publication. Anthony Cabrera led this project and organized several meetings with all co-authors. Dr. Cabrera was also responsible for writing the first draft of most of the publication, although Aaron Young was responsible for writing materials related to the CFD benchmark. I was

responsible for evaluating the SRAD and Hotspot benchmarks, and writing the corresponding sections in the document. I also proofread the entire document, and contributed to the related works sections. The material in Section 2.5 has been reduced from the original IWOCL publication to primarily focus on the areas of the project where I directly contributed and sections that I either wrote or heavily revised.

2.1 Background on FPGAs as Heterogeneous Accelerators

As discussed in Chapter I, accelerator-based heterogeneous computing, which typically employs devices such as GPUs and many-core processors, has become a mainstream approach in high-performance computing (HPC) to solve performance, power efficiency, reliability, and cost issues caused by increasing power densities in conventional von-Neumann architectures. More recently, reconfigurable computing that uses FPGAs and coarse-grained reconfigurable devices has received renewed interest due to the unique combination of performance and energy efficiency through flexible hardware customizations. FPGAs' reconfigurable nature allows these architectures to be customized to match the needs of a given application and achieve much higher energy efficiency and/or performance gains compared with conventional CPUs and GPUs. As a result, FPGAs have been deployed in various application domains, such as finance [155], database systems [156], machine learning [157], image processing [158], graph analysis algorithms [159], and others. Moreover, recent trends in FPGA technologies—such as supporting hardened floating-point data signal processing blocks and integrating CPUs, GPUs, and FPGAs as a new system-on-chip devices—make FPGA-based high-performance reconfigurable computing more attractive for serious exploration in scientific simulation and data analytics.

2.1.1 FPGA Hardware.

FPGAs are composed of digital signal processing (DSP) blocks, registers, adaptive look-up tables (ALUTs), and other specialized hardware components. At runtime, the FPGA is configured to use a subset of these hardware components using programmable interconnects. This runtime-configuration property provides several advantages for FPGAs compared to other accelerators. First, specific resources can be allocated to meet the needs of specific applications, leading to performance improvements. Additionally, because only crucial components are configured, FPGAs can maintain a low-power state. However, configuring the FPGA for specific applications has traditionally required programming in HDLs at the register-transfer level (RTL).

Figure 5 shows an example layout. While this layout is actually an abstraction layer pre-programmed to the device as part of the Intel OpenCL SDK for OpenCL (discussed in Section 2.1.3.1 below), it does highlight the hardware features accessible when using an FPGA in the context of this dissertation.

2.1.2 Traditional FPGA Programming Approaches.

Despite a huge potential to achieve high performance and flexibility with limited power consumption, FPGAs have not been widely used in HPC [160]. The most significant obstacle to realizing their potential is the lack of high-level programming models that hide implementation details. Programming FPGAs normally requires substantial knowledge about the underlying hardware design and use of low-level hardware description languages (HDLs) such as VHDL and Verilog.

RTL FPGA programming in VHDL or Verilog is inaccessible to most application programmers because it requires in-depth knowledge of the FPGA, such as cycle-by-cycle descriptions of hardware, and hardware-clock timing considerations. It also requires scientific application developers who may have

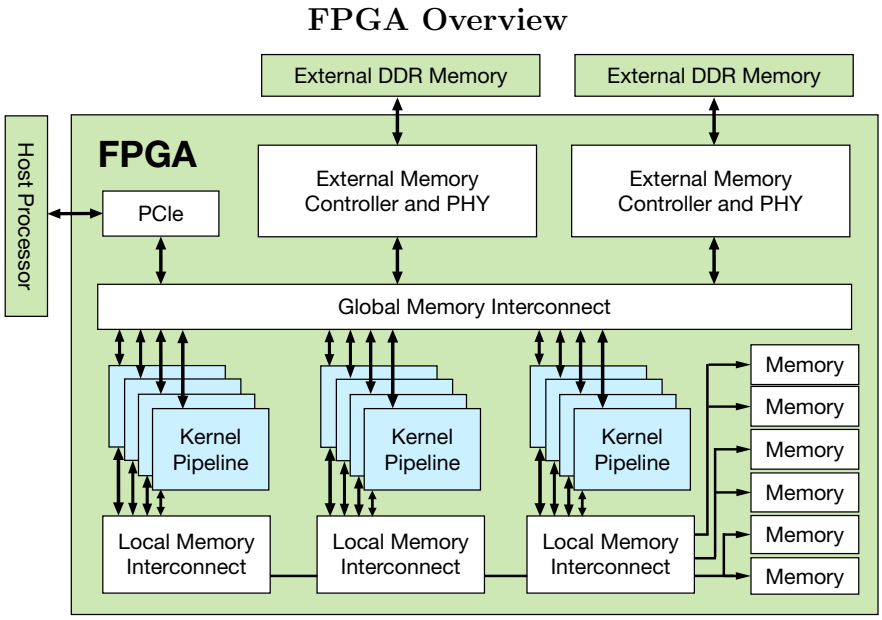


Figure 5. FPGA hardware components available through Intel OpenCL SDK for FPGAs

limited expertise on hardware architectures to design algorithms at the register transfer level (RTL) by describing them using state machines, data paths, clock management, device-specific interfaces to external memory, buffering, and so on.

2.1.3 Contemporary FPGA Programming Models. In this section we provide an overview to two high-level programming approaches for scientific computing with FPGAs: OpenCL, and as a result of the framework described in this chapter, OpenACC.

2.1.3.1 OpenCL. To alleviate the programmability concern in FPGA computing, several high-level synthesis (HLS) programming models have been proposed [161, 162, 163, 164, 165, 166]. OpenCL (Open Computing Language), introduced in Chapter I Section 1.2.2, is the first standard programming model that is functionally portable across diverse heterogeneous architectures and has been adopted by major FPGA vendors [37]. Two leading FPGA manufacturers,

Intel/Altera [108] and Xilinx [167], have provided an OpenCL-based SDK for their FPGA devices. The source-level portability of OpenCL in theory allows programmers to write applications once and run them on any OpenCL-compliant hardware accelerators, such as CPUs, GPUs, Xeon Phis, DSPs, and FPGAs.

Despite its potential to offer better programmability and portability than other HLS approaches, programming and optimizing FPGAs with OpenCL is still considered to be very complex and difficult due to the semantic gap between the OpenCL abstraction and the low-level hardware design. For example, the current OpenCL abstraction does not provide a straightforward method for programmers to express specific hardware features, such as shift registers, hardware channels, and pipeline delays. Instead, the underlying OpenCL compiler implicitly derives and synthesizes all hardware logic from an input program, and there are many practical limits in compilation for finding optimal hardware designs. Consequently, existing FPGA OpenCL compilers can be very sensitive to specific code patterns. One way to solve this problem is to lower the programming abstraction level offered by OpenCL to expose the low-level hardware design to the programmers. However, lowering the programming abstraction would sacrifice the portability benefits of OpenCL and negatively affect its programmability. In summary, OpenCL is too high-level for ideal performance, while at the same time being too low-level for ideal scientific programming.

As an aside, compilation times using the Intel SDK are significantly longer than traditional CPU or GPU compilation times, often taking several hours. This is generally true of all HLS tools. However, the Intel SDK does provide a significant amount of information about the application and how it will be mapped to hardware before attempting a full compilation. The estimated resource usages

and design layout, which are neatly presented in HTML format, were very useful for guiding optimizations, even when working at the OpenACC level.

2.1.3.2 OpenACC. OpenACC [168, 169] addresses these challenges faced by OpenCL FPGA SDKs. OpenACC (discussed in Chapter I, Section 1.2.4) is a directive-based, portable, parallel programming model for a wide variety of hardware accelerators. The model outsources device-specific implementation details to the compiler to reduce the required programming effort and increase performance portability. The OpenACC API—which consists of compiler directives, library routines, and environment variables—allows programmers to augment applications with information, exposing available parallelism within an application. A core OpenACC facility is to offload the burden of mapping parallelism directly to devices from the user to the underlying compiler. Because of its simplistic API, maintainability, usability, and portability, OpenACC is often considered as an alternative to lower level accelerator programming models.

An example of the OpenACC API in practice is shown in Chapter I, Section 1.2.4. Typically, a compute directive such as *#pragma acc parallel* annotates a for loop or other kernel region intended to be offloaded to a device. Additional clauses can be appended to this directive to apply specific types of parallelism or optimizations. Other common OpenACC directives include *#pragma acc data* for specifying data that should be transferred to and from a device.

To address the problems caused by the aforementioned semantic gap, a directive-based, high-level programming and optimization framework for efficient FPGA computing is presented in this chapter. This framework takes a standard, portable OpenACC program as input and generates an output OpenCL code, which the underlying OpenCL compiler further compiles into an FPGA hardware

configuration file. The proposed framework solves the semantic-gap issue using directive-based, high-level FPGA-specific optimizations in which programmers provide important characteristics of the input program via a set of directives. The framework then generates specific OpenCL code patterns in such a way that the underlying back-end OpenCL compiler can infer known FPGA programming paradigms, including shift registers, hardware pipelines, and sliding windows. The proposed OpenACC-to-FPGA translation framework offers enough abstraction over low-level hardware designs and complex OpenCL programming syntax and also provides high-level control over various FPGA-specific optimizations. As a result, the programmer can specify FPGA optimizations with user-friendly, high-level OpenACC directives and keywords and will leave the lower-level error-prone OpenCL FPGA-specific syntax generation to the compiler. The implementation details of the OpenACC-to-FPGA framework are presented in the following section (Chapter II, Section 2.2).

2.2 The OpenACC-to-FPGA Framework

OpenACC-to-FPGA is a directive-based, high-level FPGA-specific optimization framework, which consists of directive extensions and corresponding compiler optimizations to generate more efficient FPGA hardware configuration files from a high-level OpenACC input code. The proposed directives are designed for programmers either to provide key information necessary for the compiler to automatically generate output OpenCL code that enables FPGA-specific optimizations, or to control important tuning parameters of those optimizations.

We want to clarify how we use the term *optimization*. There is a distinction between manually-written OpenCL optimizations (like the shift-register reduction pattern and the sliding window pattern) and compiler optimizations implemented

in OpenARC (like the reduction transformation and window transformation). In the Intel FPGA SDK for OpenCL [108], programmers can use FPGA-specific features like shift registers and sliding windows by programming in OpenCL using very specific patterns. These programming patterns are non-intuitive for most OpenCL programmers and can be error-prone. Currently, the OpenCL compiler does not offer a directive- or compiler-based approach to generate these programming patterns. A primary goal of the research presented in this chapter is to create transformations in OpenARC that automatically generate these non-intuitive programming patterns from OpenACC directives. Doing so greatly simplifies the implementation of FPGA-specific features, and allows programmers without knowledge of shift registers and sliding windows to create more efficient FPGA designs.

The following optimizations were inspired by the Intel OpenCL SDK documentation [108]. We primarily chose to implement in OpenARC optimizations that potentially apply to a wide range of applications; for example, loop collapsing, scalar reduction, and branch-variant code motion optimizations are generally beneficial when they are applicable, whereas the sliding window optimization can benefit applications with stencil patterns.

This section provides a high-level overview and categorical classification of the different optimizations developed by Lee et al. [144] and by Lambert et. al [9]. The optimizations are divided into three primary categories: (1) automatically applied optimizations requiring no user intervention, (2) re-purposed directives in which existing OpenACC directives are re-implemented in an FPGA-specific way, and (3) directive extensions in which FPGA-specific directives are developed outside the established OpenACC standard.

We do note that all of the FPGA-specific optimizations are applied in the context of compute directives, and that the OpenACC data directives for a standard OpenACC applications are typically sufficient for an FPGA execution. The one exception is the *pipe* clause used as part of the channels optimization described below, which could be considered a data directive and would replace the analogous OpenACC *copyin* and *copyout* clauses.

Before discussing the OpenACC-to-FPGA framework's optimizations, we first briefly discuss the implementation of the framework itself.

2.2.1 Implementation in OpenARC. The OpenACC-to-FPGA translation framework discussed in this work is built inside the OpenARC compiler framework [39]. As discussed in Chapter I, Section 1.3.3.5, OpenARC is a research-oriented OpenACC compiler that specializes in rapidly prototyping new optimizations, API features, and device-support for emerging technologies. This makes OpenARC an ideal platform for the initial implementation of OpenACC-to-FPGA translation, which was first introduced by Lee et al. [144].

OpenARC takes an input C program that is annotated with OpenACC directives, performs several optimization and translation passes, and generates an optimized output host and kernel code in CUDA or OpenCL. The CUDA or OpenCL output is then further compiled using a low-level device compiler, such as NVCC or Intel's OpenCL compiler. In the context of the OpenACC-to-FPGA framework covered in this chapter, OpenARC is used to generate OpenCL specific to Intel FPGAs and to apply FPGA-specific optimizations.

The original baseline translation for the OpenACC-to-FPGA framework is not part of this dissertation's research, as it was developed independently by the primary OpenARC developer Seyong Lee, in the work by Lee et al. [144].

Furthermore, some of the optimizations presented in the following sections were also developed independently from this dissertation's research, and they are cited appropriately (again Lee et al. [144]) in the corresponding sections. However, the bulk of optimizations present in the OpenACC-to-FPGA framework were developed as part of this dissertation, and are cited respectively as Lambert et al. [9].

2.2.2 Automatic Optimizations. The first class of optimizations in the OpenACC-to-FPGA framework represents optimizations for which no user intervention is required. These optimizations can be safely applied any time the compiler encounters specific constructs and are applied independently from any user-supplied directives.

2.2.2.1 Dynamic Memory Transfer Alignment. In a typical FPGA-based heterogeneous system, an FPGA is attached to the host CPU via PCIe bus as a discrete device with a separate memory. Therefore, for a device kernel to access the host data and vice versa, data should be explicitly transferred between the host and device memory. Existing FPGA OpenCL runtimes, such as Intel OpenCL runtime, use direct memory access (DMA) for higher throughput and lower latency. To exploit DMA, the host-side buffer and device-side buffer should be aligned. Although device buffers are automatically allocated in an aligned way, host buffers should be allocated with special memory allocators (e.g., *posix_memalign()* in Linux). Even if both host and device buffers are allocated in an aligned way, the transfer of partial arrays might not exploit DMA if at least one of the start addresses is not aligned. The OpenACC-to-FPGA framework runtime dynamically analyzes memory alignment and employs temporary buffers to satisfy alignments without user interference, as described in Lee et al. [144].

The dynamic memory transfer alignment optimization was designed and implemented by Lee et al. [144]. However, the optimization is used in this dissertation's research to evaluate the OpenACC-to-FPGA framework on different FPGA architectures, specifically in Lambert et al. [9] and Lambert et al. [11].

2.2.2.2 Boundary Check Elimination.

When an OpenACC compute region is translated into a device kernel, each iteration in a work-sharing loop will be mapped to a device thread (work-item in OpenCL) according to the OpenACC execution model. If the total number of device threads is not the same as the number of corresponding loop iterations, then the device kernel should be executed so that only device threads with valid mapping execute the loop body, which is usually implemented using control statements. Generally, control flow divergence by control statements is less of an issue in FPGA computing than in GPU computing because the reconfigurability in FPGA can completely eliminate the diverging control paths of thread executions by using hardware predicates if the conditional structure is simple enough. However, if the device kernel has complex control structures such as thread-dependent backward branching, then the underlying OpenCL compiler cannot flatten the control structures, which can significant degrade performance by disallowing various advanced compiler optimizations, such as kernel vectorization. To alleviate the burden for the underlying OpenCL compiler to flatten the control structure, a compiler pass was developed that uses built-in symbolic analysis tools to check and eliminate unnecessary loop-boundary check code at compile time.

The boundary check elimination optimization was designed and implemented by Lee et al. [144]. However, the optimization is used in this dissertation's research

to evaluate the OpenACC-to-FPGA framework on different FPGA architectures, specifically in Lambert et al. [9] and Lambert et al. [11].

2.2.2.3 Branch-Variant Code Motion Optimization. Among devices used as hardware accelerators, the concept of directly managing hardware logic generation at the programming level is unique to FPGAs. Because programming logic is mapped directly to FPGA hardware, programming patterns and coding styles that may only affect source code length on devices like GPUs or CPUs can make concrete differences in FPGA resource usage.

Loop-invariant code motion is a common computation-reduction optimization applied across all hardware devices. In the same fashion, we can apply branch-invariant code motion. This optimization normally would not lead to a performance benefit for more traditional devices like CPUs and GPUs because the number of operations executed remains unchanged. However, when compiling for FPGAs, logic from both branches is required to be implemented in hardware, leading to increased resource usage from the redundant code. Therefore, factoring out branch-invariant code can reduce the overall resources required to implement the hardware logic, and thus the Intel OpenCL compiler supports the branch-invariant code motion optimization.

To reduce the resource usage further, we propose a branch-variant code motion optimization, which transforms branch-variant codes and factors out codes with the same computation patterns. Listing 2.1, Listing 2.2, and Listing 2.3 illustrate how the proposed optimization works: Listing 2.1 shows an input code that contains branch-variant codes, so the traditional branch-invariant code motion optimization cannot be applied. However, if we transform the code into a form in Listing 2.2, codes with common computation patterns can be hoisted out of

the conditional, as shown in Listing 2.3. The key part of this optimization is identifiying a common computation pattern, which is an expression that exists in all branch bodies and performs the same sequence of computations with branch-variant operands. For example, in assignment expressions, a common computation pattern could be statements whose *lvalues* (i.e., an object that appears on the left side of the expression) is branch-invariant, whereas the right side of the expression is branch-variant. To identify these patterns, the compiler can transform the input conditional code into a form where non-constant operands in expressions within branch bodies, except for the left sides of the assignment expressions, which are replaced with temporary variables, even though variable assignments should be done in a specific order (Listing 2.2). Then, common computation patterns existing in all branch bodies are factored out of the conditional (Listing 2.3). If the left-hand side of an assignment statement is used as an input to a subsequent statement within the branch bodies, the assignment statement and subsequent statement can be factored out only if both statements are common computation patterns. Otherwise, the conditional should split into multiple conditionals. If the conditional itself is dependant on the common statement, the code motion optimization does not apply. We can see that the number of addition and multiplication operations that require hardware implementation is halved in Listing 2.3, compared to Listing 2.1, resulting in lower FPGA resource usage.

Listing 2.1 Code Motion: Input conditional

```
if (condition) {
 output += A[i] * B[i];
} else {
 output += A[i−1] * B[i−1];
}
```

Listing 2.2 Code Motion: Modified conditional

```
if (condition) {
 t1 = A[i]; t2 = B[i];
 output += t1 * t2;
} else {
 t1 = A[i−1]; t2 = B[i−1];
 output += t1 * t2;
}
```

Listing 2.3 Code Motion: After code motion

```
if (condition) {
 t1 = A[i]; t2 = B[i];
} else {
 t1 = A[i−1]; t2 = B[i−1];
}
output += t1 * t2;
```

FPGA resource usage can indirectly impact runtime performance in several ways. High resource usage can cause the hardware design to suffer from routing congestion, negatively affecting performance. Also, applications with higher base resource usage benefit less from loop unrolling techniques because they quickly exhaust FPGA resources even with small unroll factors. Section 2.4.5 presents an example of this behavior.

The Intel OpenCL SDK compiler automatically performs simple branch-invariant code optimizations like the one in the listing above. However, in more complicated code like the HotSpot application, the optimization is not automatically applied by the OpenCL compiler. In these more complicated examples, OpenARC's high-level IR allows us to perform these kinds of optimizations automatically. OpenARC can apply branch-invariant whenever the compiler can guarantee invariance, as long as the motion takes place within an enclosing compute region.

The branch-variant code motion optimization was designed as part of this dissertation's research, referenced in Lambert et al. [9]. However, this optimization was never fully implemented in OpenARC. Although this optimization led to

significant performance improvements for HotSpot (as we see later in this chapter), other evaluated applications did not benefit directly from this optimization, making it a low priority for actual implementation.

2.2.3 Re-purposed Directives. The second class of optimizations in the OpenACC-to-FPGA framework represents optimizations that users can optionally apply using existing OpenACC directives and clauses. Many clauses are typically implemented by compilers in a specific way to optimize GPU performance. In the OpenACC-to-FPGA framework, these clauses were re-implemented to optimize FPGA performance without changing their syntax or context from a programming perspective.

2.2.3.1 Single Work-Item Optimization. A common approach in general CPU- and GPU-based computing is to develop massively parallel applications that can be partitioned across multiple computation units. Although this approach can be effective when targeting FPGAs, FPGAs alternatively offer a single-threaded approach, which is generally preferred for efficient FPGA computing. Because FPGAs can leverage deeply pipelined execution, single-threaded pipeline-parallel implementations can outperform their multi-threaded counterparts in many situations. This contrasts with GPU execution, which explicitly relies on multi-threaded execution. Because OpenACC was primarily developed with a focus on GPU execution, the default execution model assumes multi-threaded parallelism using multiple gangs, workers, and/or vectors, which can be configured using OpenACC directive clauses.

In OpenCL terminology, the massively parallel or multiple work-item approach is known as an NDRange kernel, and the single-threaded approach is referred to as a single work-item kernel [108]. Although OpenACC currently

supports directives for sequential execution, it does not currently have a specific directive for single work-item execution. However, by using existing OpenACC directives created for controlling the number of threads, we can allow a user to indicate that a region should execute in a single work-item fashion without introducing an additional directive.

We can see two examples of these directives in Listing 2.4, one using the *parallel* annotation (lines 1-2), and another with the *kernels* annotation (5-6). We have modified OpenARC to ensure that the presence of these directives leads to single work-item executions. To execute an OpenACC compute region in a single work-item fashion, the numbers of *gangs*, *workers*, and *vectors* should be explicitly set to *1*, respectively. The latest OpenACC standard (V2.6) [169] introduces a new *serial* construct, which indicates a code region should be executed in a single-threaded manner. Although the pipeline-parallel execution of FPGA single work-item kernels is not strictly single-threaded, we have extended the OpenACC-to-FPGA framework to also accept the *serial* clause (lines 3-4, 7-8 in Listing 2.4) to indicate a single work-item kernel. By setting the *num_gangs*, *num_workers*, and *vector_length* clauses of an OpenACC parallel directive to 1, or by using the OpenACC *serial* directive, OpenARC can generate the appropriate OpenCL code for the underlying back-end compiler to correctly infer a pipeline parallel execution.

Some applications, like embarrassingly parallel algorithms, are well-suited to NDRange execution. For other algorithms with data dependency or data reuse across work-items, the simple single work-item optimization alone may increase performance when executing on an FPGA. In addition to the stand-alone benefits, this optimization is notable because it is a prerequisite for the following optimizations (also discussed in this section): the collapse optimization

(Section 2.2.3.2), reduction optimization (Section 2.2.3.3), and sliding window optimization (Section 2.2.4.4).

The single work-item optimization was designed and implemented as part of this dissertation's research, first referenced in Lambert et al. [9] and later updated in Lambert et al. [11].

Listing 2.4 OpenACC Single work-item directives

```
1 #pragma acc parallel num_gangs(1) num_workers(1) vector_length(1)
2 {  ...  }
3 #pragma acc parallel serial
4 {  ...  }
5 #pragma acc kernels loop gang(1) worker(1) vector(1)
6 {  ...  }
7 #pragma acc kernels loop serial
8 {  ...  }
```

2.2.3.2 Collapse Optimization.

In a massively parallel computing approach, a loop collapse optimization is commonly used either to increase the amount of computations to be parallelized or to change the mapping of iterations to processing units. Loop collapsing is a common optimization used across several directive-based languages, including OpenMP [170] and OpenACC. In this optimization a compiler combines two tightly nested loops into a single loop, which typically requires the original iteration variables to be recalculated at each iteration. In a multi-threaded context, this recalculation can be done using division and modulus operations, deriving the old iteration index values from the collapsed iteration index value.

Loop collapsing is already a part of the OpenACC standard, and OpenARC supports the collapse clause. In Listing 2.5, we see a pair of perfectly nested loops with a collapse clause and single work-item directives. In the standard OpenARC implementation (V0.11), collapsing of perfectly nested loops is achieved by creating a new loop expression with a newly defined iteration variable. OpenARC

Listing 2.5 OpenACC nested loops with collapse clause

```
1  #pragma acc parallel loop num_gangs(1) num_workers(1) vector_length(1) collapse(2)
2  for (i = 0; i < M; i++)
3    for (j = 0; j < N; j++) {   ...   }
```

Listing 2.6 OpenACC loop after collapse transformation

```
1  // Traditional transformation
2  #pragma acc parallel loop num_gangs(1) num_workers(1) vector_length(1)
3  for (iter = 0; iter < M*N; iter++)
4  {  i = iter / N; j = iter % N;
5     ...
6  }
7
8  // FPGA-specific transformation
9  i = 0; j = 0;
10 #pragma acc parallel loop num_gangs(1) num_workers(1) vector_length(1) firstprivate(i,j)
11 for (iter = 0; iter < M*N; iter++)
12 {  ...
13    j++; if (j == N) { j = 0; i++; }
14 }
```

recalculates the values of the original iteration variables at each iteration using division and modulus operators.

In an FPGA context, these division and modulus operations are relatively expensive in terms of execution time and resource usage. However, in a single work-item context, recalculating at each iteration is unnecessary. If the given kernel is executed in the single work-item context, the OpenACC-to-FPGA framework extensions to OpenARC generate a row and column counter approach when encountering collapse clauses instead of using the costly division and modulus approach. These row and column counters are implemented as integers, one representing each loop that was collapsed, and incremented each iteration using relatively inexpensive integer additions. We can see the resulting OpenACC code after applying the OpenACC-to-FPGA collapse optimization in Listing 2.6.

The FPGA-specific collapse optimization can be automatically applied any time loop collapsing occurs within a single work-item execution context. Because the row and column counters create dependencies within the loop, in

multi-threaded contexts we revert to the traditional collapse transformation. We support application of the collapse optimization in conjunction with our reduction (Section 2.2.3.3) and sliding window (Section 2.2.4.4) optimizations. Integrating these optimizations allows application of the reduction (Section 2.2.3.3) and sliding window (Section 2.2.4.4) optimizations to a wider variety of benchmarks containing nested loops, without the performance penalty from OpenARC's traditional collapse transformation.

The collapse optimization was designed and implemented as part of this dissertation's research, referenced in Lambert et al. [9].

2.2.3.3 Reduction Optimization.

Scalar reductions are common patterns used in many algorithms, such as Rodinia's SRAD [147], to compute averages, find maximum values, and so on. Because of their popularity in applications, scalar reductions represent an operation commonly optimized by compilers. For implementations that target multi-threaded CPUs or GPUs, this optimization is typically a tree-based approach. The leaves represent the array of values, and the roots represent the combination of those values by some scalar operation. This tree-based implementation can also be used in an FPGA context and may outperform a straightforward serialized approach.

However, because a pipeline-parallel approach is often more efficient than a massively parallel approach when executing on an FPGA, an alternate FPGA-specific strategy to the scalar reduction is required. In this approach, partial sums are accumulated in a shift register, and then a final value is computed by doing a traditional reduction over the partial sums. We next describe the code transformations to realize shift-register-based reductions in the OpenACC-to-FPGA framework.

Our reduction optimization compiler technique allows users to utilize single work-item kernels and shift registers in OpenACC using only previously existing directives. When using OpenACC to target an FPGA device, the user must first indicate a single work-item execution (Section 2.2.3.1). Within a single work-item compute region, the user can then annotate any loop with the OpenACC reduction directive and a supported reduction operation. Finally, to increase the performance the user can also append an optional unroll annotation, at the cost of additional FPGA resources. Under these circumstances, we can safely and efficiently apply our reduction optimization to implement the FPGA-specific shift-register based reduction. We can see an application of the OpenACC FPGA-specific sum reduction in Listing 2.7, with N referring to the desired level of replication. OpenARC currently supports addition, multiplication, and maximum and minimum value operations for FPGA-specific reductions.

For FPGA execution, scalar reductions are an example of programming patterns where the single work-item optimization (Section 2.2.3.1) alone does not increase performance relative to the traditional NDRange implementation. Because most floating point operations on an FPGA require multiple clock cycles, traditionally programmed scalar reductions perform poorly in the pipeline parallel model or single work-item approach (Section 2.2.3.1). This results from the pipeline stalling each iteration until the dependency on the reduction variable is resolved.

These pipeline stalls during loop execution are formalized in the Intel FPGA SDK documentation by the term *initiation interval*, or *II* [108]. The initiation interval specifically refers to the number of FPGA clock cycles that a pipeline is stalled to launch each successive iteration of a loop execution. A loop with several loop-carried dependencies, like scalar reduction, may have a high *II*, while a loop

Listing 2.7 OpenACC sum reduction

```
1 #pragma acc parallel loop num_gangs(1) num_workers(1) vector_length(1) reduction(+:sum)
2 #pragma unroll N
3 for (int i = 0; i < SIZE; ++i)
4 {   sum += input[i];   }
```

without dependencies may have a lower *II*. When executing in a loop-pipeplined single work-item approach, an *II* of 1 leads to optimal performance, indicating that successive iterations are launched every clock cycle.

The stand-alone single work-item approach does not outperform the multi-threaded tree-based method for scalar reductions on an FPGA. However, a sufficiently sized shift register in addition to this approach can significantly improve performance. In the shift-register approach to scalar reductions, we use the shift register to accumulate partial results as we iterate over the input array. This is followed by a standard reduction over the much smaller shift-register array. This approach increases the reduction variable dependence distance, relaxing the loop-carried dependency on the reduction variable. As a result, the reduction loop attains the desired *II* of 1. The exact shift-register size or depth required depends on the data type, reduction operation, and unrolling or replication factor.

Fortunately, the underlying Intel OpenCL compiler provides information about loop initiation intervals at compile time that can be used to determine an appropriate shift-register depth. With this information, we performed a number of tests with different reduction configurations, and made some general observations about the relationships between the data type, reduction operation, unrolling factor, and their effects on the shift register depth required to attain the desired *II* of 1. For example, on the Stratix V FPGA, we observe that without shift registers or loop unrolling, scalar reduction using single precision floating point addition leads to an *II* of 8 cycles, while using double-precision floating point multiplication

leads to an *II* of 16 cycles. We also observe that loop unrolling acts as a multiplier to the initiation interval. For example, an unroll factor of 4 in the previous example leads to an *II* of 32 and 64 cycles, respectively. From these observations, we expect the following to be valid:

$$register\ depth \approx (operator\ latency) * (unroll\ factor) \tag{2.1}$$

In the equation above, *register depth* refers to the expected size of the shift registers required to attain an *II* of 1, and *operator latency* refers to the device-specific cost of the data type and operation used. This equation along with pre-calculated operator costs are used in the reduction optimization to calculate efficient shift register depths. However, after compiling reduction codes with different configurations, we find that the following unexpected equation holds true:

$$register\ depth \approx \frac{(operator\ latency) * (unroll\ factor)}{2} \tag{2.2}$$

That is, by halving the expected minimum register depth required for an *II* of 1, we still attain an *II* of 1.

Because of the significant performance advantages of launching successive iterations every cycle and attaining an *II* of 1, under certain situations the underlying compiler can force an *II* of 1 by intentionally throttling or reducing the maximum FPGA circuit frequency for the entire offloaded kernel [108]. That is, to reduce the number of cycles stalled each iteration, the compiler can increase the amount of time per cycle. Although the ability to successfully launch iterations every cycle may benefit a specific loop, reducing the maximum circuit frequency can negatively affect performance in other regions of the offloaded kernel. Therefore, by default in the *Reduction Optimization*, we use the original equation without halving (Equation 2.1) to calculate the register depth. We currently

Listing 2.8 OpenCL generated from OpenARC's FPGA-specific reduction transformation

```
1  #define REGISTER_DEPTH (8 * N) // OpenARC calculated shift−register depth
2
3  float shift_reg[REGISTER_DEPTH + 1] = {0}; //Create and initialize shift registers.
4
5  #pragma unroll N
6  for (int i = 0; i < SIZE; ++i) {
7    shift_reg[REGISTER_DEPTH] = shift_reg[0] + input[i]; //Perform partial reduction.
8
9    for (int j = 0; j < REGISTER_DEPTH; ++j)
10   {   shift_reg[j] = shift_reg[j + 1];   } //Shift values in shift registers.
11 }
12
13 #pragma unroll
14 for (int i = 0; i < REGISTER_DEPTH; ++i)
15 {   sum += shift_reg[i];   } //Perform final reduction on shift registers.
```

hard-code operator latencies specific to the Stratix V, but these can easily be reconfigured for other devices. In Listing 2.8, we see the OpenCL code generated by applying the reduction optimization to the OpenACC scalar reduction code from Listing 2.7, targeting a Stratix V FPGA. We see the OpenARC-calculated shift register depth is appropriately set to $8 * N$ for floating point addition and an unroll factor of N (line 1). We next declare and initialize the shift registers, used for storing the accumulated partial sums (line 3). In the main loop, we now add each successive value to the oldest partial sum present in the shift registers (line 7), followed by a shift of the entire shift register array (lines 9–10). In this execution pattern, an assigned partial result is not accessed until it has been shifted through the entire register array, which relaxes the loop-carried dependency. After accumulating partial results over the entire array, we perform a final sequential reduction over the partial results in the shift registers (lines 13–15).

The OpenCL programming patterns generated by OpenARC (Listing 2.8) direct the underlying Intel OpenCL compiler to implement scalar reduction using single work-item execution and shift registers. With the FPGA-specific reduction optimization compiler transformation, we allow users to use existing OpenACC

directives to generate these non-intuitive code patterns without specialized knowledge of shift registers, initiation intervals, and operator latencies.

The reduction optimization was developed an implemented as part of this dissertation's research and originally published by Lambert et al. [9].

2.2.4 Directive Extensions. While many FPGA-specific optimizations in the OpenACC-to-FPGA framework can be either automatically applied or applied through an alternative implementation of existing OpenACC directives, for some FPGA-specific optimizations, automatic application by the compiler is difficult. Also, there might not be a straightforward mapping to existing directives that programmers could use to optionally apply the optimizations since these optimizations might not be relevant in GPU or multi-threaded CPU contexts.

In these cases, novel directive extensions are developed that can be recognized by the OpenARC compiler framework. The goal of these extensions is to allow programmers with limited FPGA knowledge to leverage FPGA-specific optimizations that could largely affect performance.

2.2.4.1 Kernel Vectorization Directive. In the Intel FPGA OpenCL programming, kernel vectorization allows multiple work items (device threads) in an OpenCL work group to execute in a single instruction multiple data (SIMD) fashion, which is implemented by replicating the kernel data paths while sharing control logic across each SIMD vector lane. Kernel vectorization is usually beneficial, but its additional resource requirement could contend with other optimizations. Although the OpenACC *vector* clause has similar effects, the vectorization behavior in the OpenACC execution model is not the same as that of the Intel OpenCL kernel vectorization. In OpenACC, vector lanes execute only in a SIMD manner if a kernel is in vector-partitioned mode and might not execute

in a lockstep manner. In contrast, OpenCL kernel vectorization exercises a strict lockstep vectorization.

The kernel vectorization optimization was designed and implemented by Lee et al. [144]. However, the optimization is used in this dissertation's research to evaluate the OpenACC-to-FPGA framework on different FPGA architectures, specifically in Lambert et al. [9] and Lambert et al. [11].

2.2.4.2 Compute Unit Replication Directive.

The reconfigurable nature of FPGAs allows multiple compute units to be generated for each kernel so that the hardware controller in FPGA can distribute work groups to available compute units in addition to running multiple work groups in a pipeline of a compute unit. Increasing the number of compute units can achieve higher throughput, but it also increases bandwidth pressure to the global memory and requires more hardware resources, whose optimal number should be carefully tuned.

The compute unit replication optimization was designed and implemented by Lee et al. [144]. However, the optimization is used in this dissertation's research to evaluate the OpenACC-to-FPGA framework on different FPGA architectures, specifically in Lambert et al. [9] and Lambert et al. [11].

2.2.4.3 Channels Directive.

In the current OpenACC execution model, there is no mechanism to allow fine-grained synchronization between actively running device kernels, and the device kernels can communicate with each other only through the device global memory. Therefore, both kernels require reading from and writing to the global memory to communicate, and the communication is serialized due to kernel communication. Moreover, the limited bandwidth and long latency of the global memory could become another performance-limiting factor. To address these issues, the underlying Intel OpenCL

provides a hardware mechanism called *channel*, which two concurrently running kernels can use to communicate with each other in a fine-grained manner without using the expensive global memory. If two or more OpenACC kernels execute in a sequential order and communicate with each other using temporary device buffers, then these kernels might be able use the channel mechanism when running on an FPGA. However, for the kernels to use this mechanism without breaking the original execution semantics, these kernels should communicate in specific patterns, which are not easy for the compiler to detect automatically. Furthermore, the channel mechanism can only be safely applied to applications where the dependencies between kernels are iteration-specific (i.e., iteration x of a kernel only depends on the results of iteration x of a previous kernel). To enable the channel mechanism in OpenACC, a set of new backward-compatible OpenACC data clauses were proposed with the existing OpenACC data clauses that will preserve functional portability across FPGAs and non-FPGA devices.

The channels directive was designed and implemented by Lee et al. [144]. However, the optimization is used in this dissertation's research to evaluate the OpenACC-to-FPGA framework on different FPGA architectures, specifically in Lambert et al. [9] and Lambert et al. [11].

