Towards Enhancing Performance, Programmability, and Portability in Heterogeneous Computing

Konstantinos Krommydas

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Wu-chun Feng, Chair
Ali Butt
Yong Cao
Ruchira Sasanka
Eli Tilevich

November 30, 2016
Blacksburg, Virginia

Keywords: Performance, Programmability, Heterogeneous Architectures, Parallel Computing, Programming Framework

Towards Enhancing Performance, Programmability, and Portability in Heterogeneous Computing

Konstantinos Krommydas

(ABSTRACT)

The proliferation of a diverse set of heterogeneous computing platforms in conjunction with the plethora of programming languages and optimization techniques on each language for each underlying architecture exacerbate widespread adoption of such platforms. This is especially true for novice programmers and the non-technical-savvy masses that are largely precluded from the advantages of high-performance computing. Moreover, different groups within the heterogeneous computing community (e.g., computer architects, tool developers, and programmers) are presented with new challenges with respect to *performance*, *programmability*, and *portability* (or the *three P's*) of heterogeneous computing.

In this work we discuss such challenges and identify benchmarking techniques based on computation and communication patterns as an appropriate means for the systematic evaluation of heterogeneous computing with respect to the three P's. Our proposed approach is based on OpenCL implementations of the Berkeley dwarfs. We use our benchmark suite (*OpenDwarfs*) in characterizing performance of state-of-the-art parallel architectures, and as the main component of a methodology (*Telescoping Architectures*) for identifying trends in future heterogeneous architectures. Furthermore, we employ OpenDwarfs in a multi-faceted study on the gaps between the three P's in the context of the modern heterogeneous computing landscape. Our case study spans a variety of compilers, languages, optimizations, and target architectures, including the CPU, GPU, MIC, and FPGA. Based on our insights and extending aspects of prior research (e.g., in compilers, programming languages, and auto-tuning), we propose the introduction of grid-based data structures as the basis of programming frameworks and present a prototype unified framework (*GLAF*) that encompasses a novel visual programming environment with code generation, auto-parallelization, and auto-tuning capabilities. Our results, which span scientific domains, indicate

that our holistic approach constitutes a viable alternative towards enhancing the three P's and further democratizing heterogeneous, parallel computing for non-programming-savvy audiences, and especially domain scientists.

This work has been supported in part by the Institute for Critical Technology and Applied Science (ICTAS), NSF Center for High-Performance Reconfigurable Computing (CHREC), two Intel internships, and NASA/SSAI.

Towards Enhancing Performance, Programmability, and Portability in Heterogeneous Computing

Konstantinos Krommydas

(GENERAL AUDIENCE ABSTRACT)

In the past decade computing has moved from *single-core* machines, that is machines with a CPU that can execute code in a serial manner, to *multi-core* ones, i.e., machines with CPUs that can execute code in a parallel fashion. Another paradigm shift that has manifested in the past years entails computing that utilizes *heterogeneous processing*, as opposed to *homogeneous processing*. In the latter case a single type of processor (CPU) is responsible for executing a given program, whereas in the former case different types of processors (such as CPUs, graphics processors or other accelerators) collaborate in an effort to tackle computationally difficult problems in a fast, parallel manner.

The shift to *multi-core, parallel, heterogeneous computing* described above is accompanied by an associated shift in programming languages for such platforms, as well as techniques to optimize programs for high performance (i.e., execution speed). The unique complexities of parallel and heterogeneous computing exacerbate widespread adoption of such platforms. This is especially true for novice programmers and the non-technical-savvy masses that are largely precluded from the advantages of high-performance computing. Challenges include obtaining fast execution speeds (i.e., *performance*), easiness of programming (i.e., *programmability*), and the ability to execute programs across different heterogeneous platforms (i.e., *portability*). Performance, programmability, and portability constitute the *3 P's of heterogeneous computing*.

In this work we discuss the above challenges in detail and provide insights and solutions for different interest groups within the computing community, such as computer architects, tool developers and programmers. We propose an approach for evaluating existing heterogeneous computing platforms based on the concept of *dwarf-based benchmarks* (i.e., applications that are characterized

by certain computation and communication patterns). Furthermore, we propose a methodology for utilizing the dwarf concept for evaluating potential future heterogeneous platforms. In our research we attempt to quantify the trade-offs between performance, programmability, and portability in a wide set of modern heterogeneous platforms. Based on the above, we seek to bridge the 3 P's by introducing a programming framework that democratizes parallel algorithm development on heterogeneous architectures for novice programmers and domain scientists. Specifically, our framework produces parallel, optimized code implementations in multiple languages with the potential of executing across different heterogeneous platforms.

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Professor Wu-chun Feng who has guided me throughout my Ph.D journey. He has served as a role model and inspiration for me from the onset and has provided invaluable knowledge and insights, especially of the kind that textbooks cannot teach. Dr. Feng, a huge thank you for taking me in the lab back in 2010 and standing by me as my academic father.

A big thank you to my committee members: Professor Ali Butt, Doctor Yong Cao, Doctor Ruchira Sasanka, and Professor Eli Tilevich. I appreciate all your feedback on my work, as well as the time you put into serving in my Ph.D. committee.

Being part of a a great lab, the Synergy Lab, has greatly benefited me both personally and professionaly. A big thank you to all current and former members of the lab for your feedback on the work throughout the years, your support, and the good times in the lab!

I was also very lucky to have been given the opportunity to work as an intern with Intel for about a year in total. The work done there under the guidance of Dr. Ruchira Sasanka greatly steered the direction of my dissertation and I will always be grateful to him for being my mentor, and for his guidance, help, and support, not only during my time at Intel, but also afterwards.

My gratitude also goes to all the entities that funded my work: ICTAS, which honored me with a four-year fellowship and the freedom to freely pursue my research interests; the Computer Science department of Virginia Tech through teaching assistantships; the National Science Foundation CHREC program for a graduate research assistantship; Intel Corporation for two internships; NASA and SSAI for a graduate research assistantship.

As far as life outside of the lab is concerned, I've been happy to enjoy the friendship of wonderful people throughout these years. That would include the Greek community at Virginia Tech that has made me feel like home. Special thanks to my friends Nikolaos Artavanis, Manousos Valyrakis, Ioannis Kokkinidis, and Takis Apostolellis with Anna Delinikola and little Alkaios! Also, a big thank you to Engin Sengezer, Brian Jalaian, and so many more friends for all the good times in

downtown Blacksburg and beyond. Thank you, last, to my Greek buddies Christos Papagiannopoulos and Achilleas Mamalis who kept in touch all these years via our frequent Skype calls.

Finally, I would like to present my gratitude and love to my family back in Greece: my parents, Anastasios and Maria, and my sister Efstathia with her family, Achilleas and little Maria. All, for their never-ending support throughout my Ph.D. studies, and the former for investing in my lengthy education journey and instilling in me the love for learning, as well. Together with my Greek family, a last big thank you and my love to my very own family here, Armani Chien. Thank you for your love, support, for being patient and understanding, for bringing more fun in my everyday life, and most importantly for being my other half.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we motivate our work, outline the important research questions we seek to address, and present our contributions in the context of this dissertation.

1.1 Motivation

The past decade has seen a divergence from the traditional single-core computing paradigm. Single-core CPUs gave their place to multi-core processors initially, and subsequently to massively parallel platforms of heterogeneous nature. Chip multi-processors by Intel, AMD, IBM and other CPU vendors are an example of the former, whereas systems employing general-purpose graphics processing units (GPGPUs) constitute a prime example of the latter. More recent examples of heterogeneous systems that encompass varying types of cores include accelerators, like Intel

Xeon Phi of the Intel Many Integrated Cores (MIC) architecture, or even reconfigurable devices (FPGAs) and digital signal processors (DSPs).

The aforementioned deluge of heterogeneous platforms has inevitably introduced a new set of problems for the computing community, including – but not limited to – computer architects, programmers, and compiler/tool/run-time systems writers. Computer architects, for example, need a set of algorithms to use for iteratively designing, evaluating the efficacy of their designs, and identifying the architectural innovations that ultimately benefit applications. Programmers, on the other hand, face the problem of selecting a platform and language combination with which to build their applications. Last, but not least, compiler and tool writers require specific use cases to guide development and testing of back-end optimizations and tools of broad applicability.

The main challenges entailed in the context of the above landscape are tightly coupled with the themes of *performance*, *programmability* and *portability*, or the *three Ps of Heterogeneous Computing* [106]:

- **Performance** refers primarily to the execution time of a computer program. Faster execution of a program is a generally accepted metric of success. However, throughput of produced results can be more important in certain domains. Additionally, power-related considerations constitute an important part of the issue and especially on the road to exascale. Performance depends largely on the hardware itself and the underlying micro-architecture as designed by chip architects, but *performance efficiency* (or the extent to which specific hardware is efficiently utilized) is equally important. It mainly behooves the programmer, who needs

to possess deeper knowledge of the architectural details and appropriate optimizations, to "unlock" the hardware's full potential.

- **Programmability** is interrelated to the concept of performance (and performance efficiency) and refers to the level of difficulty a programmer faces in his/her attempts to exploit the hardware's theoretical peak performance. As such, any discussion on programmability entails intrinsic trade-offs between programming effort and the resulting performance. Programmability is a multi-faceted problem that includes issues related to programming languages, compilers and tools, as well as the hardware itself, as different architectures feature varying levels of difficulty in achieving sufficient performance efficiency.

- **Portability** is a characteristic of code and can be distinguished in two types: *functional* portability and *performance* portability. The former indicates that a certain computer program written in a certain language can be compiled for and run on different platforms and can generate identical output given the same input. This does not address any notion of performance. The latter, performance portability, on the other hand, implies equally fast performance across platforms (given the relative "computational power" of the platforms at hand).

All three P's [106] described above interact with each other in intricate ways and are characterized by trade-offs that interested groups (like the ones we mentioned at the very start, i.e., computer architects, programmers, compiler/tool/run-time systems writers) need to take into account in achieving their – also potentially conflicting – goals.

By design, certain heterogeneous platforms have a higher theoretical peak performance than others. This theoretical performance, however, is not only practically impossible, but it also depends on multiple factors, including the programmer's skills (which in turn depends on the programming language used), programming language abstractions, and the robustness of compilers and associated tools. All the above considered, results may also vary depending on the type of application at hand; certain types of applications that expose certain types of parallelism may be better on one platform over another, or they may be easier to express/implement and optimize in one language over another. In addition, certain parallelization opportunities or optimizations may be achieved through hardware or software (run-time systems, compilers, etc.) *Benchmark suites* have been traditionally used to perform evaluation studies and to compare computer systems. Over the past decade, benchmark suites have also been used to guide and even drive innovation in computer architecture. However, there has not been a systematic way or reasoning behind the choice of certain applications for a benchmark suite. The high-performance computing (HPC) community has merely focused on covering a large number of application areas (e.g., imaging, sorting, matrix operations) or some broad hardware-related requirements (e.g., floating point arithmetic, integer arithmetic). Moreover, the majority of benchmark suites are written in a specific language tied to a specific architecture, and potentially with architecture-specific optimizations, thus precluding fair cross-architecture evaluations and comparisons.

In order to address the problems of performance, programmability and portability in light of the above and in a systematic and generalizable way, our multi-faceted study exploits benchmarks that capture the computation and communication patterns (i.e., dwarfs or motifs) of present and future

applications. Specifically, we propose, implement changes to, and use *OpenDwarfs*, a benchmark suite based on the concept of the Berkeley dwarfs [28], i.e., cross-domain recurring patterns of computation and communication.