2.2.4.4 Sliding Window Directive. Applications relying on stencil computations are common in scientific computing. Many algorithms operating on a grid or matrix apply a stencil pattern at each input location, relying on neighboring locations. These patterns and operations can result in redundant, expensive memory operations on devices such as GPUs and FPGAs.

However, in an FPGA single work-item context, redundant memory accesses across iterations can be mitigated by using a shift-register based sliding window

approach. In the sliding window approach, we maintain the required neighborhood of relevant data in shift registers, shifting a new value in and an old value out each time an iteration begins. This approach allows us to efficiently forward data across iterations, allowing for data reuse. This also significantly reduces the number of memory operations required each iteration because we are able to access the neighboring values stored in the sliding window without pipeline delays.

Basic Sliding Window Optimization

In this section we propose an OpenARC directive extension implementing the sliding window approach to address this performance issue. The *window* directive can be applied to loops within an OpenACC compute region, specifically where the loop reads from an input array, performs computations, and writes to an output array. However, only certain types of loops can benefit from application of the window directive, such as loops where each iteration contains several non-contiguous input array accesses, and loops where the same memory locations are redundantly accessed across different loop iterations. These programming patterns are common in stencil-based scientific codes.

The window directive imposes several restrictions for safe and efficient application. The optimization requires the neighborhood of cells accessed each iteration to be a fixed size. This fixed size is used to determine the size of the sliding window. The optimization also requires that the neighbor cells (array elements) accessed each iteration have constant offsets relative to the current iteration. For example, a loop that accesses a random assortment of neighbors each iteration would not be appropriate. Finally, in the current version of the sliding window optimization, the loop iteration variable must increase monotonically and have a step size of 1. These requirements ensure that the underlying OpenCL

Listing 2.9 OpenACC with window directive

```
1  #define ROWS ...
2  #define COLS ...
3
4  #pragma acc parallel loop serial
5  #pragma openarc transform window (input, output)
6  for (int index = 0; index < ROWS*COLS; ++index) {
7    float N = input[index − COLS];
8    float S = input[index + COLS];
9    float E = input[index + 1];
10   float W = input[index − 1];
11   output[index] = input[index] + N + S + E + W;
12 }
```

compiler can successfully and effectively infer and implement a sliding window approach using shift registers. OpenARC enforces these requirements by analyzing the loop control statement and requiring the index expressions of the input array to be affine, where the coefficient of the index variable is either *1* or *-1*. Violations of these requirements cause OpenARC to issue errors or warnings, depending on the offense.

In Listing 2.9, we show an example of a simple OpenACC stencil code with the window directive applied, where each iteration in a loop contains multiple non-contiguous input array accesses. Also, each element in the input array is accessed several times over multiple iterations. Because this example code meets the requirements mentioned above, it is safe to apply the window directive.

Using only the code provided in Listing 2.9, OpenARC can analyze the input array index expressions to calculate the following values needed to implement the sliding window transformation: neighborhood size (*NBD_SIZE*), window offset (*SW_OFFSET*), and reading offset (*READ_OFFSET*). The neighborhood size refers to the smallest number of contiguous array elements needed to encapsulate the neighbors required to compute one iteration. The window offset refers to the difference between the current value of the iteration variable and the minimum index value of neighbor cells for a given iteration. This offset is used when replacing

input array accesses with accesses to the sliding window. Finally, the reading offset refers to difference between the maximum index of the current neighbors and the current index. This offset determines the index used to read from the input array each iteration and to calculate the number of initialization iterations required. These offsets are calculated internally using the following equations, where *index* refers to the index of a given iteration, and *max_index* and *min_index* refer to the largest and smallest values used to access the input array for that same iteration.

$$NBD\_SIZE = max\_index - min\_index + 1 \tag{2.3}$$

$$SW\_OFFSET = index - min\_index \tag{2.4}$$

$$READ\_OFFSET = max\_index - index \tag{2.5}$$

In the proposed sliding window optimization, calculating the above three equations is key; for this, we exploit the built-in symbolic analysis tools in OpenARC. If the target loop body does not contain inner loops, the compiler symbolically calculates the differences between any two index expressions used for the input array accesses and derives the *min_index* and *max_index* expressions by symbolically comparing those differences. If the target loop body contains inner loops, the OpenARC compiler applies a symbolic range analysis, which computes integer variables' value ranges at each program point to find the symbolic ranges of index variables of the inner loops. The calculated symbolic ranges are used to calculate the symbolic differences between two index expressions for the input array accesses.

Once the above three values (neighborhood size, window offset, and reading offset) are calculated and determined to be constant, the remaining step is to transform the target loop into a specific programming pattern so that the

underlying OpenCL compiler is able to generate the hardware logic required for efficient sliding window execution.

In Listing 2.10, we show the resulting OpenCL code after the proposed sliding window optimization has been applied. We first see the results of OpenARC's calculations using the above equations (lines 5–7), followed by a declaration for the sliding window array (line 9). The initial value of the loop iteration variable is offset by the read offset (line 11). This allows for additional iterations to properly initialize the sliding window array, ensuring that the necessary neighborhood of values is present in the sliding window for the first non-initialization iteration. Within the loop, we first shift the sliding window each iteration (lines 12–13). Although this programming pattern is inefficient on non-FPGA platforms, it is required by the underlying OpenCL compiler to infer a shift register implementation of the intended sliding window array. We next read one value from the designated input array into the sliding window array, using the pre-calculated read offset (lines 16–17). Finally, for every non-initialization iteration, we perform the calculations from the original loop (lines 20–24). We see that each read from the original input array has been replaced with one read from the sliding window array, and in the sliding window array index expressions, the iteration variable has been replaced with the window offset.

By using OpenARC to generate these specific programming patterns, as outlined in the Intel OpenCL SDK Best Practices documentation, the back-end compiler is able to generate the hardware logic required for efficient sliding window execution. Although Listing 2.9 provides an ideal case for the window directive, the sliding window compiler transformation is robust enough to handle more complex indexing expressions, including expressions within nested loops containing multiple

Listing 2.10 Transformed OpenCL sliding window code

```
#define ROWS ...
#define COLS ...

// OpenARC calculated values
#define NBD_SIZE    (2*COLS + 1)  // Neighborhood size
#define SW_OFFSET   (COLS)        // Window offset
#define READ_OFFSET (COLS)        // Read offset

float sw[NBD_SIZE]; //Create a sliding window array.

for (int index = -(READ_OFFSET); index < ROWS*COLS; ++index) {
  for (int i = 0; i < NBD_SIZE - 1; ++i)
  {  sw[i] = sw[i + 1];  } //Shift values in the sliding window array.

  //Load an input array element into the sliding window array.
  if (index + READ_OFFSET < ROWS*COLS)
  {  sw[NBD_SIZE - 1] = input[index + READ_OFFSET];  }

  if (index >= 0) { //Main computation body which uses sliding window
    float N = sw[SW_OFFSET - COLS];
    float S = sw[SW_OFFSET + COLS];
    float E = sw[SW_OFFSET + 1];
    float W = sw[SW_OFFSET - 1];
    output[index] = sw[SW_OFFSET] + N + S + E + W;
  }
}
```

iteration variables. Also, algorithms without a separate output array that write computation results back to the original input array, like the Rodinia Benchmark NW [147], are handled by the compiler transformation using special-case code. The OpenARC window directive exemplifies the need for high-level programming constructs to enable widespread adoption of FPGA programming for HPC. This OpenACC directive extension enables programmers to use the performance-critical sliding window pattern on an FPGA without specific knowledge of shift registers, neighborhood sizes, and non-intuitive OpenCL programming patterns.

Sliding Window Optimization with Loop Unrolling Like the reduction optimization (Section 2.2.3.3), we can increase the performance of the shift-register–based sliding window optimization by applying loop unrolling. This unrolling can effectively increase the pipeline depth, allowing for a higher degree of pipeline parallelism and reducing the number of iterations required. This can

decrease overall runtime but at the cost of increased FPGA resource usage. For applications with a low base resource usage, loop unrolling can be used to utilize unused resources while improving performance.

To enable loop unrolling in conjunction with the sliding window approach, users can add an additional *#pragma unroll UNROLL_FACTOR* annotation to any loop annotated with a window directive. Here *UNROLL_FACTOR* refers to the degree of unrolling and the number of times the sliding window logic should be replicated. We have integrated the sliding window approach with loop unrolling by creating an extension to the sliding window compiler transformation. Although we could simply lower the unroll directive to the underlying OpenCL compiler, we can further optimize this approach by separating the shift register and memory operations from the primary computation operations. This separation allows us to reduce the number of sliding window shifts and perform coalesced memory reads and writes, while only replicating code used in the primary computation. This models the approach used in the Intel OpenCL SKD FD3D design example [108].

We see the resulting OpenCL code generated from applying an optional loop unroll pragma along with the window directive in Listing 2.11. In this transformation, the size of the sliding window is dictated by a new compile-time constant *SW_SIZE* (line 8). The increased size of the sliding window is needed to accommodate the additional operations from loop unrolling. Because we now process multiple values each iteration, the loop step size is increased to *UNROLL_FACTOR* (line 12). Instead of shifting the sliding window one position each iteration, we now shift *UNROLL_FACTOR* positions (lines 13–14), thus reducing the overall number of shifts required. We then perform a coalesced read of *UNROLL_FACTOR* values from the input array (lines 16–19). We declare a

Listing 2.11 Transformed OpenCL sliding window code with loop unrolling

```
1  #define ROWS ...
2  #define COLS ...
3
4  // OpenARC calculated values
5  #define NBD_SIZE    (2*COLS + 1)  // Neighborhood size
6  #define SW_OFFSET   (COLS)        // Window offset
7  #define READ_OFFSET (COLS)        // Read offset
8  #define SW_SIZE     (NBD_SIZE + UNROLL_FACTOR - 1)
9
10 float sw[SW_SIZE]; //Create a sliding window array.
11
12 for (int index = -(READ_OFFSET); index < ROWS*COLS; index += UNROLL_FACTOR) {
13   for (int i = 0; i < NBD_SIZE - 1; ++i)
14   {  sw[i] = sw[i + UNROLL_FACTOR];  } //Shift UNROLL_FACTOR positions.
15   //Load UNROLL_FACTOR values to the sliding window.
16   for (int ss = 0; ss < UNROLL_FACTOR; ++ss) {
17     if (index + READ_OFFSET + ss < ROWS*COLS)
18     {  sw[NBD_SIZE - 1 + ss] = input[index + READ_OFFSET + ss];  }
19   }
20
21   float value[UNROLL_FACTOR]; //Temporary array storing outputs.
22   //Main body replicated by UNROLL_FACTOR
23   #pragma unroll
24   for (int ss = 0; ss < UNROLL_FACTOR; ++ss) {
25     if (index + ss >= 0) {
26       float N = sw[SW_OFFSET+ss - COLS];
27       float S = sw[SW_OFFSET+ss + COLS];
28       float E = sw[SW_OFFSET+ss + 1];
29       float W = sw[SW_OFFSET+ss - 1];
30       output[index] = sw[SW_OFFSET+ss] + N + S + E + W;
31       value[ss] = sw[SW_OFFSET+ss] + N + S + E + W;
32     }
33   }
34   //Store temporary outputs to the output array.
35   for (int ss = 0; ss < UNROLL_FACTOR; ++ss) {
36     if (index + ss >= 0)
37     { output[index + ss] = value[ss]; }
38   }
39 }
```

statically sized array to temporarily store output values (line 21). The primary computation is then replicated by the enclosing fully unrolled loop (lines 23–33), with each access to the sliding window offset by the unrolled loop iteration index. Finally, we perform a coalesced write from the temporary array to the output array (lines 35–38).

The loop unrolling pragma can be applied to any loop optimized with the window directive as long as the unroll factor evenly divides the iteration space of the original main loop. For example, in Listing 2.11, the user-provided unroll factor

must divide $ROWS * COLS$. Violation of this restriction results in an OpenARC compiler error.

The window directive was designed and implemented as part of this dissertation's research, referenced in Lambert et al. [9].

2.3 Experimental Setup for FPGA Platforms

In this section we discuss the benchmarks, hardware, and software platforms used in this dissertation's research to evaluate the OpenACC-to-FPGA framework and developed optimizations discussed in Section 2.2, and in the study exploring the performance portability of OpenCL between Intel and Xilinx devices (Section 2.6).

2.3.1 Benchmarks. We use multiple benchmarks to test the viability, correctness, and performance of our FPGA-specific optimizations. Table 3 provides a summary of the benchmarks and their properties.

Table 3. OpenACC and OpenCL benchmarks evaluated using FPGAs

Application	Source	Description	Input Size	Data Type
Sobel	Intel	Image edge detection algorithm	1,920 × 1,080	integer
FD3D	Intel	3D finite difference computation	64 × 64 × 64	floating-point
HotSpot	Rodinia	Compact thermal modeling	1,024 × 1,024	floating-point
SRAD	Rodinia	Speckle reducing diffusion	4,096 × 4,096	floating-point
NW	Rodinia	Needleman–Wunsch algorithm	4,096 × 4,096	integer
Pathfinder	Rodinia	Dynamic programming search.	1,000,000 × 1,000	integer
CFD	Rodinia	Computational Fluid Dynamics	1,024 × 1,024	floating-point
Jacobi	OpenARC	Jacobi kernel	8192 × 8192	floating-point
Matmul	OpenARC	Matrix mulitplication kernel	2048 × 2048	floating-point
LULESH	LLNL	Lagrangian explicit hydrodynamics	45 × 45 × 45	floating-point

The Sobel and FD3D benchmarks are taken from the Intel *High-Performance Computing Platform Examples* [108], and the HotSpot, SRAD, and NW benchmarks originate from the Rodinia Benchmark Suite 3.1 [147]. NW can be classified as a dynamic programming algorithm, but the rest can be classified

as structured grid algorithms. We use the same input sizes and input parameters as the original Intel or Rodinia source codes, with the exception of FD3D. The original FD3D OpenCL code from Intel supports an input size of $504 \times 504 \times 504$ points by dividing the input into $64 \times 64 \times 504$ blocks. This blocking is necessary to meet FPGA resource usage requirements. However, because OpenARC does not currently support this type of custom blocking with OpenACC directives, we use an input size of $64 \times 64 \times 64$ single-precision floating-point values.

Base OpenACC versions of the Intel OpenCL SDK design examples were created directly from the OpenCL code by replacing the low-level OpenCL constructs with their high-level OpenACC counterparts and removing any FPGA-specific optimizations. A primary goal of this chapter in the dissertation is to reintroduce these optimizations using directives. Base OpenACC versions of the Rodinia benchmarks were sourced from the OpenARC repository. These benchmarks were adapted from the Rodinia 1.0 OpenMP benchmarks [39], although in this study we update them with any changes in Rodinia 3.1.

The OpenCL benchmarks evaluated in Section 2.4.6 are sourced directly from [108] and [160] without modification. The OpenCL benchmarks evaluated in Section 2.6 are modified from the original versions developed in [160] in order to execute in the Xilinx environment.

The OpenMP benchmarks evaluated in Section 2.4.7 come from the Rodinia repository [147].

For the sake of generality, while conducting research for this dissertation in Lambert et al. [10] and Lambert et al. [11] the OpenACC-to-FPGA framework is evaluated using two core algorithms, Jacobi and Matmul, and the real-world proxy application LULESH [171].

The holistic evaluation of the numerous optimizations in the OpenACC-to-FPGA framework required many executions with different combinations of threading models, optimizations, kernel vectorization and compute unit replication factors, unrolling factors, and more. This process was manually guided, but it was also restricted by the applicability of optimizations to each algorithm and device resource limitations. The optimization process for each benchmark was greatly simplified by the directive-based approach because code changes between versions were very minimal. However, the large optimization search space also exposed the dire need for a more automated optimization process.

We now briefly summarize each benchmark used in this dissertation's evaluation of the OpenACC-to-FPGA framework.

2.3.1.1 Sobel. The Sobel filter, or Sobel operator, is a popular image processing method used for edge detection in image data. The method uniformly applies gradient calculations across the input image, a structured grid. Each calculation depends on a 3x3 neighborhood of cells. We use a 1920x1080 8-bit image as input, and compute one iteration.

2.3.1.2 FD3D. The 3-Dimensional Finite Difference Computation is a numerical method used in solving differential equations. FD3D iterates over a structured 3D grid and computes a difference calculation using $RADIUS * 6$ neighboring cells. We use a $RADIUS$ of 3, resulting in a 19-point 3D stencil. The original OpenCL code from Intel supports an input size of 504x504x504 points by dividing the input into 64x64x504 blocks. This blocking is necessary to meet FPGA resource usage requirements. However, because OpenARC does not currently support this type of custom blocking with OpenACC directives, we use an input size of 64x64x64 single-precision floating-point values in all experiments.

2.3.1.3 *HotSpot.*

The HotSpot application is used to simulate the thermal properties of a processor, given information about the processor's architecture and power measurements. The application takes a 2D grid of initial values and power measurements and outputs simulated thermal values after a specified number of iterations. Each iteration, all values in the 2D grid are updated based on 4 neighboring cells: north, east, south, and west. We use a 1024x1024 sized 2D grid of single-precision floating-point values as input in our experiments, and perform 10,000 iterations.

2.3.1.4 *SRAD.*

Speckle Reducing Anisotropic Diffusion is an iterative image processing algorithm, used in applications such as medical and ultrasonic imaging. Like HotSpot, SRAD operates over a 2D structured grid. SRAD first performs a scalar reduction over the input array each iteration. Subsequently, SRAD performs a 5-point stencil computation similar to HotSpot. We use a 4096x4096 image as input, where each pixel is cast to a single-precision floating-point value, and compute 100 iterations.

2.3.1.5 *NW.*

Needleman–Wunch is a dynamic programming optimization algorithm used to perform DNA sequence alignment. The input to NW is a 2D matrix, and the computation begins at the top-left corner, finishing at the bottom right corner. Each value is updated using three neighboring cells: north, northwest, and west. We use a 4096x4096 integer array as input, and compute one iteration.

2.3.1.6 *Pathfinder.*

The goal of the Pathfinder application is to find the value of a minimum-weight path from the top row of a 2D grid to the bottom row. This computation uses a dynamic programming approach. Each element in the 2D grid is populated with a nonnegative integer weight. The path to a given

element, `elt`, is determined by the taking the minimum value from the northwest, north, or northeast element relative to `elt`. The program terminates when the last row of the 2D grid has been visited.

2.3.1.7 CFD. The Computational Fluid Dynamics (CFD) application is an unstructured grid benchmark that solves 3D Euler equations for compressible flow. This application comprises three kernels: compute step factor, compute flux, and time steps. The kernels are highly compute-intensive with many single-precision floating point operations, including addition, multiplication, division, and square root. The most expensive computation is in the compute flux kernel, which calculates the artificial viscosity and accumulates flux contributions across each face.

2.3.1.8 Jacobi. The Jacobi method is an iterative solver commonly used for solving systems of linear equations in many scientific domains.

2.3.1.9 Matmul. Matrix multiplication, the cornerstone of linear algebra, is a fundamental core kernel used in applications in nearly every domain.

2.3.1.10 LULESH. The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics is widely studied proxy application and co-design effort in high-performance and exascale computing. To evaluate the OpenACC-to-FPGA framework, we target the LULESH 2.0 OpenACC implementation. Because the original application is written using C++, we target a C-based OpenACC port available in the OpenARC repository. However, because LULESH 2.0 contains few C++-specific constructs, the C and C++ versions are comparable.

2.3.2 FPGA Hardware Platforms. We use three different generations of Intel FPGAs in this dissertation: a Stratix V, an Arria 10, and a Stratix 10. The Stratix V was originally designed and released by Altera, while the

other two FPGAs were formally released by Intel (with an apparant disdain for Roman numerals), after Intel's acquisition of Altera. Details about the hardware resources available in each FPGA is presented in Table 4. We can see that each new FPGA release comes with increased availability of hardware resources. The benefits of this increased size are shown in Section 2.5 of this chapter, as we are able to apply more aggressive optimizations and replication.

On the Intel FPGAs, power usage estimations using the Quartus Power Analyzer [108] on fully compiled and routed applications. For a fair comparison with GPU and CPU power calculations, we add 2.34 W to the power estimations to account for the FPGA memory modules, as in [160]. We calculate energy (J) as runtime (s) $\times$ power (watts). Resource usage percentages are provided by the backend Intel OpenCL compiler.

For the Xilinx-based evaluations in Section 2.6, we used a Xilinx Alveo U250 Data Center accelerator card, which includes an XCU250 FPGA of the Xilinx UltraScale+ architecture, a Gen3 x16 PCIe interface, and 64 GB of DDR4, off-chip memory.

Although multi-core CPUs were used as host processors, all of the host code in the evaluations was executed using a single core.

Table 4. Intel and Xilinx Hardware Resource Features

FPGA name	Board model	ALMs	DSP blocks	RAM blocks	Host CPU
Stratix V	nallatech_385	172k	1,590	2,014	Intel Xeon E5520
Arria 10	p510t_sch_ax115	427K	1,518	2,713	Intel Xeon E5-2683 v4
Stratix 10	p520_max_sg280h	933K	5,760	11,721	Intel Xeon E5-2660 v4
Alveo U250	XCU250	1,341K	11,508	12,240*	Intel Xeon E5-2683 v4

*The Alveo board contains 2,000 "36 Kb Block RAMs" and 1,280 "288 kb Ultra Block RAMs", which is roughly analogous to 12,240 RAM blocks (when comparing to Intel devices).

2.3.3 FPGA Software Platforms. On all platforms, input OpenACC code is compiled using OpenARC V0.11 as the front end, although the specific git commit used changed frequently, especially as we continually updated OpenARC's OpenACC-to-FPGA support.

For evaluations on the Stratix V, we use the Intel FPGA SDK for OpenCL Offline Compiler V16.1.0 as the primary compiler and the back end runtime for OpenCL code.

For evaluations on the Arria 10, The back-end OpenCL code is compiled using the Intel FPGA SDK for OpenCL v17.1.0 (*aocl*). The software stack is built on CentOS Linux 7 (Core).

For the Stratix 10 devices, the back-end OpenCL code is compiled using the Intel FPGA SDK for OpenCL v19.4.0 (*aocl*). The software stack is built on Red Hat Enterprise Linux 7.

For all three devices, runtime measurements are recorded using C API calls, specifically *clock_gettime()*. Several Python scripts were also built to automate batch build, compilation, and execution processes for the FPGA. These scripts also extract resource usage and other compilation information reported by the *aocl* compiler and notify users via text message/email upon compilation completion.

Runtimes reported are the average of five executions (Stratix V) or three executions (Arria 10, Stratix 10). For the Stratix V executions, the runtime variance was below 1.5% of the mean runtime for all applications, with most variances falling below 0.1%. Similar variances were observed on the Arria 10 and Stratix 10 devices.

For our Xilinx-based experiments in Section 2.6, we used the 2020.1 version of the Vitis Core Development Kit, and the associated compiler *v++*.

Both Intel and Xilinx hardware compilers generate interactive reports that can be used to provide insight into kernel performance and opportunities for optimization. Information provided in these reports includes FPGA resource utilization and the analysis of loops within a kernel. Intel generates this report by constructing an `.html` file that can be opened in a browser, and Xilinx generates summaries that can be navigated by using the `vitis_analyzer` graphical user interface (GUI) application.

2.3.4 GPU and CPU Comparison Platforms. For the GPU comparisons in Section 2.4.7, we use an NVIDIA Tesla K40c GPU. The OpenACC code relies on the NVIDIA CUDA compiler V8.0 as the back end (the OpenACC input code is translated into CUDA by the OpenARC compiler). We calculate energy consumption using NVIDIA NVML to sample power usage every 10 ms.

For the CPU comparisons in Section 2.4.7, we use a 16-core Intel(R) Xeon(R) E5-2683 v4 CPU with 2-way hardware multi-threading. We compile the OpenMP benchmarks using GCC 4.8.5 with the *-O2* flag, and execute them using 32 OpenMP threads. We collect CPU energy usage information using the Intel Running Average Power Limit (RAPL) interface.

2.4 Intel Stratix V Evaluations

In Section 2.2, we discussed various FPGA-specific optimizations, many of which were developed as part of this dissertation's research in Lambert et. al [9]. In the following three sections, we discuss the rigorous evaluations performed across three different FPGA platforms in order to asses the performance of the OpenACC-to-FPGA framework. These evaluations span three separate publications (all included as part of this dissertation) and roughly three categories: (1) an evaluation of each developed optimization in isolation [9], (2) a holistic

evaluation of combinations of developed optimizations on new platforms [10, 11], (3) evaluations of FPGA-specific considerations and behaviors in the context of the OpenACC-to-FPGA framework [9, 11]. In this section, we discuss evaluations performed using the Stratix V platform that were originally published in Lambert et al. [9].

2.4.1 Single Work-Item Evaluation. By using directives to dictate a single work-item execution context, we can transform a traditional multi-threaded approach into an FPGA-specific pipeline-parallel single-work item approach. We evaluate the effectiveness of the single-work item approach by comparing it to the multi-threaded approach. Both approaches were programmed using OpenACC and executed on the Stratix V FPGA. Figure 6 shows the FPGA performance of the two approaches across each benchmark. In this figure, the multi-threaded approach (*NDRange*) is used as a baseline, and the single work-item approach is compared in terms of speedup. We can see that for two applications (Sobel and HotSpot), applying the single work-item alone improves runtime performance. For the other applications (FD3D, SRAD, and NW) this optimization can actually degrade performance. However, in both cases the single work-item optimization enables us to apply the more advanced collapse, reduction, and sliding window optimizations, ultimately leading to higher performance than the multi-threaded approach for all benchmarks on the Stratix V platform.

2.4.2 Collapse Evaluation. The FD3D, HotSpot, SRAD, and NW benchmarks all contain nested loops inside their main computation kernels. As a result of restrictions from the underlying OpenCL compiler (the Intel OpenCL SDK for FPGAs), to apply the sliding window and unrolling optimizations, we first need to apply loop collapsing to remove the nested loops. Traditional loop

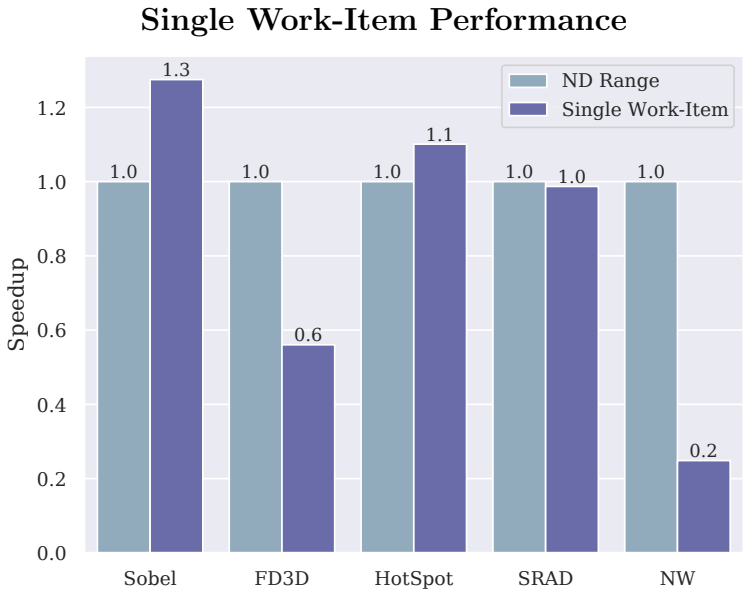


Figure 6. OpenACC-to-FPGA multi-threaded and pipeline-parallel approaches (Stratix V).

collapsing techniques can be used to remove the nested loops; however, because the sliding window and other optimizations require a single work-item context, we can apply the single work-item FPGA-specific loop collapse optimization, replacing the division and modulus operations with more efficient addition operations along with row and column counters. Table 5 demonstrates the modest performance and resource usage improvements realized when applying the FPGA-specific collapse optimization in single work-item executions.

Table 5. FPGA-specific collapse clause performance comparison (Stratix V)

Application	Collapse Type	Runtime	Resource Usage (%)
FD3D	Standard	190.935 (ms)	39
FD3D	FPGA-specific	180.149 (ms)	36
HotSpot	Standard	47.882 (s)	32
HotSpot	FPGA-specific	47.371 (s)	30

2.4.3 Reduction Evaluation. We use the SRAD benchmark to experimentally verify the observations in Section 2.2.3.3. First, we evaluate the

relationships between different programmable parameters in the FPGA-specific single-work item scalar reduction. We isolate the reduction in SRAD, removing other computations in the benchmark. This results in a single-precision floating-point sum reduction over an input array of size 4096×4096. Removing the non-reduction code allows us to better observe the relationships between shift register depth, initiation interval, resource usage, and runtime. In the initial experiment, we use a constant unroll factor of 8 and manually vary the shift register depth. In Figure 7, we see that increasing the shift register depth reduces the initiation interval, at the cost of increased resource usage. This reinforces observations about relationship between shift register depth and initiation interval introduced by Equation 2.1. As we increase the shift register depth, for certain depth values we observe an unexpected decrease in circuit frequency and a corresponding unexpected decrease in the initiation interval. These specific values indicate instances where the compiler has intentionally sacrificed or throttled the circuit frequency to attain a lower initiation interval. For example, in Figure 7, at register depths 16 and 32 we notice a decrease in II and a corresponding significant drop in circuit frequency. As the shift register depth continues to increase, the circuit frequency re-stabilizes, steadily increasing while the initiation interval remains unchanged.

In the second experiment, we evaluate the performance improvements by applying the single work-item FPGA-specific reduction optimization, compared to traditional approaches to scalar reduction. For this experiment we use the entire SRAD benchmark, as changes in the reduction implementation can also affect execution in other code regions. We compare three different approaches to

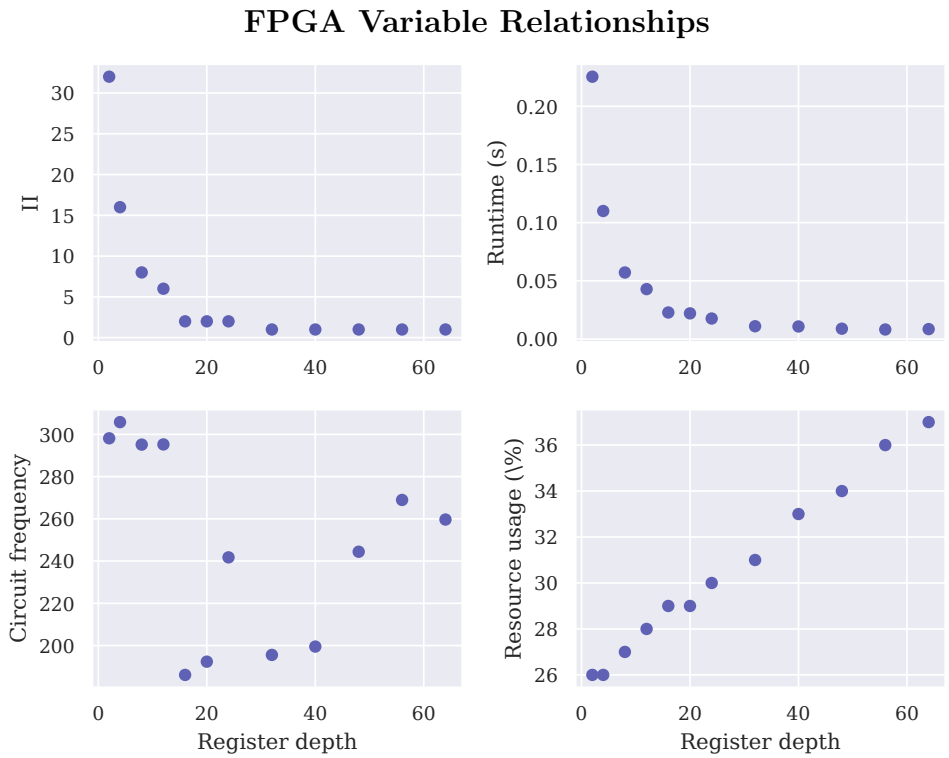


Figure 7. Initialization interval (II), circuit frequency, runtime, resource usage, and shift-register depth relationships. SRAD reduction kernel (Stratix V).

scalar reduction: (1) a tree-based reduction, (2) a basic single work-item reduction, (3) and the FPGA-specific shift register reduction.

In Table 6, we see the basic single-threaded approach performs poorly compared to the hardware-agnostic multi-threaded tree-based reduction. Consequently, scalar reduction represents a code pattern where the single work-item optimization alone does not lead to improvements in performance. However, by combining the single work-item approach with the FPGA-specific shift-register based optimization, we can significantly outperform the other approaches to scalar reduction, but this performance comes at the cost of increased resource usage.

Table 6. SRAD FPGA reduction performance comparison (Stratix V).

Reduction Type	**Runtime (s)**	**Resource Usage (%)**
Multi-threaded Tree-based	31.053	45
Single Work-item	78.307	38
Single Work-item Shift Register	23.239	50

2.4.4 Sliding Window Evaluation. In this section, we first evaluate the baseline sliding window optimization, and then evaluate the sliding window optimization with replication incorporated via customized loop unrolling.

2.4.4.1 Basic Sliding Window. The sliding window optimization (Section 2.2.4.4) can safely be applied to non-nested loops in a single work-item execution context. Therefore, by first applying the single work-item optimization (Section 2.2.3.1) and, when appropriate, the collapse optimization (Section 2.2.3.2), we can then apply the sliding window optimization to all five benchmarks.

We evaluate the effectiveness of the sliding window optimization for each benchmark by comparing a massively parallel multi-threaded approach, a basic pipeline-parallel single work-item approach, and a pipeline-parallel single work-item approach using a sliding window. We see significant performance improvements

across all benchmarks when applying the sliding window optimization. The results of the sliding window evaluation are presented in Figure 8. The runtime for the OpenACC implementation with only the single work-item optimization applied is used as a baseline, and the performance of the same OpenACC implementation with both the single work-item and sliding window optimizations applied is compared in terms of speedup.

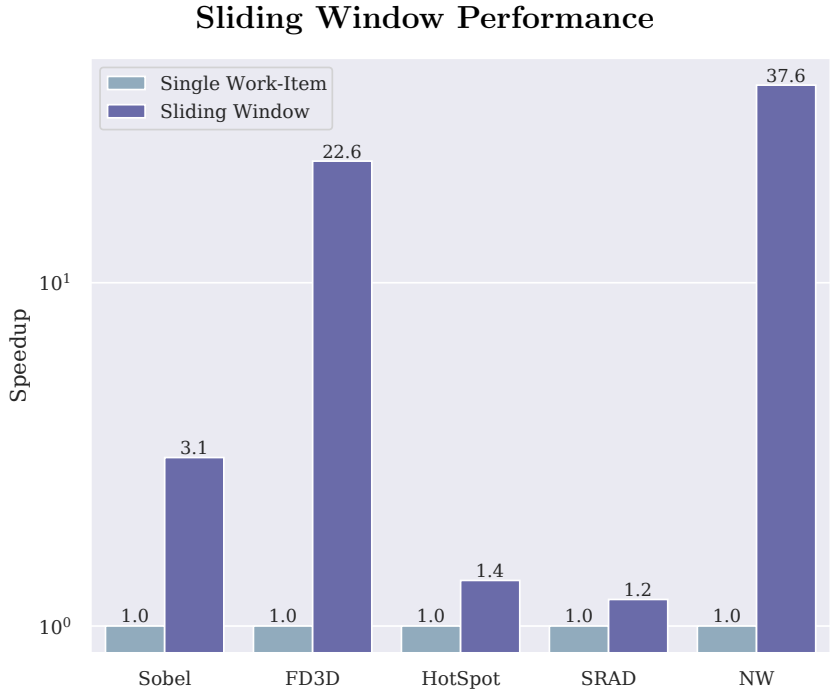


Figure 8. Comparison of a single work-item and a single work-item with shift-register sliding window approach (Stratix V)

We see that the performance of the NW benchmark improves exceptionally after applying the sliding window optimization. Unlike the other applications, NW reads from and writes to the same array, instead of writing to a separate output array. When executing in a single work-item context, this creates a memory dependency on the load and store operations to and from this array. This memory dependency causes successive iterations to be launched only once every 328 cycles,

severely degrading performance, as we see in NW's basic single work-item approach. Applying the sliding window optimization to the single work-item implementation of NW shifts the memory dependency to a local data dependency. The sliding window allows successive iterations to be launched every cycle, significantly improving performance. Additionally, the expensive load operations for neighboring array elements are replaced with sliding window, or shift register, accesses.

We can also conjecture that the degree of speedup when applying the sliding window optimization is proportional to the size of the stencil computation. For example, the Sobel (9-point stencil) and FD3D (19-point stencil) realize a greater speedup than HotSpot and SRAD (4-point stencils).

2.4.4.2 Sliding Window with Loop Unrolling. We evaluate the effectiveness of using loop unrolling in conjunction with the sliding window optimization (Section 2.2.4.4) in each benchmark by comparing the performance of the single work-item sliding window approach with various degrees of loop unrolling applied.

The results of this evaluation are presented Figure 9. For each benchmark, the runtime of the application with the sliding window optimization without unrolling (Section 2.2.4.4) is used as a baseline. These times are annotated with a 1 above the bar. We compare each baseline to the same benchmark with different unrolling factors applied, visible over each bar. In general, we see that we can utilize previously unused FPGA resources to increase runtime performance. We can also see that performance improvements diminish with high unroll factors, as resources become scarce.

We see in Figure 9 that the Sobel benchmark is an ideal candidate for loop unrolling. Because of the benchmark's low base resource usage, we can apply a

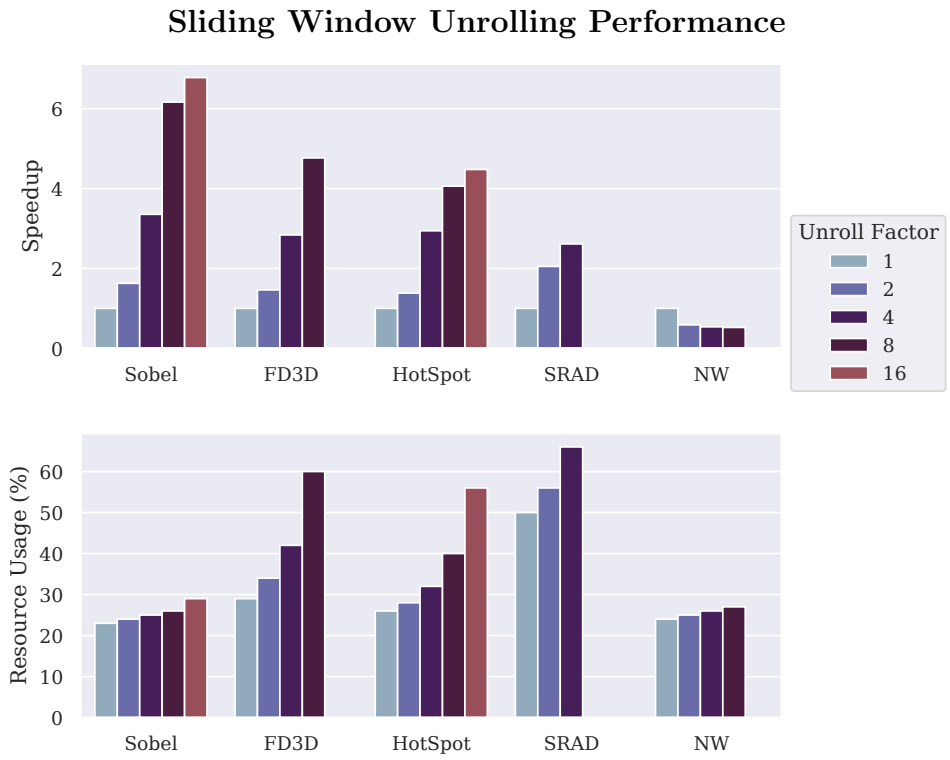


Figure 9. Sliding window optimization with different unroll factors applied (Stratix V)

high unrolling factor without exhausting FPGA resources. In contrast, applying loop unrolling to the NW benchmark actually degrades performance. As previously mentioned, the NW benchmark is unique in that the same array is used for both input and output values. This creates a dependency between loop iterations. We see performance benefits by using the sliding window optimization because of the replacement of expensive memory operations with shift register operations. However, we cannot increase the level of pipeline parallelism by unrolling the inner loop because the operations are serialized due to the loop dependency.

2.4.5 Branch-Variant Code Motion Evaluation. We use the HotSpot benchmark to measure the performance and resource usage effects of the branch-variant code motion optimization. In this benchmark, a nine-way conditional is used to determine if the current index is an edge, corner, or neither. Several common operations occur within each branch of this conditional, including several multiplication and addition operations, and an expensive load operation. These common operations result in a relatively high base resource usage for the application. By applying branch-variant code motion, we can factor or hoist the common computation code from each branch, significantly reducing the number of multiplications, additions, and loads required to be mapped to the hardware logic. This results in a lower base resource usage.

Table 7 shows the results of executing HotSpot with the sliding window applied and different loop unroll factors with and without branch-variant code motion. We see that applying the resource usage reduction optimization does not directly or significantly affect runtime. However, as we unroll the inner loop, the version with the common operations in each branch of the conditional quickly encounters performance degradation due to resource exhaustion, while

the optimized version with the hoisted code continues to see performance improvements.

Table 7. HotSpot code motion performance evaluation (Stratix V)

	Base		Hoisted	
Unroll Factor	**Resource Usage (%)**	**Runtime (s)**	**Resource Usage (%)**	**Runtime (s)**
1	28	36.842	26	35.622
2	31	24.656	28	25.796
4	39	16.625	32	12.106
8	54	29.442	40	8.770
16	84	50.702	56	7.953

2.4.6 OpenACC and OpenCL Performance Comparison.

To explore the viability of using a high-level language like OpenACC for FPGA programming, we compare the performance of all five benchmarks to the performance of those same benchmarks implemented directly in OpenCL. The OpenCL versions manually implement several of the same optimizations generated by the OpenARC compiler, but they also contain other FPGA-specific optimizations not currently supported by OpenARC, such as blocking and halo regions with sliding window arrays.

We can see the comparison between the best-performing OpenACC implementation and the manual OpenCL implementations in Figure 10. In this figure, the OpenACC runtimes are used as baselines, and the OpenCL runtimes are compared in terms of speedup. We can see that the OpenACC applications FD3D, HotSpot, and SRAD perform comparably to the manual OpenCL versions, with performances varying by less than a factor of 2.

The OpenACC version of the NW benchmark is roughly 10 times slower than the OpenCL version. This is because Rodinia's OpenCL version of NW,

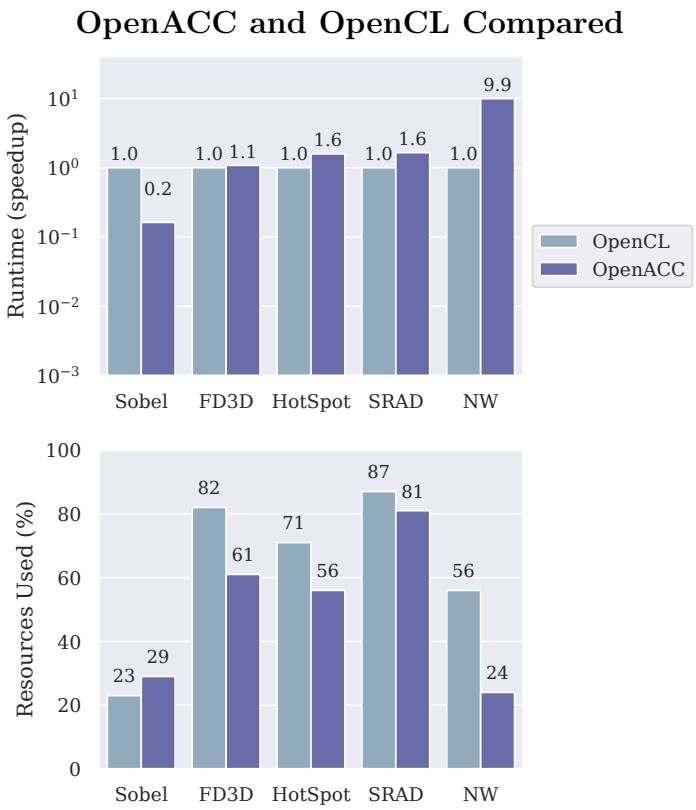


Figure 10. OpenACC and OpenCL with FPGA-specific optimizations (Stratix V).

on which the FPGA-specific OpenCL version in is based, employs a significantly different programming pattern than the straightforward serial version of NW used to develop our OpenACC version. These patterns are not currently reproducible using our OpenACC directives for FPGA-specific optimizations, so NW represents a class of applications where our current FPGA-specific optimizations fail to realize the performance of manually tuned OpenCL. In contrast, the OpenACC version of the Sobel Filter actually outperforms the OpenCL version from the Intel SDK design examples. Although the manual code also uses a sliding window approach, it does not perform loop unrolling, resulting in the performance differences observed.

2.4.7 Performance and Power Comparisons of FPGAs, GPUs, and CPUs.

To evaluate the viability of OpenACC FPGA programming, we compare OpenMP programs executed on a CPU and OpenACC programs executed on a GPU (Section 2.3.4) against OpenACC programs executed on an FPGA (Section 2.3.2). The results of this evaluation are shown in Figure 11. In this figure we compare runtimes, measured in terms of speedup from the CPU baseline, and energy consumption, measured in Joules and normalized to a CPU baseline of 1.

The NW benchmark performs relatively poorly both in terms of runtime and power usage on the FPGA. This stems from the same algorithmic differences mentioned in Section 2.4.6. However, for every other benchmark, the FPGA outperforms at least one of the other newer devices in either runtime or power usage when programmed using high-level frameworks.

2.5 Intel Arria 10 and Stratix 10 Evaluations

In Section 2.4, we evaluated each *optimization* individually. In this section, we evaluate each *application* individually on two new FPGA platforms and one new benchmark, holistically applying optimizations from both Lambert et al. [9]

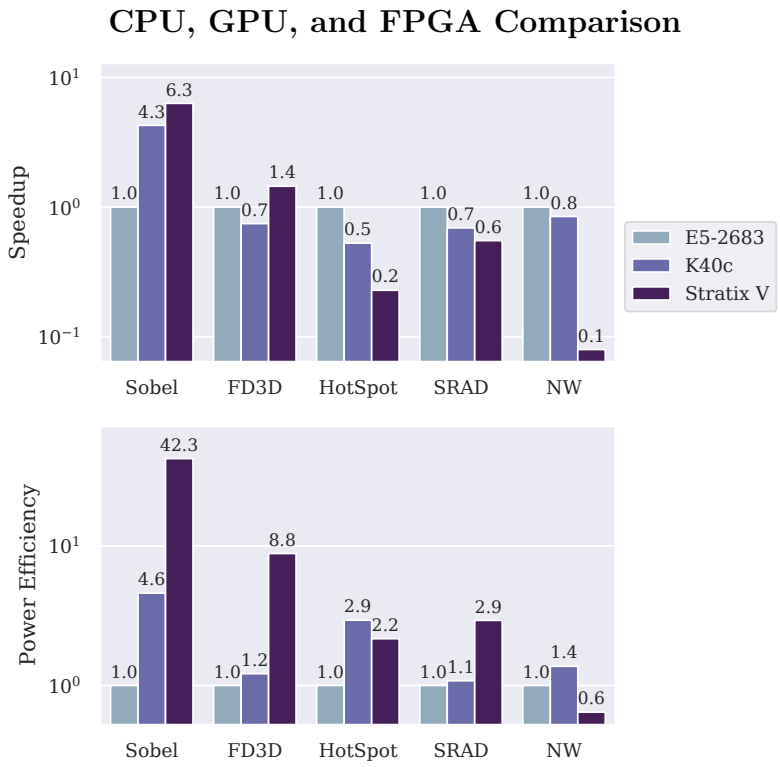


Figure 11. Comparison of OpenMP CPU (Xeon x32) executions, OpenACC GPU (K40c) executions, and OpenACC FPGA (Stratix V).

and Lee et al. [144]. We then discuss evaluations exploring FPGA resource usages, compilation times, and performance portability. The evaluations in this section were originally performed in Lambert et al. [10] and Lambert et al. [11].

To evaluate each individual benchmark, first only the multi-threaded and single work-item implementations of each kernel were evaluated. Then, optimization directives and clauses were incrementally applied to each version where possible. Finally, different replication factors were tested when possible by varying the number of compute units and SIMD parallelism in the multi-threaded kernels and by varying the reduction and sliding window unrolling factors in the single work-item kernels. In Subsections 2.5.1-2.5.4, we explore the Sobel, SRAD, Jacobi, and Matmul benchmarks, while LULESH is explored separately in Subsection 2.5.8. In Figures 12–16, the version name is a concatenation of the applied optimizations:

- `nd`: multi-threaded kernel
- `numcX`: number of compute units (`X`: replication factor)
- `simdX`: vectorization (`X`: replication factor)
- `elim`: kernel boundary elimination optimization
- `coll` collapse optimization
- `swi`: single work-item kernel
- `redX`: reduction optimization (`X`: unroll factor)
- `swX`: sliding window optimization (`X`: unroll factor)
- `hoist`: code motion optimization
- `flat`: 2D arrays manually flattened to 1D array.

2.5.1 Sobel Holistic Evaluation. The Sobel benchmark iterates over a 1D image array and performs a stencil operation. Sobel is unique among

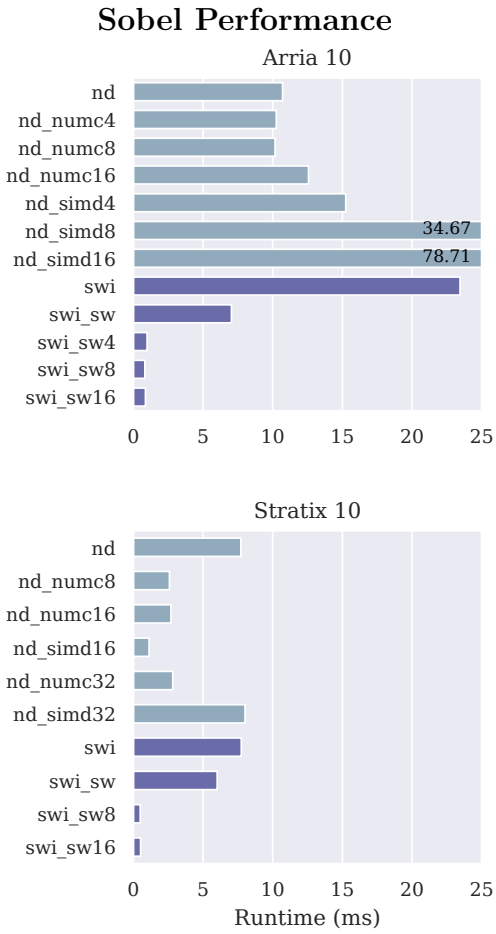


Figure 12. Runtime performance (in seconds) of Sobel with different FPGA-specific optimizations applied (Arria 10 and Stratix 10). Blue bars indicate the multi-threaded approach, and purple bars indicate the single work-item approach (smaller is better).

the evaluated benchmarks because it relies on integer operations, which can be implemented very efficiently in FPGA logic. In Figure 12, on the Arria 10 the baseline `nd` implementation outperforms the baseline `swi` implementation. However, if the replication factors are scaled using `simd` and `numc` in the multi-threaded version, then the performance degrades significantly. This is most likely due to the high cost of the memory operations for the 9-point stencil, relative to the cheap cost of integer and bit arithmetic. On the Stratix 10, some multi-threaded replication improves performance, but again excessive replication degrades performance (`nd_simd32`).

Conversely, applying additional optimizations and replication to the single work item significantly improves performance on both devices. Applying the sliding window pattern effectively reduces the ratio of memory operations to computation, and applying loop unrolling significantly increases the parallelism.

On the Arria 10, Sobel represents a unique but critical example in which the multi-threaded kernel initially outperforms the single work-item kernel until applying sufficient optimization results in a performance reversal. This example exposes a pitfall of manual intuition-guided optimization and further motivates an automated analytical optimization solution.

One major difference between the Arria 10 and Stratix 10 implementations is the effectiveness of `simd` replication. This could be a result of the newer compiler version (v17.1 compared to v19.4) used on the Stratix 10 and its abilities to apply `simd` replication successfully.

2.5.1.1 HotSpot.

Like Sobel, HotSpot also consists of a stencil operation, although HotSpot relies on a 5-point stencil instead of a 9-point stencil and uses floating-point values instead of integer values.

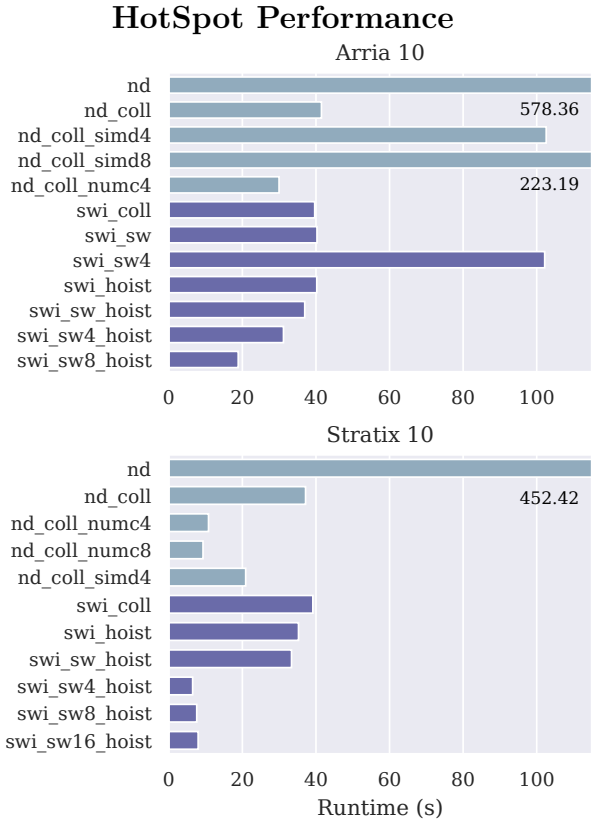


Figure 13. Runtime performance (in seconds) of HotSpot with different FPGA-specific optimizations applied (Arria 10 and Stratix 10). Blue bars indicate the multi-threaded approach, and purple bars indicate the single work-item approach (smaller is better).

As shown in Fig. 13, HotSpot experiences a significant performance imporvment if the collapse optimization is applied on both the Arria 10 and Stratix 10. This result contradicts previous results on the Stratix V [9] in which the collapse optimization achieved only modest performance. This could be attributed to the difference in FPGAs, but it is more likely an artifact of the different compiler versions and how v17.1 and v19.4 of the SDK interpret the nested loops.

In the multi-threaded kernels on the Arria 10, HotSpot does not respond well to kernel vectorization, likely due to the high degree of branching and numerous conditionals. However, HotSpot does experience modest performance improvement from compute unit replication (41.49 s vs. 29.97 s).

However on the Stratix 10, like the Sobel application, we see much better performance improvements from multi-threaded replication, especially *simd*.

If only the collapse optimization is applied, the single work-item kernels perform very similarly to the multi-threaded kernels. Only with significantly more optimization (collapse, code motion, sliding window, and unrolling) does the single work-item approach achieve a lower runtime (18.87 s vs. 29.97 s on Arria 10). As with the Stratix V device evaluations in Lambert et al. [9], no performance improvements were seen from the window optimization with replication unless the common operation hoisting optimization (code motion) was also applied.

2.5.2 SRAD Holistic Evaluation. The SRAD algorithm consists of three separate kernels: one large reduction, and two 5-point stencil loops. In Figure 14, the `device` and `host` keywords for the multi-threaded kernels indicate whether the reduction is performed on the FPGA device or on the host (and are updated via a *#pragma acc update* directive). For the single work-item kernels, the reduction is always performed on the FPGA device. The distinct sliding window

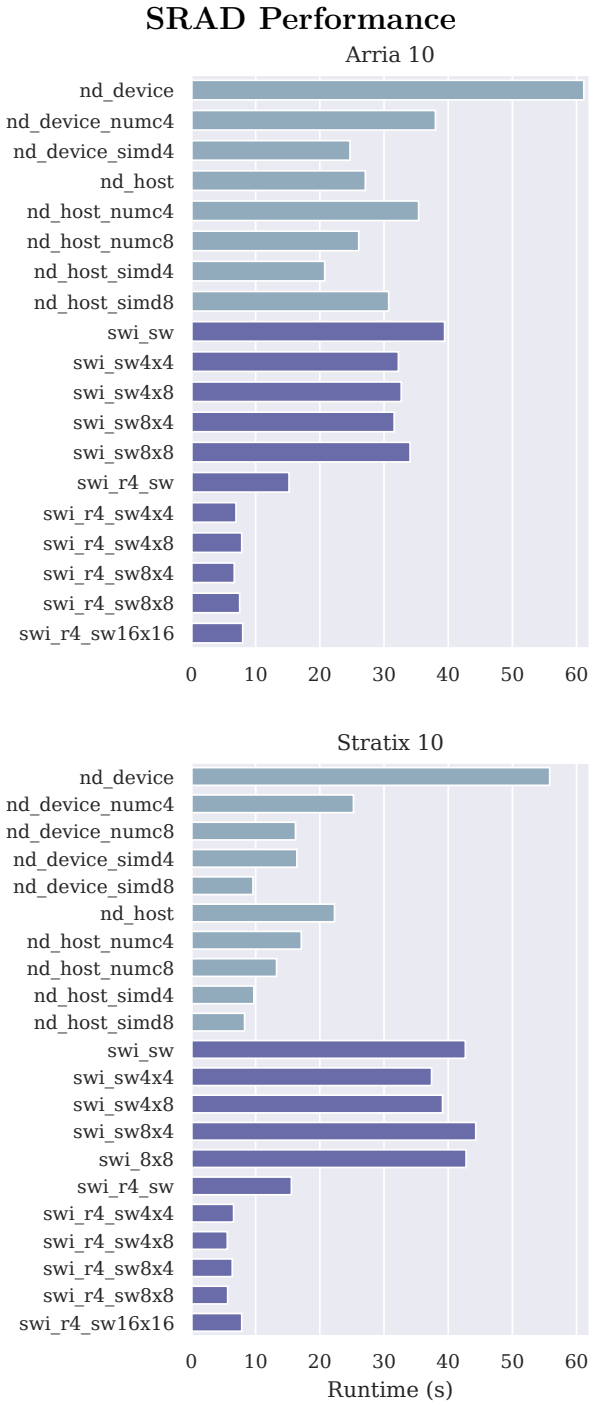


Figure 14. Runtime performance (in seconds) of SRAD with different FPGA-specific optimizations applied (Arria 10 and Stratix 10). Blue bars indicate the multi-threaded approach, and purple bars indicate the single work-item approach (smaller is better).

unrolling factors for the first and second stencil loops are separated by the `x`. For example, in the `nd_host_simd8` version, the reduction is performed on the host CPU and the results that are copied to the device, and the two stencil loops are vectorized with a `simd` factor of 8. In the `swi_r4_sw_8x4`, the first kernel is optimized using the reduction optimization with an unroll factor of 4, and the second and third kernels are optimized using the sliding window approach with unroll factors of 8 and 4, respectively.

Unlike HotSpot and Sobel, SRAD multi-threaded kernels respond well to compute unit replication and kernel vectorization. Also, the performance patterns are very similar for both the Arria 10 and Stratix 10.

The vectorized multi-threaded kernels outperform many single-threaded kernels, even after applying the sliding window and unrolling optimizations to the second two loops. However, after applying a combination of sliding window and reduction optimizations, the single-threaded kernel performance significantly surpassed the multi-threaded counterpart. Again, we see a trend where sufficiently optimized single-threaded kernels outperform multi-threaded kernels.

2.5.3 MatMul Holistic Evaluation. Compared to the other evaluated benchmarks, MatMul is the simplest and has no conditionals, limited arithmetic, and few memory operations.

As shown in Figure 15, there is little opportunity for optimization in the single-threaded approach because only the collapse optimization can be applied with no significant difference in performance.

As a result of the simplicity, MatMul responds very well to kernel vectorization and compute unit replication. MatMul is the only example in which the multi-threaded kernels significantly outperform the single work-item

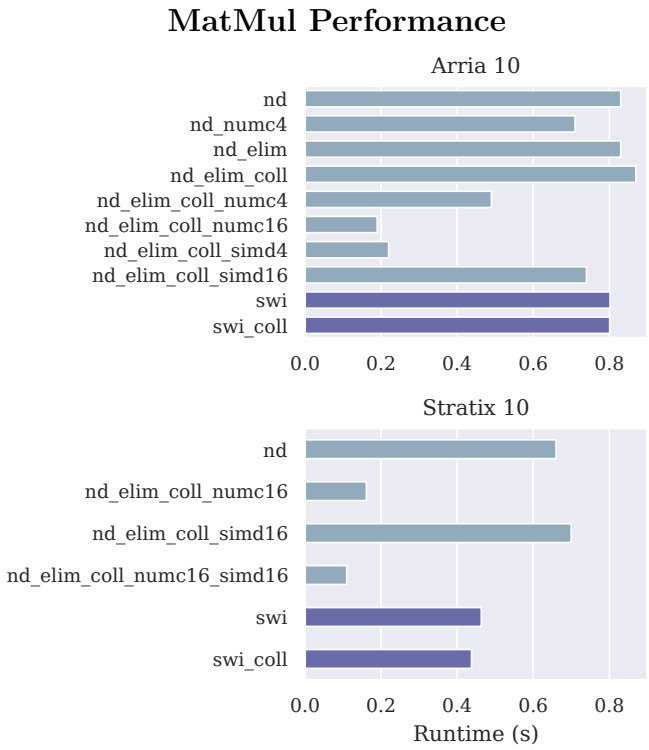


Figure 15. Runtime performance (in seconds) of MatMul with different FPGA-specific optimizations applied (Arria 10 and Stratix 10). Blue bars indicate the multi-threaded approach, and purple bars indicate the single work-item approach (smaller is better) .

counterparts. We evaluated fewer versions on the Stratix 10, focusing only on the versions with more aggressive replication.

Matmul could be an example in which the OpenACC-to-FPGA contains insufficient single work-item optimizations. Including more advanced optimizations that are not yet supported by the framework (e.g., blocking, tiling, local memory buffering) could improve performance.

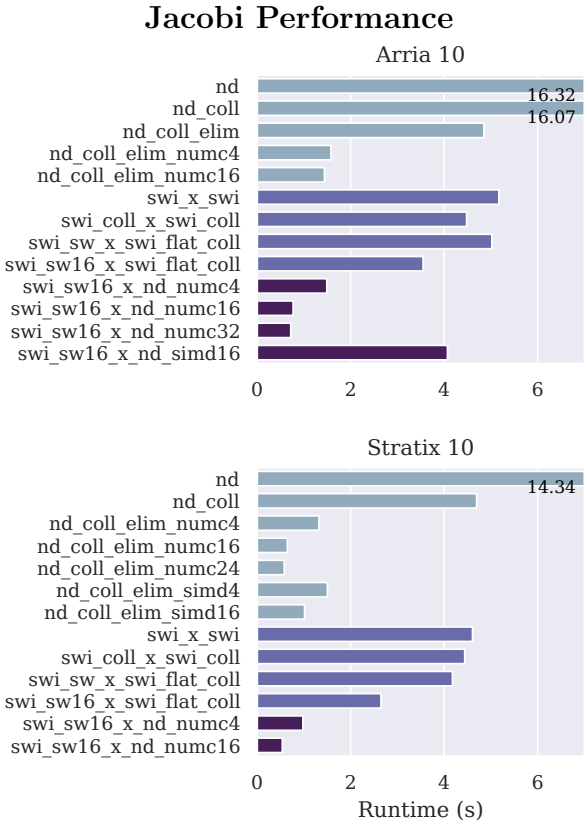


Figure 16. Runtime performance (in seconds) of Jacobi with different FPGA-specific optimizations applied (Arria 10 and Stratix 10). Blue bars indicate the multi-threaded approach, purple bars indicate the single work-item approach, and green bars indicate a hybrid multi-threaded+single work-item approach (smaller is better).

2.5.4 Jacobi Holistic Evaluation.

Jacobi consists of two kernels: one 5-point stencil operation with very little arithmetic (floating point), and an array-copy kernel.

In Figure 16, there are three different groupings of performance measurements. The top grouping represents versions in which a multi-threaded approach was used for both kernels by applying the same optimizations to each kernel.

The second grouping conversely represents versions in which the single-threaded or single work-item approach was used for both kernels. The `x` separates the optimizations applied to the first and second kernels. This distinction was made because the window optimization applies only to the first stencil-based loop. The `flat` keyword represents versions in which the original 2D array is replaced with a 1D array, and the indices are adjusted accordingly. Currently, the window optimization supports only 1D arrays, so this manual code modification was needed even though manually modifying code is generally avoided and only directives are applied.

The third grouping represents a novel hybrid approach in which a single work-item approach is used for the first kernel and a multi-threaded approach is used for the second kernel. Although the `flat` keyword is omitted, these versions also revert to a 1D array to apply the window optimization.

As shown in Figure 16, the best-performing hybrid approach (0.72 s Arria 10, 0.53 s Stratix 10) outperforms the best-performing multi-threaded kernel approach (2.8 s Arria 10, 0.58 s Stratix 10) and the best-performing single work-item kernel approach (3.55 s Arria 10, 2.65 s Stratix 10). These results further complicate the manual optimization process because different threading models can

lead to optimal performance, even within a single application. Fortunately, using the high-level directives in the OpenACC-to-FPGA framework to switch between threading models requires modifying only two clauses in the enclosing OpenACC directive, as compared with OpenCL, which requires modifying the host and device code.

In the above benchmark optimization sections, we evaluated the runtime performance of the five applications after applying different FPGA-specific optimizations. We show that both the Arria 10 and Stratix 10 executions benefit from these optimizations, with the Stratix 10 device allowing for higher replication factors and generally achieving lower runtimes. We also see that both multi-threaded and single-threaded kernels can perform well, but that if applicable, single-threaded optimizations generally result in the best performance.

2.5.5 Resource Usage Evaluation. This section evaluates the relationship between the reported resource usages and kernel frequency (fmax) and runtime performance using the two benchmarks with the highest variety of code versions, SRAD and Jacobi.

Table 8. SRAD benchmark resource usage data (Arria 10)

Model	Version	DSPs	RAMs	fmax	Runtime (s)
multi	nd_reduce	88	594	247	61.13
multi	nd_reduce_simd4	271	1025	229	24.7
multi	nd_update	58	537	258	27.09
multi	**nd_update_simd4**	**193**	**898**	**227**	**20.76**
multi	nd_update_simd8	373	1441	214	30.73
swi	swi_sw	59	545	257	39.44
swi	swi_r4_sw	129	516	252	15.2
swi	swi_r4_sw4x4	264	601	285	6.94
swi	swi_r4_sw4x8	284	643	257	7.83
swi	**swi_r4_sw8x4**	**424**	**695**	**287**	**6.67**
swi	swi_r4_sw8x8	444	737	270	7.51
swi	swi_r4_sw16x16	804	1064	245	7.98

Table 9. SRAD benchmark resource usage data (Stratix 10)

Model	Version	DSPs	RAMs	fmax	Runtime (s)
multi	nd_device	68	1151	264	55.81
multi	nd_device_numc4	272	2147	205	25.24
multi	nd_device_numc8	544	3475	172	16.21
multi	nd_device_simd4	217	1565	221	16.42
multi	nd_device_simd8	415	1923	216	9.58
multi	nd_host	46	1062	256	22.28
multi	nd_host_numc4	184	1791	234	17.11
multi	nd_host_numc8	368	2763	183	13.26
multi	nd_host_simd4	153	1310	250	9.73
multi	**nd_host_simd8**	**295**	**1469**	**234**	**8.27**
swi	swi_sw	51	984	282	42.67
swi	swi_sw4x4	158	1200	258	37.41
swi	swi_sw4x8	178	1196	240	39.11
swi	swi_sw8x4	284	1404	224	44.29
swi	swi_8x8	304	1400	227	42.79
swi	swi_r4_sw	121	1110	256	15.56
swi	swi_r4_sw4x4	228	1426	231	6.56
swi	**swi_r4_sw4x8**	**248**	**1429**	**237**	**5.58**
swi	swi_r4_sw8x4	354	1689	255	6.35
swi	swi_r4_sw8x8	374	1692	241	5.64
swi	swi_r4_sw16x16	666	2244	222	7.83

2.5.5.1 SRAD Resource Evaluation. In the SRAD multi-threaded kernels, increasing the replication factors scaled the resources used. Conventional logic suggests that using more of the FPGA resources would result in higher performance, and this is often the case. However, scaling the replication factors—and thus the degree of parallelism—also lowers the operating fmax, which typically degrades performance. Therefore, there is often a trade-off between resource usage and operating frequencies, and the goal is often to achieve a balance between increasing parallelism and maintaining a high fmax. The effects of this relationship are shown in Tables 8 and 9. On the Arria 10, the lowest runtime multi-threaded version employed a sufficient degree of 4-way `simd` parallelism

while still maintaining a high fmax relative to the other versions at 226.8 MHz. On the larger Stratix 10, 8-way SIMD parallelism performed well with an fmax of 216.91 MHz.

In the single work-item kernels, a similar behavior was observed. Although replication generally increases performance, with higher degrees of unrolling the trade-offs between parallelism and fmax are relevant. On the Arria 10, the best-performing kernel achieved a balance between a high degree of parallelism (replication factors of 4, 8, and 4) and a high fmax (286.69 MHz). On the larger Stratix 10 slightly higher replication factors (4, 8, 8) performed optimally.

Table 10. Jacobi benchmark resource usage data (Arria 10)

Model	Version	DSPs	RAMs	fmax	Runtime (s)
multi	nd	3	416	290	16.32
multi	nd_coll	3	409	279	16.07
multi	nd_coll_elim	3	403	285	4.85
multi	nd_coll_elim_numc4	12	634	262	1.58
multi	**nd_coll_elim_numc16**	**48**	**1558**	**210**	**1.44**
swi	swi_x_swi	3	544	267	5.17
swi	swi_coll_x_swi_coll	3	407	311	4.48
swi	swi_sw_x_swi_flat_coll	3	421	287	5.02
swi	**swi_sw16_x_swi_flat_coll**	**768**	**852**	**208**	**3.55**
hybrid	swi_sw16_x_nd_numc4	768	914	217	1.49
hybrid	swi_sw16_x_nd_numc16	768	1154	212	0.77
hybrid	swi_sw16_x_nd_numc32	768	1474	196	**0.72**
hybrid	swi_sw16_x_nd_simd16	768	854	220	4.07

2.5.5.2 Jacobi Resource Evaluation. By using the multi-threaded kernel approach in the Jacobi kernel (Tables 10 and 11), applying the collapse and kernel boundary elimination optimizations significantly improved performance without significantly changing resource use. On both the Arria 10 and Stratix 10, as we apply multi-threaded replication, the resource usage increases, and the fmax decreases. In this case, the trade off does result in lower overall runtimes.

Table 11. Jacobi benchmark resource usage data (Stratix 10)

Model	Version	DSPs	RAMs	fmax	Runtime (s)
multi	nd	3	754	298	14.34
multi	nd_coll	3	790	297	4.60
multi	nd_coll	3	789	285	4.79
multi	nd_coll_elim_numc4	12	1323	271	1.32
multi	nd_coll_elim_numc16	48	2835	211	0.65
multi	**nd_coll_elim_numc24**	**72**	**3843**	**226**	**0.58**
multi	nd_coll_elim_simd4	12	1276	257	1.50
multi	nd_coll_elim_simd16	48	2834	213	1.02
swi	swi_x_swi	3	790	299	4.61
swi	swi_coll_x_swi_coll	3	788	309	4.44
swi	swi_sw_x_swi_flat_coll	3	645	329	4.18
swi	**swi_sw16_x_swi_flat_coll**	**768**	**1479**	**278**	**2.65**
hybrid	swi_sw16_x_nd_numc4	768	1218	269	0.98
hybrid	**swi_sw16_x_nd_numc16**	**768**	**1638**	**231**	**0.54**

In the single work-item approach, on the Arria 10 the *window* optimization and aggressive unrolling significantly decreased the fmax. However, as with the multi-threaded case, the large increase in parallelism more than offsets the decrease in fmax, which resulted in a lower overall runtime. Also, applying the window and unrolling optimizations significantly increased the DSP usage, which is a powerful resource in the Arria 10 and Stratix 10 FPGAs that generally improves performance when used fully [172].

In the hybrid-threading approach, on both the Arria 10 and Stratix 10 devices, the window optimization still uses a significant number of DSPs. However, the multi-threaded nature of the array copy kernel also consumes a significant portion of RAM blocks. Using these resources resulted in the lowest overall runtime, even with a lower fmax.

In this section we explored the relationships between FPGA resources and performance for two of the studied benchmarks. In general, we see that both a high

resource utilization and a high operating frequency are desirable for performance, but that these two quantities are inversely correlated. The best-performing version for both benchmarks strikes a balance between these two metrics.

2.5.6 Compilation Times. In this section, we briefly investigate the effect of different performance optimizations on the compilation time, and trade-offs between performance and higher compilation costs.

In Figure 17 we highlight a specific application, SRAD compiled on the Stratix 10 device, to explore the effects of compilation time. Generally, we see that as more advanced optimizations and loop unrolling are applied, the compilation time increases. As a result, there is generally an inverse relationship between runtime performance and compilation time. This is not unexpected, as the replication optimizations utilize more FPGA resources, which increases the placing and routing demands, a main factor contributing to the compilation times.

Not every optimization equally contributes to increases in compilation time. With the first three bars of Figure 17, we see that increasing the compute unit replication (`numc4`, `numc8`) improves the performance, but significantly increases the compilation time as well. Alternatively, we see that applying `simd` parallelization also increases performance, but has a much smaller effect on compilation time. As a result, `simd` parallelization may be a more attractive optimization option if compilation time is a concern.

The figure also shows that many of the single work-item optimizations do not significantly contribute to the compilation time until aggressive sliding window replication is applied (see the final set of bars in Figure 17). This provides another motivation for application developers to target single work-item instead of multi work-item kernels for a lower time-investment option.