1.2 Research Questions

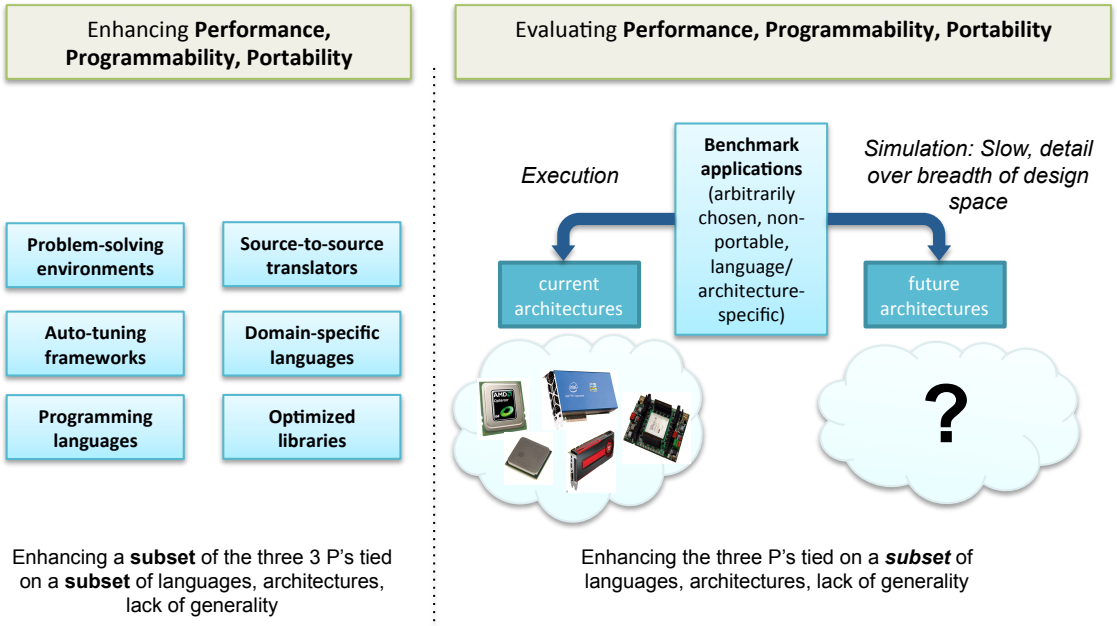
In short, the three key research questions we identify and address in this work are the following:

(1) What are the performance implications of architectural features found in modern heterogeneous architectures, and what is an appropriate means for their systematic identification? How can we extend such a methodology to determine the future of heterogeneous architectures?

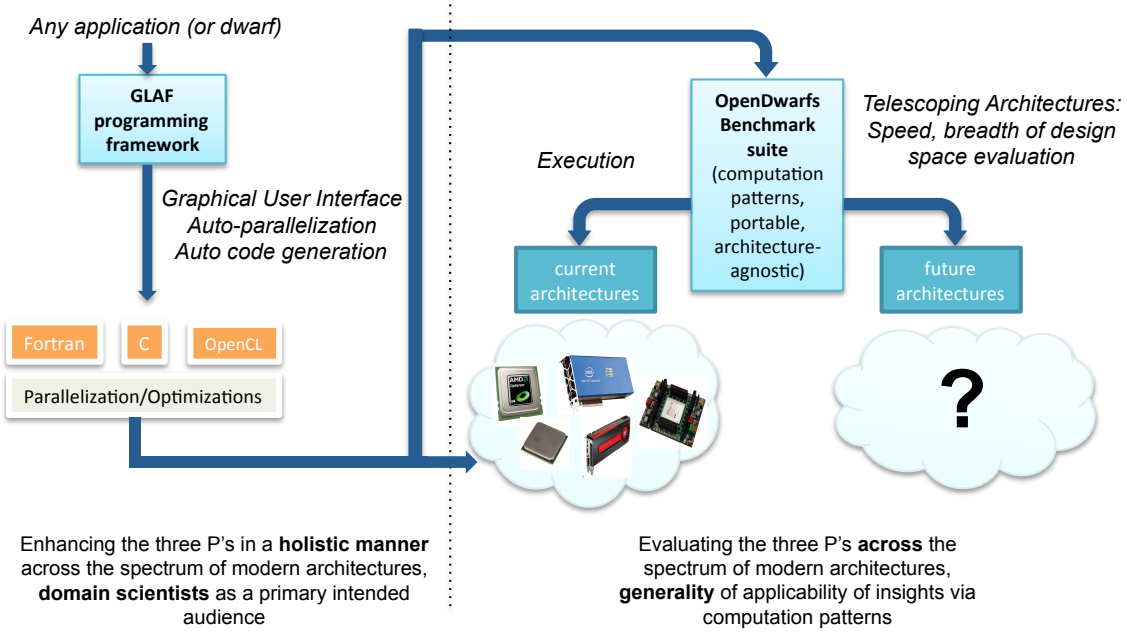
(2) What are the trade-offs between performance, programmability, and portability for heterogeneous computing, given the complex ecosystem of languages, compilers, tools, and optimization methods?

(3) How and to what degree can we enhance performance and programmability and ensure functional portability and a certain level of performance *portability* via tools and frameworks?

Figure 1.1 depicts the scope of the above questions and the utility of our work in the context of the hardware/software stack. We discuss the details of prior work and differences to our approach in more detail in the *Related Work* chapter (Chapter 3).



(a) Existing approaches for the three P's



(b) Proposed approaches for the three P's

Figure 1.1: High-level overview: Existing and proposed approaches to evaluating and enhancing performance, programmability, and portability of current and future heterogeneous platforms

1.3 Contributions

This dissertation seeks to shed light on the conflicting issues of performance, programmability, and portability (functional and performance) in the context of an ever-evolving heterogeneous computing landscape. The overarching contributions of this PhD dissertation are the following:

- We present a multi-targeted *performance* evaluation on a subset of uniformly optimized OpenCL dwarfs across a diverse set of parallel platforms [176, 175] (Section 4.1). Subsequently, we characterize performance of gradually optimized implementations of an n-body dwarf across the programming languages and platforms spectrum [180, 177] (Sections 4.2 and 4.3) and use OpenDwarfs as a means to identify trends towards future heterogeneous architectures [174] (Section 4.4).

- We present a study on programmability and the trade-offs between performance, programmability, and portability in a multi-dimensional study that includes multiple optimization levels, programming languages and techniques, and target architectures to include CPUs, GPUs, Intel MIC [180] (Section 5.2).

- We attempt to enhance performance, programmability, and portability by proposing the introduction of grid-based data structures and presenting a prototype implementation of a framework, called GLAF, that seeks to address the three P's for CPUs, Intel MIC, GPUs, and FPGAs [178, 179] (Section 5.3), in a holistic approach by encompassing a novel visual programming environment and auto-parallelizing, auto-tuning framework. In addition

to dwarf-based benchmarks, we use GLAF to develop and evaluate a large-scale satellite imaging application by NASA (Section 5.4).

1.4 Outline

The rest of the dissertation is organized as follows:

Chapter 2 discusses relevant background information and Chapter 3 presents related work. Chapters 4 and 5 contain the core of our research contributions, as discussed above. Specifically, in Chapter 4 we explore issues related to performance from various standpoints by using OpenDwarfs and discuss how insights drawn can benefit different interest groups. We examine performance obtained by applying architecture-aware optimizations in the context of a specific dwarf (n-body) across current heterogeneous platforms and programming languages. We then go beyond existing architectures and propose a dwarf-based methodology for evaluating future deeply heterogeneous architectures. Chapter 5 extends Chapter 4 by exploring the dimension of programmability and the entailed trade-offs in the context of an n-body dwarf, as well as portability. In order to narrow the identified gaps between performance, programmability and portability, we set forth to address the issue of enhancing programmability and portability by introducing a grid-based programming abstraction in the context of frameworks like GLAF, our relevant research prototype. Finally, Chapter 6 summarizes this dissertation and proposes related future research work.

Chapter 2

Background

This work seeks to address the issues of performance, programmability and portability in the context of heterogeneous architectures. In this chapter we provide background information on GPUs, the Intel Xeon Phi co-processor and FPGAs, key representatives of heterogeneous architectures. Also, we introduce OpenCL, Altera OpenCL and SOpenCL, i.e., major programming languages and tools at the disposal of programmers of such architectures that we will encounter throughout this work.

2.1 Heterogeneous Architectures

In this section we present an architectural overview of heterogeneous architectures that we use throughout this dissertation, namely the general-purpose graphics processing unit (GPGPU), the Intel Many Integrated Cores architecture (MIC), and the field-programmable gate array (FPGA).

2.1.1 General-Purpose Graphics Processing Unit (GPGPU)

Graphics Processing Units (GPUs) have been traditionally used for graphics rendering. With the advent of programmable shaders, a GPU programmer could recast certain problems in terms of graphics primitives. This process, using graphics-oriented APIs (OpenGL, DirectX), was low-level and counter-intuitive. The GPU architectures have since evolved to support general-purpose computation using higher-level programming languages, such as Brook [49], originally, and CUDA and OpenCL [19], subsequently.

A GPU is in fact a *stream processor*. Each of the two major GPU manufacturers employs different terminology for the building block of a GPU: AMD uses the term *compute unit*, while NVIDIA dubs it *streaming multiprocessor*. Both are effectively wide SIMD processors. The GPU further differentiates itself from the CPU via its memory design, which consists of a hierarchy of *manually* managed memories along with *relaxed coherence semantics*. The most abundant type of memory is global memory whose bandwidth and latency are both high. Moreover, GPUs contain a cache hierarchy, where the lower-level cache usually functions as software-managed scratch space. The way such caches are divided among compute units/streaming multiprocessors is highly dependent on the GPU architecture family.

NVIDIA, AMD, and Intel are the main vendors manufacturing GPGPUs. The former two provide both *discrete* and *integrated* GPU variants targeting the computer gaming community, as well as the HPC segment, while Intel mainly targets the laptop market with its integrated GPU line. *Discrete GPUs* come in the form of PCIe cards and data transfers between the GPU and the host CPU

are required due to memory spaces' physical separation. *Integrated GPUs* (as part of *Accelerated Processing Units (APUs))* are physically located on the same die with the CPU cores, hence obviating the need of (costly) PCIe communication.

All three vendors support programming their GPUs using OpenCL. Due to its extended use in our work, we provide more details for OpenCL in Section 2.2.1. NVIDIA GPUs can also be programmed using CUDA, which bears many similarities to OpenCL. CUDA is NVIDIA's flagship programming model for the GPU. It is a C-like parallel language with support for single-instruction, multiple-thread (SIMT) processing. A CUDA program comprises a host part executing on the CPU and one (or more) kernel(s) that are executed on the GPU. The level of parallelization depends on how the programmer partitions computation in grids, thread blocks, and threads. At the highest level, a kernel consists of a number of blocks organized into grids, wherein each block is further composed of threads.

Throughout our work, we make use of GPUs from both NVIDIA and AMD that span both categories mentioned above (i.e., discrete, integrated). To illustrate the basic GPU features presented above, we provide further discussion for NVIDIA K20c, which belongs to the Kepler microarchitecture family. K20c is a representative GPU used in the HPC segment (many architectural concepts from NVIDIA GPUs are found in AMD ones, and vice versa). We use this specific GPU extensively in our evaluations in Sections 4.2 and 5.2.

Figure 2.1 provides an architectural block diagram of the NVIDIA Kepler K20 discrete GPU. K20 contains a number of next-generation streaming multiprocessors (SMXs), which effectively serve as "wide SIMD" processors, ranging in width from 32 to 192. Each SMX includes 64KB

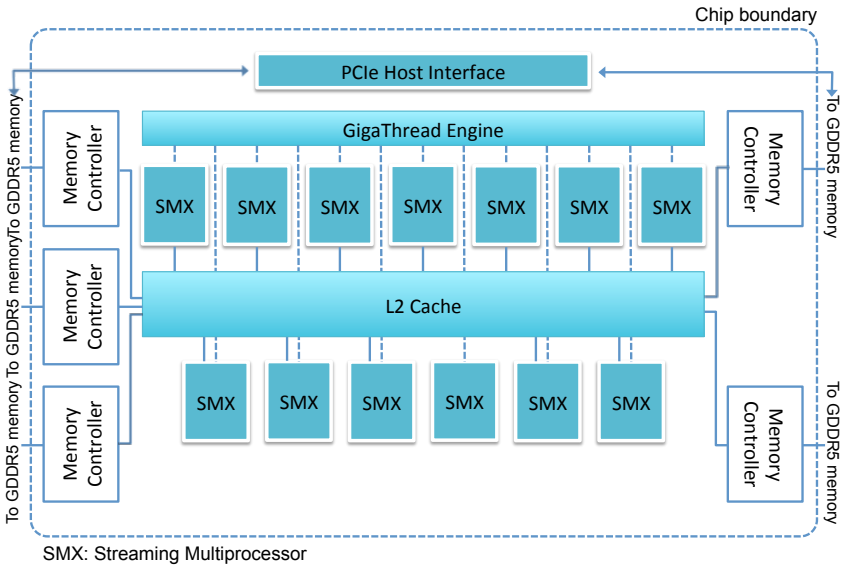


Figure 2.1: NVIDIA Kepler K20 block diagram

of configurable L1 cache that can also be used as software-managed scratch space (i.e., *shared memory*). Each block's threads are executed in parallel on a single SMX in groups of 32, called warps. In Kepler, four warp schedulers per SMX orchestrate the scheduling of warps for execution on available execution units. Having a large number of blocks (and hence warps) allows the high latency to global memory to be hidden, as the scheduler brings in another available warp while the other stalls waiting for data from global memory. For SIMT operation, if all the threads of a warp execute the same instruction, then GPU resources are fully utilized. When threads do not execute the same instruction at any given moment (e.g., because of data-dependent conditional branches), execution is serialized, thus adversely impacting performance.