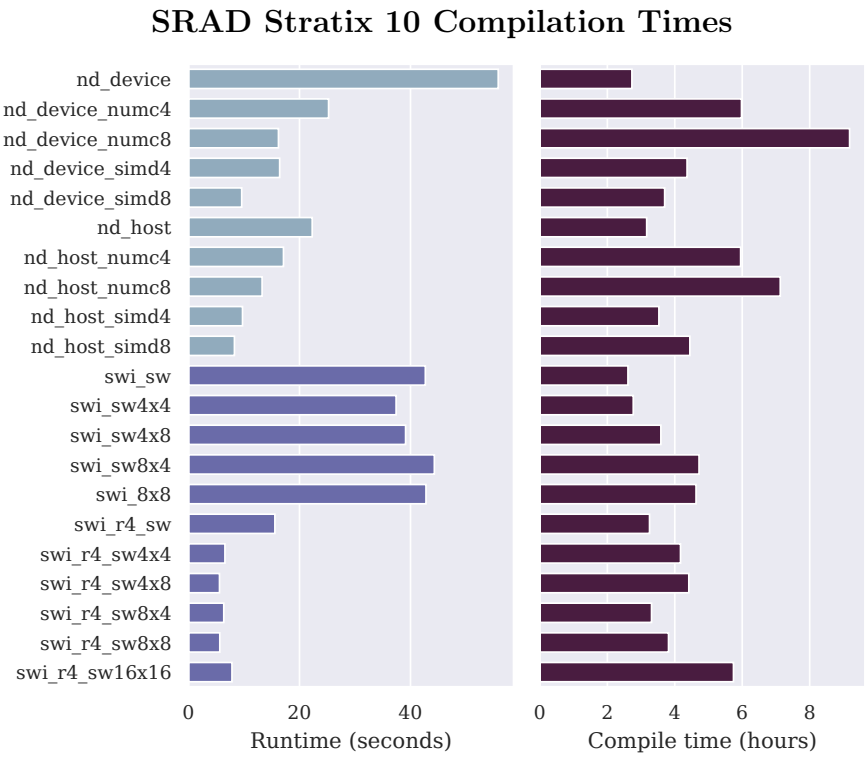


Figure 17. SRAD Runtime performance (in seconds) compared to compilation time (in hours) (Stratix 10)

2.5.7 Performance Portability. In this section we briefly investigate the performance portability between the Arria 10 and Stratix 10 devices.

In Figure 18, on the left we compare the best-performing Arria kernel with the best-performing Stratix kernel, all executed on the Arria 10 device. Essentially we evaluate how well the Stratix 10 codes port to the Arria 10. We see the converse evaluation on the right of Figure 18, evaluating how well the Arria 10 codes port to the Stratix 10.

The *hotspot multi* and *jacobi multi* Stratix 10 kernels failed to compile for the Arria 10 device, primarily due to the differences in resource availability across the two devices. As expected, the best-performing Stratix 10 kernels generally utilize more hardware resources than the best-performing Arria 10 kernels.

In Figure 19, we quantify the performance portability as an averaged fraction of peak performance across the two devices. The Arria 10 kernels typically perform well on the Stratix 10 device (averaging 89%), as they just slightly under-utilize the larger device. The Stratix 10 kernels, when successfully built, achieve lower performance on the Arria 10 device (averaging 70%), as they typically challenge the resource limitations.

2.5.8 LULESH Initial Evaluation. In this section, we explore an initial evaluation of the LULESH 2.0 proxy application [171] on the Stratix 10 FPGA. During the evaluation, we first targeted the entire LULESH application for offloading to the Stratix 10 device. However, due to the significant number of kernels (95) in the application, the FPGA resources were quickly exhausted (see row 1 of Table 12). Executing the entire application on a single FPGA would require re-flashing the device mid execution. This behavior is currently not possible

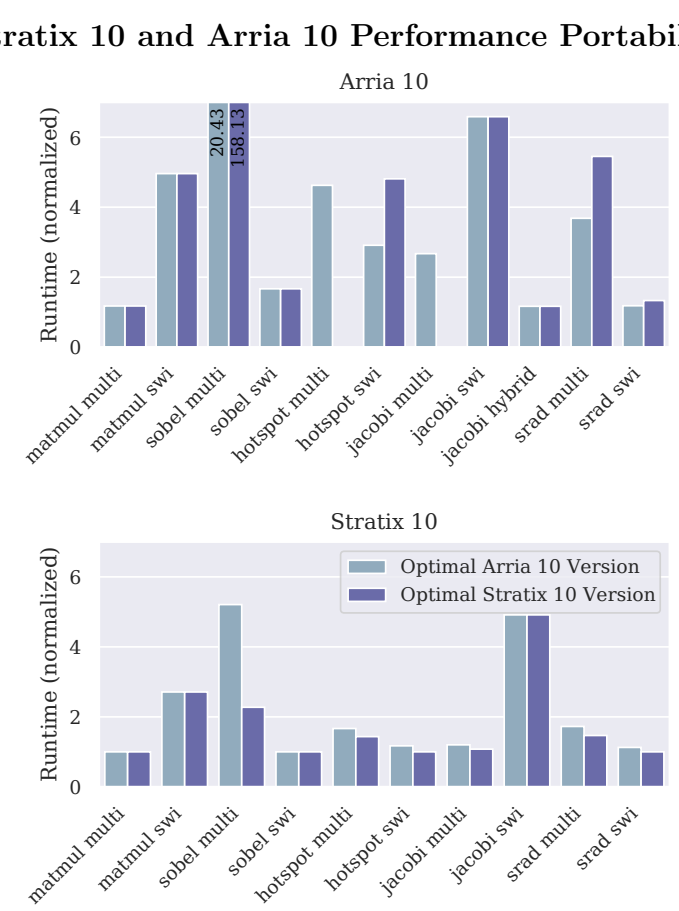


Figure 18. A performance portability evaluation of the best-performing Stratix 10 and Arria 10 versions run on the Arria 10 (left), and the best-performing Stratix 10 and Arria 10 versions run on the Stratix 10 (right). Runtimes are normalized such that the best-performing version across both devices is represented as 1.

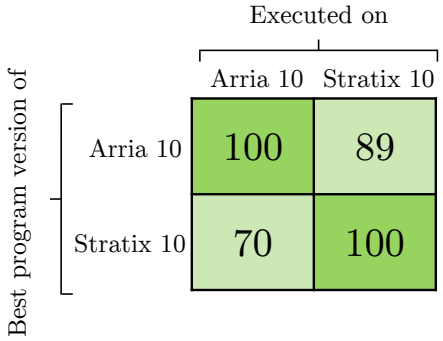


Figure 19. The average percentage of peak performance achieved when executing program versions optimized for each device across the two different devices.

with the OpenACC-to-FPGA framework, but is an interesting direction for future works targeting real-world applications.

We next focused on a single routine in the LULESH application, *EvalEOSForElems*. This same kernel was targeted by Jin et al. [173] in their evaluation of the LULESH OpenCL version for FPGAs, as it is representative of the range of kernels found in LULESH and does not immediately exhaust the FPGA resources. As we see in Table 12, this single function successfully compiles on the Stratix 10 device if the replication factors (*simd* for multi-threaded kernels and standard loop unrolling for the single-threaded kernels), are small.

Table 12. LULESH benchmark resource usage data (Stratix 10)

Version	Total Logic	DSPs	RAMs	Compiled
LULESH	650%	91%	415%	No
EvalEOS nd	47%	4%	27%	Yes
EvalEOS nd_simd2	64%	9%	35%	Yes
EvalEOS nd_simd4	96%	36%	79%	No
EvalEOS swi	48%	4%	27%	Yes
EvalEOS swi_fused	46%	4%	25%	Yes
EvalEOS swi_fused2	63%	9%	32%	Yes
EvalEOS swi_fused4	94%	36%	86%	No

In Table 12 and Figure 20 we refer to several different versions of the *EvalEOSForElems* kernel (abbreviated as *EvalEOS*). Like the previous applications, the *nd* keyword refers to multi-threaded kernels and the *simdX* keyword refers to simd replication with a replication factor of *X*. The *swi* keyword refers to single-work-item executions. The *fusedX* keyword is specific to LULESH. In these code versions, instead of using separate OpenACC parallel regions for each loop in the *EvalEOS* function, we combine all loops into a single parallel region. This avoids multiple kernel launches by the underling Intel FPGA SDK for OpenCL. Finally,

the X in *fusedX* refers to the degree of unrolling applied to each loop (via *#pragma unroll X*).

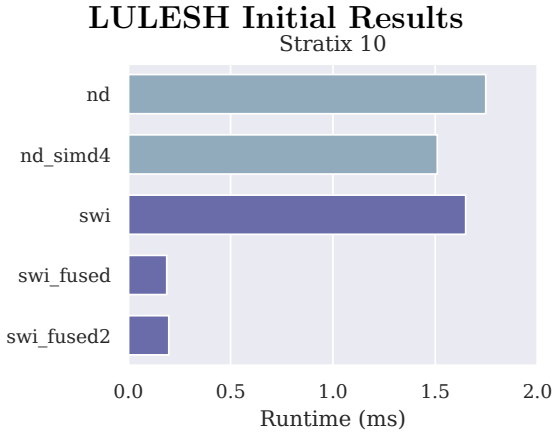


Figure 20. Runtime performance (in ms) of LULESH proxy application with different FPGA-specific optimizations applied (Stratix 10 and Arria 10). Blue bars indicate the multi-threaded approach, and purple bars indicate the single work-item approach (smaller is better).

Figure 20 shows the runtime performance of the different kernels using an input size of 45^3. Like many of the other applications evaluated, the baseline multi-threaded and single-threaded *EvalEOS* executions perform similarly. We see modest performance improvements when using *simd* replication, and significant performance improvements when using the fused kernel. This is most likely due to reducing the overhead of launching several smaller kernels. Surprisingly, we do not see a performance improvement when applying loop unrolling to the single-work item kernel, which warrants further investigation. Because *EvalEOSForElems* contains no nested loops, reduction variables, or stencil patterns, the more advanced single work-item optimizations could not be applied.

2.6 Intel and Xilinx OpenCL Portability Study

Since Intel's acquisition of Altera in 2013, Intel and Xilinx have been the two most dominant FPGA device manufacturers. In this Section, we discuss a

study exploring the performance portability of applications executing on Xilinx FPGA architectures that were originally written for Intel FPGA architectures. Although both vendors support the OpenCL standard and incorporate FPGA-specific programming patterns like shift registers, it is unclear how similar these OpenCL implementations are, as each implementation contains vendor-specific extensions. Although this study does not directly involve directive-based programming, it has major implications for the incorporation of Xilinx FPGAs into the OpenCL-based OpenACC-to-FPGA framework evaluated previously in this dissertation.

Because of OpenCL's portability, an OpenCL kernel that is authored for an Intel FPGA should, in theory, be synthesizable for a Xilinx FPGA and vice versa. However, in practice, although many HLS-based application kernels exist for Xilinx and Intel hardware, little has been reported about the actual portability of HLS kernels between these two device families. Even if one kernel can be compiled to run correctly on both platforms, performance portability between platforms is far from guaranteed. Therefore, understanding the commonalities between Intel and Xilinx HLS tools and the quirks peculiar to each is a worthwhile topic for investigation. If performance portability between these FPGA families can be achieved, then it would enable application designers to confidently author their kernels once in OpenCL C and achieve high performance with each family by using its respective HLS tools.

This section presents an initial evaluation of portability and performance of OpenCL C kernels that were originally written for an Intel FPGA and then reconfigured for a Xilinx FPGA. We use the Intel FPGA implementations from Zohouri et al. [160] of the Rodinia benchmark suite [147] as a baseline and

investigate the process and impact of porting these implementations to a Xilinx FPGA. The research in this section was performed as a co-authored collaboration, Cabrera et al [12].

2.6.1 Porting Intel Applications to Xilinx Hardware. Table 13 lists the particular kernel versions of each benchmark that we used and ported in our evaluation. The version numbering follows that of Zohouri et al. [160] in which odd-numbered kernels use the single work-item execution model.

Table 13. List of kernels ported from Intel OpenCL to Xilinx OpenCL

Application	**Baseline**	**Best**
Pathfinder	v1	v5
CFD	v1	v5
SRAD	v1	v5
HotSpot	v1	v5

For each application examined, we ported the both baseline and the best performing kernels (*Baseline* and *Best*, respectively, in Table 13). The baseline versions are single work-item kernels in which there are no FPGA optimizations supplied as hints to the hardware compiler aside from use of the single work-item model itself. The best performing kernels are the versions that were reported to give the best performance among all kernel versions for each tested application in the original work by Zohouri et al. (i.e., the best evaluated kernel when targeting Intel-based Stratix V and Arria 10 FPGAs).

We evaluated the portability and performance of these kernels by performing the minimum modifications required to port the annotated hardware optimizations for each kernel from the Intel specification to the Xilinx specification. Although using OpenCL C gives us a foundation for porting kernels between the two platforms, the way in which optimizations are specified between the Xilinx and

Intel platforms is different. Intel uses a combination of specific programming patterns, `#pragmas` and `__attributes__`, to provide guidance to the hardware compiler, whereas Xilinx uses only `__attributes__`. Additionally, although there is sometimes a one-to-one mapping of kernel optimizations between platforms, this is not always the case. The following sections detail the loop unrolling and shift register FPGA optimizations at the OpenCL level, how they are expressed for an Intel platform, and the changes we made to express that same construct on a Xilinx platform. We note that both of these optimizations at the OpenCL level are exactly those abstracted by the OpenACC-to-FPGA framework discussed in Section 2.2.

2.6.1.1 Loop Unrolling. As previously discussed in Section 2.2, loop unrolling is a common optimization in FPGA programming. In both the Intel and Xilinx tools, loop unrolling hints allow the hardware compiler to use additional resources to replicate the loop body. In a single-work item execution context, this allows for more deeply nested pipelines, higher FPGA resource utilization, and typically better overall performance. Intel and Xilinx support unrolling loops through compiler hints. For Intel OpenCL kernels, a loop is preceded with

`#pragma unroll N`.

For Xilinx, the previous pragma is replaced with

`__attribute__((opencl_unroll_hint(N)))`.

In both cases, `N` is the loop unrolling factor. Therefore, the mapping between loop unrolling for Intel and Xilinx OpenCL is straightforward. The hardware compiler will determine whether it is possible to unroll the loop given available resources of the target FPGA. Also, the Intel and Xilinx compilers will both attempt to analyze

and automatically unroll non-annotated loops, but in our experience, manually applying the directives and attributes results in more consistent compilations and performance.

2.6.1.2 Shift Registers. Also discussed in Section 2.2, shift registers are an FPGA construct that aid in efficient pipelining of loop iterations by storing data to satisfy inter-loop dependencies and avoiding redundant loads from global FPGA memory. How these shift registers are constructed in OpenCL depends on the vendor. Both vendors support using registers within the FPGA fabric. Depending on the size, the Intel hardware compiler might try to synthesize a shift register from user-supplied programming patterns by using on-chip memories. Xilinx supplies a header file that allows the shift register to be synthesized by configuring lookup tables in the FPGA fabric to act as a RAM-based shift register. Unlike the case for loop unrolling, there is not a one-to-one mapping for inferring shift registers between vendors.

We show a minimal example of how to infer a shift register for Intel and Xilinx in Listings2.12. For Intel, a private buffer is declared (line 1), and the size of this buffer is a compile-time constant. Shift register shifting is orchestrated in the inner loop (line 5). For the hardware compiler to infer a shift operation, the inner loop must be unrolled by prepending a `pragma`, as described above. The Xilinx code example is similar. Again, a private buffer must be declared, but an additional attribute (line 2) must be appended to this buffer. This attribute is a hint to the hardware compiler that the kernel designer wants to completely decompose the buffer into a collection of registers. The `complete` keyword indicates that the buffer must be completely decomposed into a collection of registers, and the 0 argument implies that we are performing this decomposition among all dimensions of the

buffer. The inner loop that orchestrates the shifting (line 6) is then unrolled, as described above, by appending an attribute (line 5).

Listing 2.12 Inferring a shift register using the Intel and Xilinx platforms.

```
// Intel OpenCL SDK for FPGAs example
int shift_reg[SR_SIZE]; // where SR_SIZE is a compile time constant
for (int n = 0; n < N; n++) {
    shift_reg[SR_SIZE − 1] = input_arr[n];
    #pragma unroll SR_SIZE − 1
    for(int i = 0; i < SR_SIZE − 1; i++)
        shift_reg[i] = shift_reg[i + 1];
}

// Xilinx Vitis example
int shift_reg[SR_SIZE]
    __attribute__((xcl_array_partition(complete,0)));
for (int n = 0; n < N; n++) {
    shift_reg[SR_SIZE − 1] = input_arr[n];
    __attribute__((opencl_unroll_hint(SR_SIZE − 1))
    for(int i = 0; i < SR_SIZE − 1; i++)
        shift_reg[i] = shift_reg[i + 1];
}
```

2.6.2 Minimum Modification Porting Evaluation. Figure 21 shows the results of porting the baseline best performing (on the original Intel hardware) kernel versions, as detailed in Section 2.6.1. The following sections detail the process of porting each kernel to the Xilinx platform. In this section, we only evaluate a minimally modified kernel ported from Intel OpenCL to Xilinx OpenCL. Cabrera et al. [12] also present a more in-depth evaluation using the Pathfinder application, but that evaluation was done independently from this dissertation's research, and is not included in this document.

2.6.2.1 Pathfinder Porting and Evaluation. The `pathfinder` kernel version `v1` was straightforward to port because there were no vendor-specific

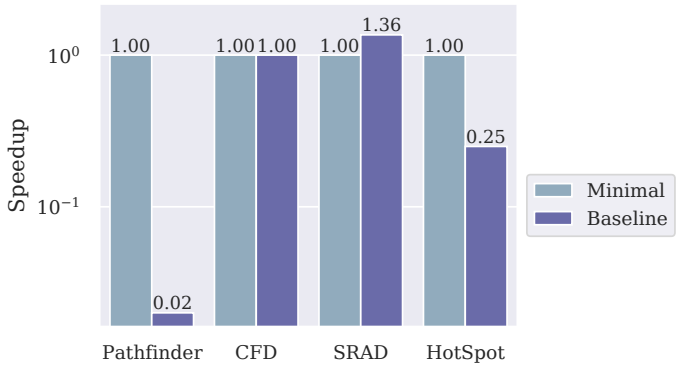


Figure 21. Each application's performance on the Xilinx platform for the port of their respective baseline kernel and the port with minimum modification of the best performing kernel when targeting the Intel platform. The performance is reported as speedup relative to the Xilinx baseline result.

compiler optimizations to port. For `pathfinder` kernel version `v5`, we set two marcos at compile time when building the kernel: `BSIZE` and `SSIZE`. We describe these two macros below.

Because the `pathfinder` kernel performs a stencil operation, an FPGA-specific shift register or sliding window can be used to reduce redundant memory accesses, as mentioned in Section 2.6.1.2. The minimum size of the shift register is constrained by the smallest number of contiguous array elements required to encapsulate a single iteration of the stencil operation. In the `pathfinder` application, this equates to one complete row of the input array plus one additional element. For large input data sizes, even one row of the input dataset can be too large for implementation in a single shift register. The `BSIZE` macro controls this column size and thus indirectly controls the resulting shift register size.

Although one output array element is assigned each iteration by default, the second macro `SSIZE` allows multiple stencil operations per iteration. By allowing multiple operations per iteration, we can reduce the total number of iterations and increase the FPGA utilization. Increasing `SSIZE` can significantly improve

performance if the FPGA has enough resources to support the hardware needed to perform multiple stencil operations and the hardware compiler does not have to increase the loop initiation interval or decrease the compute unit operating frequency.

In our initial port, we used the best performing macro values listed from Zohouri et al. [160], which sets `BSIZE` $= 32,768$ and `SSIZE` $= 32$. However, we found that the kernel using this parameterization is not synthesizable immediately when building on Xilinx; the hardware compiler only allows a buffer to be partitioned 1,024 times. Therefore, it is not possible for us to infer a shift register by using the original parameterization. To use the given parameters, then, we do not partition the array. We next replace the Intel loop unrolling construct with the Xilinx one, as detailed in Section 2.6.1.1. At this point, the kernel was successfully built and executed on the Xilinx FPGA.

The best performing Pathfinder kernel on Intel is 43 times slower than the baseline version when both kernels are ported to the Xilinx FPGA. Performance was expected to decrease because, with the given information, the Xilinx compiler was not able to infer a shift register. Also, the output logs generated by the Xilinx hardware compiler reported that the main loop of computation in this kernel was not successfully pipelined.

2.6.2.2 CFD Porting and Evaluation.

The version `v1` kernel of CFD is a straightforward single work-item port of the kernel used in the GPU OpenCL kernel available in the original Rodinia [147] benchmark suite. Because the `v1` version did not have any Intel-specific pragmas in the kernel, no changes were made to the `v1` kernel to target the Xilinx FPGA. The version `v5` CFD kernel was the best performing on the Intel Stratix V FPGA, according to Zohouri et al.

This kernel had various optimizations, including adding the restrict qualifier to the input arrays and using a shift register-based reduction to accumulate the flux contribution (both optimizations that were automated by the OpenACC-to-FPGA framework in Section 2.2). The `v5` version also had an unroll pragma, which was re-written using the Xilinx unroll hint attribute.

The generated compiler information for version `v1` reported that the Xilinx compiler was unable to flatten the main computation loop because the outer loop was not a perfectly nested loop. It also reported that there was a data dependency in the loop, which greatly reduced the loop iteration interval. The build reports generated by the Xilinx compiler for version `v5` showed a lower loop initiation interval than the `v1` kernel. However, the main iteration loop was still unable to be flattened because the the outer loop had nontrivial logic in the loop latch. Despite the reported lower initiation interval, the two kernels took essentially about the same amount of time to execute; the v5 kernel executed slightly faster with a 0.23% reduction in kernel execution time.

Overall, the performance of directly porting CFD kernels from Intel to Xilinx FPGAs was quite poor, with a 70× increase in kernel execution time compared with Zohouri et al.'s work [160]. The performance degradation is not surprising when looking at the large loop latencies and initiation intervals reported in the Xilinx build reports, which indicates further Xilinx specific optimizations and compiler hints are needed.

2.6.2.3 SRAD Porting and Evaluation.

Because the SRAD application also implements an iterative stencil algorithm, it shares many of the same FPGA-specific optimizations and tuning parameters with the Pathfinder application. The SRAD `v1` kernel implements a straightforward approach that

extends the source Rodinia OpenCL kernel with restrict keywords on input array variables and creates single work-item kernels. The highest performing kernel on the Intel platform version `v5` combines the five separate kernels into a single kernel and implements a shift register-based reduction and shift register-based sliding window.

We make several changes for the minimally modified Xilinx analog kernel of the Intel-based `v5`. We replaced an Intel-specific attribute applied to the entire kernel,

```
__attribute__((max_global_work_dim(0)))),
```

with an analogous one recognized by the Xilinx platform,

```
__attribute__((reqd_work_group_size(1, 1, 1))).
```

We also replaced instances of `#pragma unroll` with the previously mentioned Xilinx-specific attribute and annotated the shift register with the following Xilinx-specific attribute:

```
__attribute__((xcl_array_partition(complete, 0)))
```

For this application, we were able to completely partition the array used for the shift register operation and were not required to do a block or cyclic partition. The Xilinx platform's restrictions on the size of the shift register array are not necessarily a limitation. Although the Intel platform successfully compiles with larger shift register sizes, the larger arrays can significantly degrade performance, which is why the manual partitioning via the `BSIZE` variable and logic is present, even in the Intel-optimized code. Finally, we left the `SSIZE` replication factor at it's default value of 2.

As shown in Figure 21, the SRAD `v5` kernel represents the only example in which directly porting an Intel-optimized kernel to use analogous Xilinx constructs improves performance over a more platform-agnostic baseline. This application demonstrates that directly translating constructs can improve performance over a baseline in some cases, although we do note that the absolute performance of the baseline and `v5` SRAD underperform their Intel counterparts. That is, there is still a significant amount of room for Xilinx-specific improvement in these kernels.

2.6.2.4 HotSpot Porting and Evaluation. Like Pathfinder and SRAD, the HotSpot application implements an iterative stencil. Again, the `v1` version of the kernel is directly adapted from the original OpenCL, only adding restrict keywords and switching to a single work-item kernel. In the `v5` version, we again replaced the Intel-specific loop unrolling, kernel dimension attributes, and directives with Xilinx-specific attributes. Like Pathfinder, the default `BSIZE` value results in a shift register that is slightly too large for complete partitioning by the Xilinx compiler with a size of 1,032 elements against the restriction of 1,024. However, instead of defaulting to a blocking or cyclic partition scheme—which typically leads to poor performance, as shown in the following section—for the results presented in Figure 21, we instead reduced the value of `BSIZE`, which allowed full compilation with complete partitioning on the shift register array. Like SRAD, we again maintain the default `SSIZE` replication factor, which is 16 for the HotSpot application.

Figure 21 shows that, as with the Pathfinder application, directly translating the Intel-specific optimizations to their Xilinx counterparts in the `v5` version degrades performance compared with a more agnostic, less-optimized baseline.

HotSpot represents another example in which one-to-one kernel optimization ports do not lead to portable performance.

2.7 Directive-based FPGA Programming: Related Works

Watanabe et al. [131] presented preliminary results on a very closely related project and targeted OpenACC using the OmpSs compiler [174, 46], which, like OpenARC, can generate output OpenCL. However, instead of developing optimizations inside the OpenACC to OpenCL translation, they generate SPD code for SPGen (Stream Processing Generator) alongside OpenCL. The separation program designator (SPD) code bypasses the OpenCL abstraction layer, translating directly into HDL. This seems to be a promising project to complement the OpenACC-to-FPGA framework, as it performs a similar function but uses a very different software stack not reliant on the Intel FPGA SDK for OpenCL, which has both advantages and disadvantages.

Sommer et al. [175] presented a fully functional implementation of the OpenMP device offloading for Xilinx FPGAs. The work integrated a custom compiler toolflow into the LLVM/Clang OpenMP offloading infrastructure. The input program contains one or more OpenMP target directives. The compiler generates a complete FPGA design, including a ThreadPoolComposer device software executable, Vivado HLS input file, and kernel description. In the prototype, the FPGA offloaded versions show slower performance than one 4-core CPU.

The Scalable Parallel Computing Laboratory (SPCL) at ETH Zurich is also developing several tools for high-level FPGA programming, although their approach is very different from the OpenACC-to-FPGA framework. One work done by de Fine Licht and Hoefler incorporates software engineering design principles into HLS

development [176]. These works are somewhat similar to our work in that they try to account for differences when targeting an Intel or Xilinx FPGA through a C++ library they developed called `hlslib`. In contrast, our work focused on using OpenCL C for Intel and Xilinx FPGA kernels to evaluate the portability and performance of starting from a kernel optimized for an Intel platform and then porting that specification to a Xilinx platform.

Another work from SPCL, the DataCentric (DaCe) project [177] recently integrated FPGA support. DaCe relied on Python and a graphical user interface-based dataflow diagram to map computations to hardware. Although this abstraction level is significantly different from that of traditional HPC applications, it could map well to dataflow architecture of FPGAs for certain applications. DaCe might be an interesting option for new FPGA-centric applications, but it would require significant code restructuring and algorithm modifications to existing applications.

2.8 Directive-based FPGA Programming: Conclusions

This chapter presents a directive-based, high-level FPGA-specific optimization framework, consisting of a set of user directives and corresponding compiler optimizations, for more efficient FPGA computing. The proposed framework enables directive-based interactive programming by allowing users to provide important information to the compiler using directives. These directives instruct the compiler to automate FPGA-specific optimizations and allow control of important tuning options at a high level. We have developed several FPGA-specific optimizations in the OpenARC compiler framework, such as a reduction optimization to exploit shift registers, sliding window optimizations to enable more

efficient pipelining, and branch-variant code motion optimization to reduce overall resource usage.

We first evaluate the proposed framework by porting five OpenACC benchmarks and comparing them against manually optimized OpenCL versions on an Intel Stratix V FPGA. The results show that the directive-based, semi-automatic optimizations can successfully realize performance comparable to the hand-written, low-level codes in many cases, and that OpenACC FPGA programs can have performance benefits over OpenACC GPU programs and OpenMP CPU programs in terms of runtime and power usage.

Next, these optimizations were holistically evaluated against a set of representative benchmarks using Arria 10 and Stratix 10 FPGAs. The experimental results show that multi-threaded and single-threaded kernels can perform well on FPGAs, depending on which optimizations can be applied to a specific application. For example, most applications that allow for advanced single-threaded optimizations outperform their multi-threaded counterparts. In contrast, applications in which these single-threaded optimizations do not apply might perform best using multi-threaded compute unit or SIMD replication.

The relationship between resource usage and runtime performance was also explored. In general, higher resource usage indicates better utilization that typically results from replication, which leads to better performance. However, there are also several exceptions to this trend. In some cases, if two benchmark versions employ the same degree of parallelism, then higher resource usage can indicate less-efficient routing and could hurt performance. In other cases, if additional logic is implemented that results in higher resource usage and sacrifices the kernel fmax, then performance can suffer. Finally, even if more logic were

implemented without sacrificing the fmax, lower performance is still observed if the initiation interval is increased.

The impact of optimizations on compilation time is mostly straightforward; as more aggressive optimizations are applied, compilation times increase. However, some optimizations have a smaller impact on compilation times, such as SIMD replication for multi work-item kernels and non-replicating optimizations for single work-item kernels. When investigating performance portability between the two devices, we see that the Arria 10-optimized kernels typically perform well on the newer devices, but the Stratix 10-optimized kernels may not be portable to the older device, or may perform poorly.

Additionally, an initial evaluation on the LULESH 2.0 proxy application was discussed. We showed that, while the entire application cannot be mapped to the Stratix 10 FPGA, we could map and execute a representative kernel, and apply OpenACC-based optimizations to improve performance.

A study comparing the performance portability of Intel-FPGA-specific and Xilinx-FPGA-specific OpenCL was discussed. We saw that even though OpenCL is a portable standard, applications typically could not be ported between the two devices without minimal modifications, and that these minimal modifications are far from sufficient for portable performance. These evaluations further motivate the need for a higher-level abstraction for general scientific FPGA programming, for example the OpenACC-to-FPGA framework.

We plan to aggressively extend the OpenACC-to-FPGA framework in the future. An immediate target is to integrate the Aspen performance modeling tool [143, 142] into the OpenACC-to-FPGA translation to automate the

optimization process, including threading-model selection and lower level tuning of replication and unrolling factors.

We also aim to support Xilinx devices in the near future. Developing *hlslib* [176] and other cross-platform tools at the OpenCL level can greatly simplify multidevice support in the OpenACC-to-FPGA framework. Also, the work presented in Section 2.6.1 is a major step toward supporting Xilinx hardware.

Although the presented directive-based framework has exclusively relied on OpenACC as the front-end programming model, we envision supporting OpenACC and OpenMP within this framework due to the introduction and increased popularity of the OpenMP offloading model. By employing tools such as CCAMP [13, 14] (the main subject of Chapter III) for OpenMP to OpenACC translation and developing an analogous FPGA-specific API for OpenMP, the OpenACC-to-FPGA framework can be extended to support OpenMP offloading models.

Finally, with the introduction of the OneAPI framework, Intel's FPGA support is projected to shift from OpenCL to OneAPI's SYCL/DPC++ implementation. Likewise, our long-term goal is to migrate the OpenACC-to-FPGA framework to use these newer intermediate representations.

CHAPTER III

AN INTEGRATED TRANSLATION AND OPTIMIZATION FRAMEWORK FOR OPENMP AND OPENACC

This chapter contains previously published material with co-authorship. All of the presented research in this chapter was conducted as a collaboration between the University of Oregon and Oak Ridge National Laboratory. The original translation passes (Sections 3.3, 3.5.2) and evaluation were presented at HeteroPar 2019 [13]. The rest of the work in this chapter was presented at SC 2020 [14]. For both publications, Seyong Lee was instrumental in the conceptualization of the projects, and provided continued support, suggestions, and advice throughout the projects with weekly meetings. Dr. Lee was also responsible for writing the original OpenARC translation pass for the OpenMP to OpenACC direction, and for translating several algorithms and pseudocode into concrete OpenARC compiler passes for the device-specific optimizations. Finally, Dr. Lee assisted with revisions to the documents, and sometimes portions of the writing, typically in the introductions and conclusions. Allen Malony and Jeffrey Vetter both provided high-level guidance and advice during all three projects. They both also assisted with revisions, and contributed information for the introduction and conclusions sections. I researched, designed, and, with help from Dr. Lee as mentioned above, implemented the compiler passes for works published at HeteroPar 2019 and SC 2020. I also collected all data, performed all experiments, and did the bulk of writing for both publications.

3.1 OpenMP and OpenACC Interoperable Framework: Introduction

Recent trends toward the end of Dennard scaling and Moore's law indicate that future computing systems will become more specialized and comprise more

complex architectures in terms of processors, accelerators, memory hierarchies, on-chip interconnection networks, storage, and so on; this trend has been broadly labeled as *extreme heterogeneity* [6]. Heterogeneous systems that contain more than one type of device (e.g., multicore CPUs, GPUs, field-programmable gate arrays [FPGAs], digital signal processors) have already been observed as the new norm in high-performance computing (HPC), machine learning, and embedded computing communities [178, 179]. Heterogeneous computing allows programmers from different application domains to accelerate their applications by mapping computations to workload-specific devices. However, exploiting these devices often requires low-level, heterogeneous programming models such as CUDA and OpenCL, which often require expertise in the underlying hardware and force programmers to adapt their applications specifically to unique devices, incurring programmability and performance portability issues [180].

Directive-based, high-level programming models, such as OpenMP [170] and OpenACC [168], have evolved to alleviate these programming challenges in heterogeneous computing. These directive-based approaches allow programmers to provide the compilers with important application characteristics (e.g., parallelism and data sharing) via a set of directives to transfer much of the low-level programming and optimization burdens to the compilers. However, as shown in the following sections, device-specific implementations and varying levels of language support and maturity across compilers make it difficult for the existing directive solutions to achieve the ideal performance and portability promised by these standards.

To address these issues, in this chapter we propose CCAMP, an integrated translation and optimization framework for OpenACC and OpenMP. CCAMP is

built on top of OpenARC [39], and performs: (1) automatic translations between the two directive models to enable better performance portability by letting programmers choose more mature programming solutions preferred by the target device and (2) automatic optimizations to better map computations to the target device in a way preferred by the back-end compilers on the given device.

OpenARC uses a high-level intermediate representation and is equipped with various built-in compiler analysis and transformation passes, including OpenMP directive parsing capabilities. The proposed CCAMP framework is built on top of the existing OpenARC and leverages OpenARC's OpenMP and OpenACC parsers, initial lexical analysis, and abstract syntax tree generation. However, the CCAMP translation and optimization layers are novel contributions in this project, as well as modifications of the OpenARC parsers to accommodate directive extensions.

The main contributions of this chapter include:

- the design and implementation of CCAMP Translation, an automatic framework that transforms OpenMP 4+ to OpenACC and vice versa;
- the design and implementation of CCAMP Optimization, a general optimization strategy to map computations to devices in a way preferred by the back-end compilers;
- an evaluation of the proposed framework across an array of devices (e.g., Intel Xeon CPU, IBM Power 9, Nvidia P100, V100) and compilers (e.g., clang, PGI, XLC, GCC) by using the SPEC Accel Benchmark Suite, two kernel benchmarks, and LULESH 2.0; and

– the comparison and evaluation of OpenMP 4+ and OpenACC performance variability.

3.2 CCAMP: Background

3.2.1 OpenACC and OpenMP. As discussed in Chapter I, Section 1.2, OpenACC and OpenMP are two popular programming models for directive-based high-level heterogeneous computing. Although OpenACC was originally developed as a high-level alternative to CUDA for GPU programming, because OpenACC was designed with accelerator-based heterogeneous computing in mind, it has been adopted for various accelerators, such as FPGAs (Chapter II), Xeon Phis [181], and custom CPUs, such as those in the Sunway TaihuLight supercomputer [182, 183].

In contrast, OpenMP has been used for decades as an essential tool for thread-based parallel programming on shared memory systems, such as multicore CPUs. However, from version 4 onward, OpenMP has adopted offloading constructs [184]. OpenMP 4+ and OpenACC share the common goal of providing programmers with a high-level approach to heterogeneous programming. However, there are several important issues and setbacks to using these standards.

One primary issue is that existing directive solutions might not provide portability across diverse architectures. Although OpenACC and OpenMP seek to offer a portable, high-performance, cross-platform solution, they are often at the mercy of vendor-specific compiler implementations. Many devices achieve high performance when using the vendor compiler tied to the device, which often only supports either OpenACC or OpenMP, but not both.

However, even among compilers that prefer specific directive standards, the level of language support, implementation quality, and strategies for the same

standard can vary greatly. This discrepancy is partially caused by the fact that the level of parallelisms that OpenACC and OpenMP offer might be different from those in the target devices. For example, although OpenACC and OpenMP offer three levels of parallelism—gangs, workers, and vectors in OpenACC and teams, threads, and vectors in OpenMP—typical GPUs and CPUs offer only two levels of parallelism: threadblocks and threads in Nvidia GPUs and threads and single instruction, multiple data (SIMD) in Intel CPUs. Therefore, different compilers can choose different mapping strategies.

As a result of these issues, existing OpenACC and OpenMP 4+ implementations do not achieve the goal of being portable for heterogeneous systems. A primary goal of the CCAMP framework is to allow programmers to fully use the existing OpenMP and OpenACC implementations to achieve performance portability across heterogeneous devices.

3.2.2 OpenARC. As described in Chapter I, Section 1.3, OpenARC [39] is an open-source OpenACC compiler built on top of the Cetus compiler framework [141], which performs source-to-source translations of an input OpenACC program into an output CUDA/OpenCL program, depending on target devices (e.g., Nvidia/AMD GPUs, Intel Xeon Phis, Intel FPGAs). OpenARC uses a high-level intermediate representation and is equipped with various built-in compiler analysis and transformation passes, including OpenMP directive parsing capabilities.