2.1.2 Intel Many Integrated Cores Architecture (MIC)

Intel Xeon Phi (codenamed *Knight's Corner*) is the first commercially available product in Intel's MIC co-processor line. It integrates the low-latency of multi-core CPUs and higher-latency/higher-throughput of vector-like computing (*a la* GPU) for co-processing.

Figure 2.2 shows an architectural diagram of Intel Xeon Phi, which is realized as a PCIe card. Whereas a traditional Intel Xeon contains 2, 4, or 8 cores clocked at a high frequency (e.g., 3.0 GHz), Intel Xeon Phi contains an order of magnitude more processors (i.e., 61), where each processor features four-way multi-threading while running at a pedestrian 1.09 GHz, for a total of 244 hardware threads. Although each of the lower-clocked CPU-based cores of Xeon Phi is based on an older Pentium core design, each core comes with enhancements that differentiate it from the traditional Xeon. In addition to a 64-bit architecture, it features a dual-issue, in-order execution pipeline, where a 512-bit-wide vector and a scalar unit can accommodate corresponding instructions simultaneously.

The memory hierarchy consists of 32KB of 8-way L1 instruction and data caches and a 512KB unified L2 cache per core for a total of 32MB of fully coherent L2 cache per card as well as 8GB GDDR5 memory.

For communication, a specialized version of the Intel ring bus interconnects the Xeon Phi cores together. For communication outside of Xeon Phi, each Xeon Phi card can communicate with the host CPU using TCP/IP over PCIe. For details on the MIC architecture, see [74].

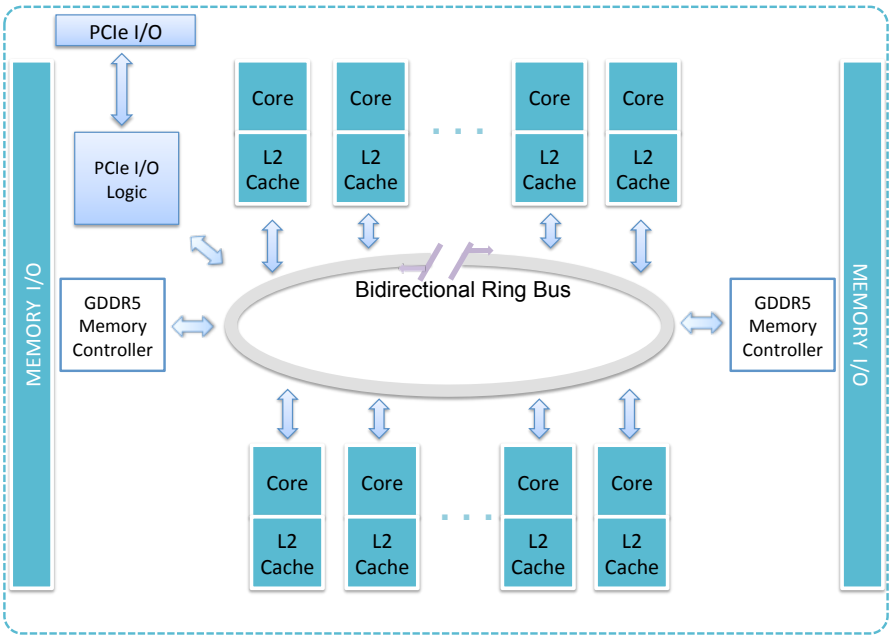


Figure 2.2: Intel Xeon Phi block diagram

Relative to programmability, Xeon Phi targets the portability and performance of both legacy code and new code by adopting a parallel programming paradigm that is compatible with the x86 instruction set architecture (ISA). Using C/C++ with OpenMP or Pthreads is a typical way of programming Xeon Phi, while other methods using proprietary software like Intel Cilk [241] or Thread Building Blocks (TBB) [231] can be used. Xeon Phi runs a minimal operating system, which can be used to treat the card itself as a distinct computer in its own right. Xeon Phi's uOS is based on the Linux kernel with BusyBox and other basic Linux services (e.g., services required to communicate with the host). The above allows native execution on the co-processor, in addition to an offload computational model that is similar to that of a GPU. Specifically, the parallel part of the code is offloaded to the co-processor by use of simple compiler directives.

2.1.3 Field-Programmable Gate Array (FPGA)

Compared to the fixed hardware of the CPU and GPU architectures, FPGAs (*Field-Programmable Gate Arrays*) are configured post-fabrication. FPGAs are configurable integrated circuits (ICs) that include large numbers of adaptive logic modules (ALMs), digital signal processing (DSP) blocks, as well as memory blocks. An FPGA board may include PCIe, Ethernet, or other interfaces, memory controller interfaces, as well as an integrated CPU to form a system on chip (SoC) solution. "Programming" the FPGA, entails setting the configuration bits that specify the functionality of the configurable high-density arrays of *blocks* mentioned above and the programmable routing channels (*reconfigurable interconnects*) between them.

FPGAs offer the highest degree of flexibility in tailoring the architecture to match the application, since they essentially emulate the functionality of an ASIC (Application Specific Integrated Circuit). Moreover, they avoid the overheads of the traditional ISA-based von Neumann architecture followed by CPUs and GPUs and can trade-off computing resources and performance by selecting the appropriate level of parallelism to implement an algorithm. Since reconfigurable logic is more efficient in implementing specific applications than multicore CPUs, it enjoys higher power efficiency than any general-purpose computing substrate.

The main drawbacks of FPGAs are two-fold. First, they are traditionally programmed using Hardware Description Languages (VHDL or Verilog), a time-consuming and labor-intensive task, which requires deep knowledge of low-level hardware details. In this work we abstain from using VHDL or Verilog. Instead, we turn to using SOpenCL [225] and Altera OpenCL [25]. The former

constitutes one of the first attempts to use OpenCL as a means to describe hardware and targets Xilinx FPGAs. The latter is the first commercial attempt, by Altera, to use OpenCL for programming Altera FPGAs. Using these higher-level languages we alleviate the burden of implementing accelerators in FPGAs by utilizing the same (or similar) OpenCL code-base used for CPU and GPU programming. Second, the achievable clock frequency in reconfigurable devices is lower (by an order of magnitude) compared to high-performance processors. In fact, most FPGA designs operate in a clock frequency less than 200 MHz, despite aggressive technology scaling. As we see in next chapters, this does not – by itself – imply worse performance.

FPGAs have many applications in diverse fields. Examples include aerospace and defense, automotive industry, audio/video/image processing and wireless communications. The cost and performance of FPGAs make them a potentially cost-efficient alternative over ASICs for certain applications. FPGAs have also started being used in the context of large-scale datacenter services (specifically Microsoft Bing search-engine) [237]. With Altera's acquisition by Intel, it is expected that FPGA fabric will make its way onto the CPU die with the intent to act as a reconfigurable accelerator for specific compute- or data-intensive tasks.

2.2 Programming Languages for Heterogeneous Architectures

In this section we discuss three main means of parallel programming of heterogeneous systems that we extensively use in this dissertation. Specifically, we introduce OpenCL and Altera OpenCL – its FPGA-specific instantiation. Last, we present Silicon OpenCL (SOpenCL), a means of pro-

gramming FPGAs using OpenCL that predates commercial FPGA-specific implementations of OpenCL.

2.2.1 OpenCL

OpenCL provides a parallel programming framework for a variety of devices, ranging from conventional Chip Multiprocessors (CMPs) to combinations of heterogeneous cores such as CMPs, GPUs, and FPGAs. OpenCL (Open Compute Language) represents a coordinated standardization effort to allow cross-platform (e.g., CPU, GPU, MIC, FPGA) programming in support of a truly unified heterogeneous ecosystem. Key hardware vendors spanning the CPU, GPU, FPGA domains provide their OpenCL implementation, allowing code written for one platform to be (functionally) portable to another. OpenCL is based on a subset of ISO C99 with appropriate extensions to accommodate expressing parallelism, data transfers between the host/device, error-management, etc.

Its platform model comprises a *host* processor and a number of *compute devices*. Each device consists of a number of compute units, which are subsequently subdivided into a number of processing elements. An OpenCL application is organized as a *host program* and a number of *kernel functions*. The host part executes on the host processor and submits commands that refer to either the execution of a kernel function or the manipulation of memory objects. Kernel functions contain the computational part of an application and are executed on the compute devices. The work

that corresponds to a single invocation of a kernel is called a *work-item*. Multiple work-items are organized in a *work-group*.

OpenCL allows for geometrical partitioning of the grid of independent computations to an N-dimensional space of work-groups, with each work-group being subsequently partitioned to an N-dimensional space of work-items, where $1 \leq N \leq 3$. Once a command that refers to the execution of a kernel function is submitted, the host part of the application defines an abstract index space, and each work-item executes for a single point in the index space. A work-item is identified by a tuple of IDs, defining its position within the work-group, as well as the position of the work-group within the computation grid. Based on these IDs, a work-item is able to access different data elements (SIMD style) or follow a different path of execution.

Data transfers between host and device occur via the PCIe interface in the cases of discrete GPUs and other types of co-processors like Intel Xeon Phi. In such cases, the large gap between the (high) computation capability of the device and the (comparatively low) PCIe bandwidth may incur significant overall performance deterioration. The problem is aggravated when an algorithmic pattern demands multiple kernel launches between costly host-to-device and device-to-host data transfers. Daga et al. [79] re-visit Amdahl's law to account for the parallel overhead incurred by data transfers in accelerators like discrete or fused GPUs. Similar behavior, with respect to restricting available parallelism is observed in CPUs and APUs, too, when no special considerations are taken during OpenCL memory buffer creation and manipulation. In CPUs (CPU-as-device scenario) with appropriate buffer allocation and transfers the data transfer part can be practically eliminated. In APUs, due to the tight coupling of the CPU and GPU core on the same die, and

depending on the exact architecture, more data transfer options are available for faster data transfers between the CPU and GPU side. Lee et al. [184] and Spafford et al. [262] have studied the tradeoffs of fused memory hierarchies.

2.2.2 Altera OpenCL (AOCL)

Altera was the first FPGA vendor to support programming its FPGAs using OpenCL. Specifically, Altera provides a compiler (*AOC*) and a set of tools (including a profiler) that a programmer can use to build and run OpenCL applications on Altera FPGAs. AOCL supports the embedded profile of version 1.0 of the OpenCL specification, however it extends it with support for *AOCL channels* and features of OpenCL 2.0 (such as *shared virtual memory (SVM)* and *OpenCL pipes*).

The Altera OpenCL toolset supports a software-like programming workflow. Specifically, Altera recommends OpenCL development in stages, rather than single-step development and compilation. The main reason is that full compilation is in the order of hours (depending on the complexity of the OpenCL kernels). Thus, a simple error in the implemented algorithm would require re-compilation of the code if found after the fact. On the other hand, *multistep AOCL design flow* splits the design process into four discrete steps, highlighting algorithmic design rather than low-level hardware implementation details:

(a) *Intermediate compilation:* Pertains to a first-order compilation that does not generate the hardware configuration file (.aocx). Rather, it generates the appropriate Verilog files that describe the input OpenCL kernel(s) in Hardware Design Language (HDL) and provides an original estimation

about the hardware resources that will be used (based on the target board that the compilation takes place for). This step is fast (in the order of minutes), identifies any syntax errors and generates an *Altera Offline Compiler Object file (.aoco)* that can be used in step (b), that is *emulation*. Along with the above object file, it generates other intermediate files with useful information, among which an optimization report. Information in this file can be used to further optimize the OpenCL kernel code.

(b) *Emulation:* In this step the user can emulate the kernels that were compiled in step (a) on an x86 host and detect any functional errors in the code. Symbolic debugging (for object files compiled with the appropriate flag) is possible and facilitates locating errors in the implemented algorithm.

(c) *Profiling:* After ensuring syntactic and functional correctness of the OpenCL kernel code, the user can re-compile the kernel code with a profiling flag. This flag instructs AOC to instrument performance counters in the generated hardware, i.e., the Altera Offline Compiler Executable file (.aocx). This compilation step is in the order of hours. On execution, information from performance counters is collected in a utility file (.mon) and can be used within the provided profiler to identify performance bottlenecks.

(d) *Full deployment:* The above steps (a)-(c) are iterated over until satisfactory kernel performance is obtained. Once this is done, the user can procede to the last step. In this step AOC performs full compilation to generate the final .aocx file that will be used.

The resulting file (.aocx) of the above workflow corresponds to the OpenCL kernel file (.cl) and is executed on the FPGA. Once we obtain the .aocx file, OpenCL on Altera FPGAs follows the

same execution flow as its GPU counterparts (Section 2.2.1), the only difference being that the *device* is now the FPGA instead of the GPU. The host program that runs on the x86 host (or an on-board CPU in system-on-chip (SoC) solutions) is compiled using gcc/g++ and linking with the appropriate Altera libraries. In the general case, execution starts on the host, where appropriate initializations, memory allocation on the FPGA device and data transfers take place, and then kernel execution is invoked on the FPGA. Once computation is over, output data is transferred back to the host. The above is illustrated in Figure 2.3.

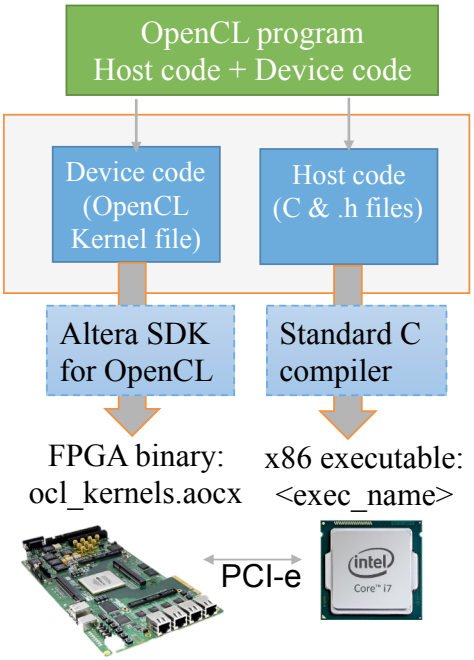


Figure 2.3: Altera OpenCL execution flow

Kernel execution on the hardware level on the FPGA fundamentally differs from that on GPUs. Specifically, the latter follows a SIMT (Single Instruction Multiple Threads) paradigm, where

parallel threads execute the same instruction on different input data concurrently. In FPGAs parallelism is mainly exposed as *pipeline parallelism* and is achieved via deeply pipelined hardware circuitry; that is generated hardware circuitry handles different stages of the kernel for different input data in a pipelined fashion. Depending on available hardware resources each pipeline can be replicated in order to achieve even higher degrees of parallelism. Due largely to the fundamental differences discussed above and the fact that "programming" FPGAs effectively implies *configuring* re-configurable hardware (rather than *utilizing* fixed hardware), optimization strategies for FPGAs are quite different than the GPU ones.

2.2.3 Silicon OpenCL (SOpenCL)

SOpenCL [225] is the first academic attempt to automatically generate hardware accelerators using OpenCL, thus dramatically minimizing development time and increasing productivity. SOpenCL enables quick exploration of different architectural scenarios and evaluation of the quality of the design in terms of computational bandwidth, clock frequency, and size. The final output of this procedure is synthesizable Verilog, functionally equivalent to the original OpenCL kernels, which can in turn be used to configure an FPGA. Despite the above merits of SOpenCL, and the advantages of high-level synthesis (HLS) tools in general, a well-thought manual hardware implementation in a hardware design language by an experienced designer is expected to be better-performing. The large span of potential solutions, stemming from the customization capabilities of the FPGA fabric, further exacerbates the job of a HLS tool.

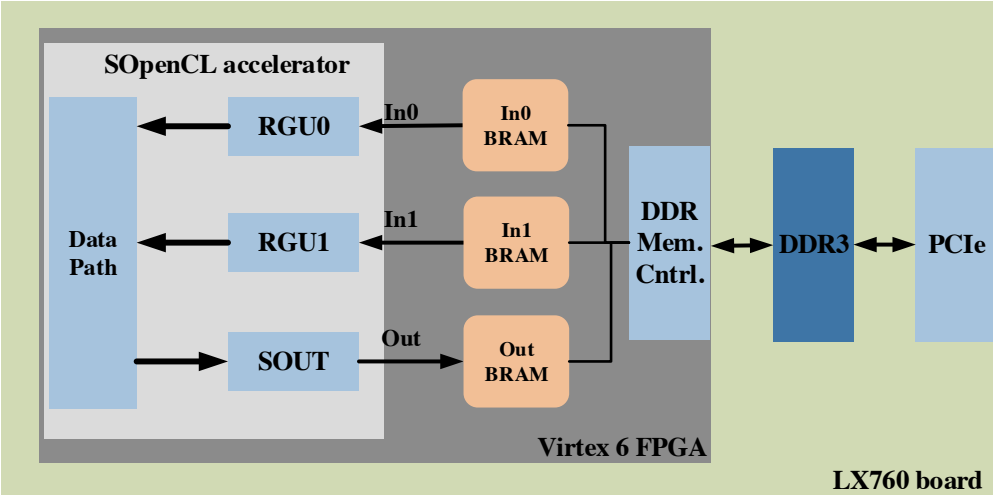


Figure 2.4: FPGA memory organization

One of the design choices lies, for example, in the number of accelerators that may fit in a given FPGA. Figure 2.4 shows a single-accelerator SOpenCL design in a Xilinx LX760 board with a Virtex 6 FPGA. In a SOpenCL design the program data inputs are initially transferred to the DDR3 board-level memory from where they are transferred to the on-chip FPGA BRAMs. Likewise, all outputs are transferred from the BRAMs back to the DDR3 memory. For a single FPGA accelerator, input data are stored sequentially in BRAMs without any special partitioning across multiple BRAM banks. This is typically the biggest obstacle for achieving high bandwidth and, hence, high performance. For multiple accelerators, manual partitioning of the data across multiple BRAMs is required in order to exploit the increased bandwidth requirements.

SOpenCL includes a *front-end* that is a source-to-source compiler that adjusts the parallelism granularity of an OpenCL kernel to better match the hardware capabilities of the FPGA. OpenCL kernel code specifies computation at a work-item granularity. A straightforward approach would map a work-item to an invocation of the hardware accelerator. This approach is suboptimal for FPGAs

which incur heavy overhead to initiate thousands of work-items of fine granularity. SOpenCL, instead, applies source-to-source transformations that collectively aim to coarsen the granularity of a kernel function at a work-group level. The main step in this series of transformations is logical thread serialization. Work-items inside a work-group can be executed in any sequence, provided that no synchronization operation is present inside a kernel function. Based on this observation, the execution of work-items is serialized by enclosing the instructions in the body of a kernel function into a triple nested loop, given that the maximum number of dimensions in the abstract index space within a work-group is three. Each loop nest enumerates the work-items in the corresponding dimension, thus serializing their execution. The output of this stage is a semantically equivalent C code at the work-group granularity.

SOpenCL's *back-end* flow is based on the LLVM compiler infrastructure [183] and generates the synthesizable Verilog for synthesizing the final hardware modules of the accelerator. The functionality of the back-end supports *bitwidth optimization*, *predication*, and *swing modulo scheduling* (SMS) [197] as separate LLVM compilation passes: (a) *Bitwidth optimization* is used to minimize the width of functional units and wiring connecting them, to the maximum expected width of operands at each level of the circuit, based on the expected range of input data and the type of operations performed on input and intermediate data. Experimental evaluation on several integer benchmarks shows significant area and performance improvement due to bitwidth optimizations. (b) *Predication* converts control dependencies to data dependences in the inner loop, transforming its body to a single basic block. This is a prerequisite in order to apply modulo scheduling in the subsequent step. (c) *Swing modulo scheduling* is used to generate a schedule for the inner loops.

The scheduler identifies an iterative pattern of instructions and their assignment to functional units (FUs), so that each iteration can be initiated before the previous ones terminate. SMS creates software pipelines under the criterion of minimizing the Initiation Interval (II), which is the constant interval between launches of successive work-items. Lower values of Initiation Interval correspond to higher throughput since more work-items are initiated and, therefore, more results are produced per cycle. That makes the Initiation Interval the main factor affecting computational bandwidth in modulo scheduled loop code.

Chapter 3

Related Work

Heterogeneous computing has seen important advances and increasing adoption since the mid-2000s, and significant work has been done that spans the areas of performance, programmability and portability – the three P's we seek to address in this dissertation. To better understand the research themes in the above areas, and put our work in perspective, we conduct a thorough survey of related research published in related venues. [1]

First, we discuss work that adress various aspects of performance (Section 3.1), and then programmability and portability (Section 3.2). As the above aspects are deeply interrelated (per our motivation discussion in Chapter 1), one should keep in mind that the works presented in one category may have direct implications in another, too. For example, auto-tuning research can have

[1] PPoPP, HPDC, PACT, ICS, PoPL, CGO, ASPLOS, IPDPS, OOPSLA (SPLASH), PLDI, SC, CCGrid, IISWC, ISC, CF, Cluster, ICSE, ICSM

positive impact in both performance and programmability and may be presented in the most appropriate category below.

3.1 On Performance

In this section we discuss prior research works that focus on the aspect of performance. First, we present related work on performance evaluation and benchmarking, and then we highlight prior work on enhancing performance via software or hardware approaches.

3.1.1 Performance Evaluation and Benchmarking

Heterogeneous computing has emerged to address limitations in transistor scaling, prohibitive heat dissipation levels, and power/energy related constraints. The need to circumvent the above roadblocks ultimately lies on the ongoing demand for faster devices, i.e., increased performance. In order to address the issue of performance from various perspectives (e.g., programmer's, software's, tools', and hardware's), HPC engineering and research have highlighted the importance of developing *benchmark suites*. In [141] Hoefler and Belli outline twelve ways to improve measuring and reporting parallel computing performance results.

Benchmarks have traditionally followed a design concept in which different applications stress different subsystems of a computing device (such as memory, network, processor) or different capabilities within a subsystem (e.g., integer or floating point arithmetic in the CPU). While useful

for the purpose they were created, traditional benchmark suites (e.g., EEMBC [101], SPEC [263], PARSEC [42], ALPBench [192]) are of limited utility in the context of heterogeneous computing for various reasons. Specifically, such suites are written in languages that target CPU execution and cannot execute on heterogeneous architectures. Also, they tend to focus on concrete implementations of specific applications, which is inherently restrictive in that it goes against the very concept of generality required to capture trends in parallel computing.

The emergence of OpenCL as a common programming model for heterogeneous architectures facilitated a solution towards the first problem mentioned above. To address the latter shortcoming of traditional benchmark suites, a different approach has been proposed, which entails benchmarks that capture high-level computation and communication patterns in an attempt to stress heterogeneous devices in a holistic way. In [256] the authors emphasize the need for benchmarks to be related to scientific *paradigms*, where a paradigm defines what the important problems in a scientific domain are and what the set of accepted solutions is. This notion of paradigm parallels that of the *computational dwarf*. A dwarf (or *motif*) is an algorithmic method that encapsulates a specific computation and communication pattern. The seven original dwarfs, attributed to P. Colella's unpublished work, became known as *Berkeley's dwarfs*, after Asanovic et al. [28] formalized the dwarf concept and complemented the original set of dwarfs with six more. Based in part on the dwarfs, Keutzer et al. later attempted to define a pattern language for parallel programming [166].

Various new benchmark suites have emerged in an attempt to target heterogeneous architectures. Notable ones include *Rodinia* [63], *OpenDwarfs* [107], *SHOC* [84], *Parboil* [244] and *SPEC ACCEL* [161]. Of these benchmark suites, only two follow the dwarfs classification. **Rodinia** [63],

originally released in 2010, was built around the concept of dwarfs and included applications written in CUDA (to target GPU architectures) and OpenMP (for shared-memory multi-processors). Starting in 2013, Rodinia began translating their benchmarks to OpenCL. **OpenDwarfs** [107] (originally known as "OpenCL and the 13 Dwarfs" [2]) was first released in 2012 and was the first benchmark suite to provide a thorough collection of dwarf-based benchmarks written in OpenCL, which enabled seamless cross-platform support. Part of the benchmarks were developed in-house, while others were derived from corresponding CUDA implementations in Rodinia. The newer version of OpenDwarfs extends support to Altera FPGAs via Altera OpenCL.