The proposed CCAMP framework is built on top of the existing OpenARC and leverages OpenARC's OpenMP and OpenACC parsers, initial lexical analysis, and abstract syntax tree generation. However, the CCAMP translation and

optimization layers are novel contributions in this chapter, as well as modifications of the OpenARC parsers to accommodate directive extensions.

3.3 CCAMP: Automated Translation between OpenMP and OpenACC

As mentioned previously, CCAMP consists of two primary functions: (1) automated translation between OpenMP 4+ and OpenACC and (2) automated optimization within OpenMP 4+ and OpenACC. This section discusses the rationale and implementation details for the translation function.

Because we preserve the semantics of the original application and host code, features such as MPI support, CUDA memory management, other low-level optimizations of legacy OpenACC applications, and asynchronous and concurrent kernels are fully supported and unaffected by CCAMP's optimization and translation passes.

CCAMP's Translation facilities includes two primary translation passes:

- OpenMP 4+ to OpenACC and
- OpenACC to OpenMP 4+.

CCAMP translation can be leveraged to migrate codes to systems with different software support or target devices—for example, those supporting only either OpenACC or OpenMP, which is common among current accelerators and compilers. The translation pass can be used alone or in combination with the CCAMP Optimization passes, as shown in Section 3.4.

Generally, the translations were developed by analyzing how relative parallelism is expressed in the two different standards and by carefully reviewing the intentions and restrictions of the individual directives in the OpenMP 4+ and OpenACC standards' documentations. Although CCAMP does not support the

entire OpenACC and OpenMP standards, many unsupported constructs are also not supported by underlying back-end compilers, especially when offloading to accelerator devices. By omitting these constructs and focusing on the directives and clauses most commonly used by programmers and implemented by back-end compiler writers, optimized and translated codes can be more confidently generated across different ecosystems. Even with these unsupported constructs, CCAMP still generates functionally correct output programs by serializing or ignoring them, similar to current back-end compilers.

A significant portion of the OpenMP 4+ and OpenACC standards are interchangeable and can be directly substituted by using a pattern-matching approach or simple *sed* script. Table 14 gives an example of many of these analogous directives, clauses, and API calls. However, there are several situations and constructs that require more than direct substitution.

3.3.1 OpenMP 4+ to OpenACC. Of the two primary translations, OpenMP 4+ to OpenACC is the more straightforward direction. In a traditional view, OpenMP is a prescriptive set of directives in which users explicitly define the intended parallelism, variable scoping, and so on. Thus, most of the information necessary for translating to OpenACC is user-provided and requires no additional analysis. However, the prescriptive nature of OpenMP is shifting with the introduction of new OpenMP 5 features, as discussed in Chapter I, Section 1.2.

A key exception is the OpenMP critical region. By design, OpenACC does not contain an analogous directive for creating critical regions or regions to be executed in a mutually exclusive manner. Because OpenACC's initial intended use and primary current use involve offloading code to GPU accelerators, the standard designers intentionally omitted a directive for creating mutually exclusive code

Table 14. Examples of straightforward directive translations implemented in CCAMP.

	OpenACC	OpenMP 4+
Data directives	#pragma acc data	#pragma omp target data
	#pragma acc data enter	#pragma omp target enter data
	#pragma acc data exit	#pragma omp target exit data
Data clauses	create	alloc
	copyin	to
	copyout	from
	copy	tofrom
	present	assert(omp_target_is_present())
Parallel directives	#pragma acc parallel loop *#pragma acc kernels loop*	#pragma omp target teams
Parallel clauses	gang	distribute distribute parallel for
	worker	parallel for
	vector	simd parallel for simd
Parallel size clauses	num_gangs	num_teams
	num_workers	num_threads
	vector_length	simdlen
Other clauses	collapse	collapse
	if	if
	private	private
	reduction	reduction
API calls	acc_set_device_num	omp_set_default_device
	acc_get_device_num	omp_get_default_device
	acc_on_device	!omp_is_initial_device
	acc_malloc	omp_target_alloc
	acc_free	omp_target_free

regions since these types of approaches typically perform very poorly on GPUs. Instead, the designers encouraged algorithm writers to rethink their designs.

However, one special case of OpenMP critical regions can be appropriately translated by CCAMP. Critical regions in OpenMP are commonly used to express array reductions. By using OpenARC's auto-reduction analysis, CCAMP can determine whether a critical region is used for an array reduction and instead generate appropriate OpenACC reduction statements. For other non-reduction instances of OpenMP critical regions, CCAMP cannot translate the code and reports this to the user.

Another incompatibility when translating OpenMP 4+ involves the recently introduced OpenMP tasking directives. CCAMP currently serializes these directives, which is semantically correct but inefficient. The authors aim to address this in future works.

3.3.2 OpenACC to OpenMP 4+. Translating OpenACC into OpenMP 4+ represents a greater challenge than the converse direction. Unlike OpenMP, OpenACC at its core is a descriptive set of directives. OpenACC programmers can often elect to shift the burden of mapping parallelism to hardware to the underlying compiler. As a result, perfectly valid OpenACC programs might omit a significant amount of information that would typically be required in an analogous OpenMP program. Translation between the two standards requires an analysis of loop constructs, available parallelism, vectorization considerations, variable scoping and memory access, and other information typically omitted in OpenACC to generate the necessary information.

For example, in Table 14, OpenACC "gang" and "worker" are equated with OpenMP 4+ "teams distribute" and "parallel for," respectively. However,

OpenACC programmers can omit these directives, often without consequence. Omission in OpenMP would result in a serial execution. Therefore, before translating, CCAMP attempts to supply any missing parallelization clauses (e.g., gang, worker, and vector) by using a combination of the provided OpenACC directives and OpenARC's auto-parallelization pass [141]. Analyzed loops are marked as *independent* or *sequential* and then annotated with an appropriate parallelization clause before being translated into OpenMP 4+. This is very similar to the process performed in the initial stage of the CCAMP Optimization pass, as described in Section 3.4.

Besides generating necessary clauses, other issues resulting from OpenACC's descriptive nature must be addressed. In OpenACC, reduction statements for shared variables in a nested loop can be placed on the nested loop or the outermost loop. However, OpenMP requires the reduction clause to be placed with the *teams* directive. Therefore, CCAMP migrates any reduction clauses before the final translation. Similarly, clauses in OpenACC specifying thread counts and work group sizes (num_gangs, num_workers, vector_length) are typically placed with the initial parallel directive, whereas the OpenMP analogs (num_teams, num_threads, simdlen) are required to reside with the nested parallelism directives. Again, the CCAMP translation pass automatically migrates these clauses before direct translation.

3.4 CCAMP: Automated Optimization of OpenMP and OpenACC

Section 3.3 focuses on the legality of translation and adhering to the standard. However, generating translated code that satisfies the corresponding standard does not guarantee performance portability. Different compilers have varying levels of language support and implementation maturity, and the popular

compilers differ in their preferred mapping strategies between parallelism defined by the standard and parallelism available in target devices.

As a result, no single translation strategy achieves the best available performance in all device + compiler combinations. To address this, this section discusses a generalized optimization strategy, which was implemented as a compiler pass within the CCAMP framework. These optimizations can be applied in conjunction with CCAMP Translation or independently for applications that do not require translation. CCAMP's optimization strategy first employs a generalized parallelism identification pass, followed by a language- and device-specific optimization pass. Although psuedocode algorithms are provided for the optimization passes, several details and corner cases are omitted for brevity and readability.

3.4.1 Extracting Parallelism. Algorithm 1, which is implemented in the CCAMP framework, is applied regardless of the input language and target device. The CCAMP Optimization pass first identifies user-defined loop independence, parallelism, and vector status via OpenMP or OpenACC loop-related clauses, and then it appropriately marks loops by using internal notation. This internal notation is used in additional passes to reapply parallelism directives for specific target devices. Although the OpenMP and OpenACC standards do not strictly require loops annotated with parallelization clauses to be independent, this is typically the intention of programmers; thus, CCAMP provides an option to assume that these loops are sequentially independent. However, for programmers that require a more strict adherence to the standards, CCAMP also provides an alternative option that performs OpenARC analysis, even on loops annotated

by users with parallelism directives, and emits any inconsistencies as compiler warnings.

After marking user-annotated loops with internal annotation, CCAMP performs a second sweep to automatically categorize any unmarked loops not explicitly annotated by the user. Loop independence and viability for parallelism are analyzed by using OpenARC's auto-parallelization analysis pass. Viability for vectorization can also be automatically determined in many cases. However, because many underlying compilers are very conservative when applying vectorization and often ignore user-supplied vectorization clauses, CCAMP also conservatively marks loops for vectorization by using a vector-friendly analysis with the following criteria. The loop: (1) is either parallelizable or vectorizable from a strictly theoretical sense without breaking program semantics, (2) has compile-time constant loop bounds, and (3) does not have control flow divergence, irregular array accesses, function calls, or inner loops, which might not disqualify a loop from being strictly parallelizable or vectorizable but could have significant performance disadvantages. Although most compilers ignore superfluous vectorization directives, these directives can inhibit opportunities for aggressive loop collapsing. CCAMP reports loops marked by compiler analysis to the user, providing the user with an opportunity to overwrite the compiler's behavior by manually applying additional directives.

The final loop in Algorithm 1 performs a loop nesting analysis, automatically determining which loops are tightly nested and suitable for loop collapsing. For a pair of collapsible loops, only the inner loop is marked with the internal notation, which allows the collapse clause to percolate up through parent loops as the internal notation is consumed.

Algorithm 1 Extract Parallelism and Tightly-Nested Loops

function ARC_ANALYSIS(Loop L)
  Perform loop auto-parallelization analysis and mark L
   with `arc_loop_para` if parallelizable
  Perform loop vector-friendly analysis and mark L
   with `arc_loop_vectfrd` if vector-friendly

for loops L in OpenMP regions
  if L annotated with *teams distribute* **or** *parallel for*
    Mark L with `arc_loop_para`
  if L annotated with *simd*
    Mark L with `arc_loop_vect`

for loops L in OpenACC regions
  if L annotated with *loop independent, gang, or worker*
    Mark L with `arc_loop_para`
  if L annotated with *loop vector*
    Mark L with `arc_loop_vect`

for loops L in OpenMP/OpenACC regions
  Call ARC_ANALYSIS(L)
  if L tightly nested in enclosing loop
    Mark L with `arc_loop_tnest`

3.4.2 OpenMP Mapping on CPUs. Because of the large disparity in thread count and clock speeds between CPUs and GPUs, OpenMP directives must be configured differently for the two devices to optimize performance. CCAMP's CPU-specific OpenMP Optimization, outlined in Algorithm 2, first focuses on applying SIMD parallelism. CPU compilers have mature vector parallelization facilities, and exploiting SIMD parallelism significantly affects CPU performance.

Specifically, CCAMP annotates the innermost loop marked as vector-friendly (by OpenARC vector-friendly analysis or by the user) with an OpenMP SIMD directive. CCAMP then prioritizes loop collapsing, which is generally beneficial in the evaluated benchmarks. Finally, CCAMP applies a single parallelization directive, *#pragma openmp teams distribute parallel for*, to the outermost loop marked as parallelizable. Although this single directive could be separated and applied to the loops in a nested fashion, because of the coarse granularity and lower core count of CPUs, the performance on evaluated applications was typically higher with the conjoined directive.

3.4.3 OpenMP Mapping on GPUs. Instead of prioritizing SIMD use, CCAMP's OpenMP GPU Optimization focuses on maximizing thread counts through loop collapsing and nested parallelism. In the evaluated applications, LLVM clang and IBM XLC, two OpenMP compilers with offloading support, largely ignored SIMD clauses when targeting GPUs. Therefore, in Algorithm 3, CCAMP first collapses all tightly nested loops based on the analysis in the parallelism extraction phase. CCAMP then applies parallelism directives at the two outermost tightly nested parellelizable loops: *#pragma omp teams distribute* at the outermost nested loops and *#pragma omp parallel for* at the second

Algorithm 2 CCAMP OpenMP CPU Optimization

```
for each OpenMP loop nest N
    for each loop L (innermost to outermost)
        if L marked with arc_loop_vectfrd
            Annotate L with #pragma omp simd
            Remove arc_loop_tnest mark
            Break
    for each loop L (outermost to innermost)
        if L marked with arc_loop_para
            Let n be the nesting level of tightly-nested
                parallel loops with arc_loop_para
                starting from L
            Annotate L with #pragma omp teams
                distribute parallel for
            Annotate L with collapse(n) if n > 1
            Break
```

outermost nested loops. Unlike the CPU case in which these clauses were applied in a single conjoined directive, when targeting GPUs, the additional parallelism from the nested approach leads to higher GPU utilization and performance. As shown in Section 3.5.4.1, unlike clang and XLC, GCC SIMD clauses are critical when targeting the GPU. In future iterations of CCAMP, this behavior must be incorporated or compiler-specific variants must be created.

3.4.4 OpenACC Mapping. Although the OpenMP optimizations required separate mappings for GPUs and CPUs, the same mapping is employed for both devices with OpenACC. A single set of generalized directives seem to perform well across devices with the same source code.

In contrast to the OpenMP SIMD clause, the OpenACC vector clause is recognized by the PGI compiler on GPU and CPU devices. However, due to looser restrictions in OpenACC, it might not map threads directly to vector units. Because of this, CCAMP's OpenACC Optimization, outlined in Algorithm 4, first applies a vector directive to the innermost vector-friendly loop, if present.

Algorithm 3 CCAMP OpenMP GPU Optimization

```
for each OpenMP loop nest N
    for each loop L (outermost to innermost)
        if L marked with arc_loop_para
            Let n be the nesting level of tightly-nested
                parallel loops with arc_loop_para
                starting from L
            Annotate L with collapse(n) if n > 1
            Break
for each tightly-nested loops M with arc_loop_para
    if M is the outermost nested loop
        Annotate M with #pragma omp teams distribute
        if M has immediate inner nested parallel loops
                with arc_loop_para, K
            Annotate K with #pragma omp parallel for
        else Annotate M with #pragma omp parallel for
```

CCAMP then collapses all nested parallelizable loops. Finally, all loops marked as parallel are annotated with a *#pragma acc loop independent* directive. Although additional clauses (i.e., *gang*, *worker*) could be appended for specificity, this did not significantly affect performance with PGI in the evaluated benchmarks.

Algorithm 4 CCAMP OpenACC CPU and GPU Optimization

```
for each OpenACC loop nest N
    for each loop L (innermost to outermost)
        if L marked with arc_loop_vectfrd
            Annotate L with #pragma acc loop vector
            Remove arc_loop_tnest mark
            Break
    for each loop L (outermost to innermost)
        if L marked with arc_loop_para
            Annotate L with #pragma acc loop independent
            Let n be the nesting level of tightly-nested parallel loops with
arc_loop_para starting from L
            Annotate L with collapse(n) if n > 1
            Break
```

3.4.5 Optimization Code Examples. Listing 3.1 demonstrates CCAMP's optimization algorithms executed on a simple example application. This application, along with a similarly coded Matmul application, is the basis of the fundamental kernel used in the evaluation described in Section 3.5.1.3, although many details are omitted, such as data movement directives and variable initialization.

Listing 3.1 contains four versions of the OpenMP Jacobi application: (1) unmodified input program, (2) code with internal annotations after applying parallelism extraction, (3) CPU-optimized code after applying Algorithm 2, and (4) GPU-optimized code after applying Algorithm 3.

Lines 14–29 of Listing 3.1 show how CCAMP's parallelism extraction internally annotates loops by using user-supplied directives, auto-parallelizaion and vector-friendliness analyses, and loop-nesting analysis. For the CPU optimization (lines 31–42), CCAMP prioritizes SIMD parallelism, and for the GPU optimization (lines 44–53), CCAMP prioritizes loop collapsing.

Listing 3.2 contains three versions of the OpenACC Matmul application: unmodified input, code with internal annotations, and optimized code after applying Algorithm 4.

Lines 12-24 show the resulting internal annotations after CCAMP extracts parallelism and tightly nested loops. Lines 26-36 show the resulting parallelism mapping after optimizing with Algorithm 4. The loop lines 6-7 fails to meet the criteria for CCAMP's vector-friendly analysis, and so is annotated with a sequential clause. When targeting the CPU with the PGI compiler, inclusion or exclusion of a vectorization directive for this loop does not affect performance. However, annotating the loop with an OpenACC *vector* directive when targeting GPU

Listing 3.1 Naive OpenMP Jacobi CCAMP Optimization

```
1  // Naive OpenMP Jacobi
2  #pragma omp target teams distribute
3  for (i = 1; i <= SIZE; i++)
4    #pragma omp parallel for
5    for (j = 1; j <= SIZE; j++)
6      a[i][j] = (b[i − 1][j] + b[i + 1][j] + b[i][j − 1] + b[i][j + 1]) / 4.0f;
7
8  #pragma omp target teams distribute
9  for (i = 1; i <= SIZE; i++)
10   #pragma omp parallel for
11   for (j = 1; j <= SIZE; j++)
12     b[i][j] = a[i][j];
13
14 // CCAMP Parallelism Extraction and Loop Nesting
15 #pragma arc_loop_para
16 for (i = 1; i <= SIZE; i++)
17   #pragma arc_loop_para
18   #pragma arc_loop_vectfrd
19   #pragma arc_loop_tnest
20   for (j = 1; j <= SIZE; j++)
21     a[i][j] = (b[i − 1][j] + b[i + 1][j] + b[i][j − 1] + b[i][j + 1]) / 4.0f;
22
23 #pragma arc_loop_para
24 for (i = 1; i <= SIZE; i++)
25   #pragma arc_loop_para
26   #pragma arc_loop_vectfrd
27   #pragma arc_loop_tnest
28   for (j = 1; j <= SIZE; j++)
29     b[i][j] = a[i][j];
30
31 // CCAMP CPU Optimization
32 #pragma omp teams distribute parallel for
33 for (i = 1; i <= SIZE; i++)
34   #pragma omp simd
35   for (j = 1; j <= SIZE; j++)
36     a[i][j] = (b[i − 1][j] + b[i + 1][j] + b[i][j − 1] + b[i][j + 1]) / 4.0f;
37
38 #pragma omp teams distribute parallel for
39 for (i = 1; i <= SIZE; i++)
40   #pragma omp simd
41   for (j = 1; j <= SIZE; j++)
42     b[i][j] = a[i][j];
43
44 // CCAMP GPU Optimization
45 #pragma omp teams distribute parallel for collapse(2)
46 for (i = 1; i <= SIZE; i++)
47   for (j = 1; j <= SIZE; j++)
48     a[i][j] = (b[i − 1][j] + b[i + 1][j] + b[i][j − 1] + b[i][j + 1]) / 4.0f;
49
50 #pragma omp teams distribute parallel for collapse(2)
51 for (i = 1; i <= SIZE; i++)
52   for (j = 1; j <= SIZE; j++)
53     b[i][j] = a[i][j];
```

devices significantly degrades performance. Essentially, the PGI compiler maps the vectorization clause to thread-level parallelism. Omitting the vector clause in this

Listing 3.2 Naive OpenACC Matmul Optimization

```
1  // Naive OpenACC Matmul
2  #pragma acc parallel loop
3  for (i=0; i<M; i++) {
4    for (j=0; j<N; j++) {
5      float sum = 0.0F;
6      for (k=0; k<P; k++)
7        sum += b[i*P+k]*c[k*N+j];
8      a[i*N+j] = sum ;
9    }
10   }
11
12 // CCAMP Parallelism Extraction and Loop Nesting
13 #pragma arc_loop_para
14 for (i=0; i<M; i++) {
15   #pragma arc_loop_para
16   #pragma arc_loop_tnest
17   for (j=0; j<N; j++) {
18     float sum = 0.0F;
19     #pragma arc_loop_vect
20     for (k=0; k<P; k++)
21       sum += b[i*P+k]*c[k*N+j];
22     a[i*N+j] = sum ;
23   }
24   }
25
26 // CCAMP Optimization
27 #pragma acc loop collapse(2)
28 for (i=0; i<M; i++) {
29   for (j=0; j<N; j++) {
30     float sum = 0.0F;
31     #pragma acc loop seq
32     for (k=0; k<P; k++)
33       sum += b[i*P+k]*c[k*N+j];
34     a[i*N+j] = sum ;
35   }
36 }
```

instance allows the compiler to map thread-level parallelism to the parent collapsed loops, leading to higher overall utilization.

These code examples represent simple applications in which these optimizations can be easily applied manually. However, large code bases (e.g., several of the applications evaluated in this work) can contain hundreds of directives and far more complicated loop interactions and relationships, making manual optimization application tedious and error-prone. This project addresses this issue with automatic applications of the aforementioned algorithms implemented in the CCAMP framework.

3.5 Evaluation of CCAMP Framework

3.5.1 Experimental Setup of Intel, IBM, and Nvidia Platforms.

3.5.1.1 Devices. CCAMP was evaluated by using four multithreaded devices: two multicore CPUs and two manycore GPUs. These four devices are contained within two separate nodes. The first is an Intel-based cluster node with attached Nvidia GPUs (P100). The second is an IBM-based node with an attached Nvidia GPU (V100), modeled after the nodes of the Summit supercomputer.

Intel Cluster:

- Xeon CPU: Intel(R) Xeon(R) CPU E5-2683 v4 @
- P100 GPU: Nvidia Tesla P100-PCIE-12GB (Pascal)

Summit Node:

- Power9: IBM POWER9, altivec supported, 176 CPUs, 4 threads per core, 22 cores per socket, 2 sockets
- V100: Nvidia Tesla V100 SXM2 16GB (Volta)

Each device is typically coupled with a vendor-supplied compiler that strongly prefers either OpenACC or OpenMP 4+. This preference is one of the main motivations for finding a fluid way to translate between the standards.

3.5.1.2 Compilers. Along with the devices mentioned, three compiler frameworks were used in the evaluation: two specific to device vendors (i.e., PGI, XLC) and one open-source solution (i.e., clang). The PGI compiler recognizes both OpenACC and OpenMP 4+ directives, although it does not yet support offloading for OpenMP 4+, only host execution. Currently, we use the latest-released community edition of the compiler, PGI 19.4-0 (LLVM 64-bit for the Intel

node and Linuxpower target for the IBM node). Each compilation includes the following flags: "-V19.4 -Mllvm -fast -acc -mp -Mnouniform." A device-specific flag for each target architecture was also included: "-ta=multicore" for the Xeon CPU and Power9, "-ta=tesla:cc60" for the P100, and "-ta=tesla:cc70" for the V100.

The IBM XLC compiler only recognizes OpenMP 4+ directives, although it does support offloading. This compiler is only available on the IBM node (Power9/V100 devices), whereas the clang and PGI compilers are available on both evaluated nodes. We specifically use IBM XL C/C++ for Linux, V16.1.1 (Community Edition), the most recently released version. The flags "-O3 -qarch=pwr9 -qsmp=omp -qnooffload" are used on the Power9 and "-O3 -qsmp=omp -qoffload" are used on the V100.

The LLVM project, including clang, is an open-source project that is not tied to a specific vendor. As a result, clang is easily installed on both evaluated nodes and supports all the evaluated devices. However, the current version of clang still supports only the OpenMP 4+ directives, not OpenACC. LLVM version 9.0.0 (git tag llvmorg-9.0.0-rc6) is used. For each device, the "-fopenmp" flag is used, and "-fopenmp-targets" is set to a device-triple flag specific to each architecture: "x86_64-unknown-linux-gnu" for the Xeon CPU, "nvptx64-nvidia-cuda" for the V100 and P100, and "ppc64le-unknown-linux-gnu" for the Power9. We also include "–cuda-gpu-arch=sm_60" and "–cuda-gpu-arch=sm_70" for the P100 and V100, respectively.

Finally, an initial evaluation is included by using the GNU GCC compiler, version 10.1, installed via spack with nvptx-none support for OpenACC and OpenMP. Details on specific compiler versions and optimization flags used across different devices are found in the accompanying artifact description. Applications

are built with GCC using the "-foffload=-lm" flag, and "-fopenacc" and "-fopenmp" for OpenACC and OpenMP, respectively.

3.5.1.3 Benchmarks. The SPEC Accel Benchmark suite, first introdcued in Chapter I, Section 1.4 was used to evaluate CCAMP primarily because SPEC Accel is one of the few benchmark suites with both OpenACC and OpenMP 4+ versions of several benchmarks. This was required to directly evaluate the performance of CCAMP-translated codes against hand-coded applications. The authors evaluated the OpenMP and OpenACC SPEC Accel benchmarks written in C because CCAMP does not support Fortran.

- *X03 ostencil* (303 for OpenACC and 503 for OpenMP), a thermodynamics stencil kernel (also referred to as *os*).
- *X14 omriq*, a convolution-based Hessian multiplication.
- *X52 ep*, an embarrassingly parallel application.
- *X54 cg*, a conjugate gradient kernel.
- *X57 csp*, a scalar penta-diagonal solver.
- *X70 bt*, a block tridiagonal solver for 3D PDEs.

Within the 503.ostencil, 557.sp, and 570.bt benchmarks, two different sets of OpenMP 4+ directives are included, distinguished by the SPEC_INNER_SIMD macro. These are treated as separate benchmarks in the evaluations and are denoted by an asterisk suffix (X03*, X57*, X70*).

CCAMP was also evaluated by using standard Jacobi and Matmul kernels adopted from implementations available in the OpenARC repository[39]. The

Jacobi kernel was parallelized by using a "naive" OpenMP 4+ approach, and the Matmul kernel was parallelized by using a "naive" OpenACC approach. The goal was to recreate what a beginner OpenACC or OpenMP user might find as a reasonable implementation after reviewing the standards and introductory documentation.

For the naive Jacobi kernel shown in Listing 3.1, OpenMP 4+'s two levels of parallelism were applied in a nested way because there was a pair of nested loops. An even simpler approach was taken for the naive Matmul kernel in Listing 3.2; only an OpenACC parallelism directive was applied to the outermost loop. Although this seems overly naive, a "less is more" approach often results in good performance for OpenACC applications due to the descriptive nature of OpenACC and the maturity of the PGI OpenACC compiler.

Finally, with the goal of targeting a more realistic application, the authors include an evaluation that uses the LULESH 2.0 proxy application [171]. LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) is a widely studied application related to codesign efforts for exascale computing. Because LULESH is written in C++, it must be ported to C for evaluation with CCAMP. However, because LULESH 2.0 contained few C++ constructs, the port was relatively straightforward.

To express the scope and size of the different SPEC benchmarks and LULESH, Table 15 lists several different attributes of each SPEC benchmark. Using *grep*, *wc*, and manual observations, the authors recorded the total number of C-code lines ("Lines of C"), OpenACC ("ACC") and OpenMP directives ("OMP"), compute kernels or parallel regions ("Kernels"), compute kernels with nested loops ("Nests"), and the number of vector-friendly clauses added for semi-automated

Table 15. SPEC Accel Benchmark Attributes

	Lines of C	ACC	OMP	Kernels	Nests	VF
X03	1,245	13	6	1	1	1
X14	1,179	4	4	2	1	0
X52	957	11	23	5	3	0
X54	1,457	34	43	24	2	1
X57	3,586	78	285	66	65	0
X70	7,773	61	126	43	43	0
LULESH	8,594	95		31	2	0

compilation ("VF"). Only kernels, nests, and vector-friendly clauses were recorded once because their numbers are consistent between the OpenACC and OpenMP versions.

3.5.2 Evaluation of CCAMP Translation. As mentioned previously, the SPEC benchmark suite contains OpenMP 4+ and OpenACC versions of the same applications, which were leveraged to develop and evaluate CCAMP's translation facility. After translating a SPEC OpenACC application to OpenMP 4+, the new OpenMP code and run time performance can be directly compared with the corresponding SPEC OpenMP 4+ application, provided that input data, iteration count, and other application-specific inputs are carefully accounted for. Similarly, the SPEC OpenACC applications can be used to evaluate the OpenMP 4+ to OpenACC translation.

Figure 22 shows the results of an evaluation performed by using the clang and PGI compilers on Intel Xeon and Nvidia P100 devices. First, manually coded OpenMP 4+ applications unmodified from the SPEC benchmark suite were compared with the code automatically generated by CCAMP Translation, which originated from the corresponding SPEC OpenACC application. The performance evaluation of the OpenACC to OpenMP 4+ translation is shown in the top row of Fig. 22. The bottom row represents the performance evaluation of CCAMP's

OpenMP 4+ to OpenACC translation and compares manually coded OpenACC and automatically generated OpenACC.

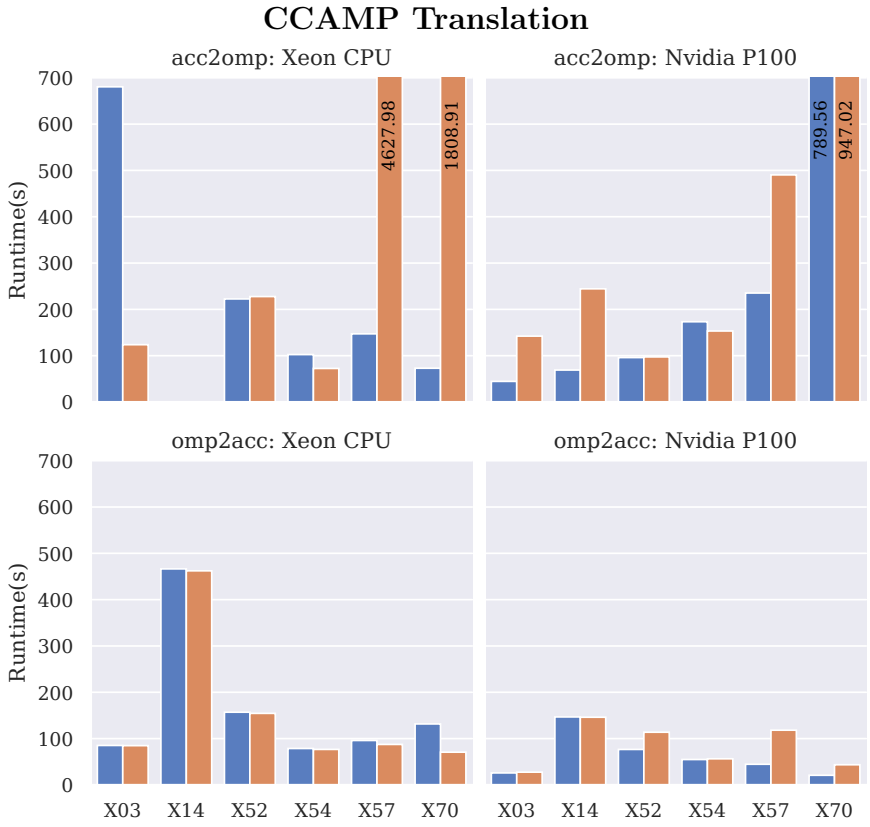


Figure 22. Run time performance comparison of manually coded applications (blue) and CCAMP-translated applications (orange) using CCAMP's OpenACC to OpenMP 4+ (acc2omp) and OpenMP 4+ to OpenACC (omp2acc) translation.

As shown in Fig. 22, no single mapping or translation can consistently provide high performance across different architectures, especially for the more prescriptive OpenMP 4+ standard. This further motivates the need for the device-specific optimization presented in this section.

However, there could be instances in which users want to apply CCAMP Translation but not CCAMP Optimization. One current limitation of the CCAMP Optimization passes is their omission of clauses that specify thread and work group sizes: "num_teams, num_threads, simdlen" for OpenMP 4+ and "num_gangs,

num_workers, vector_length" for OpenACC. If an application's directives are already optimized for a specific device in one standard, then translation alone might be enough to generate optimized code in the counterpart standard. The CCAMP Optimization passes generate code specific to a class of devices (i.e., GPU-friendly or CPU-friendly) but do not generate code for a specific device model. As mentioned in Section 3.7, one future goal of CCAMP is to improve device-specific optimization by scraping system information on thread limits and core counts.

3.5.3 Evaluation of CCAMP Optimization. CCAMP's Optimization passes are independently evaluated for different device+compiler combinations. The results from three experiments with OpenMP 4+ (i.e., Clang, PGI, XLC) and one experiment with OpenACC (i.e., PGI), are shown using the aforementioned SPEC benchmarks.

3.5.3.1 OpenMP 4+ Optimization with Clang. In Fig. 23, there is a stark difference in performance for *X03* on the CPU and for *X03** on the GPUs. The SPEC benchmark developers noticed these distinctions and implemented two different parallelization strategies, likely intending users to choose *X03* on GPU-like devices and *X03** on CPU-like devices. However, these differences immediately illuminate the advantages of using a framework such as CCAMP. CCAMP aims to optimize performance across all devices with the same source code, whereas the manually coded SPEC benchmarks require two separate source codes.

There are modest performance improvements for the *X52* and *X54* benchmarks on the GPU devices. The manually coded SEPC applications place all the parallelism clauses on the outermost loops, whereas CCAMP Optimization employs a nested parallelism approach when targeting GPUs.

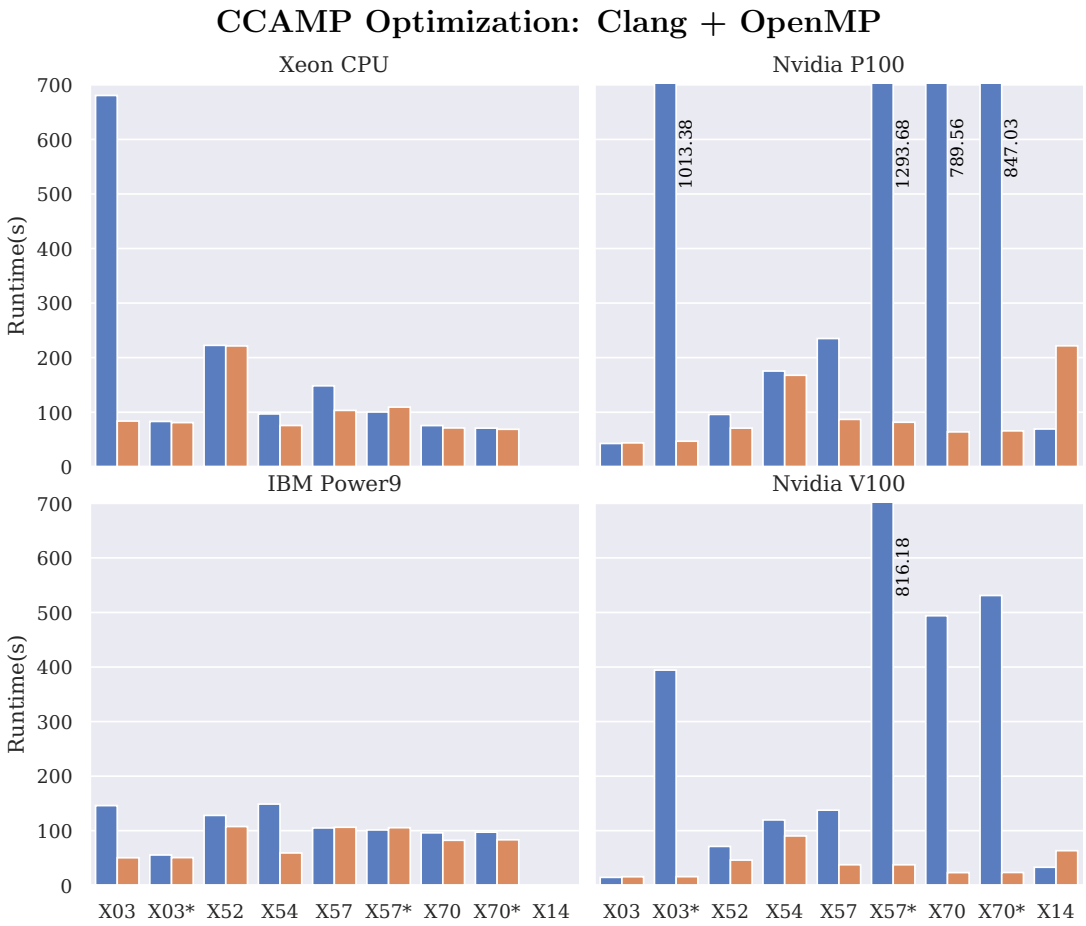


Figure 23. Clang + OpenMP. Run time comparison of SPEC hand-optimized (blue) and CCAMP automated optimization (orange). (Smaller is better.)

The large *X57*, *X57**, *X70*, and *X70** demonstrate modest performance improvements on the CPU devices and significant improvements on the GPU devices when comparing CCAMP-optimized versions with baseline unmodified versions. These applications highlight the advantages of CCAMP since manually modifying the hundreds of kernels across the applications would be extremely error prone and time consuming.

Finally, *X14* represents an outlier in which CCAMP Optimization either fails to complete (CPUs) or actually leads to a slight performance degradation (GPUs), indicating CCAMP still has room for improvement.

3.5.3.2 *OpenMP 4+ Optimization with PGI.*

When evaluating CCAMP's Optimization with OpenMP 4+ and the PGI compiler, the evaluation is restricted to CPU devices since PGI does not support OpenMP 4+ offloading. As with clang, significant performance improvements are seen when optimizing the *X03* and *X03** applications (Fig. 24). Compared with clang, PGI-compiled binaries typically result in lower overall run times. This could indicate that clang's OpenMP 4+ development is focused on GPU optimization, whereas PGI is still limited to CPU executions. This could also be an artifact of different optimization levels between the compilers: "-fast" for PGI and "-O3" for clang.

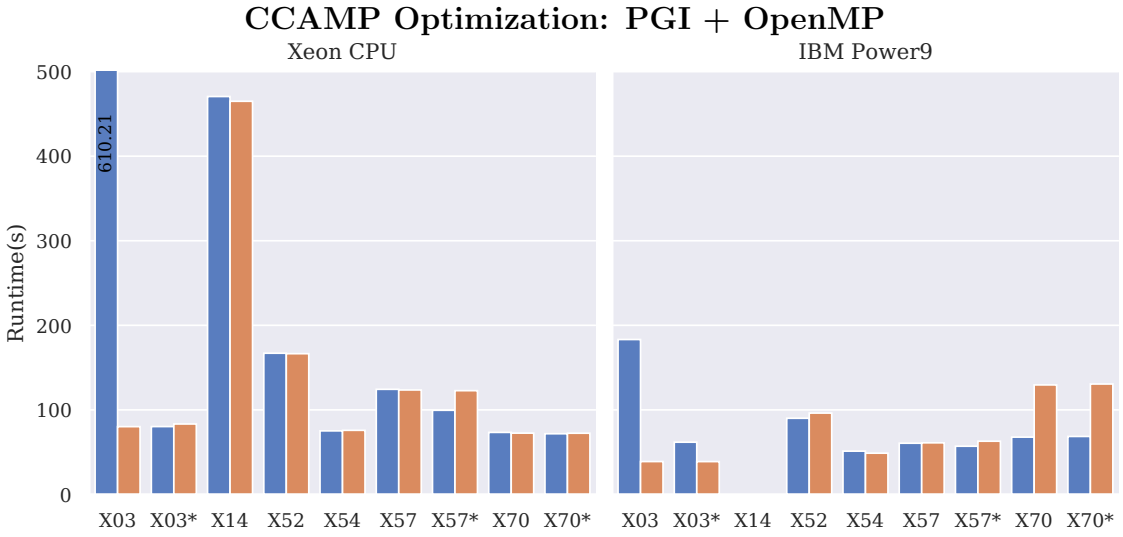


Figure 24. PGI + OpenMP. Run time comparison of SPEC hand-optimized (blue) and CCAMP automated optimization (orange). (Smaller is better.)

3.5.3.3 *OpenMP 4+ Optimization with XLC.*

When evaluating OpenMP with the IBM compiler, efforts are restricted to the IBM node devices, the Power9, and V100. Figure 25 shows a similar performance pattern with *X03* on the Power9 and *X03** on the V100. On the V100, CCAMP Optimization significantly outperforms the manual baselines for most applications.

XLC demonstrates more erratic behavior than the other compilers. For some applications, XLC significantly outperforms clang and PGI on the Power9 (*X52*) and the V100 (*X03*, *X54*) devices. This suggests that XLC could take advantage of IBM-specific architecture features. For other applications, such as *X54* on the Power9, XLC experiences relatively catastrophic performance execution (run time was extrapolated from three iterations). This most likely results from XLC's failure to vectorize OpenMP SIMD loops, even though the loops are identified as vectorizable by OpenARC analysis and the other compiler frameworks. Generally, across all compilers on CPU devices, recognizing and successfully vectorizing OpenMP SIMD loops significantly affects performance.

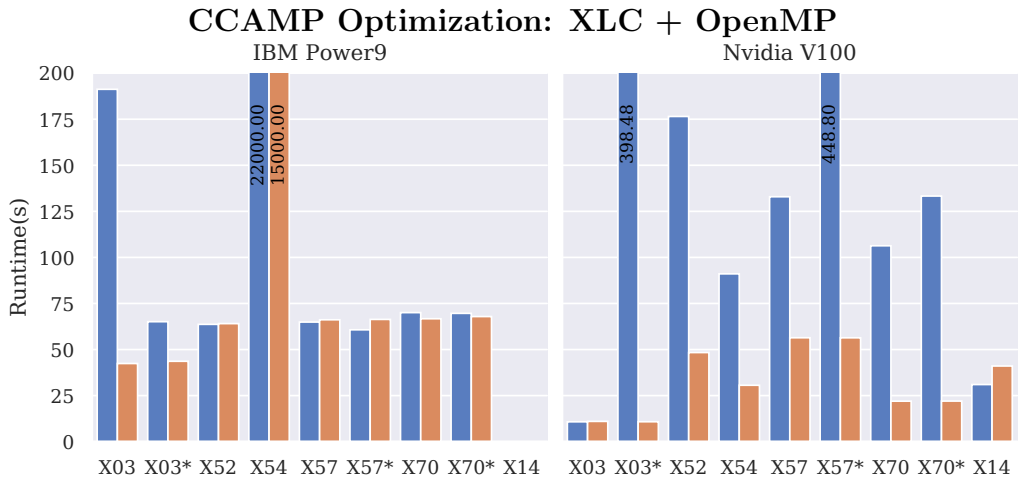


Figure 25. XLC + OpenMP. Run time comparison of SPEC hand-optimized (blue) and CCAMP automated optimization (orange). (Smaller is better.)

3.5.3.4 OpenACC Optimization with PGI.

Because the authors are still in the initial stages of GCC OpenACC evaluation, only the PGI compiler is used to evaluate CCAMP's OpenACC optimization on the SPEC Accel benchmarks (Fig. 26). For most applications and devices, there was, at most, a modest performance improvement over the manually coded applications. As shown in Section 3.5.4.3, OpenACC performance is typically much less sensitive to the

specific directive configuration, leaving fewer opportunities and a smaller need for optimization.

For most applications, the manual and CCAMP-optimized OpenACC versions perform similarly to the CCAMP-optimized OpenMP 4+ version. However, there are several exceptions. OpenMP 4+ *X03* and *X52* on Power9 + XLC significantly outperform the OpenACC Power9 + PGI counterpart. Conversely, the OpenACC *X54* implementation on P100 + PGI significantly outperforms the CCAMP-optimized OpenMP 4+ version compiled with clang. The SPEC developers mention that the OpenMP 4+ and OpenACC versions cannot always be directly compared for run time performance, which could result in the differences observed previously, although the authors did verify that the same input data and host code were used in both versions. Otherwise, this might be a motivation to translate between standards not just for portability but also for performance, especially if tools exist to automate the translation, such as CCAMP.

3.5.3.5 Putting it Together: CCAMP Translation and Optimization.

This section demonstrates how general programmers can use CCAMP's Translation and Optimization facilities in tandem to develop optimized and portable OpenACC and OpenMP 4+ applications. Table 16 shows the performance of the naive Jacobi OpenMP 4+ implementation, which is described in Section 4.3, across different device + compiler combinations and the resulting performance after applying CCAMP's Optimization and Translation passes.

Using language translation enables two new device + compiler combinations—PGI + P100 and PGI + V100—to be targeted by using OpenACC. Significant performance improvements are also seen across the different CPU devices since CCAMP avoids applying nested parallelism on CPU devices, which

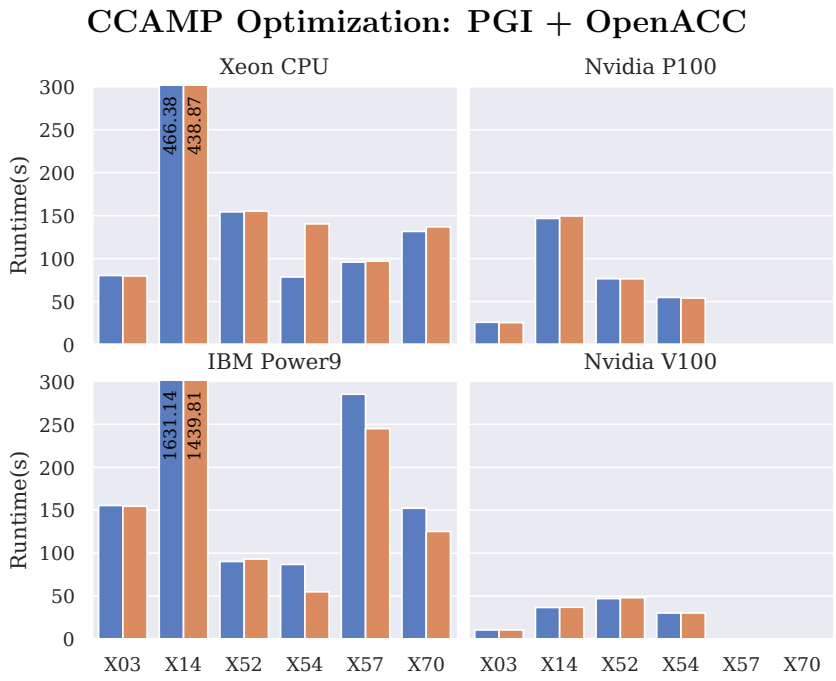


Figure 26. PGI + OpenACC. Run time comparison of SPEC hand-optimized (blue) and CCAMP automated optimization (orange). (Smaller is better.)

can lead to very poor performance. The naive OpenMP 4+ implementation could have been designed to be more CPU-friendly instead of GPU-friendly, which would have resulted in more significant CCAMP improvements for the GPU code instead of the CPU code.

Modest improvements are seen on the GPU devices primarily because the CCAMP Optimization applies loop collapsing. Table 16 also shows the performance of the naive Matmul OpenACC implementation. Modest performance improvements are seen for the device + PGI combinations, again primarily due to CCAMP's automated loop collapsing function. However, there is a significant increase in code portability when using CCAMP's translation mechanism, enabling significantly more device + compiler combinations.

Table 16. Naive Jacobi and Matmul OpenMP 4+ Run Times Optimized with CCAMP. Average of three executions. CCAMP Translation indicates OpenMP 4+ to OpenACC translation was applied.

	Naive Jacobi	Jacobi Tr.	Jacobi Opt.	Naive Matmul	Matmul Tr.	Matmul Opt.
pgi+xeon	13.523	-	3.707	26.22	-	24.979
pgi+P9	149.095	-	2.269	5.981	-	5.621
pgi+p100	-	Yes	2.339	15.440	-	1.817
pgi+v100	-	Yes	0.931	2.514	-	0.678
clang+xeon	13.558	-	4.210	-	Yes	7.195
clang+P9	25.727	-	2.740	-	Yes	25.169
clang+p100	1.646	-	1.448	-	Yes	1.103
clang+v100	1.008	-	0.578	-	Yes	0.406
xl+P9	2.178	-	2.186	-	Yes	7.169
xl+v100	0.863	-	0.601	-	Yes	0.463

The translated and optimized OpenMP 4+ versions compiled with clang outperform the optimized OpenACC version on the V100 and P100 devices. This could indicate that CCAMP's OpenACC optimization needs further improvement, or it could indicate that clang is better able to optimize this specific application over the "more mature" PGI compiler, motivating the need for code to be portable between compilers.

3.5.4 Additional CCAMP Evaluations.

3.5.4.1 GCC: Initial Evaluation. An initial performance evaluation was performed by using the GNU GCC compiler. Although the support could be immature relative to PGI and clang, GCC is unique in that it supports offloading in both standards.

Figure 27 highlights performance comparisons between manual and CCAMP-optimized SPEC applications, although *X57* and *X70* are excluded due to compilation failures.

In OpenMP, reasonable performance is achieved for the manual versions and modest speedups when applying CCAMP Optimization. Interestingly, unlike clang

and XLC, GCC recognizes and even depends on SIMD clauses when offloading to GPUs. Because of this, CCAMP's CPU-specific optimizations were applied to achieve the reported performance numbers. This suggests that CCAMP might need to implement compiler-specific optimizations or further generalize the existing GPU and CPU-specific optimizations to incorporate GCC moving forward.

Besides *X03*, which performs well, the OpenACC evaluations performed very poorly or failed to compile. This could be due to SPEC's frequent use of and GCC's relative lack of support for the OpenACC *kernels* directive, although further investigation is needed.

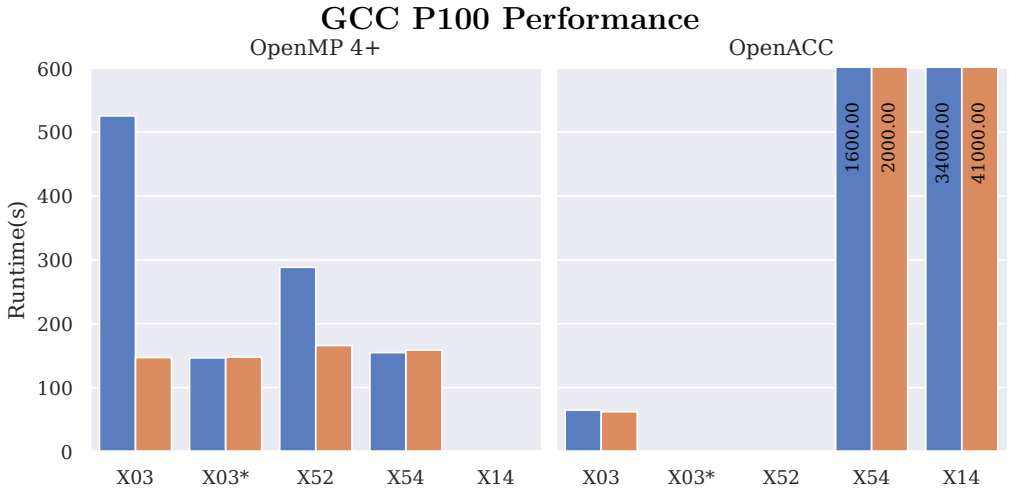


Figure 27. (Left) Performance of OpenACC manual (blue) and CCAMP optimized (orange). (Right) Performance of OpenMP manual (blue) and CCAMP optimized (orange).

3.5.4.2 LULESH 2.0. To assess CCAMP's performance on a more realistic application, an evaluation was performed by using the OpenACC LULESH 2.0 application on the Xeon CPU and Nvidia P100 GPU. For the OpenACC evaluations, the authors evaluated with the manual unmodifed code (blue bars) and after applying CCAMP's OpenACC Optimization. For the OpenMP evaluations,

the CCAMP-translated code (blue bars) and CCAMP-translated and optimized code (orange bars) were evaluated.

Figure 28 shows the results of the LULESH performance evaluations. The extremely high run time was immediately noted when targeting the CPU with PGI (extrapolated from one iteration), resulting from a vector clause placed on the outermost loop of one kernel. It is assumed that the OpenACC LULESH implementation was not intended to be run on the CPU. The CCAMP-optimized version performs significantly better. Also, relatively strong performance was achieved with the OpenMP translated versions, and clang-compiled versions achieved a faster run time than their OpenACC + PGI counterparts on the CPU and competitive performance on the GPU. Finally, poor performance was seen across the board with GCC, which required small modifications to successfully compile, as described in the artifact description. This could be a reflection of the level of support in GCC and the need for more compatibility between CCAMP and GCC.

3.5.4.3 Performance Variability.

This experiment highlights the performance variability for different directive configurations across the two standards: OpenMP 4+ and OpenACC. This was quantified by using the SPEC Accel *X03* benchmark, which comprises a single kernel with a triply nested loop, allowing for a large degree of variation in directive placement.

Figure 29 compares different versions of *X03* OpenMP that were compiled by using the clang compiler for the Xeon CPU (top left) and Nvidia P100 (top right) with different versions of *X03* OpenACC compiled by using the PGI compiler targeting the Xeon CPU (bottom left) and Nvidia P100 (bottom right).

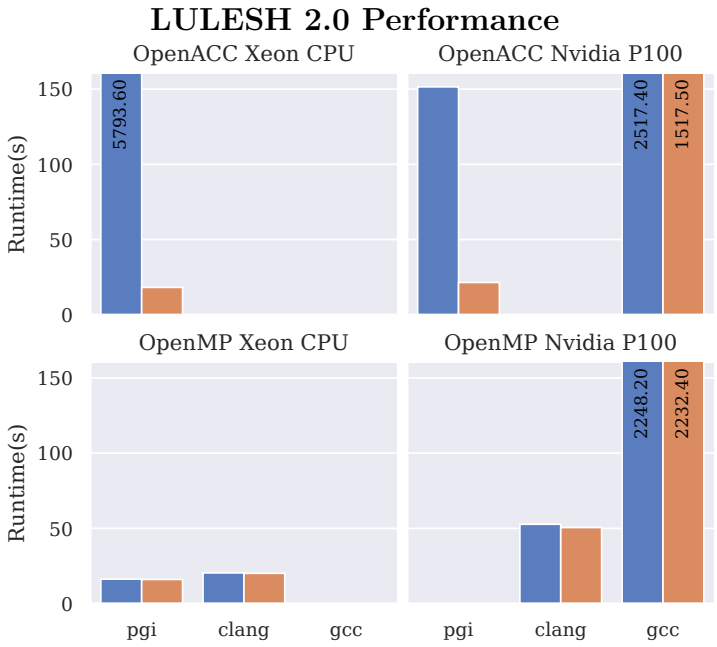


Figure 28. (Top) Performance of OpenACC manual (blue) and CCAMP optimized (orange). (Bottom) Performance of OpenMP translated (blue) and OpenMP translated + optimized (orange).

The OpenMP 4+ version numbers (VX) refer to directive placements as follows, in which *(outer)* refers to the outermost loop, *(middle)* refers to the first nested loop, and *(inner)* refers to the innermost nested loop. The abbreviated "parfor" represents the OpenMP "parallel for" clause, and "coll" represents the "collapse" clause.

- V0: (outer) teams distribute parfor coll(3) SIMD.
- V1: (outer) teams distribute parfor coll(2) (inner) SIMD.
- V2: (outer) teams distribute parfor coll(2) .
- V3: (outer) teams distribute (middle) parfor coll(2).
- V4: (outer) teams distribute (middle) parfor (inner) SIMD.

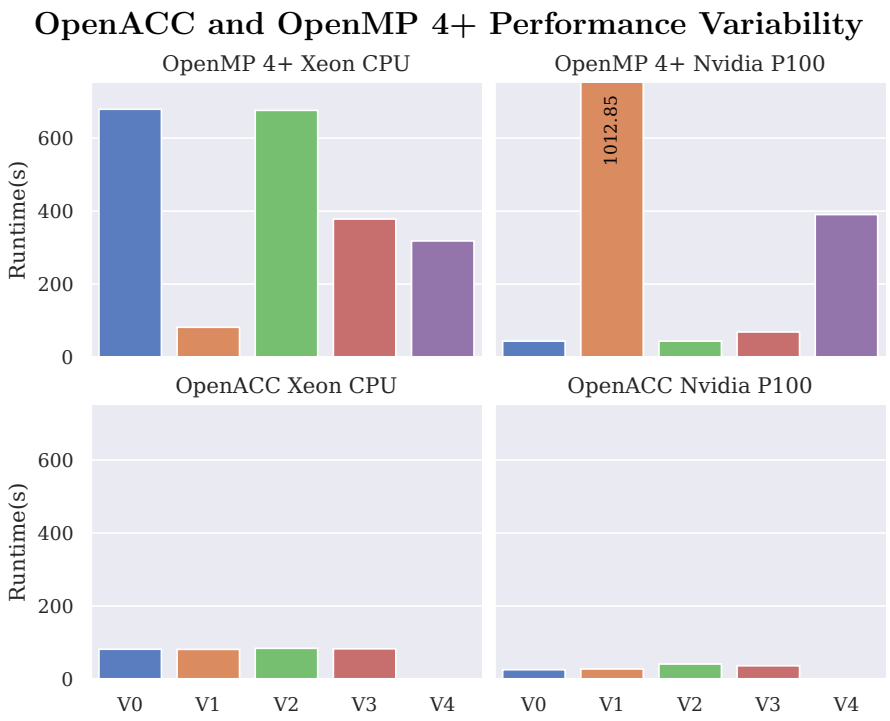


Figure 29. Comparison of performance variability with different sets of directives between OpenMP (top) and OpenACC (bottom) by using the SPEC X03 (ostencil) Benchmark.

The OpenACC version numbers refer to the following directive placements:

- V0: (outer) parallel gang coll(2) (inner) vector.
- V1: (outer) parallel gang worker vector coll(3).
- V2: (outer) parallel gang (middle) worker vector coll(2).
- V3: (outer) parallel gang (middle) worker (inner) vector.

The OpenMP 4+ versions demonstrate a much higher performance variance than the OpenACC versions: OpenMP 4+ and Xeon CPU $\sigma = 52,000$, OpenMP 4+ and P100 $\sigma = 14,000$, OpenACC and Xeon CPU $\sigma = 1.4$, and OpenACC and P100 $\sigma = 40$.

These performance differences might not be direct artifacts of the standards themselves but of the relative maturity of the underlying compilers. The PGI OpenACC compiler actually predates the OpenACC standard since PGI adopted its previous directive-based parallel compiler to handle OpenACC directives. By comparison, clang's OpenMP 4+ offload support is relatively new and is undergoing active development.

However, these results also are not entirely surprising given the fundamental differences in standard design; OpenACC is more descriptive, and OpenMP is more prescriptive. The OpenACC compiler retains more liberty to optimize as it sees fit, independent of exactly how a user specifies directives. Conversely, the OpenMP 4+ compiler is more strictly bound by the standard to adhere to the directives specified by the user, even when ill-advised.

With the more novel features introduced in OpenMP 5, the standard has shifted to also allow a more descriptive approach, similar to the abstraction level of OpenACC. For example, the OpenMP *loop* directive is a highly descriptive

directive that outsources much of the management to the underlying compiler. Additionally, the OpenMP *metadirectives* mirror some of the functionalities and goals present in CCAMP, although in a less automated way because they require programmers to manually provide alternative options for each device type per compute region. As compilers work to implement these features, a shift in how the results vary between the two standards might occur, and the mapping strategies within CCAMP might need to be adjusted accordingly.

3.6 OpenMP and OpenACC Interoperable Framework: Related Work

Several previous works explore the performance and portability of directive-based approaches across heterogeneous systems. Vergara et al. [185] evaluates OpenMP applications on IBM Power8 and Nvidia Tesla devices by using IBM and LLVM clang compilers. Lopez et al. [186] experiments with OpenACC and OpenMP implementations of core computational kernels, including Daxpy, Dgemv, Jacobi, and HACCmk. Lopez et al. evaluates the performance of these implementations by using Cray, Intel, and PGI compilers on Nvidia GPU and Intel Xeon Phi devices. Gayatri et al. [187] implements a material science kernel and evaluates OpenMP 3.0, OpenMP 4.0, OpenACC, and CUDA implementations on Xeon CPUs, Xeon Phis, Nvidia P100s, and Nvidia V100s. Gayatri et al. also discusses experiences with different compilers—including PGI, Intel, IBM, and GCC compilers—and the then-current status of the work's directive-based language support. These works all highlight the high performance variability of directive-based approaches across different compiler and device combinations, which help motivate the utility of frameworks such as CCAMP.

Several previous works researched the potential of an OpenACC and OpenMP translation framework. Wolfe [188] explores this idea and discusses

some obvious and some subtler challenges that would arise if implementing such a framework. Wolfe also discusses motivations and the significance of developing such a framework, which are in line with the motivations presented here. Sultana et al.[189] presents a prototype OpenACC to OpenMP translation scheme, which consists of a combination of automated directive translation performed by using the Eclipse user interface and manual user-performed code restructuring. This work represents a promising first attempt to develop an automated translation framework, although it evaluates only a single benchmark and supports only a subset of the OpenACC standard. Pino et al. [190] describes a mapping between the most common directives of OpenACC and OpenMP and compares the performance between the two different sets of directives on several SHOC and (The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite) NAS Parallel (developed and maintained by the NASA Advanced Supercomputing (NAS) Division) benchmarks. However, Pino et al. does not propose any automated scheme or framework to perform the actual translation. Denny et al. [112] presents an ongoing work to develop an OpenACC to OpenMP 4.5 translator (Clacc) within the clang compiler to allow clang to support OpenACC. Clacc represents a rigorous effort to develop a translation scheme that supports the full OpenACC standard, which accomplishes the goal of the OpenACC to OpenMP 4.5 baseline translation. However, Clacc is constrained by the clang compiler, preventing it from using the maturity of device-specific back-end compilers.

3.7 OpenMP and OpenACC Interoperable Framework: Conclusions

As systems become more exotic and specialized, the HPC community has experienced an increased demand for high-level portable programming solutions. Although directive-based standards and approaches aim to provide a solution,

they fail to realize this goal due to competition between vendor compilers and inconsistent levels of standard support.

This chapter presents the CCAMP framework, with the goal of allowing programmers to seamlessly flow between different directive sets and eventually select the directive set best-suited to a target device. Automatic translation and optimization passes are introduced and shown to generate output code across different directive contexts that perform competitively with hand-coded programs.

Although some OpenACC and most OpenMP 4+ compilers are still somewhat immature, these tools are already being used to develop current and future high-performance systems. Additionally, support for directive-based offloading in the exisiting tools is constantly improving; for example, each new clang release significantly improves offloading support. CCAMP can help application developers port their applications with the OpenMP 4+/OpenACC compilers by enhancing their performance with the CCAMP optimizer. Moving forward, the authors hope that the CCAMP translation capabilities can be used to help migrate large code bases to future systems; for example, CCAMP can be used to help port large OpenACC applications developed for the current GPU-based systems (e.g., Summit at Oak Ridge National Laboratory) to future OpenMP-only systems (e.g., upcoming exascale systems, such as Aurora at Argonne National Laboratory).

In the future, we plan to develop and extend CCAMP in several ways. One primary goal is to develop more device-specific and algorithm-specific optimizations that can produce not only generalized directive sets in different languages but also directive sets specifically catered toward an indented target device. We also plan to improve the presentation of CCAMP's compilation output on loop

analysis and incorporate suggestions for performance to increase user-friendliness for programmers interested in a more semi-automated optimization approach. We plan to incorporate other compilers (e.g., Intel, Clacc) and other devices (e.g., FPGAs). We want to expand CCAMP to cover more of the OpenMP and OpenACC standards, especially the OpenMP 5-specific features, such as tasking, the loop directive, and metadirectives. As support for these features improves across compilers and applications begin to leverage them, it will be critical for CCAMP to incorporate these features in the translation and optimization passes.

CHAPTER IV

EXPLORING HETEROGENEOUS PROGRAMMING FOR FUTURE DIVERSE EXASCALE PLATFORMS

This chapter contains unpublished material with co-authorship. All of the presented research in this chapter was conducted as a collaboration between the University of Oregon and Oak Ridge National Laboratory. Seyong Lee was instrumental in the conceptualization of the projects, and provided continued support, suggestions, and advice throughout the projects with weekly meetings. Allen Malony and Jeffrey Vetter both provided high-level guidance and advice during this project. Allen Malony also wrote the introduction section. Sameer Shende provided guidance and support related to the TAU performance system, and Mohammad Alaul Haque Monil assisted with the other profiling tools and evaluations related to profiling. I performed all of the non-profiling experiments, collected all of the non-profiling data, and did the bulk of the writing besides the introduction for this project.

4.1 Exploration of Exascale Platforms: Introduction

In the last 10 years, there has been a steady transition in high-performance computing (HPC) from homogeneous systems, where node-level architectures utilizes general purpose processors (i.e., CPUs), towards heterogeneous systems, where different processor devices (e.g., CPU, GPU, FPGA) are used together. The tremendous computing power of manycore devices, exemplified by GPU SIMT architectures, could not be ignored in HPC platforms and heterogeneous systems are now the status quo for high-end supercomputing.

However, the potential for heterogeneous machines can only be realized if it is possible to program them. Herein lies the rub. Heterogeneous processors are

more complex to program because their architectures require different programming models, and the interaction between processing devices is critical to achieving performance. That interaction must be programmed as well. The standard parallel programming methods for homogeneous computing are insufficient for these purposes. The key challenge for heterogeneous programming integrating the parallel execution capabilities found in the heterogeneous processors under a unified programming umbrella. While research may be trending in this direction, the reality is that there is a variety of programming techniques covering sparsely a growing space of accelerator technologies. A productive near-term focus to address programming and performance portability could concentrate on compiler-based translation and coupling of parallel programming models.

In this chapter, we explore and evaluate the diversity of programming models likely to be featured in upcoming exascale machines. This chapter makes the following contributions.

- A survey of exascale platforms and discussion of experimental pre-exascale systems,
- An evaluation of the aforementioned pre-exascale systems, using a single source code and a source-to-source translator to efficiently target each platform

4.2 Exascale Platforms and Programming Models

Before exploring the upcoming exascale programming models, we should first quickly discuss the upcoming exascale machines themselves. By definition, an exascale machine is a machine capable of executing 10^{18} floating point operations per second. Although the 1 exaflop cutoff is somewhat arbitrary, the exascale

designation represents a class of next-generation machines that will have significant impacts and contributions to science as a whole.

Three major exascale systems have been announced, with delivery dates as early as late 2021. All three machines are currently being developed by the US Department of Energy and US National Nuclear and Security Administration.

- **ANL Aurora:** Developed by Cray, Intel, and Argonne National Lab, the ANL Auora machine [4] is slated to release in 2021. Aurora will contain over 9000 nodes, each containing two Intel Xeon "Sapphire Rapids" CPUs and six Intel Xe "Ponte Vecchio" GPUs (the HPC counterpart to the Intel Xe Max evaluated in this work). The per-node performance is expected to be 130 double-precision TFLOPs, which puts the performance of the entire functioning machine at just over 1 exaflop.

- **ORNL Frontier:** Developed by Cray, AMD, and Oak Ridge National Lab, the ORNL Frontier machine [3] is also slated to release in 2021. Each Frontier Node will contain one HPC and AI oriented AMD EPYC CPU and four "purpose-built" AMD Radeon Instinct GPUs, likely to be similar to the Radeon Instinct GPUs evaluated in this project. The entire Frontier system is expected to achieve over 1.5 exaflops.

- **LLNL El Capitan:** Also developed by Cray and AMD, Lawerence Livermore National Lab's El Capitan machine [5] is scheduled to deploy in 2023. This machine will likely feature similar hardware to Frontier, albeit upgraded, and is expected to achieve over 2 double-precision exaflops.

Other exascale machines are certain to follow, likely candidates including China's Tianhe-3 machine, a machine developed by the European High-Performance Computing Joint Undertaking, and many others.

4.2.1 Exascale Programming Models. Although the proposed machines above are undeniably significant feats of human engineering, the actual utility of these systems is dependent on the availability, performance, maintainability, and robustness of associated programming models and software stacks. In this section we survey several programming models likely to be featured during the early days of exascale machines. These are also precisely the languages used in this project's evaluations. Although these models are discussed in Chapter I, Section 1.2, we briefly reintroduce them here in the context of this chapter.

4.2.1.1 OpenMP. The OpenMP programming standard [170] unquestionably has the most illustrious past of models explored in this project. Although OpenMP began as a multi-core shared-memory CPU programming standard, with the introduction of offloading directives in versions 4.0 and later OpenMP has evolved to encompass heterogeneous GPU-based computing. As a directive-based standard, application programmers can annotate an existing C, C++, or FORTRAN application with parallelization directives without major changes to the underlying source code. Additionally, because of the prevalence of OpenMP in HPC ecosystems, OpenMP support is present in nearly all compilers evaluated in this project, although the degree of support varies between implementations, especially in terms of performance. Nevertheless, OpenMP is certain to be a primary target on all upcoming exascale systems.

4.2.1.2 *OpenACC.*

Before the release of OpenMP offloading directives, OpenACC stood alone as the sole directive-based standard for heterogeneous computing. OpenACC was originally constructed as a high-level alternative to lower level heterogeneous programming approaches like CUDA and OpenCL (discussed in subsequent sections), and offered an approach palatable to programmers accustomed to directive-based CPU parallelization approaches like OpenMP. Although OpenMP offloading directives were introduced shortly after the first installation of the OpenACC standard, OpenACC has remained relevant in HPC largely due to the availability of the mature production-level compiler developed and managed first by the Portland Group as the PGI compiler [43], and now by Nvidia as part of the NVHPC Toolkit [105]. Furthermore, several applications, including LULESH [171], the evaluated SPEC Accel benchmarks, and several large DOE applications are written in OpenACC, and as a result OpenACC is very likely to be featured on upcoming exascale systems alongside OpenMP.

4.2.1.3 *CUDA.*

The Nvidia CUDA programming model [8] has been immensely successful since its inception in 2007. CUDA can be considered a low-level programming model, as it requires specific knowledge of GPU devices, and significant rewriting of the most computationally intensive portions of an application. Despite 1) CUDA's low-level nature, which can be a significant barrier for scientific application developers interested in heterogeneous computing, and 2) the development of numerous high-level alternatives including directive-based standards like OpenMP and OpenACC, many programmers still choose to program directly using CUDA. CUDA's success can be partially attributed to the robustness of the proprietary CUDA software stack, including compilers, debugging and profiling tools, and professional training.

Although Nvidia GPUs and CUDA are currently absent in the ecosystems of announced exascale machines, several of the current leading supercomputers, including ORNL's Summit [1] and LLNL's Sierra [2] are built using these hardware and software stacks. It is no stretch then to assume that future machines, including future exascale machines, will likely feature the CUDA programming model.

4.2.1.4 OpenCL. While the proprietary, and well-funded, nature of CUDA certainly attributed to its success, it also limited the CUDA's heterogeneous landscape to Nvidia-developed devices. Shortly after the release of CUDA, an open-source alternative was introduced. OpenCL [37], first managed by Apple and now by Khronos, shares a similar low-level nature with CUDA, but is intended to run on any device with a sufficient OpenCL implementation. While OpenCL's adoption and uptake has not experienced the same degree of success as CUDA, the cross-platform and portable potential of OpenCL has made it an attractive option for upcoming exascale systems, both as a front-end programming model and as a backend intermediate representation for higher level models like SYCL, as discussed below.

4.2.1.5 HIP/ROCm. Nvidia's main competitor in the GPU market, traditionally in the consumer market but more recently also in the high-performance and scientific community, is AMD. Unlike Nvidia, AMD has not developed a proprietary programming approach and vendor compiler for heterogeneous computing. Instead, to support its GPU architectures, AMD has developed the open-source ROCm (Radeon Open Compute) suite [41]. ROCm is a collection of APIs, drivers, and development tools that support heterogeneous execution on AMD GPUs, but also other architectures like Nvidia GPUs. The actual programming model developed as part of ROCm is HIP, another low-

level approach with a similar abstraction level to CUDA and OpenCL. However, the ROCm toolkit and associated compilers also support OpenMP and OpenCL applications. The compilers, libraries, and debuggers for ROCm are available from the open source github [42].

Table 17 lists the programming models and associated implementations, where available, used as part of this project.

4.2.1.6 Other Notable Models.

Of course programming models likely to be featured on future exascale systems are not limited to the above list. Especially in the distant future, new or existing programming models are likely to be adopted on exascale systems to meet the demands of new applications. In the near-future, some programming models likely to be featured at exascale include general languages like SYCL [55], Kokkos [47], and Raja [52] and domain specific approaches (DSLs) like Tensorflow [7] and Keras [91]. SYCL, and by extension DPC++, is already a staple of the Intel OneAPI [38] programming ecosystem, and therefore intended to be a prominent approach on the Intel-based Aurora machine. While SYCL was originally developed as an OpenCL abstraction layer, more recently it has been promoted as stand-alone product that can target other backends besides OpenCL. Kokkos and Raja are high-level alternative to the low-level approaches like OpenCL and CUDA, and attractive options for scientific programmers. Finally, with the ubiquity of deep learning, DSLs like Tensorflow and Keras implementations and support will be necessary for exascale systems. Although these models will certainly be relevant in Exascale systems, they are omitted from this project to limit the scope of the study. However, we do suggest exploration of some these models during Section 4.6's discussion of future works.

Table 17. Exascale Programming Models and Implementations Explored

Nvidia A100

CUDA	nvcc (CUDA Toolkit)
HIP	nvcc_detail header (ROCm) + nvcc (CUDA Toolkit)
OpenCL	nvcc (CUDA Toolkit) clang (LLVM)
OpenMP	nvc (Nvidia HPC Toolkit) clang (LLVM)
OpenACC	nvc (Nvidia HPC Toolkit)

AMD Instinct

CUDA	hipify-perl (ROCm) + hipcc (ROCm)
HIP	hipcc (ROCm)
OpenCL	hipcc (ROCm)
OpenMP	hipcc (ROCm)
OpenACC	NA[2]

Intel Xe

CUDA	NA
HIP	NA
OpenCL	icpx (Intel OneAPI)
OpenMP	icpx (Intel OneAPI)
OpenACC	NA

[1]Partial support available through the ECP Clacc project, a fork of LLVM, but not evaluated in this project

4.3 Exploration of Exascale Platforms: Experimental Setup

In this section we detail the specific compilers, compiler versions, software platforms, and hardware platforms targeted in this project. We also briefly discuss the benchmarks used in the following evaluations.

4.3.1 AMD Platform. The specific AMD GPU evaluated in this work is a Radeon Instinct MI50 Accelerator (gfx906). Officially released in November 2018, the MI50 is based on the AMD Vega 20 architecture. This 7nm device advertises a peak throughput of 13.3 single-precision TFLOPs (FP32) and 6.6 double-precision TFLOPs (FP64).

The host processor attached to this accelerator is an AMD EPYC 7402 24-Core processor, although all host code in the evaluations is executed using a single thread. The platform operating system is CentOS Linux 8.