The remaining benchmark suites do not follow the dwarfs classification. **SHOC** [84] (Scalable Heterogeneous Computing) benchmark suite does not follow the dwarfs classification. Its applications, written in OpenCL and CUDA, are divided into two categories: a) stability tests that stress OpenCL devices by running computationally demanding kernels, and b) performance tests that are divided into three levels that span from testing low level device characteristics to high-level device performance via real application kernels. **Parboil** [244], developed by the IMPACT Research Group at University of Illinois at Urbana-Champaign in 2008, originally included CUDA implementations of benchmarks derived from various sources. Later [269], it evolved to include OpenCL implementations (including architecture-aware optimized versions). While Parboil includes benchmarks that span application domains, it is not explicitly organized in dwarf categories. However, at the core of its design philosophy lies the concept of providing largely architecture-agnostic implementations in addition to GPU-optimized ones. **SPEC ACCEL** [161], released in

[2] OpenDwarfs has been a collaborative effort within the Synergy Lab at Virginia Tech

2014 by the SPEC High Performance Group (HPG), contains two application suites in OpenCL (19 benchmarks) and OpenACC (15 benchmarks). Both SPEC ACCEL suites are largely derived from Parboil and Rodinia, as well as NAS Parallel Benchmarks [218, 33] (originally in MPI and OpenMP).

According to a similar approach, benchmarks seek to capture the behavior of real-world, full-scale applications, assuming the role of a proxy in terms of application characteristics (*proxy-apps*). Notably, they do so by obfuscating any details unecessary in terms of the algorithm. Benchmarks can range from larger applications to minimal ones (or mini-apps). Department of Energy (DOE) labs (e.g., Livermore, Sandia, and Los Alamos National Labs) maintain repositories of proxy applications of interest, for example Lawrence Livermore National Lab (LLNL) [2], Los Alamos National Lab (LANL) [11] and Sandia National Lab [12].

While benchmark suites, contain collections of applications, as discussed above, there is the possibility of certain levels of *redundancy*. In [67] the authors explore the "characteristics of workloads used in high performance and technical computing" and seek to quantify the diversity of such performance characteristics and compare with commercial applications. They explore the memory access patterns of the benchmarks under consideration (like NAS Parallel Benchmarks) and analyze their instruction decomposition. A similar approach is followed in [130], where the authors present a framework to automatically recognize performance idioms in scientific applications. Via a detailed study, they conclude that the proposed idioms (similar in concept to the dwarfs idea) can fully cover 100% of six NAS Parallel Benchmark benchmarks and that performance approximations of the full benchmarks using the idioms can be possible. Computation and communication

patterns or idioms are one way of identifying diversity (or lack thereof) in benchmarks. Other studies [88] use the concept of codelets, i.e., small fragments of code that do not overlap. Such codelets may be general enough to coincide with idioms (as in [130]) or even dwarfs, or much simpler code fragments. With codelets redundancy can be identified at the granularity of a single benchmark rather at the granularity of a benchmark suite. The latter case corresponds to studies, which identify similarities in benchmark suite components via characterization and diversity analysis, a more formal process of comparing applications to one another in the context of a benchmark suite. Examples include [160, 230], which examine benchmark similarity and redundancy in SPEC CPU2006, and show that a subset of the benchmarks can be used to estimate certain average benchmark characteristics. In [117] the authors conduct diversity analysis on the NVIDIA CUDA SDK, Parboil, and Rodinia for the GPU, while in [41, 64] the authors focus on SPLASH-2 and Rodinia, respectively, and its similarities with Parsec on the CPU and GPU. In [20] Adhinarayanan et al. perform diversity analysis and subset GPGPU workloads in SPEC ACCEL, as well as other benchmark suites. As shown above, it is a fact indeed that benchmark suites are typically characterized by a certain degree of redundancy. Eliminating such redundancy is beneficial, especially in the cases of architectural design, where simulation time can be orders of magnitude higher compared to execution time on actual hardware. In the latter case, it may be faster to execute more benchmarks (i.e., even redundant one) than perform a diversity analysis and subsetting process.

From a practical standpoint, one of the main uses of benchmark suites is to characterize architectures. In works like [63, 64, 84, 244, 248] and [269] the authors discuss architectural differences between contemporary CPUs and GPUs of the time using many of the benchmark suites that we

mentioned before. One of the main limitations of these original works is that they mostly focus on CUDA benchmark implementations and NVIDIA GPUs. Works like [198, 223] extend characterization of heterogeneous platforms on more recent architectures, like Intel MIC, Kepler-based NVIDIA GPUs, etc. and provide comparisons with the OpenACC programming model. A more detailed discussion on the implications of architectural features with respect to algorithms and insight on future architectural design requirements is given in [191], while in [212] the authors conduct a preliminary study with a subset of Rodinia benchmarks on the Intel Xeon Phi co-processor. With respect to FPGAs, there is early work done with respect to using OpenCL as a programming model for hardware design and comparing its performance with hardware design languages (HDLs) or high-level synthesis (HLS) languages. Most works focus on a single application case study. Examples include [249, 257, 65, 110]. OpenDwarfs was one of the first benchmark suites to be used for evaluating Xilinx FPGAs using OpenCL via SOpenCL [3] and to later include support for Altera FPGAs via OpenCL. Another work towards this direction is [220]. As we mention in Chapter 1 benchmarks constitute the basis for multi-faceted studies on performance that target computer architects, compiler, tool, run-time systems developers, and programmers. There is a myriad of examples for each of the above categories; below we provide just a representative sampling. Benchmark suites, mainly Rodinia and Parboil, have been widely used to characterize performance of heterogeneous architectures and suggest alternatives with respect to a diverse set of architectural features, such as memory architecture [209, 156, 113] and pipelines [136]. With respect to run-time systems, compilers and tools, examples include Starchart [157], OpenARC [190], automatic memory management in GPUs using compiler-assisted runtime [226], a

[3] FPGA work contributed by Dr. M. Owaida, Dr. C. D. Antonopoulos, and Dr. N. Bellas

high-level programming model [96], and scheduling work [185, 247]. Various efforts in determining optimization techniques for programmers to use with multi-core CPUs and GPUs have been reported. Detailed studies on optimization techniques for the CPU and GPU, along with architectural comparisons with respect to performance differences are presented in [191, 267, 245]. None of the above work considers the optimization search-space for Intel Xeon Phi.

In our attempt to evaluate performance across heterogeneous platforms and address the issue from multiple standpoints, we utilize benchmarks that follow the dwarfs categorization, specifically the OpenDwarfs benchmark suite. Such a benchmark suite, whose application selection delineates modern parallel application requirements, can constitute the basis for comparing and guiding hardware and architectural design. As we mention above, on a parallel path with OpenDwarfs, which was based on OpenCL from the onset, many existing benchmark suites were re-implemented in OpenCL and new ones were released (e.g., Rodinia [63], SHOC [84]). Most of them were originally developed as GPU benchmarks, translated to OpenCL from CUDA implementations, and as such still carried optimizations that favor GPU platforms (and in most cases NVIDIA-based). OpenDwarfs, too, itself originally belonged to this category. This, however, violates the *portability* requirement for benchmarks that mandates a lack of bias for one platform over another [28, 256] and prevents drawing broader conclusions to be drawn with respect to hardware innovation. In our work we *revise* and *extend* the original OpenDwarfs [107] in an attempt to present an all-encompassing benchmark suite for heterogeneous computing. Specifically, we build on the dwarf-based categorization of applications, also followed in Rodinia, but at the same time provide *architecture-agnostic* implementations, like Parboil, thereby combining the best features

of the two and making OpenDwarfs the first all-around complete benchmark suite for heterogeneous computing. We describe our contributions to the original OpenDwarfs benchmark suite in detail in Section 4.1 in detail.

In this work (Section 4.1), [4] we complement prior research by characterizing OpenDwarfs on a diverse set of modern parallel architectures, including CPUs, Accelerated Processing Units (APU), discrete GPUs, the Intel Xeon Phi co-processor, as well as on Xilinx FPGAs. As a matter of fact, the parts of the work that focus on AMD APUs and Intel Xeon Phi (code-named Knight's Corner) are among the first of their kind and related experiments were able to evaluate the corresponding architectures before their official release. In contrast to most related work that stresses the optimizations required for performance, our evaluation work with OpenDwarfs focuses on architecture-agnostic benchmark implementations in order to highlight the computation and communication patterns themselves. The trend of incorporating GPUs and co-processors like Intel Xeon Phi in computer clusters made such up-to-date studies imperative (four supercomputers in Top500 list's top ten [276] make use of such accelerators). Even more so, when the benchmarks used in such studies capture characteristics of real-world applications (i.e., benchmarks based on the dwarf concept) that such systems are routinely used for.

[4] Most of this work predates or is contemporary to the most recent ones mentioned in this chapter.

3.1.2 Optimizing Performance: Software Approaches

Optimizing performance via software approaches can be split into two large categories: (a) general tools with broader scope and (b) enhancing the performance of specific algorithms using architecture-aware optimizations.

General Tools for Optimizing Performance

This category includes compilers, optimized libraries, domain-specific languages and problem-solving environments, as well as auto-tuning frameworks and run-time systems. These tools we term *general* in the sense that they have broader scope than the complementary category that lists architecture-aware optimizations in specific applications.

Compilers, range from the widely used GNU and Intel compilers for traditional languages to more exotic and/or research-oriented ones. The latter category includes works like [35] and [291] for GPUs. In [35] the authors describe a compiler framework for auto-parallelization and associated optimizations of affine loop nests, while in [291] optimizations such as memory coalescing and tiling are automatically applied for GPGPU compiled codes. Similar works (e.g., Apricot [238]) exist for other heterogeneous platforms like Intel MIC. Last, OpenARC [190] is an open-source compiler that seeks to accelerate research for directive-based heterogeneous computing.

As far as optimized libraries are concerned there is a wealth of them, targeting heterogeneous architectures – mainly GPUs. For example, cuFFT [5] provides fast GPU implementations for CUDA-enabled NVIDIA GPUs, while clFFT [1] extends similar functionalities to AMD GPUs.

cuBLAS [3], MAGMA [275] (with OpenCL GPU [94] and Intel MIC [126] ports), include GPU- and Xeon-Phi-accelerated linear algebra libraries, respectively. CUSP [6] is the corresponding library for sparse linear algebra and graph computations in NVIDIA GPUs. Thrust [18] is a parallel template library that provides support for CUDA (among others) by including optimized GPU versions [39] of algorithms like sort, reduce, scan, etc. Last, NVIDIA Performance Primitives (NPP) [15] and CUDA Math Library [4] provide fast implementations of image and signal processing functions and standard math functions, respectively, on the GPU. A similar collection by Intel (Intel Integrated Performance Primitives [8] and Intel Math Kernel Library [9]) targets Intel CPUs and Intel MIC.

With respect to domain-specific languages, problem-solving environments and auto-tuning frameworks plenty of research has been done that covers a broad range of scientific domains and applications. Due to their extra importance with respect to programmability and portability we discuss the above categories in detail in the following related sections (Section 3.2.1 and Section 3.2.2).