As previously mentioned, OpenACC is used as the front-end programming model in all of this project's evaluations. At this time, no major compiler fully supports OpenACC compilation for AMD GPUs, although partial support is being developed as part of the ECP clacc project [112]. However, using OpenARC source-to-source translation, the input OpenACC applications can be used to generate code that is supported by major AMD GPU compilers, including OpenMP, OpenCL, HIP, and CUDA.

For the backend compilation of these supported languages (after lowering from OpenACC), we rely on a system-installed ROCm 3.9.0. For the OpenMP, OpenCL, and HIP backends, we use *hipcc* (HIP version: 3.9.20412-6d111f85). The ROCm *hipcc* utility is a compiler driver that internally invokes either AMD's HCC compiler or the AMD branch of the LLVM *clang* compiler, in this case built on top of LLVM version 12.0.0. Relevant flags to *hipcc* for OpenMP compilation include

"-target x86_64-pc-linux-gpu -fopenmp -fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx906".

For the CUDA backend, we first run *hipify-perl* (included in ROCm 3.9.0) to translate the CUDA code into an analogous HIP code, and then run *hipcc* similar to the other backends. Although ROCm also provides a more robust translation tool, *hipify-clang*, the *hipify-perl* tool successfully translated all the applications evaluated in this project, while the LLVM-based tool encountered several errors.

4.3.2 Nvidia Platform. In this project we evaluate the Nvidia Ampere (A100-PCIE-40GB) GPU, code-named GA100, which was officially released in September 2020. This 7nm micro-architecture claims a peak throughput of 19.5 single-precision TFLOPs (FP32) and 9.7 double-precision TFLOPs (FP64). The host processor attached to this accelerator is also an AMD EPYC processor, identical to the one described in the AMD GPU Platform section above.

Unlike the AMD environment, Nvidia devices do have major compiler support for OpenACC. Previously known as the PGI compiler, the re-branded Nvidia High Performance Computing SDK (NVHPC) supports OpenACC compilation through its associated compiler, *nvc*. When targeting OpenACC as the backend programming model, for consistency when comparing with other programming models we still run the input OpenACC application through OpenARC source-to-source translation. Although both the input source and output source are OpenACC, OpenARC does apply a series of compiler passes that can result in small changes in the output code. For this project, we use NVHPC version 20.11, installed via spack [191] with package "nvhpc@20.11". We also use this specific NVHPC and associated *nvc* to evaluate NVHPC's OpenMP support in Section 4.4.5.

To compile OpenARC-generated CUDA and OpenCL for the A100 device, we use the NVIDIA CUDA Toolkit directly, in this case version 11.0.194, also installed via spack. This installation contains both CUDA and OpenCL headers and runtime libraries. To execute HIP applications on the A100, we use *hipcc* and the *nvcc_detail* header files, included in ROCm 3.9.0. These header files effectively redefine HIP API calls as thin wrappers over CUDA API calls, and *hipcc* subsequently calls a present CUDA installation internally, in this case the same version 11.0.194. When compiling CUDA, we include the "-03" and "-lcuda" flags. When compiling OpenCL, notable flags include "-03", "-lstdc++", and "-lOpenCL".

We also compile OpenARC-generated OpenCL and OpenMP using LLVM *clang*, directly built from source (llvmorg-11.1.0-rc2). In Section 4.4.5 we compare the LLVM-based open-source implementations of OpenMP and OpenCL against the Nvidia-owned NVHPC SDK and CUDA Toolkit implementations.

4.3.3 Intel Platform. Finally, we evaluate the Intel Iris Xe Max GPU (0x4905), also known as a DG1 card, which launched in quarter four of 2020. The Xe Max is built using 10nm semiconductor technology, and claims a peak throughput of 2.46 single-precision TFLOPs. Unlike the AMD and Nvidia cards evaluated, this Intel GPU is not HPC oriented, and instead is intended to ship with smaller portable laptops. However, the programming models and other software artifacts relevant to the Xe Max are very likely to also be relevant in upcoming Intel GPU releases, including those likely to be featured in the upcoming Aurora supercomputer [4]. The host processor on the Intel Platform is a 56-core Intel Xeon CPU E5-2680 v4 @ 2.40GHz.

Because of the novelty of Intel GPUs as a whole, the programming model and compiler availability is relatively limited when compared to options for AMD and Nvidia devices. In this project, we evaluate only two OpenARC-generated backend programming models for the Intel GPU Platform: OpenMP and OpenCL. Both programming models are compiled using the proprietary Intel compiler *icpx*, available as part of the Intel oneAPI Toolkit. Specifically we target a system-installed Intel oneAPI DPC++ Compiler 2021.1.2 (2020.10.0.1214).

4.3.4 Benchmarks. The SPEC Accel Benchmarks are ideal candidates for this project's evaluation because they 1) contain C-based OpenACC implementations of several applications, 2) represent applications from several different scientific domains, and 3) are professionally maintained and updated by The Standard Performance Evaluation Corporation (SPEC). In this project, we specifically use SPEC Accel v1.3. From the selection of benchmarks available in SPEC Accel, our evaluations in this chapter focused on the following benchmarks (the same applications targeted in Chapter III):

- *303 ostencil*, a thermodynamics stencil kernel (also referred to as *os*).
- *314 omriq*, a convolution-based Hessian multiplication.
- *352 ep*, an embarrassingly parallel application.
- *354 cg*, a conjugate gradient kernel.
- *357 csp*, a scalar penta-diagonal solver.
- *370 bt*, a block tridiagonal solver for 3D PDEs.

4.4 Evaluation of Heterogeneous Platforms with OpenACC, OpenARC, and CCAMP

As previously mentioned, OpenARC and the SPEC Accel benchmarks create an ideal sandbox in which we can explore and evaluate a diversity of different exascale-bound programming models. In each of the following evaluations, we have used OpenARC source-to-source translation to translate a single set of OpenACC SPEC applications in to a variety of different backends. We then evaluate these backends, exploring different aspects of performance, and highlighting interesting patterns and discrepancies.

4.4.1 Relative Performance of Each Programming Model Across Devices.

Given an application written using a specific programming model, it is reasonable to ask which hardware platform would be most appropriate to target. That is, which hardware platforms are more likely to approach peak performance for a given model. Similarly, we may ask which hardware platforms have mature implementations of a specific programming model. For vendor-specific programming models like HIP and CUDA, the obvious choice would be the corresponding hardware platform developed by the vendor. For other portable models like OpenMP, OpenACC, and OpenCL, the situation is less clear.

In Figure 30, we compare the relative performance of programming models (subplots) for each device (bar colors). Although we could directly compare the absolute runtime of each application on each device, this comparison would be inherently biased because the devices have different release dates, different semi-conductor fabrics, and different peak performances. Directly comparing runtimes may not accurately represent 1) the eventual suitability of programming models for each device family and 2) the maturity of the compiler implementations.

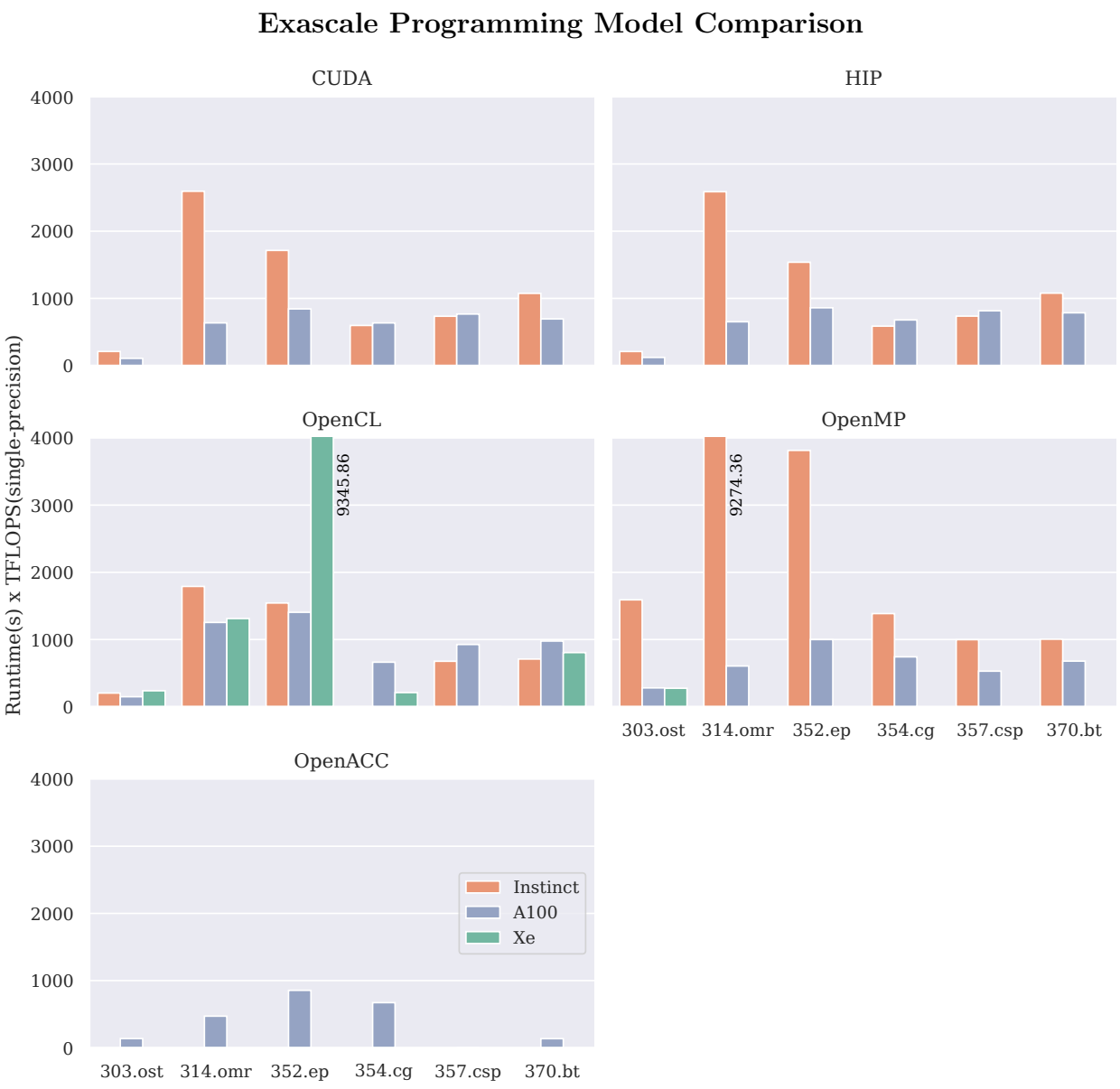


Figure 30. Relative runtime comparison (lower is better) of programming models (generated from OpenACC source code via OpenARC) across devices (distinguished by bar color). Relative runtime is estimated as absolute runtime (s) multiplied by theoretical peak performance (FP32 TFLOPS) for each device. Missing bars indicate an unsupported programming model or a failed compilation or execution.

Therefore in Figure 30 the y-axis is an estimation of relative performance, calculated by multiplying the runtime achieved for each specific benchmark on each platform by the theoretical peak performance reported for that platform (lower is better). As mentioned in the previous section, the reported peak performance in single-precision TFLOPs is 13.3 for the Instinct device, 19.5 for the A100, and 2.46 for the Xe Max. For example, the Xe GPU (Xe Max 0x4905), used in this evaluation is intended for low-power ecosystems and has limited double-precision support, while the Xe GPUs intended for Aurora will have more advanced FP64 support and fewer power constraints. Below, we briefly break down each subplot in Figure 30

CUDA Subplot: In Subplot 1, for the A100 device CUDA is compiled using *nvcc* (CUDA Toolkit), and for the Instinct device compilation is done using *hipify-perl* and *hipcc*. Unsurprisingly, the A100 achieves the best relative performance for the CUDA programming model for 4 of 6 benchmarks, although the AMD device achieves comparable performance for the remaining two applications (354.ct and 357.csp). Because no CUDA implementation exists for the Xe platform, those measurements are absent in this figure.

HIP Subplot: Interestingly, the A100 also achieves the best relative performance for 4 of 6 benchmarks when targeting the HIP backend (generated from OpenACC using OpenARC). Furthermore, the first two subplots (CUDA and HIP) are nearly identical. This is a testament to the success and efficiency of the *nvcc_detail* header file used to execute HIP applications on Nvidia hardware (orange bars in Subplot 2), and the *hipify-perl* tool used to execute the CUDA applications on the AMD hardware (blue bars in Subplot 1). Both *nvcc_detail* and *hipify-perl* are maintained by AMD and released as part of ROCm. Although a

single portable programming model is the ideal solution to create an application that can be run across several platforms, robust translation tools and compatibility libraries like those for CUDA and HIP can provide an alternative solution.

However, the library and translation solutions for portability also have downsides. Internal translation can make it more difficult for tools that have expectations about the runtime execution of an application. For example, in this project OpenARC had to be extended to support the *nvcc_detail* execution header because of the unexpected presence of CUDA constructs in a HIP execution context. Also, it may be more difficult for profiling tools to provide relate runtime information to the original source code.

OpenCL Subplot: In the third subplot, *nvcc* is used to compile OpenCL for the A100, *hipcc* is used for the Instinct device, and *icpx* is used for the Xe Max. The *hipcc* implementation targeting the Instinct device (blue bars) performed comparably to the other OpenCL implementations for most benchmarks, and actually achieved the lowest relative runtime for two applications, 357.csp and 370.bt. However, the *hipcc* driver failed to successfully execute the 354.cg benchmark, reporting a memory access fault.

The Intel implementation (green bars) also failed to successfully execute one of the SPEC Accel applications, 357.cp, and experienced unusually poor performance for the 352.ep application. However, for those applications that executed successfully, the relative performance of the Intel implementation on the Xe Max GPU was comparable to the AMD and Nvidia devices, and even achieved the lowest relative runtime in the case of 354.cg.

Unlike the Intel and Nvidia implementations, the Nvidia implementation (*nvcc*) successfully compiled and executed each OpenCL application for the A100

device. The relative runtime of the *nvcc* executions (orange bars) is comparable for most applications, and lowest across the evaluated devices for 303.ost, 314.omr, and 352.ep.

We do note that, although OpenCL may not be the most popular programming model, it does achieve the best coverage in terms of successful executions across all devices, covering 15/18 of the potential device and compiler combinations evaluated in for this figure.

OpenMP Subplot: In the fourth subplot, *hipcc* and *icpx* are again used for the Instinct and Xe Max devices. However, for OpenMP, we use the LLVM *clang* implementation when targeting the Nvidia 100. Although Nvidia also supports an OpenMP implementation as part of the NVHPC SDK, the LLVM-based implementation currently achieves more consistent performance (we explore this further in Section 4.4.5). To generate OpenMP from the input OpenACC codes, OpenARC's CCAMP translation was applied, including the optimization passes. The best performing mapping is evaluated in this figure. Performance differences between OpenMP mappings is explored further in Section 4.4.3.

For the single application where *icpx* successfully compiles and executes for the Xe Max, 303.ost, the relative runtime (blue bar) is very promising, significantly outperforming the relative runtime of the *hipcc* implementation on Instinct and performing comparably with the *clang* implementation on the A100. However, for the other 5 of 6 benchmarks, the OpenMP experiences runtime execution errors, most often segmentation faults, likely due to the relative immaturity of the OpenMP offloading features of *icpx*, 2) the lack of double-precision support, required for the 352.ep, 354.cg, 357.cp, and 370.bt benchmarks. In order to successfully compile the benchmarks, double-precision emulation was required,

enabled via the *OverrideDefaultFP64Settings* and *IGC_EnableDPEmulation* environment variables. Future releases of *icpx* and next generation Xe GPUs are likely to address these issues.

The *hipcc* implementation for the Instinct device (orange bars) performs relatively poorly compared to the LLVM implementation for the A100, even though when targeting OpenMP the *hipcc* compiler-driver internally relies on LLVM. This likely indicates that the LLVM OpenMP implementation has previously focused on offloading specifically to Nvidia devices, which not surprising given Nvidia's prevalence in contemporary systems, including the supercomputers Summit and Sierra. However, with the transition to AMD devices in many of the upcoming exascale machines, performance improvements in LLVM's, and indirectly *hipcc*'s, OpenMP implementations when targeting the AMD devices families will be essential.

OpenACC Subplot: In the final subplot, instead of investigating other programming models generated from OpenACC via OpenARC, we asses OpenACC directly. The only evaluated implementation that supports OpenACC compilation directly is *nvc* from the NVHPC SDK, and thus only bars for the A100 are visible. However, the Clacc Project [112], currently under development as part of the ECP project, promises to bring OpenACC support to LLVM, building off of LLVM's OpenMP implementation. Clacc will bring OpenACC support for AMD devices, likely performing similarly to the LLVM-based OpenMP implementations from the previous subplot.

4.4.2 Absolute Performance of Programming Models on Each Device.

Although the relative performance and runtime metrics used the previous section are useful for directly comparing performance between devices

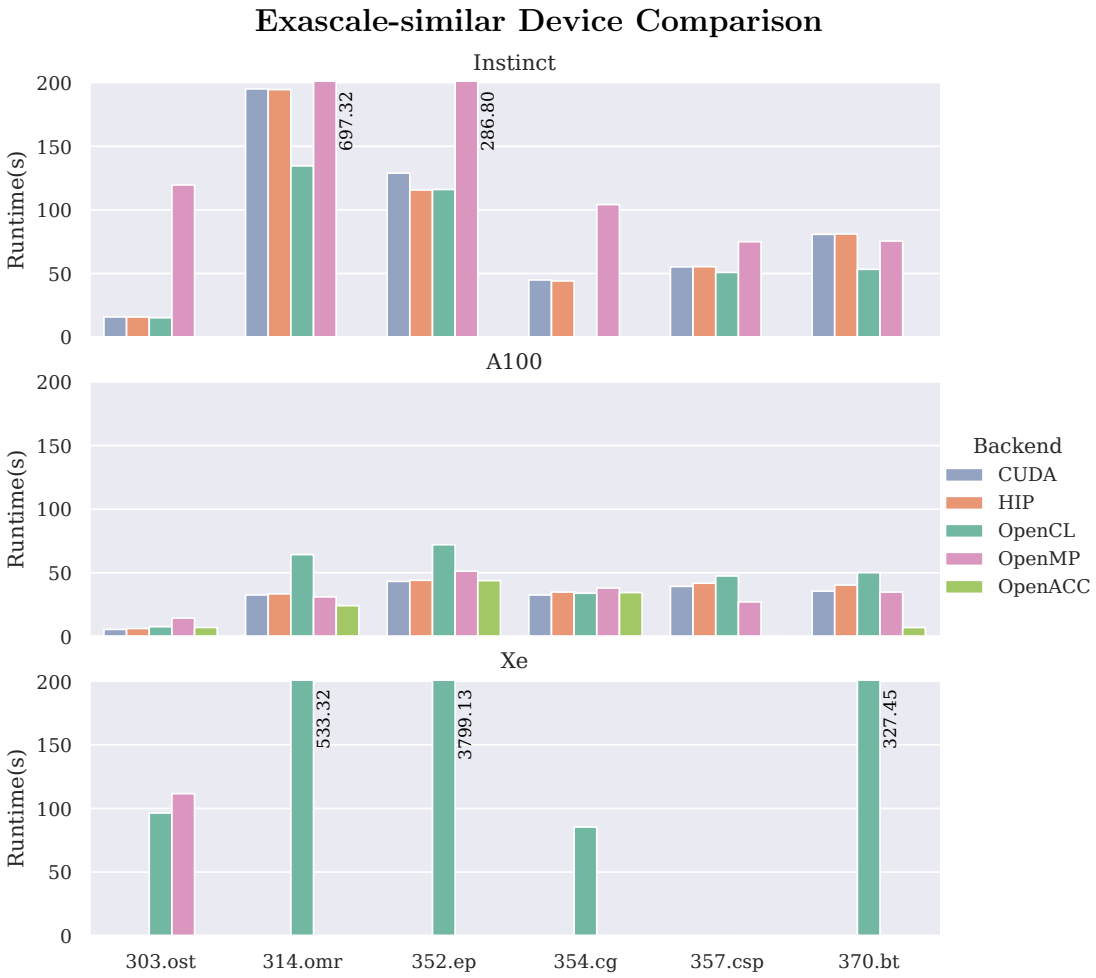


Figure 31. Absolute runtime performance comparison of different programming models (generated from OpenACC source code via OpenARC) on each device. Missing bars indicate an unsupported programming model or a failed compilation/execution.

and assessing relative maturity of implementations between devices, we are also interested in exploring the absolute performance of available implementations on a single device. That is, instead of starting with an application and choosing an appropriate device, we want to explore the hypothetical of starting with a device and choosing a programming model likely to perform optimally.

In Figure 31, we begin with the same data set used in the previous figure, but present the data in a new way (without normalizing by peak FLOPs) in order to explore different patterns and discrepancies. Again, absent bars represent either unsupported backends by the evaluated implementations or compilation or runtime failures that prevented collection of an accurate runtime. Runtimes that are reported were verified over multiple executions.

Instinct Subplot: In the first subplot, we evaluate the performance of all supported programming models on the AMD Instinct device. We first notice that the OpenCL model, compiled via *hipcc* (green bars) performs either comparably or significantly better than other programming models for all applications except 354.cg, where we experience a runtime error related to a memory access fault. As previously mentioned, CUDA and HIP, compiled with *hipcc* and *hipify-perl* perform similarly for each application. Although they lag slightly behind the OpenCL in terms of performance for some applications, the *hipcc* CUDA and HIP implementations are able to successfully compile and execute all applications. The OpenMP model, again compiled by *hipcc*, also successfully compiles every application. However, OpenMP's runtime performance is significantly slower than other the implementations for all applications except 370.bt, where it is surprisingly faster than the HIP and CUDA implementation but still slower than the OpenCL

implementation. As previously mentioned, OpenACC as a backend is not evaluated for the AMD device.

A100 Subplot: The first things we notice in the A100 subplot are the consistently low runtimes for all applications and programming models compared to the other subplots. Although the A100 is a newer device with a higher peak throughput, this also evidence of the relative maturity of programming model implementations when targeting Nvidia devices, a predictable outcome given Nvidia's dominance in high-performance heterogeneous computing over the last decade.

Interestingly OpenCL, compiled with *nvcc* (green bars), has the longest runtime of all programming models for 4 of 6 applications, contrasting significantly from the OpenCL implementation on the AMD platform.

It is probably safe to assume that a sufficiently hand-optimized CUDA implementation would likely outperform other programming models for all applications on the A100 device. However, the OpenARC-generated CUDA (blue bars), while still optimized via OpenARC compiler passes, results in the lowest runtime for only 3 of 6 applications. OpenACC, compiled via *nvc* (pink bars) claims that position for two other applications, and OpenMP compiled with *clang* actually achieves the lowest runtime for the 357.csp application. In general, *clang*-compiled OpenMP performance is consistent with the other programming models on the A100 device. Again, HIP executions, compiled using *hipcc* and the *nvcc_detail* header file (orange bars), perform nearly identically to CUDA.

Xe Max Subplot: The first thing we notice about the Xe Max subplot is that it is sparsely populated compared to the other subplots. Only the 303.ost application was successfully executed with the OpenMP programming model, and

neither the generated OpenMP or generated OpenCL codes, both compiled with *icpx*, successfully executed the 357.csp application. As previosuly mentioned, this lack of success is likely due to the relative immaturity of GPU compilation from the *icpx* implementation and the emulation of double-precision support. However, the relative inexperience of the authors with this platform is also likely a contributing factor. However, with the imminent release of Aurora, significant efforts are being made to develop support for the Xe family of devices, including support for the SYCL and DPC++ programming models not evaluated in this work.

4.4.3 OpenMP Mappings. In this section, we briefly explore the performance of the different OpenMP codes generated from the input OpenACC applications after applying OpenARC's CCAMP translation and optimization. As described in Section 1.3.3.5, the three different mappings of OpenMP directives generated include 1) literal translation from OpenACC directives with the intent to maintain the same level of parallelism and computation patterns as the original application ("default" in Figure 32), 2) optimization of the generated OpenMP directives specifically tailored for GPU devices ("gpu-friendly") and 3) optimization of the generated OpenMP directives tailored for CPU executions ("cpu-friendly").

In Figure 32, we see the runtime performance results from executing each mapping across each device and application. The Instinct executions (Subplot 1) were compiled with *hipcc*, which as previously mentioned relies on LLVM internally. The A100 executions were compiled with LLVM directly via *clang*. The Xe Max executions were compiled using *icpx*.

For all benchmarks, applying device-specific optimizations led to either comparable or improved performance over a more literal translation from OpenACC. For 5 of 6 benchmarks, the "gpu-friendly" translation performed either

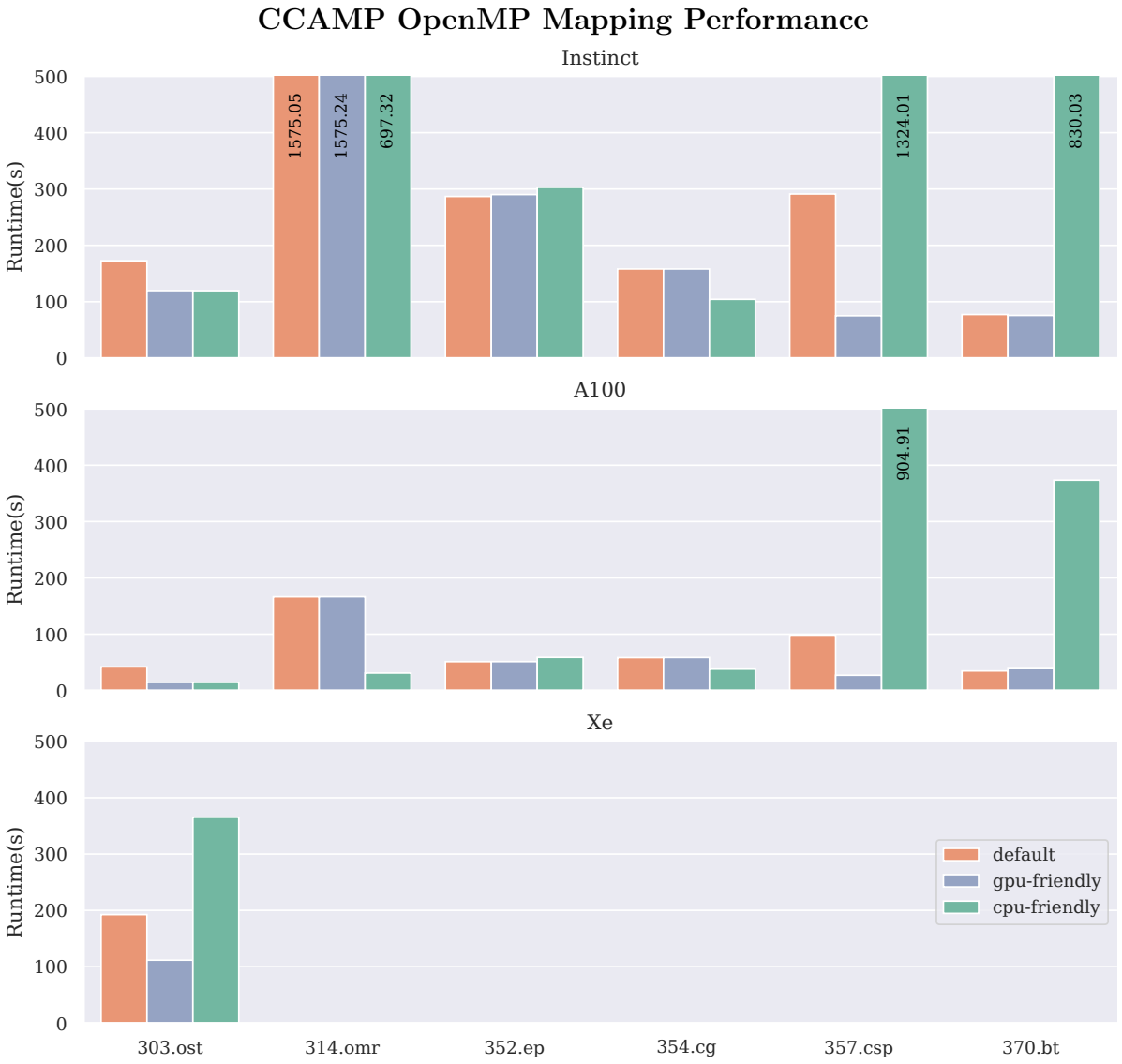


Figure 32. Runtime performance comparison of different CCAMP OpenMP mappings across different architectures.

similarly or much better than the other mappings, which is consistent with the intent of the device-specific optimizations. The lone outlier, 314.omr, performs best with the "cpu-friendly" mapping on both the Instinct and A100 device. This is consistent with the results in the original CCAMP project [14] (Chapter III of this dissertation). In 314.omr, a reduction and small loop trip count on an inner loop of a computationally intensive kernel causes the nested parallelism of the "gpu-friendly" approach to perform more poorly than the outer-loop parallelism focused "cpu-friendly" approach.

Overall, this evaluation demonstrates that OpenMP directive configuration still plays a huge role in OpenMP performance on these exascale-similar hardware platforms, and further motivates the need for device-specific optimization, preferably automated, as CCAMP's distinct mappings can be configured using a single command-line argument. However, improvements are needed to tools like OpenARC's CCAMP for more consistent performance across a wide array of applications.

4.4.4 Intel *icpx* and Intermediate Representations for OpenMP.

In this section we briefly explore the two different intermediate representations generated during Intel *icpx* compilation of OpenMP when targeting the Xe Max GPU. During the compilation of an OpenMP application, the OpenMP application is lowered by *icpx* to either Level0 or OpenCL.

Level0 is an intermediate representation developed as part of the Intel oneAPI framework. The goal of Level0 is to provide a driver-level API in order to interface between the different programming models supported under the oneAPI umbrella and the different hardware devices developed by Intel, including Intel GPUs, AI chips, and FPGAs. By default when compiling OpenMP the Level0 API

is targeted. However, this can be reconfigured at runtime to target OpenCL via the "LIBOMPTARGET_PLUGIN" environment variable. We initially experimented with this variable in an attempt to successfully execute more OpenMP applications on the Xe Max device, but even with "LIBOMPTARGET_PLUGIN=OPENCL" we only successfully executed the 303.ostencil application.

Table 18 shows the runtime performance of the 303.ostencil application, with the three different OpenMP "mappings" generated by OpenARC and CCAMP from the OpenACC source code. In the second column, the OpenMP codes were lowered by *icpx* to the Level0 API, while in the third column the OpenCL alternative internal API is targeted. We see that the performance is nearly identical for two of the three "mappings", but that for the "CPU-friendly" mapping we see a nearly 25% difference in performance, with the Level0 backend being more efficient. However, to adequately establish patterns in performance more evaluations need to be performed with either an updated implementation of the Intel OpenMP compiler, a next-generation Intel Xe GPU, or a different benchmark set with fewer double-precision applications.

Table 18. Runtime performance comparison of Level0 and OpenCL backends for *icpx* OpenMP compilations (303.ostencil)

Mapping	Level0	OpenCL
Default	193.91 (s)	192.52 (s)
GPU-friendly	*111.65 (s)*	*139.19 (s)*
CPU-friendly	365.36 (s)	365.68 (s)

4.4.5 LLVM and Nvidia Implementation Comparison for OpenCL and OpenMP.

Few devices have multiple implementations available for a single programming model. For example, CUDA, OpenACC, and HIP each have one implementation on supported devices (ignoring the translation tools

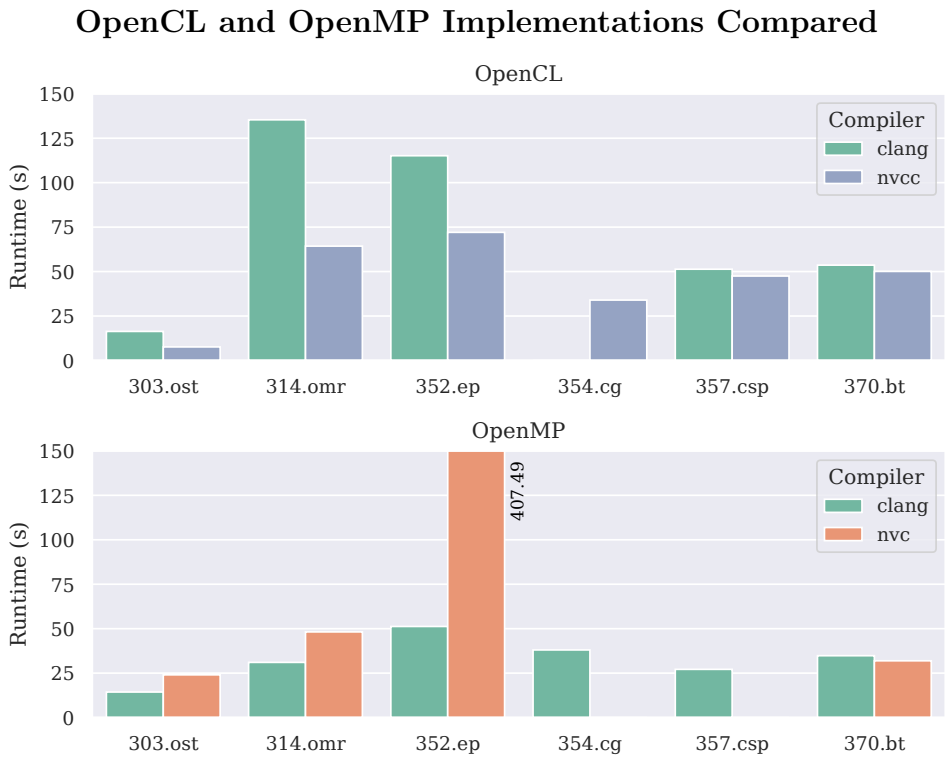


Figure 33. Runtime performance comparison of two OpenCL implementations and two OpenMP implementations (A100).

nvcc_detail, *hipify*, and OpenARC). Furthermore, on the Intel Xe GPU only a single implementation is evaluated in this work for both supported programming models.

However, for the Nvidia A100 device we explored two implementations for two different programming models; both the CUDA Toolkit's *nvc* and LLVM's *clang* for OpenCL, and both the NVHPC SDK's *nvc* and LLVM's *clang* for OpenMP. In this section we briefly compare and contrast these implementations.

Figure 33 shows the runtime performance differences between the two evaluated implementations for OpenCL (Subplot 1) and OpenMP (Subplot 2). For OpenCL, Nvidia's *nvcc* outperforms LLVM *clang* for each applications, in some cases significantly. Not only does *clang* perform more poorly than *nvcc*, but *clang* also fails to successfully execute 354.cg due to a memory error. However, nearly the exact converse is true for the OpenMP implementations. LLVM *clang* outperforms NVHPC *nvc* for nearly every every application, significantly so for 352.ep. In order to compile several of the applications with *nvc*, the *linear()* clause needed to be manually removed, as the current version of *nvc* does not yet support this clause. Even after removing unsupported clauses, *nvc* still failed to successfully compile the 354.cg and 357.csp applications due to internal compiler errors. However, OpenMP support in NVHPC's *nvc* is still relatively novel, being recently adapted from the implementation of the Nvidia-acquired PGI OpenACC compiler.

4.5 Exploration of Exascale Platforms: Related Work

In 2012, shortly after the release of the Exascale Software Project Roadmap [192], Lee et al. [193] performed an early evaluation of directive-based GPU programming models for productive exascale computing. They surveyed the then-current programming models, including OpenACC, HMPP, OpenMPC [140], and Rstream, and compared performance with CUDA applications. These models

have developed significantly over the past decade (OpenACC), or in other cases become deprecated (HMPP, OpenMPC), but the authors do identify several considerations that are still relevant for today's exascale programming models: functionality, scalability, tunabilty, and debugability.

In a more recent work (2018), Gayatari et al. [187] explore the performance of a single application (GPP) with OpenMP 4.5, OpenACC, and CUDA. They find that OpenMP and OpenACC initially fail to match the performance of the CUDA implementation, but after sufficient optimization the performances are similar. They also find that the GPU-intended OpenMP implementation performs poorly on the CPU device, an observation that we confirm and address with the CCAMP OpenMP optimizations in our evaluations.

A recent work (2020) by Davis et al. [194] assesses the performance of different OpenMP compilers on the Nvidia V100 device. The evaluate the Cray, IBM, Nvidia, and LLVM clang OpenMP compilers on several different benchmarks, and observe general programming patterns. Their results are consistent with the OpenMP backend results we experience in this work, although they are limited to a single OpenMP mapping without an automated mapping strategy like CCAMP.

Also in 2020, Usha et al. [195] compare the performance OpenACC and OpenMP 4.5 for Nvidia GPUs, specifically P100 and V100 devices., on several generic benchmarks, including matrix multiplication, Jacboi kernels, and Monte-Carlo simulations. They experience more success with OpenACC on the Nvidia devices, and have difficulties optimizing the OpenMP implementations.

Finally, in 2020 Bertoni et al. [196] perform a performance portability evaluation of OpenCL benchmarks across Intel and Nvidia Platforms. Specifically, they evaluate using an Intel Integrated (G9) GPU, an Intel SkyLake CPU, and an

Nvidia V100 GPU. Their project focuses on developing a metric for and measuring the performance-portability of OpenCL applications between platforms, and they conclude that a significant effort is needed to realistically achieve sufficient performance portability with OpenCL. In the meantime, this motivates tools like OpenARC that can generate several specialized output codes using a single portable input programming model.