Architecture-Aware Optimizations in Applications

Research that targets optimizing specific applications is abundant, given the different platform architectures, languages, optimization choices, as well as the number of unique applications itself. Hence, the below is not meant to be a complete collection of related optimization work, as this could easily be a whole dissertation in its own merit. Instead, in this section we seek to provide a

Table 3.1: Applications and architecture-aware optimizations

Dwarf	Platform	Notes
N-Body Methods		
GROMACS Cell [224]	Cell	Porting GROMACS molecular dynamics code on CellBE
Rodriguez et al. [243]	GPU	GPU acceleration of cutoff pair potentials
NAMD Cell [254]	Cell	Non-bonded force-field MD on STI CellBE
Williams et al. [59]	(see notes)	Fast multipole method on Intel, AMD, Sun CPUs and GPU
Grape-8 [202]	(see notes)	Special-purpose accelerator for gravitational n-body sim.
Pennycook et al. [229]	MIC	Extreme vectorization for molecular dynamics
Anton2 [250]	(see notes)	Special-purpose molecular dynamics supercomputer
Sparse Linear Algebra		
Tang et al. [273]	MIC	Storage format and optimizations for SpMV on Intel MIC
Dalton et al. [83]	GPU	Optimization of SpMV using merge path
Williams et al. [284]	(see notes)	SpMV on AMD, Intel, Sun CPU and STI CellBE
Choi et al. [71]	GPU	BSCR, BELLPACK, auto-tuning of SpMV on GPU
Bell et al. [38]	GPU	Focus on sparse formats for GPU SpMV
Nath et al. [219]	GPU	Symmetric matrix-vector product (SYMV)
yaSpMV [288]	GPU	SpMV framework, reduces bandwidth issues
Graph Traversal		
CRONO [22]	CPU	For futuristic multi-core targets
Pannotia [62]	GPU	Irregular GPU graph application benchmarks
Enterprise [196]	GPU	Energy efficient and fast BFS (#45 in Graph 500)
Paredes et al. [228]	MIC	Fastest Intel Xeon Phi Top-down BFS algorithm
Beamer et al. [36]	GPU	Direction-optimizing BFS for low-diameter graphs
Wu et al. [287]	GPU	Uses high-level programming for GPU graph analytics
Spectral Methods		
Nukada et al. [222]	GPU	Multi-GPU 3D FFT for TSUBAME 2.0
cusFFT [280]	GPU	Sparse FFT
Structured Grids		
Datta et al. [85]	(see notes)	Stencil for Intel, AMD, Sun, IBM CPUs and NVIDIA GPU
Dynamic Programming		
Rucci et al. [186]	MIC	Smith-Waterman optimizations in hybrid CPU and MIC

sampling of works whose main target is heterogeneous architectures and that have been published in top-tier conferences.

We present some of the most current work that targets heterogeneous computing in Table 3.1. We group the applications using the dwarfs categorization and put special focus on n-body methods, due to the fact that we use GEM, an n-body application, as our case study for discussing performance optimizations and programmability vs. performance trade-offs in Chapters 4 and 5, respectively. We include notes that highlight important aspects of the respective works.

With respect to architecture-aware optimizations (Section 4.2) for increased performance in heterogeneous platforms, we provide a detailed case study on an electrostatic surface potential (ESP) calculation algorithm that falls under the n-body dwarf category, called GEM. The need for faster electrostatic surface potential (ESP) calculation execution, as part of molecular dynamics applications, has led to the development of multi- and many-core CPU and GPU implementations [163, 266, 80], as well as implementations on other heterogeneous computing platforms (IBM Cell [254]). Specialized hardware implementations using application-specific integrated circuits (ASIC), such as MD-GRAPE3 [217] have been deployed to provide molecular dynamics acceleration, as well. In contrast to these works, which optimize molecular dynamics on previous generations of parallel architectures, our work is one of the first to address optimization of a molecular modeling application on the most recent NVIDIA GPU architecture at the time (i.e., Kepler) and most importantly the Intel Xeon Phi co-processor. Working with pre-release versions of the hardware we conduct and evaluate performance optimizations on the new architectures, including both application and evaluation of traditional optimizations, as well as unique novel optimizations only applicable to

the new hardware. Our careful design of experiments following gradually optimized implementations serves as the basis for a detailed study that explores the performance and programmability trade-offs (Section 5.2). An important contribution of our work, compared to related work, is the evaluation of OpenCL as a means to program hardware (FPGAs). Our work in Chapter 4 is once more among the first to publish experiments with OpenCL on FPGA targets. Combined with our implicit evaluation of OpenCL optimizations in Chapter 5 it is the first, (together with [187] which also implicitly does so) to provide an analysis of basic Altera OpenCL design choices and the first, to the best of our knowledge, to provide a detailed analysis the more advanced Altera OpenCL optimizations for the FPGA.

3.1.3 Optimizing Performance: Hardware Approaches

One aspect of optimizing performance lies in enhancing the software, via architecture-specific optimizations, given a certain, existing platform (e.g., GPU). Another aspect, lies in changing the platform itself, so it is amenable to providing higher performance gains for applications. In the previous section we discussed the former aspect, while here we present work related to the former. Specifically, we discuss research that seeks to increase heterogeneity at the node level.

In [45], Borkar discusses the prospect of many-core architectures that comprise hundrends or thousands of cores, as an answer to the unreasonable power envelop of integrating multiple complex cores on a die. In the proposed solution the cores correspond to simpler cores versus “fat” cores (e.g., typical Xeon cores). This concept materialized with Intel Many Integrated Core (MIC) ar-

chitecture (up to 61 cores in Knight's Corner and 72 cores in Knight's Landing co-processors). Borkar's work refers to homogeneous cores, as opposed to the following works, however it provides useful background related to fine-grain power management, memory bandwidth, on-die networks and system resiliency for many-core systems that are also relevant in the heterogeneous context.

In [181] Kumar et al. study single-ISA heterogeneous multi-core architectures. Specifically, they present a chip-level multi-processor with four Alpha cores of varying complexity and power consumption, with the same ISA. Their evaluation is simulation-based and includes certain assumptions (e.g., oracle scheduling, assumed architectural and power-related characteristics). One assumption is that different workload phases can be assigned to the most appropriate core type, while the non-appropriate type of cores are switched off. The authors conclude that the proposed multi-core architecture demonstrates up to three times higher energy efficiency with small sacrifices in performance.

Chung et al. [75] try to answer the question: *"Does the future include custom logic, FPGAs, and GPGPUs?"* To address the question in the context of performance scaling and energy efficiency they investigate designs that place *unconventional cores* alongside traditional CPU cores. They focus on an analytical model that extends Hill and Marty's work [139] to include unconventional cores, i.e., custom logic, FPGAs, and GPUs. As in [181] their methodology includes assumptions with respect to the parallel workloads, like perfect scheduling and infinite divisibility.

In [199] Lukefahr et al. propose *Composite Cores*, an architecture that brings the notion of heterogeneity within a *single* core, with the main target being to reduce *switching overheads* – typically

observed in other heterogeneous computing approaches. Composite Cores is based around the concept of big and little compute micro-engines that are characterized by high performance and high energy efficiency, respectively. Cycle-accurate simulations show their design achieves considerable energy savings (18%) at minimal performance loss (5%). Along a similar path, in [153] Ipek et al. introduce *Core Fusion*, a reconfigurable CMP where smaller cores can dynamically morph into a larger CPU, according to an application's needs. In [54] the authors propose HeteNode, a novel node architecture where the CPU(s) and co-processors are directly connected via a system controller, in an effort to decrease the communication overhead observed with the traditional co-processing paradigm.

Last, Chien et al. [68] propose an alternative approach to heterogeneity and energy efficiency through *hardware customization*. Specifically, they argue about steering away from the traditional 90/10 optimization paradigm that guides architectural designs and addresses the "common case". Instead, they propose the *10x10* architecture that includes cores (or *micro-engines*) optimized for ten different 10% cases (where the number ten is arbitrarily chosen). Guha et al. [122] examine a broad selection of benchmarks from major benchmark suites in order to cluster applications by computation and memory behavior. Conceptually, each of these clusters would correspond to the "10%" cases optimally executed by each of the "10" corresponding micro-engines. [69] presents a case study, on a higher-level, of a "7x7" architecture.

Beyond the above detailed approaches, other studies have attempted to project the technology and architecture trends (which are indeed expected to be heterogeneous). One such key study is the 2008 DARPA Exascale Technology Report (ETR). Another appears in [171] and uses the data

from the TOP500 list (like the 2008 DARPA ETR) to update the ETR projections and identify an additional *heterogeneous* class of architectures, beyond the previously proposed categories (*heavy-weight*, *light-weight*).

Our work, complements prior work in that it tries to address heterogeneous computing, but does so in the context of what we term *telescoping architectures*. In contrast to the above works, we omit a theoretical analysis or simulator-based approaches, in favor of a high-level practical approach with considerable breadth, where we conceptually combine existing heterogeneous architectures and evaluate their performance, under certain assumptions. As shown in previous works above, assumptions in initial approaches for novel architectures are acceptable (but need to be refined and addressed in subsequent work). Clusters on a Chip (CoCs), as defined in Section 4.4, in contrast to [181] have disparate ISAs. However, as opposed to some of the other works, they can be universally programmed using OpenCL. In our work, we employ dwarf-based benchmarks for evaluation purposes that allow a certain level of generalization of conclusions. The majority of the work, except "10x10"-related papers that use the notion of clustering of computation and communication patterns, uses standard benchmarks that may not allow such broader insights.

3.2 On Programmability and Portability

In this section we present related work that addresses the aspects of programmability and portability of parallel heterogeneous systems. Among others, we discuss approaches that entail domain-

specific languages, auto-tuning frameworks, programming languages, and library-based frameworks.

3.2.1 Programmability

Parallel computing, both homogeneous and heterogeneous, and its ensuing computational power is of paramount importance across scientific domains, with many applications in engineering, mathematics, physics, biology, and elsewhere. We see many specific examples in Section 3.1. Parallel computing has enabled scientists to perform complex computations faster, do so with unprecedented amounts of data, and obtain more accurate results previously sacrificed over computation speed.

Until the recent past, high-performance computing was a privilege of government labs, large research universities and companies that could afford to build and maintain expensive (and power-demanding) supercomputers. With the parallel computing revolution of the mid-2000s parallelism has become mainstream and what was once considered "supercomputing" is now available at reasonable cost to a broader audience.

At the heart of accessible parallel computing lies different types of cores (i.e., *heterogeneous cores*), among which CPUs, GPUs, co-processors (e.g., Intel Xeon Phi), and reconfigurable architectures (i.e., FPGAs). Parallel programming, especially in the context of a heterogeneous environment, can prove challenging for expert programmers, and even more so for non-programmers, or novice programmers. Making heterogeneous parallel programming easier, i.e., enhancing the

programmability aspect, is crucial; domain scientists typically belong to that second category. Accelerating scientific advances hinges on domain scientists' ability to focus on their science, while being able to express and solve the computational aspect of their problems *fast*.

A survey among a high number of researchers that span scientific domains that was presented in Supercomputing (SC) 2011 [235] is illustrative of the state of the practice in computational science. The survey concludes that:

- Programming constitutes a high percentage of overall research time.
- For 50% of the participants in the survey the program execution time is less than 24 hours (of which 20% less than one hour).
- When it comes to strategies for enhancing performance, half participants replied they use none, while smaller percentages use techniques such as data layout optimizations (5%), loop optimizations (17%), or compiler flags (9%).
- Specifically for parallelism, 33% does not use any form of parallelism, 11% uses loop-level parallelism, and 7% thread-level parallelism. The majority uses job parallelism (51%).
- 40% of the researchers use desktop-computing only to address their computational needs.

A 2009 study [129] that takes into account the answers of 2000 users reaches similar conclusions.

Given the above, namely the importance of parallel computing for domain scientists (as well as general-purpose users) and their reluctance to commit time for learning to write efficient code,

researchers have followed various approaches to address the programmability aspect of parallel computing.

The most *transparent* way to make parallel platforms more programmable requires the programmer to only provide a sequential version of the algorithm. Then, modern compilers, like GCC, Intel/Cray/PGI, perform loop parallelization and code vectorization, among a wider collection of optimizations. Exploiting the auto-vectorization and auto-parallelization features of compilers can be challenging, and failure may ensue due to restrictions enforced by the *conservative* nature of compiler optimization algorithms or specific programming practices followed by the programmer (although ongoing research tries to address issues like vectorizing partially vectorizable loops [32]). While the above completely automated methods (i.e., most programmable) can offer great performance for more trivial cases, they cannot easily address the general case.

In this section, we first discuss related studies on the subject of programmability, and then we present representative examples from the corpus of related work as categorized in three broad categories:

1. Domain-specific languages (DSLs), problem-solving environments (PSEs), and auto-tuning frameworks.

2. Programming languages.

3. Source-to-source translators.

Previous Studies on Programmability

Various studies provide insights on various aspects of programming parallel platforms. We provide an overview below:

Hochstein et al. [140] present an interesting study on parallel programmer productivity. Their study is unique in that it constitutes one of the first empirical studies in the area that compares parallel programming models used in CS graduate curricula across four major colleges. The authors seek to evaluate programmability not as a matter of number of source lines of code (SLOC), but more so as the *effort per SLOC* from a practical standpoint. Another experimental study [159] highlights the best practices and common pitfalls when teaching parallel computing to science faculty. The concepts from this work (e.g., importance of showing parallelism, interactivity, visual results) are important and can be implemented in the context of programming frameworks targeted towards novice programmers (e.g, domain scientists) to facilitate embracing a parallel mindset for developing applications. The work of Burckhardt et al. [51] focuses on the importance of visual results and interactivity, also mentioned above, and discusses a departure from the "traditional edit-compile-run cycle" to what is termed "live programming" characterized by continuous feedback in the form of a user interface.

In [53] the authors propose several metrics for measuring programmability and productivity. While they focus on evaluating Unified Parallel C (UPC) [77] in a CPU environment (which they find to be equally programmable to MPI), these metrics have a more general applicability (e.g., when extending UPC for GPU cluster computing [66]). As far as GPU computing is concerned, an

early study on CAPS HMPP and PGI directives is provided in [133]. A later work by Lee and Vetter [189] shows a thorough evaluation of directive-based models for GPUs via 13 applications and comparisons with hand-tuned CUDA implementations. The paper concludes that directive-based models can provide reasonable performance with increased programmability.

A more extensive study [72] includes an insightful discussion on programmability and portability experimenting with three benchmarks from the dwarfs taxonomy across 12 different programming languages and platforms (including Intel, AMD and IBM CPUs, NVIDIA GPU, IBM Cell Broadband Engine). The quantitative measures include lines of code, development time, and achieved performance (percentage over peak for a given platform). The above work does not consider OpenCL [19], since it was still in beta release at the time. In [245] the authors attempt to quantify the *Ninja Gap*, i.e., the programming effort required to close the performance gap between mostly automated (i.e., more programmable) parallel implementations and hand-tuned code versions. Also, they discuss means of increasing programmability via hardware support, focusing on Intel Many Integrated Cores (MIC) architecture. Subsequent studies [293, 268] specifically focus on performance portability of OpenCL programs. The former studies three benchmarks – again from the dwarfs taxonomy, across Intel and AMD CPUs, discrete NVIDIA GPU, integrated Intel GPU, as well as an AMD APU. The latter explores performance portability of six benchmarks across the CPU and GPU only. In both cases, the authors identify the performance portability gap between naive and architecture-aware implementations and identify appropriate tuning knobs for performance portable programming. Other works compare the performance versus pro-

grammability gap across languages for CPUs or across heterogeneous architectures (OpenMP versus OpenCL [253], CUDA versus OpenCL [104], OpenACC versus OpenMP [283]).

Last, more recent studies include [81] and [204]. In the former Daga et al. conduct a programmability analysis of C++ AMP and OpenACC versus OpenCL on AMD discrete GPUs and APUs, and conclude that despite the increased programmability, performance is not in par with OpenCL hand-written code. In the latter, the authors evaluate emerging parallel programming models like Kokkos [39], RAJA [145], OpenACC, and OpenMP 4.0 versus CUDA and OpenCL. Their conclusion is that performance of the above models lie within a 5-20% of the CUDA/OpenCL code for a much better programmability and that programmability of each of these models will be the deciding factor for the breadth of their adoption.

Domain-Specific Languages, Problem-Solving Environments, and Auto-Tuning Frameworks

Domain-specific languages (DSLs), problem-solving Environments (PSEs), and auto-tuning frameworks are an approach often favored by domain scientists. Such solutions offer the added benefit of high-performance, often a result of automatic optimizations based on domain knowledge. This specific nature of the above, however, is a double-edged sword, in that it also introduces the problem of lack of generality of applicability. Such solutions focus to very function-specific codes, like dense linear algebra code (ATLAS) [76] or FFT (FFTW) [108], or even more specific subareas within a class of problems [115, 87, 98].

Irrespective of that, most are also restrictive in terms of target language and/or architecture on which the code is to be executed (e.g., GPU, CUDA, OpenCL [86, 193]) thereby lacking the aspect of portability. Additionally, they tend to have reduced utility as replacement blocks of code in pre-existing, legacy scientific code. Auto-tuning frameworks incorporate techniques such as those discussed in the performance-related discussion (Section 3.1). Different DSLs, PSEs, and auto-tuning frameworks span different domains. Stencil algorithms is an area that has been thoroughly researched and many auto-tuning frameworks have been proposed. Some focus on GPU architectures (NVIDIA, AMD, or both) [277, 142, 203], while others, like PATUS [73] or FAST [200], target both the CPU and GPU. While the above works can exploit single-node heterogeneous environments, works like [292] provide opportunities for running stencil codes on a cluster of GPUs. Examples from other areas include sparse linear algebra [272], graph analytics (Green Marl) [144], machine learning (OptiML) [57], mesh-based PDE solvers (Liszt) [91], and biomedical image analysis and visualization (Diderot) [70]. In [271] the authors describe Chestnut, a domain-specific parallel GPU programming language for parallel multi-dimensional grid applications, and Roccom [158] discusses a software integration framework in the context of parallel multi-physics simulations. In [165] the authors conduct a comprehensive review of domain-specific languages for FPGA computing.

Generally, DSLs/PSEs can be more generic within a domain (e.g., handle multiple dimensions of stencils), while others can be more specific (e.g., focus on code generation of 3D stencils [292]). Moreover, certain DSLs/PSEs can perform auto-tuning without user annotation (e.g., [200]), while others, like Mint [277] require the user to annotate code, from which the DSL/PSE can provide a

GPU implementation, for instance. Requiring user annotation adds a level of indirection that reduces programmability, but can typically provide better performance. In [146, 172] the authors focus on SIMD architectures. The former, ASPAS, addresses parallel sorting with code vectorization on x86-based architectures, focusing on Intel Xeon Phi. The latter conducts high-level restructuring of a program to expose ISA-independent vectorizable codelets and then generates ISA-specific code with appropriate SIMD-related optimizations. Finally, on the task-level, [215] constitutes an empirical auto-tuning framework that estimates performance benefits from auto-parallelization on the section-level and optimizes the program accordingly.

Another category of tools seeks to exploit familiarity of computational scientists with software like Mathematica or MATLAB. Early on, in works like [43], loop-based MATLAB code is transformed to the more efficient array-based MATLAB code. Moreover, opportunities for utilizing MATLAB's (faster) functions are identified and substitute the appropriate blocks of code. Later, with the advent of general-purpose GPU computing, research focused on providing automatic support of GPUs. For instance, in [236] the authors address automatic compilation of MATLAB programs to allow for synergistic execution on a CPU+GPU heterogeneous environment, while in [251] the authors seek to automate GPU computing by using MATLAB. Wolfram and Maple have themselves added GPU support in their products, Mathematica and MATLAB, respectively. For example, Mathematica provides the CUDALink [147] package that contains GPU-accelerated functions from areas like linear algebra and image processing, while also giving users the ability to develop their own GPU-accelerated functions, in a way arguably simpler than writing CUDA itself. Except for CUDALink, Mathematica provides analogous support for OpenCL, with their

OpenCLLink [148], thus obviating the need for OpenCL programming. MATLAB provides similar functionality, which is currently limited to NVIDIA GPUs only, via the Parallel Computing Toolbox [149]. The latter provides high-level constructs that can substitute OpenMP for within-chip multi-threading, as well as MPI for cluster computing.

Programming Languages for Programmability

On the programming languages side, there have been many proposed approaches that either define a *new programming model and programming language* from scratch, provide *higher-level abstractions* (e.g., directive-based approaches), or *libraries* that can be used in the context of existing languages.

In the case of shared-memory chip multi-processors (CMPs), Pthreads [52] and OpenMP [82] have long remained the "traditional" ways to approach parallelism, with the latter being a far more programmable solution compared to the former. As alternatives to the above, Intel provides Intel Cilk Plus [241] and Thread Building Blocks [231]. The former provides C/C++ language extensions to allow expressing task and data parallelism within applications. The latter is a library that includes, among others, a number of generic parallel algorithms, synchronization primitives, and support for dependency and data flow graphs. The above allow easy expression of loop- and task-level parallelism.

One method that is efficient in expressing computations at a more abstract level is *array programming*, as introduced by K. Iverson and implemented in APL [154]. In such cases, for example,

the operation X+Y on arrays is allowed, performs the addition operation across all elements of X and Y, and returns the result in an array form. Concepts of array programming are found in modern mathematical software (such as Mathematica and MATLAB), or languages like Fortran90. Similarly, NumPy arrays [278] provide a means to accomodate multi-dimensional generic data and is incorporated in the same-named package for numerical computation with Python, together with tools for integrating C/C++ and Fortran code. POOMA [240], a library originally developed at the Los Alamos National Lab, includes similar functionalities for arrays (among other data structures), with an emphasis on parallelism and ease of programming. Other approaches, like OOPAL [214] unify the concept of array programming and object-oriented programming to provide the combined advantages of developing code using object interfaces and expressiveness of array programming. In [124] the authors propose Hierarchically Tiled Arrays (HTAs); in many applications spanning scientific domains tiles or blocks constitute an important data arrangement in terms of performance (parallelism and data locality). HTAs is a data type that provides an easy way of manipulating (creation, usage, dynamic partitioning, etc.) tiles for sequential and more importantly parallel programming.

Following, we provide a representative sampling from the literature that showcases further means of enhancing programmability:

SWARM [31] is an open-source programming framework for the development of efficient multi-core programs. SWARM, as opposed to the OpenMP compiler-based approach, provides library functions and directives (a la OpenMP) and supports loop-level parallelism, as well as task-level parallelism. In [195] the authors introduce Merge, another library-based programming model for

heterogeneous multi-core systems that aims at increasing programmability. In Merge, the user develops a program by using high-level language extensions that are based on the MapReduce model. Mapping and distribution of the computation in parallel across heterogeneous components is automated by the Merge compiler and runtime.

Chapel [58] is a language designed as part of DARPA's High Productivity Computing Systems (HPCS) program and is specifically focused on programmability. It presents an alternative programming approach, encompassing a full-blown programming language, designed with programmability and portability in mind, that supports a multi-threaded execution model via high-level abstractions for task-, loop-, data-level parallelism and concurrency. The original version of Chapel has since been improved with added support for accelerators [255]. The same, single program written in Chapel can hence be now used to program across the spectrum of CPUs, Intel MIC, and GPUs.

PetaBricks [27], on the other hand, specifically attempts to solve programmability when the need arises to employ hybrid algorithms (i.e., a combination of algorithms, based on problem size and/or input data set features). Specifically, PetaBricks is a language that allows the user to express algorithmic choices, while letting the compiler performing the appropriate optimizations. The authors' evaluation highlights the importance of enabling auto-tuned hybrid algorithms, given their higher performance than individual ones.

In [281] the authors present a programming approach whereby users develop object-oriented stream programs using *aspects*. The proposed aspect-based method reportedly simplifies programming accelerators and provides performance close (approximately 80%) to hand-optimized CUDA code on

GPUs. Lime [97] is another proposed language, Java-compatible, that seeks to eliminate low-level requirements of languages like OpenCL and CUDA (e.g., explicit memory allocation on the device side and accompanying data transfers). The high-level object-oriented language provided by Lime constitutes an extension to Java with constructs that address heterogeneous architectures like GPUs and FPGAs. Experiments show that benchmarks developed with Lime achieve performance that lies between 75-140% of the corresponding hand-tuned OpenCL implementations.

HMPP, presented in [44] is a programming "workbench" that is based on the codelets concept (analogous to what a kernel is in OpenCL). HMPP includes HMPP directives for declaring, executing codelets, as well as directives for memory data transfers between a CPU and an accelerator. After a two-pass compilation, HMPP compiler produces an executable for the host-side program and the accelerator implementation of the codelets in the form of dynamic shared libraries. hiCUDA [128] is an attempt to provide yet another higher level abstraction for GPU programming, by providing directives that hide issues such as GPU memory allocation and data transfers from the programmer. Wolfe et al. [286] introduce the PGI accelerator directives for heterogeneous computing, the "ancestor" of OpenACC [282]. The latter, co-developed by Cray, CAPS, NVIDIA, and PGI, provides a standardized directive-based way to program heterogeneous systems that has effectively rendered it the "OpenMP analogous" for heterogeneous computing.