All of the works discussed in this section complement and confirm the results of this project. However they all address either a single programming model, experiment with a single or small number of benchmarks, evaluate specific limited device families, or rely on now outdated hardware and software platforms. In contrast, in this project we explore a wide diversity of programming models, several different bleeding-edge hardware and software platforms, and an extensive set of benchmark applications for a more comprehensive overview of the state of exascale programming approaches.

4.6 Exploration of Exascale Platforms: Conclusions

The rapidly approaching horizon of exascale machines promise to deliver incredible performance, but inevitably create incredible challenges, including the availability, implementations, and performance of programming approaches. In this work we explore several programming approaches guaranteed to occupy a spot in the exascale landscape. We investigate both the individual performance of these approaches on exascale-intended hardware, and the feasibility of generating these specialized approaches using a single source code and source-to-source translation.

First, this work is only made possible by the availability of quality OpenACC benchmarks from SPEC Accel and the source-to-source capabilities of OpenARC. The idea of developing and maintaining separate OpenACC,

OpenMP, OpenCL, CUDA, and HIP implementations of an array of applications is intimidating at best, if not impossible. Furthermore, different devices may prefer different code versions even within a single backend programming model, as we see with OpenMP. An automated compilation framework is critical to comprehensively explore and evaluate this diversity of programming models across several different platforms.

Our evaluations highlight several important considerations for exascale-intended platforms. When comparing programming models across platforms, we immediately see that the engineering of the Nvidia GPU and maturity of the CUDA implementations outclass the other platforms and programming approaches. The Nvidia-focused evaluations achieved consistently low performance on all benchmarks, a testament to the focus the HPC and heterogeneous computing communities have granted Nvidia over the past decade. Even the LLVM-based OpenMP implementations are mature when targeting Nvidia devices, but lag significantly behind for AMD and Intel GPUs. However, we also see significant successes with the other platforms.

AMD, with ROCm and HIP, have developed a mature open-source environment for compilation and execution on their platforms. They have also developed high-quality tools and header files for interaction with Nvidia software and platforms, effectively allowing them to leverage work done by Nvidia and CUDA developers instead of re-inventing the wheel.

Furthermore, OpenCL, while not the most popular choice for developers, was the most functionally portable programming model evaluated, followed by OpenMP. OpenACC is still largely limited to Nvidia devices, but the Clacc [112] framework may make OpenACC available on a wide variety of platforms that

support LLVM OpenMP implementations. On that note, we also observe that LLVM is a core technology in nearly every evaluated programming approach, either directly via clang, indirectly as part of a compiler-driver, or internally as a compiler builder.

Finally, we observe that Intel's Xe platform requires a significant amount of effort to match the performance of the other platforms, at least with the evaluated programming models. Failed compilations and executions, both with OpenMP and OpenCL, and a lack of support for double-precision made evaluating the Xe platform with the SPEC Accel benchmarks challenging. This is concerning with the imminent release of Aurora. However, it is possible that the Xe platform experiences more success with the OneAPI, SYCL, and DPC++ programming approaches which were not evaluated in this work.

To that end, a relevant future work involves creating a SYCL backend for OpenARC. We also hope to extend CCAMP to support device-specific optimizations for other programming models (OpenCL, HIP, etc.). We also hope to extend our evaluations of OpenACC by incorporating the Clacc framework.

CHAPTER V

CONCLUSION

Since their first conceptualization with the PASM and TRAC machines in the early 80s, heterogeneous computing and heterogeneous programming approaches have shifted in and out of vogue. In Chapter I, we recounted how distributed heterogeneous computing rose with the promise of robust diverse and distributed systems. We also saw how these systems were eventually eclipsed by homogeneous supercomputers, homogeneous cloud servers, and CPU-chip advancements. We explored the rebirth of heterogeneous computing through accelerator-based computing, as well as the explosion of GPU-based computing.

Although the contexts are distinct, the challenges faced by distributed heterogeneous systems and contemporary accelerator-based systems are not so different. Many of the challenges early developers faced are being constantly revived and re-imagined, especially in the face of the extreme heterogeneity of next-generation systems. Many of the conceptual models, theoretical road maps, programming approaches, technical requirements and restrictions, and strategies for success from distributed heterogeneous research apply directly to accelerator-based systems. The original Figure 3, first published in 1995, would look right at home in a 2021 publication exploring extreme heterogeneity, albeit with improved graphics.

In Chapter I, we introduced several significant challenges related to current and next-generation heterogeneous programming and computing: the diversity of hardware and of programming models, finding the appropriate abstraction level for different types of science, and the balance between different types of funding for programming platform development. Although we present these concepts as challenges, in reality, they are also indications of progress. Any claim to solve these

challenges in totality would be less a scientific achievement and more an indication of stagnation in computational development.

However, working toward solutions is important and necessary. Creating portable, automated, and optimized programming solutions for extremely heterogeneous environments is crucial as we encounter increasingly diverse and specialized accelerators. The research in this dissertation makes an adventurous step closer to addressing the challenges of contemporary heterogeneous computing.

In Chapter II, we introduce a high-level directive-based framework designed to bring FPGAs, previously an outcast, under the umbrella of high-performance computing. We develop automated and compiler-based optimizations, empowering scientific programmers to both write palatable applications and produce highly specialized code, effectively bridging the semantic gap between hardware-level, low-level, and high-level FPGA programming.

In Chapter III, we present an interoperable framework integrating the two most common directive-based standards in high-performance computing. By conceptually merging the two standards, we stretch the capabilities, contexts, and ultimately the performance of applications written in either standard. Finally, in Chapter IV we present an exploration and evaluation of exascale-intended programming approaches. Exascale systems, the near-term pinnacle of the heterogeneous timeline, are far from immune to the challenges outlined above, and will feature a diverse set of hardware and programming models. In Chapter IV, we leverage a single programming model to explore this diversity, again relying on automated compiler optimizations and code generation.

Heterogeneity in computing is fated to an endless cycle of divergence and specialization, encapsulation and integration. In the future, these intertwined

notions may result in more exotic hardware like quantum or neuromorphic accelerators, and more evolved software concepts like AI-inspired compilation and machine programming—and these advances will require the very same incremental steps presented in this dissertation: the development of high-level, portable programming approaches that can deliver specialized performance.

REFERENCES CITED

[1] Summit Supercomputer. Oak Ridge National Laboratory. [Online]. Available: https://www.olcf.ornl.gov/summit/

[2] Sierra Supercomputer. Lawrence Livermore National Laboratory. [Online]. Available: https://computing.llnl.gov/computers/sierra

[3] Frontier Supercomputer. Oak Ridge National Laboratory. [Online]. Available: https://www.olcf.ornl.gov/frontier/

[4] Aurora Supercomputer. Argonne National Laboratory. [Online]. Available: https://alcf.anl.gov/aurora

[5] El Capitan Supercomputer. Lawrence Livermore National Laboratory. [Online]. Available: https://www.llnl.gov/news/llnl-and-hpe-partner-amd-el-capitan-projected-worlds-fastest-supercomputer

[6] J. S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf, K. Antypas, D. Donofrio, T. Humble, C. Schuman, B. V. Essen, S. Yoo, A. Aiken, D. Bernholdt, S. Byna, K. Cameron, F. Cappello, B. Chapman, A. Chien, M. Hall, R. Hartman-Baker, Z. Lan, M. Lang, J. Leidel, S. Li, R. Lucas, J. Mellor-Crummey, P. P. Jr, T. Peterka, M. Strout, and J. Wilke, "Extreme Heterogeneity 2018 - Productive Computational Science in the Era of Extreme Heterogeneity: Report for DOE ASCR Workshop on Extreme Heterogeneity," USDOE Office of Science (SC) (United States), Tech. Rep., 2018.

[7] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, and others, "Tensorflow: A system for large-scale machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[8] D. Kirk and others, "NVIDIA CUDA software and GPU parallel computing architecture," in *ISMM*, vol. 7, 2007, pp. 103–104.

[9] J. Lambert, S. Lee, J. Kim, J. S. Vetter, and A. D. Malony, "Directive-Based, High-Level Programming and Optimizations for High-Performance Computing with FPGAs," in *ACM International Conference on Supercomputing (ICS18)*, Jun. 2018.

[10] J. Lambert, S. Lee, J. S. Vetter, and A. Malony, "In-Depth Optimization with the OpenACC-to-FPGA Framework on an Arria 10 FPGA," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2020, pp. 460–470.

[11] J. Lambert, S. Lee, J. S. Vetter, and A. D. Malony, "Optimization with the openacc-to-fpga framework on the arria 10 and stratix 10 fpgas," *Parallel Computing*, vol. 104-105, p. 102784, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819121000417

[12] A. M. Cabrera, A. R. Young, J. Lambert, Z. Xiao, A. An, S. Lee, Z. Jin, J. Kim, J. Buhler, R. D. Chamberlain, and J. S. Vetter, "Toward evaluating high-level synthesis portability and performance between intel and xilinx fpgas," in *IWOCL'21: International Workshop on OpenCL, Munich Germany, April, 2021*, S. McIntosh-Smith, Ed. ACM, 2021, pp. 7:1–7:9. [Online]. Available: https://doi.org/10.1145/3456669.3456699

[13] J. Lambert, S. Lee, A. D. Malony, and J. S. Vetter, "CCAMP: OpenMP and OpenACC Interoperable Framework, Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms," in *Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar), in conjunction with Euro-Par19*, Gottigen, Germany, 2019.

[14] J. Lambert, S. Lee, A. Malony, and J. S. Vetter, "CCAMP: An Integrated Translation and Optimization Framework for OpenACC and OpenMP," in *SuperComputing*, 2020.

[15] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, H. E. Smalley, and S. D. Smith, "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Transactions on computers*, vol. 100, no. 12, pp. 934–947, 1981, publisher: IEEE.

[16] N. E. Abel, P. P. Budnik, D. J. Kuck, Y. Muraoka, R. S. Northcote, and R. B. Wilhelmson, "TRANQUIL: a language for an array processing computer," in *Proceedings of the May 14-16, 1969, spring joint computer conference*. ACM, 1969, pp. 57–73.

[17] M. C. Sejnowski, E. T. Upchurch, R. N. Kapur, D. P. Charlu, and G. J. Lipovski, "An overview of the Texas reconfigurable array computer," in *afips*. IEEE, 1980, p. 631.

[18] G. J. Lipovski, "SIMD and MIMD processing in the Texas Reconfigurable Array Computer," in *Proceedings COMPSAC 88: The Twelfth Annual International Computer Software & Applications Conference*. IEEE, 1988, pp. 268–269.

[19] M. Ercegovac, "Heterogeneity in supercomputer architectures," *Parallel Computing*, vol. 7, no. 3, pp. 367–372, 1988, publisher: Elsevier.

[20] H. J. Siegel, J. K. Antonio, R. C. Metzger, M. Tan, and Y. A. Li, "Goals of and open problems in high-performance heterogeneous computing," in *23rd AIPR Workshop: Image and Information Systems: Applications and Opportunities*, vol. 2368. International Society for Optics and Photonics, 1995, pp. 206–217.

[21] V. S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: practice and experience*, vol. 2, no. 4, pp. 315–339, 1990, publisher: Wiley Online Library.

[22] A. Beguelin, J. Dongarra, A. Geist, and V. Sunderam, "Visualization and debugging in a heterogeneous environment," *Computer*, vol. 26, no. 6, pp. 88–95, 1993, publisher: IEEE.

[23] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, K. Moore, and V. Sunderam, "PVM and HeNCE: Tools for heterogeneous network computing," in *Software for Parallel Computation*. Springer, 1993, pp. 91–99.

[24] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and K. Moore, "HeNCE: A heterogeneous network computing environment," *Scientific Programming*, vol. 3, no. 1, pp. 49–60, 1994, publisher: IOS Press.

[25] R. M. Butler and E. L. Lusk, "Monitors, messages, and clusters: The p4 parallel programming system," *Parallel Computing*, vol. 20, no. 4, pp. 547–564, 1994, publisher: Elsevier.

[26] A. S. Grimshaw, "Easy-to-use object-oriented parallel processing with Mentat," *Computer*, no. 5, pp. 39–51, 1993, publisher: IEEE.

[27] A. S. Grimshaw, J. B. Weissman, E. A. West, and E. C. Loyot, "Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 21, no. 3, pp. 257–270, 1994, publisher: Elsevier.

[28] B. Bland, "Jaguar: Powering and cooling the beast," *Conference on High-Speed Computing*, 2009.

[29] Y. Cui, K. B. Olsen, T. H. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. K. Panda, A. Chourasia, and others, "Scalable earthquake simulation on petascale supercomputers," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–20.

[30] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in cell's multicore architecture," *IEEE micro*, vol. 26, no. 2, pp. 10–24, 2006, publisher: IEEE.

[31] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, "Entering the petaflop era: the architecture and performance of Roadrunner," in *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, 2008, pp. 1–11.

[32] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming," *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2012, publisher: Elsevier.

[33] NVIDIA Developer Blog. [Online]. Available: https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c/

[34] X.-J. Yang, X.-K. Liao, K. Lu, Q.-F. Hu, J.-Q. Song, and J.-S. Su, "The TianHe-1A supercomputer: its hardware and software," *Journal of computer science and technology*, vol. 26, no. 3, pp. 344–351, 2011, publisher: Springer.

[35] B. Bland, "Titan-early experience with the titan system at oak ridge national laboratory," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 2012, pp. 2189–2211.

[36] OpenACC Hackathon Online Reference. OpenACC. [Online]. Available: https://www.openacc.org/hackathons

[37] Open Compute Language Online Portal. Khronos Group. [Online]. Available: http://www.khronos.org/opencl/

[38] Intel OneAPI Online Portal. Intel. [Online]. Available: https://www.intel.com/content/www/us/en/develop/tools/oneapi/components/dpc-library.html

[39] S. Lee and J. S. Vetter, "OpenARC: open accelerator research compiler for directive-based, efficient heterogeneous computing," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, 2014, pp. 115–120.

[40] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, and others, "TVM: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[41] ROCm Documentation Online. AMD. [Online]. Available: https://rocmdocs.amd.com/en/latest/

[42] AMD ROCm Online Repository. AMD. [Online]. Available: https://github.com/RadeonOpenCompute/ROCm

[43] M. Wolfe and P. C. Engineer, "The PGI Accelerator Programming Model on NVIDIA GPUs Part 1," *PGI Group*, 2009.

[44] M. Wolfe, "Implementing the PGI accelerator model," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010, pp. 43–50.

[45] J. Lee and M. Sato, "Implementation and performance evaluation of xcalablemp: A parallel programming language for distributed memory systems," in *2010 39th International Conference on Parallel Processing Workshops*. IEEE, 2010, pp. 413–420.

[46] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel processing letters*, vol. 21, no. 02, pp. 173–193, 2011, publisher: World Scientific.

[47] H. C. Edwards and C. R. Trott, "Kokkos: Enabling performance portability across manycore architectures," in *2013 Extreme Scaling Workshop (xsw 2013)*. IEEE, 2013, pp. 18–24.

[48] H. C. Edwards, D. Sunderland, V. Porter, C. Amsler, and S. Mish, "Manycore performance-portability: Kokkos multidimensional array library," *Scientific Programming*, vol. 20, no. 2, pp. 89–114, 2012, publisher: IOS Press.

[49] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014, publisher: Elsevier.

[50] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, 2014, pp. 1–11.

[51] F. Mueller, "Pthreads library interface," *Florida State University*, 1993, publisher: Citeseer.

[52] R. D. Hornung and J. A. Keasler, "The RAJA portability layer: overview and status," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2014.

[53] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "RAJA: Portable performance for large-scale scientific applications," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 2019, pp. 71–81.

[54] M. Martineau, S. McIntosh-Smith, M. Boulton, W. Gaudin, and D. Beckingsale, "A performance evaluation of Kokkos & RAJA using the TeaLeaf mini-app," in *The International Conference for High Performance Computing, Networking, Storage and Analysis, SC15*, 2015.

[55] R. Keryell, R. Reyes, and L. Howes, "Khronos SYCL for OpenCL: a tutorial," in *Proceedings of the 3rd International Workshop on OpenCL*, 2015, pp. 1–1.

[56] B. Ashbaugh, A. Bader, J. Brodman, J. Hammond, M. Kinsner, J. Pennycook, R. Schulz, and J. Sewall, "Data Parallel C++ Enhancing SYCL Through Extensions for Productivity and Performance," in *Proceedings of the International Workshop on OpenCL*, 2020, pp. 1–2.

[57] Intel OneAPI Press Release. Intel. [Online]. Available: https://newsroom.intel.com/news-releases/intel-unveils-new-gpu-architecture-optimized-for-hpc-ai-oneapi/

[58] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–11.

[59] The Legion Parallel Programming System. Stanford Legion. [Online]. Available: https://legion.stanford.edu/

[60] The Legion Online Repository. Stanford Legion. [Online]. Available: https://github.com/StanfordLegion/legion

[61] The Legion ECP Project. Stanford Legion. [Online]. Available: https://www.exascaleproject.org/research-group/programming-models-runtimes/

[62] HPX Online Repository. High Performance ParalleX. [Online]. Available: https://github.com/STEllAR-GROUP/hpx

[63] HPX Online Portal. High Performance ParalleX. [Online]. Available: http://stellar.cct.lsu.edu/projects/hpx/

[64] P. Diehl, M. Seshadri, T. Heller, and H. Kaiser, "Integration of CUDA Processing within the C++ Library for Parallelism and Concurrency (HPX)," in *2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. IEEE, 2018, pp. 19–28.

[65] M. Copik and H. Kaiser, "Using sycl as an implementation framework for hpx. compute," in *Proceedings of the 5th International Workshop on OpenCL*, 2017, pp. 1–7.

[66] K. Gregory and A. Miller, *C++ AMP*. Microsoft Press, 2012.

[67] B. Schäling, *The boost C++ libraries*. Boris Schäling, 2011.

[68] J. Szuppe, "Boost. Compute: A parallel computing library for C++ based on OpenCL," in *Proceedings of the 4th International Workshop on OpenCL*, 2016, pp. 1–39.

[69] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for CUDA," in *GPU computing gems Jade edition*. Elsevier, 2012, pp. 359–371.

[70] Bolt template Library. C++ template library for heterogeneous computing. AMD. [Online]. Available: https://hsa-libraries.github.io/Bolt/html/

[71] D. Demidov, "VexCL: Vector expression template library for OpenCL," 2012.

[72] NVIDIA cuBLAS Library. NVIDIA. [Online]. Available: https://developer.nvidia.com/cublas

[73] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse library," in *GPU Technology Conference*, 2010.

[74] Toolkit 4.1 CUFFT Library. NVIDIA CUDA. [Online]. Available: https://developer.nvidia.com/cufft

[75] ROCm Blas Library. AMD. [Online]. Available: https://rocmdocs.amd.com/en/latest/ROCm_Tools/clBLA.html

[76] S. Tomov, J. Dongarra, V. Volkov, and J. Demmel, "Magma library," *Univ. of Tennessee and Univ. of California, Knoxville, TN, and Berkeley, CA*, 2009.

[77] S. Tomov, R. Nath, P. Du, and J. Dongarra, "MAGMA Users' Guide," *ICL, UTK (November 2009)*, 2011.

[78] Eigen C++ Library Online Reference. Eigen. [Online]. Available: http://eigen.tuxfamily.org/index.php

[79] K. Ahnert and M. Mulansky, "Odeint–solving ordinary differential equations in C++," in *AIP Conference Proceedings*, vol. 1389. American Institute of Physics, 2011, pp. 1586–1589, issue: 1.

[80] Odient Online Repository. Odient Library. [Online]. Available: http://headmyshoulder.github.io/odeint-v2/

[81] Spiral Online Portal. Spiral. [Online]. Available: https://www.spiral.net/

[82] M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, and others, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005, publisher: IEEE.

[83] F. B. Hamzah, C. Lau, H. Nazri, D. Ligot, G. Lee, C. Tan, and others, "CoronaTracker: worldwide COVID-19 outbreak data analysis and prediction," *Bull World Health Organ*, vol. 1, p. 32, 2020.

[84] J. C. Machicao, "Covid-19 infection speed and acceleration as better tools for monitoring with uncertain data in Peru."

[85] D. J. Duffy, "Analysis of covid-19 mathematical and software models: Or how not to set up a softward project," *Wilmott*, vol. 2020, no. 110, pp. 66–71, 2020.

[86] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, 2013, issue: 6 Pages: 519–530 Publisher: ACM New York, NY, USA Volume: 48.

[87] Halide Online Portal. Halide. [Online]. Available: https://halide-lang.org/

[88] Halide Online Repository. Halide. [Online]. Available: https://github.com/halide/Halide

[89] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[90] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[91] A. Gulli and S. Pal, *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[92] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, and others, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[93] S. Tokui, K. Oono, S. Hido, and J. Clayton, "Chainer: a next-generation open source framework for deep learning," in *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, vol. 5, 2015, pp. 1–6.

[94] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[95] MIOpen - AMD's Machine Intelligence Library. AMD. [Online]. Available: https://github.com/ROCmSoftwarePlatform/MIOpen

[96] J. Khan, P. Fultz, A. Tamazov, D. Lowell, C. Liu, M. Melesse, M. Nandhimandalam, K. Nasyrov, I. Perminov, T. Shah, and others, "MIOpen: An Open Source Library For Deep Learning Primitives," *arXiv preprint arXiv:1910.00078*, 2019.

[97] K. Moreland, C. Sewell, W. Usher, L.-t. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K.-L. Ma, H. Childs, and others, "Vtk-m: Accelerating the visualization toolkit for massively threaded architectures," *IEEE computer graphics and applications*, vol. 36, no. 3, pp. 48–58, 2016, publisher: IEEE.

[98] M. Larsen, J. Ahrens, U. Ayachit, E. Brugger, H. Childs, B. Geveci, and C. Harrison, "The alpine in situ infrastructure: Ascending from the ashes of strawman," in *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, 2017, pp. 42–46.

[99] V. Clement, S. Ferrachat, O. Fuhrer, X. Lapillonne, C. E. Osuna, R. Pincus, J. Rood, and W. Sawyer, "The CLAW DSL: Abstractions for performance portable weather and climate models," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, 2018, pp. 1–10.

[100] CLAW Project Online Repository. ETH CLAW Project. [Online]. Available: https://claw-project.github.io/

[101] NVIDIA. Nvcc cuda compiler online reference. [Online]. Available: https://developer.nvidia.com/cuda-llvm-compiler

[102] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pages: 75–86 Publisher: IEEE.

[103] PGI OpenCL Press Release. The Portland Group. [Online]. Available: https://www.khronos.org/news/permalink/pgi-opencl-compiler-for-arm

[104] PGI Compiler Web Reference. The Portland Group. [Online]. Available: https://www.pgroup.com/index.htm

[105] Nvidia Math Libraries. NVIDIA. [Online]. Available: https://developer.nvidia.com/hpc-sdk

[106] AMD ROCm AOCC Compiler. AMD. [Online]. Available: https://developer.amd.com/amd-aocc

[107] C. J. Newburn, S. Dmitriev, R. Narayanaswamy, J. Wiegert, R. Murty, F. Chinchilla, R. Deodhar, and R. McGuire, "Offload compiler runtime for the Intel® Xeon Phi coprocessor," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 2013, pp. 1213–1225.

[108] Intel FPGA SDK for OpenCL. Intel. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html

[109] C. Lattner, "LLVM and Clang: Next generation compiler technology," in *The BSD conference*, vol. 5, 2008.

[110] LLVM Compiler Framework Online Portal. LLVM Project. [Online]. Available: https://llvm.org/

[111] Clang Compiler Framework Online Portal. LLVM Project. [Online]. Available: https://clang.llvm.org/

[112] J. E. Denny, S. Lee, and J. S. Vetter, "Clacc: Translating OpenACC to OpenMP in Clang," in *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE, 2018, pp. 18–29.

[113] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: A Compiler Infrastructure for the End of Moore's Law," *arXiv:2002.11054 [cs]*, Feb. 2020. [Online]. Available: http://arxiv.org/abs/2002.11054

[114] Multi-Level IR Compiler Framework Online Portal. LLVM Project. [Online]. Available: https://mlir.llvm.org/

[115] Tensorflow MLIR Project Page. Tensorflow. [Online]. Available: https://www.tensorflow.org/mlir

[116] Flang Compiler Online Repository. LLVM Project. [Online]. Available: https://github.com/flang-compiler

[117] GCC Compiler. GNU Project. [Online]. Available: https://gcc.gnu.org/

[118] GNU OpenACC Status Wiki. GNU Project. [Online]. Available: https://gcc.gnu.org/wiki/OpenACC

[119] GNU OpenMP Status Wiki. GNU Project. [Online]. Available: https://gcc.gnu.org/wiki/openmp

[120] D. Quinlan, "ROSE: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, no. 02n03, pp. 215–226, 2000, publisher: World Scientific.

[121] Rose Compiler Online Repository. Lawrence Livermore National Laboratory. [Online]. Available: https://github.com/rose-compiler/rose

[122] C. Liao, Y. Yan, B. R. De Supinski, D. J. Quinlan, and B. Chapman, "Early experiences with the OpenMP accelerator model," *International Workshop on OpenMP*, 2013, pages: 84–98 Publisher: Springer.

[123] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng, "OpenUH: An optimizing, portable OpenMP compiler," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 18, pp. 2317–2332, 2007, publisher: Wiley Online Library.

[124] OpenUH Compiler Online Repository. HPCTools at UH. [Online]. Available: https://github.com/uhhpctools/openuh

[125] Open64 Compiler Online Repository. Open64. [Online]. Available: https://github.com/open64-compiler/open64

[126] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka, "Design of OpenMP compiler for an SMP cluster," in *Proc. of the 1st European Workshop on OpenMP*, 1999, pp. 32–39.

[127] Omni Compiler Online Portal. Omni Compiler. [Online]. Available: https://omni-compiler.org/

[128] A. Tabuchi, M. Nakao, and M. Sato, "A source-to-source OpenACC compiler for CUDA," in *European Conference on Parallel Processing*. Springer, 2013, pp. 178–187.

[129] M. Nakao, H. Murai, T. Shimosaka, A. Tabuchi, T. Hanawa, Y. Kodama, T. Boku, and M. Sato, "XcalableACC: Extension of XcalableMP PGAS language using OpenACC for accelerator clusters," in *2014 First Workshop on Accelerator Programming using Directives*. IEEE, 2014, pp. 27–36.

[130] T. Boku, T. Hanawa, H. Murai, M. Nakao, Y. Miki, H. Amano, and M. Umemura, "GPU-accelerated language and communication support by FPGA," in *Advanced Software Technologies for Post-Peta Scale Computing*. Springer, 2019, pp. 301–317.

[131] Y. Watanabe, J. Lee, K. Sano, T. Boku, and M. Sato, "Design and Preliminary Evaluation of OpenACC Compiler for FPGA with OpenCL and Stream Processing DSL," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops*, 2020, pp. 10–16.

[132] J. Planas, R. M. Badia, E. Ayguade, and J. Labarta, "Self-adaptive OmpSs tasks in heterogeneous environments," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 138–149.

[133] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with StarSs," *The International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 284–299, 2009, publisher: SAGE Publications Sage UK: London, England.

[134] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, "Productive cluster programming with OmpSs," in *European Conference on Parallel Processing*. Springer, 2011, pp. 555–566.

[135] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta, "Productive programming of GPU clusters with OmpSs," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 557–568.

[136] V. K. Elangovan, R. M. Badia, and E. A. Parra, "OmpSs-OpenCL programming model for heterogeneous systems," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2012, pp. 96–111.

[137] M. Perelló Bacardit, "Porting Rodinia Applications to OmpSs@ FPGA," B.S. thesis, Universitat Politècnica de Catalunya, 2019.

[138] J. Bosch, A. Filgueras, M. Vidal, D. Jimenez-Gonzalez, C. Alvarez, and X. Martorell, "Exploiting parallelism on GPUs and FPGAs with OmpSs," in *Proceedings of ANDARE - the 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems*, 2017, pp. 1–5.

[139] J. Bosch, X. Tan, A. Filgueras, M. Vidal, M. Mateu, D. Jiménez-González, C. Álvarez, X. Martorell, E. Ayguadé, and J. Labarta, "Application acceleration on fpgas with ompss@ fpga," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 70–77.

[140] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP programming and tuning for GPUs," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–11.

[141] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A Source-to-Source Compiler Infrastructure for Multicores," *IEEE Computer*, vol. 42, no. 12, pp. 36–42, 2009, publisher: IEEE. [Online]. Available: http://www.ecn.purdue.edu/ParaMount/publications/ieeecomputer-Cetus-09.pdf

[142] S. Lee, J. S. Meredith, and J. S. Vetter, "Compass: A framework for automated performance modeling and prediction," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 405–414.

[143] K. L. Spafford and J. S. Vetter, "Aspen: A domain specific language for performance modeling," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–11.

[144] S. Lee, J. Kim, and J. S. Vetter, "OpenACC to FPGA: A Framework for Directive-based High-Performance Reconfigurable Computing," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016.

[145] M. Kotsifakou, P. Srivastava, M. D. Sinclair, R. Komuravelli, V. Adve, and S. Adve, "Hpvm: Heterogeneous parallel virtual machine," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2018, pp. 68–80.

[146] HPVM Online Portal. Heterogeneous Parallel Virtual Machine. [Online]. Available: https://publish.illinois.edu/hpvm-project/

[147] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-h. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[148] G. Juckeland, W. Brantley, S. Chandrasekaran, B. Chapman, S. Che, M. Colgrove, H. Feng, A. Grund, R. Henschel, W.-M. W. Hwu, and others, "Spec accel: A standard application suite for measuring hardware accelerator performance," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 2014, pp. 46–67.

[149] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 63–74.

[150] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[151] W. Feng, H. Lin, T. Scogland, and J. Zhang, "OpenCL and the 13 dwarfs: a work in progress," in *Proceedings of the 3rd acm/spec international conference on performance engineering*, 2012, pp. 291–294.

[152] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and others, "The landscape of parallel computing research: A view from berkeley," 2006, publisher: eScholarship, University of California.

[153] EPCC OpenACC Benchmarks. EPCC. [Online]. Available: https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openacc-benchmark-suite

[154] EPCC Online Repository. EPCC. [Online]. Available: https://github.com/EPCCed/epcc-openacc-benchmarks

[155] Q. Tang, L. Jiang, M. Su, and Q. Dai, "A pipelined market data processing architecture to overcome financial data dependency," in *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*, Dec. 2016, pp. 1–8.

[156] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad, "Database Analytics Acceleration Using FPGAs," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 411–420. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370874

[157] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang, "FPGA-Accelerated Dense Linear Machine Learning: A Precision-Convergence Trade-Off," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Apr. 2017, pp. 160–167.

[158] Y. Kim, S. Jadhav, and C. S. Gloster, "Dataflow to Hardware Synthesis Framework on FPGAs," in *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, Oct. 2016, pp. 91–96.

[159] B. Betkaoui, D. B. Thomas, W. Luk, and N. Przulj, "A framework for FPGA acceleration of large graph problems: Graphlet counting case study," in *2011 International Conference on Field-Programmable Technology*, Dec. 2011, pp. 1–8.

[160] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16, 2016, pp. 35:1–35:12.

[161] Mentor SDK Design Suite. Handle-C. [Online]. Available: https://www.mentor.com/products/fpga/handel-c/dk-design-suite/

[162] The Vitis software development platform. Xilinx. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html

[163] M. Aubury, I. Page, G. Randall, J. Saul, and R. Watts, *Handel-C language reference guide*, 1996.

[164] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11, 2011, pp. 33–36.

[165] P. R. Panda, "SystemC: A Modeling Platform Supporting Multiple Design Abstractions," in *Proceedings of the 14th International Symposium on Systems Synthesis*, ser. ISSS '01, 2001, pp. 75–80.

[166] M. C. Smith, J. S. Vetter, and X. Liang, "Accelerating Scientific Applications with the SRC-6 Reconfigurable Computer: Methodologies and Analysis," in *19th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '05, 2005.

[167] The SDAccel software development platform. Xilinx. [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html

[168] OpenACC, "OpenACC: Directives for Accelerators," 2011, published: [Online]. Available: http://www.openacc.org.

[169] OpenACC Online Portal. OpenACC. [Online]. Available: https://www.openacc.org/

[170] "OpenMP Reference," 1999, OpenMP Standards Organization.

[171] I. Karlin, J. Keasler, and R. Neely, "LULESH 2.0 Updates and Changes," Livermore, CA, Tech. Rep. LLNL-TR-641973, Aug. 2013.

[172] B. Veenboer and J. W. Romein, "Radio-Astronomical Imaging: FPGAs vs GPUs," in *European Conference on Parallel Processing*. Springer, 2019, pp. 509–521.

[173] Z. Jin and H. Finkel, "Evaluating LULESH kernels on opencl FPGA," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2019, pp. 199–213.

[174] The OmpSs Programming Model. [Online]. Available: https://pm.bsc.es/ompss

[175] L. Sommer, J. Korinth, and A. Koch, "OpenMP device offloading to FPGA accelerators," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Jul. 2017, pp. 201–205.

[176] J. de Fine Licht and T. Hoefler, "hlslib: Software engineering for hardware design," *arXiv preprint arXiv:1910.04436*, 2019.

[177] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, "Stateful Dataflow Multigraphs: A data-centric model for performance portability on heterogeneous architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–14.

[178] M. J. Schulte, M. Ignatowski, G. H. Loh, B. M. Beckmann, W. C. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S. K. Reinhardt, and G. Rodgers, "Achieving exascale capabilities through heterogeneous computing," *Micro, IEEE*, vol. 35, no. 4, pp. 26–36, 2015.

[179] Y. LeCun, "Deep learning hardware: Past, present, and future," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2019, pp. 12–19.

[180] S. J. Pennycook, J. D. Sewall, and V. W. Lee, "Implications of a metric for performance portability," *Future Generation Computer Systems*, vol. 92, pp. 947–958, 2019.

[181] A. Sabne, P. Sakdhnagool, S. Lee, and J. S. Vetter, "Evaluating Performance Portability of OpenACC," in *Languages and Compilers for Parallel Computing*, 2015, pp. 51–66.

[182] L. Cai, Y. Wang, W. Tang, B. Wang, S. Ethier, Z. Liu, and J. Lin, "OpenACC vs the Native Programming on Sunway TaihuLight: A Case Study with GTC-P," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 88–97.

[183] J. Lin, Z. Xu, L. Cai, A. Nukada, and S. Matsuoka, "Evaluating the SW26010 Many-core Processor with a Micro-benchmark Suite for Performance Optimizations," *Parallel Computing*, vol. 77, pp. 128 – 143, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819118301820

[184] B. R. d. Supinski, T. R. W. Scogland, A. Duran, M. Klemm, S. M. Bellido, S. L. Olivier, C. Terboven, and T. G. Mattson, "The Ongoing Evolution of OpenMP," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 2004–2019, 2018.

[185] V. V. Larrea, W. Joubert, M. G. Lopez, and O. Hernandez, "Early Experiences Writing Performance Portable OpenMP 4 Codes," in *Proc. Cray User Group Meeting, London, England*, 2016.

[186] M. G. Lopez, V. V. Larrea, W. Joubert, O. Hernandez, A. Haidar, S. Tomov, and J. Dongarra, "Towards Achieving Performance Portability Using Directives for Accelerators," in *2016 Third Workshop on Accelerator Programming Using Directives (WACCPD)*. IEEE, 2016, pp. 13–24.

[187] R. Gayatri, C. Yang, T. Kurth, and J. Deslippe, "A Case Study for Performance Portability Using OpenMP 4.5," in *International Workshop on Accelerator Programming Using Directives*. Springer, 2018, pp. 75–95.

[188] M. Wolfe, *Compilers and More: OpenACC to OpenMP (and Back Again)*, hpcwire.com, Ed., Jun. 2016.

[189] G. Arnold, A. Calvert, J. Overbey, and N. Sultana, "From OpenACC to OpenMP 4: Toward Automatic Translation," *XCEDE16, Miami, FL*, 2016.

[190] S. Pino, L. Pollock, and S. Chandrasekaran, "Exploring Translation of OpenMP to OpenACC 2.5: Lessons Learned," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 673–682.

[191] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The Spack Package Manager: Bringing Order to HPC Software Chaos," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, 2015, event-place: Austin, Texas. [Online]. Available: https://doi.org/10.1145/2807591.2807623

[192] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig *et al.*, "The international exascale software project roadmap," *The international journal of high performance computing applications*, vol. 25, no. 1, pp. 3–60, 2011.

[193] S. Lee and J. S. Vetter, "Early evaluation of directive-based GPU programming models for productive exascale computing," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–11.

[194] J. H. Davis, C. Daley, S. Pophale, T. Huber, S. Chandrasekaran, and N. J. Wright, "Performance Assessment of OpenMP Compilers Targeting NVIDIA V100 GPUs," *arXiv preprint arXiv:2010.09454*, 2020.

[195] R. Usha, P. Pandey, and N. Mangala, "A Comprehensive Comparison and Analysis of OpenACC and OpenMP 4.5 for NVIDIA GPUs," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020, pp. 1–6.

[196] C. Bertoni, J. Kwack, T. Applencourt, Y. Ghadar, B. Homerding, C. Knight, B. Videau, H. Zheng, V. Morozov, and S. Parker, "Performance Portability Evaluation of OpenCL Benchmarks across Intel and NVIDIA Platforms," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2020, pp. 330–339.