In [78] the authors discuss extensions of x10 [61] for GPU, i.e., compiling x10 to CUDA. Other works [290, 201] extend the Habanero execution model [56] – a Java implementation of x10 – for modern heterogeneous architectures and exhibit significant improvements with respect to programmability and portability.

With OpenMPC [186] Lee and Eigenmann propose a high-level programming interface that extends OpenMP with appropriate directives in support of a higher-performing version of what was presented in [188]. With these extensions not only can programs originally written with OpenMP directives run on (NVIDIA) GPUs, but also can be tuned for performance without having to resort to expressing related optimizations in CUDA. Another OpenMP-like programming model, OmpSs [99, 102], uses StarSs [233] to address programmability and portability. Experiments with six benchmarks indicate increased productivity, as well increased performance over the corresponding OpenCL and OpenMP implementations on the CPU and GPU.

Latest developments in the programming languages category include Regent [258] and Tangram [60]. The former is a language for HPC programs composed with tasks and logical regions. The latter is a programming language based on the synthesis of codelets, i.e., reusable code building blocks.

Programmability in a heterogeneous context *within* a compute node is one thing. Extending the notion of heterogeneity in a *multi-node* environment is yet another challenging problem. In his keynote at CCGrid 2013 [260], Mark Snir discusses programming models for high-performance (cluster) computing. When it comes to programmability/productivity, he notes that coding productivity is overrated, because coding constitutes a small fraction compared to debugging, tuning, and testing. He claims that MPI is good enough and should continue to be used for exascale computing. But, he notes that it is not just MPI, but MPI+X, which may be a problem, in which case research should focus on X, rather than on the MPI part. In the latest work by Kim, Lee, and Vetter [169] the authors propose an MPI+OpenACC framework for heterogeneous clusters based on accelerators. Based on evaluations that include the Titan supercomputer the authors conclude that

IMPACC achieves better performance for an easier programming approach than MPI+OpenACC separately. Earlier works, such as MPI-ACC [23] attempt to integrate support for auxiliary memory systems (e.g., GPU), beyond the CPU memory space that is de facto supported in MPI. Works like SnuCL [170] and LibWater [118] extend OpenCL for cluster environments, while rCUDA [95] does the same using CUDA. In [93] the authors target cluster computing by presenting a hybrid parallel programming model with Unified Parallel C (UPC) and MPI. In [210] the authors discuss X10, a PGAS language, and specific examples in the context of scientific computation and discuss aspects of programmability for domain scientists. XMP is a PGAS language and in [216] the authors explore in detail the productivity of the language and compare it to UPC. In [50] the authors seek to enhance productivity of clusters that include GPUs with OpmSs, an extension to the directive-based StarSs [233]. In [242, 155] the authors attempt to address programmability of HPC cluster computing via algorithmic skeleton interfaces (Triolet), and via a framework for hybrid parallel programming that combines Charm++ [164] and MPI, respectively. Last, in [289] the authors similarly propose novel language extensions to OpenMP, but in this case the target is to support multiple accelerators on which data and computation regions can be offloaded. Associated compiler and runtime support is provided to handle multi-GPU scenaria.

Works addressing other architectures like Cell Broadband Engine Architecture (CBEA) [162] include CellSs [40] and the work by Kunzman and Kale [182]. Both works seek to enhance programmability of the CellBE heterogeneous platform by annotating existing code; CellSs is based on OpenMP-like annotation that is handled by the CellSs compiler, while [40] extends the Charm++[164] programming model with *accelerated entry methods*.

Source-to-Source Translation

Another category of tools that provide a certain degree of programmability are source-to-source translators. Such tools, take a certain language at their input and output the equivalent code in another. In the case of source-to-source translators the user does not need to know both languages (i.e., input and output); knowing one implies being able to obtain code in the other, supported by the source-to-source translator. While source-to-source translators affect programmability, their main advantage remains portability, so we present some important work in source-to-source translation in our discussion on portability (Section 3.2.2). Here we focus on three example cases where a source-to-source translator facilitates entry in a non-CPU domain via a CPU or GPU-oriented language/programming paradigm. For example, in [188] the authors introduce OpenMP to CUDA source-to-source translation. The proposed compilation framework includes techniques for reducing memory transfers and optimizing global memory accesses. In [238] the authors present *Apricot*, an optimizing compiler framework that automates translation of OpenMP code to Intel MIC offload language extensions. This allows direct execution of OpenMP programs on Intel Xeon Phi, while a cost model selects at run-time code regions to be offloaded onto the co-processor, also keeping data transfers to the minumum necessary. Our third example, [187] uses OpenARC [190], an open-source compiler framework for directive-based accelerator computing. It focuses on enabling OpenACC to OpenCL for FPGA source-to-source translation. This includes extensions to the OpenACC directives to assist FPGA-friendly optimizations, as well as FPGA-targeted optimizations folded into the compiler itself.

Programmability of FPGAs

Programmability in heterogeneous computing, where platforms may include multi-core CPUs, GPUs, Intel MIC, as is evident from the above is a challenging task. Programmability in heterogeneous computing when we factor in FPGA targets arguably becomes a daunting endeavor. We present most work related to programmability in FPGAs separately here.

Industry-led and academic attempts have long seeked to facilitate FPGA programming via different methods [30] than direct use of RTL-level hardware design languages (HDLs), like Verilog or VHDL. HDL design is largely based on structural descriptions of the design. While using behavioral descriptions in HDL is possible, it is considered a bad design practice by many hardware programmers. Behavioral description of a program, conversely, is the traditional way of programming in software engineering. As such, to make FPGA programming accessible to non-hardware programmers, switching to a *high-level*, *behavioral* approach led to *High-Level Synthesis (HLS)*.

High-level synthesis approaches can be broadly divided to two main categories, *text-based* and *model-based/GUI-based*:

Text-based: This approach adopts text-based programming in languages that allow behavioral algorithm description on a higher level than HDL. There are languages specifically devised for HLS, like *Bluespec* [221], but typically require a steep learning curve and cannot take advantage of existing code. Most HLS languages are based on C/C++ (e.g., C-to-Verilog, Impulse C, Catapult C, Mitrion C, Synphony C by Synopsys, Vivado HLS by Xilinx). Discussion on HLS frameworks based on C are provided in [30, 208]. While the core of these languages is C/C++, there are restric-

tions on certain language features (e.g., recursion) and the code often needs to be annotated with language-specific constructs. A big advantage is the ability for fairly easy reuse of existing C/C++ code. A big disadvantage is that these variations are based on C/C++ that was designed as sequential language, thereby the majority lacks *intrinsic* support for describing parallelism. HLS based on languages like CUDA (FCUDA [227]) or OpenCL (Altera OpenCL [24], SOpenCL [225]), which were designed with parallelism in mind address this problem. FCUDA requires CUDA code annotation, which is then source-to-source translated to AutoPilot C and synthesized to an RTL design. The OpenCL tools listed above follow a similar approach. At the last step HDL code is generated and synthesized to produce the FPGA binary. Liquid Metal [150] by IBM offers a unified programming language (Lime) and can target a wide range of heterogeneous architectures, including multi-core CPUs, GPUs, and FPGAs.

Model-based/GUI-based: Tools in this category are based on graphical interfaces. *NI LabVIEW FPGA Module* [151] extends the capabilities of LabVIEW graphical development workflow using functional blocks and interconnects and allows targeting NI FPGAs. *Matlab HDL Coder* [206] follows a similar model-based approach allowing using Simulink models and Matlab functions to generate portable, synthesizable VHDL and Verilog code. Generated code can target Xilinx and Altera FPGAs, while both vendors (Xilinx with SysGen and Altera with DSP Builder) provide their Simulink blocksets that allow efficient synthesis of pre-defined functions. Other graphical model-based design tools include *SystemVue* [167] and *VisualSim* [211].

3.2.2 Portability

Among the three P's of heterogeneous computing (Performance, Programmability, Portability), *portability* is the last, but not least important aspect we seek to address in this work. It incorporates the aspects of *functional* portability, i.e., the ability of a given program to execute across similar or even fundamentally different platforms, and *performance* portability, i.e., the ability of a given program to not only execute correctly across platforms, but also do so at a proportionally equal performance. As is evident, performance portability *implies* functional portability.

With heterogeneous computing being a (relatively) new and ongoing trend, the bulk of research efforts have targeted performance and programmability (in this order) as a first-order concern, rather than programmability. Despite this, though, there is still plenty of research on the portability domain with different approaches aspiring to provide functional or performance portability.

We classify the related work in three main categories and present representative examples of each below:

- Library-based frameworks.
- Compiler frameworks and source-to-source translation.
- Programming language frameworks.

Library-Based Frameworks

The first approach entails *library-based frameworks*. Specifically, libraries of frequently used algorithms (where each library contains multiple functionally equivalent implementations of an algorithm) are available in the context of a framework. APIs are presented to the user, where the API is typically disjoint from any specific implementation. Such a framework may automatically identify the best performing implementation for a given set of input data and target execution platform and transparently execute in any one of the supported target platforms. Implementation and target platform selection is achieved by means of a manual or automatic modeling process. In [274] the authors present such an approach in the context of STAPL [26], where implementation is selected via an automatic model. Other libraries, that have been widely used include Thrust [39], and Kokkos [100]. The former is a library of parallel algorithms that resembles the C++ Standard Template Library (STL). As is the case with other representative libraries in this category, Thrust provides a high-level interface that enables programmers to write portable programs across CPUs and GPUs. The latter (Kokkos) [100] is available as part of Trilinos [134] and defines manycore parallel abstractions that can be used to form applications that benefit from Kokkos backends' performance portability. Specifically, a series of mini-apps are implemented using Kokkos and their performance on multi-core CPUs and GPUs fall within 90% of the device-specific optimized code. Other libraries in the related literature include SkePU [103] and SkelCL [265], both of which are based on algorithmic skeletons. Both these programming libraries support CPU and GPU systems (SkePU also supports multi-GPU systems). MAGMA [275] is a linear algebra for multi-core architectures, GPU, and Xeon Phi, effectively providing LAPACK/ScaLAPACK functionality on hybrid

architectures. MAGMA provides features like multi-precision support for high performance, hybrid algorithms and multi-GPU support. Most recently, Helal et al. [131] present MetaMorph, a library framework that seeks to provide performance portability in heterogeneous targets. Other works discuss performance portability of domain-specific libraries, like [89] for FFT in GPU architectures. Beyond the above, we discuss other important libraries in Section 3.1.2 (where we focus more on their performance aspect).

Compiler Frameworks and Source-to-Source Translation

The second approach we study here is with regards to *compiler frameworks* and *source-to-source translation*. In [123] Gummaraju et al. present Twin Peaks, a software platform that enables OpenCL code originally developed for GPU execution to run efficiently on the CPU via runtime-assisted cache and functional unit utilization techniques. Similarly, Ocelot [92] is a dynamic compilation framework, which allows translating, optimizing, and executing code originally written for GPUs (CUDA/PTX) via LLVM to other non-GPU multi-threaded targets (i.e., multi-core CPUs). Prior work, like MCUDA [270] also attempts to provide portability across the GPU and CPU, this time by mapping the CUDA programming model to multi-core CPU architectures, showing efficient execution on both types of architectures. In [205] Martinez et al. present CU2CL, the first CUDA-to-OpenCL source-to-source translator claiming to provide performance for OpenCL automatically generated codes that is "on par with the manually ported counterparts." Subsequent work [112] extends [205] and enhances the robustness of the tool, also providing proof of (more) successul translation for a broader set of benchmarks, including larger ones (in terms of lines of

code). Along a similar path with CU2CL, in [168] the authors discuss similarities and differences between CUDA and OpenCL and propose an automatic translation framework between the two. Their implementation of such a framework reportedly provides a certain degree of performance portability. OpenCL can run across platforms, including the GPU, so the OpenCL-to-CUDA part of the translator offers code portability, but adds little to the gamut of supported platforms. On the other hand, similar to CU2CL, the automatic CUDA-to-OpenCL translation opens up more opportunities for alternate target platforms for users. The authors in [187] develop an OpenACC-to-FPGA translation framework, based on OpenARC [190]. This allows code originally written for GPU targets to be functionally, and – to a certain degree – performance portable on (Altera) FPGA targets. This is one of the first works to address performance portability via a high-level programming approach on the FPGA domain. Last, DMML (Distributed Multiloop Language) [48] constitutes another tool for achieving performance portability by means of an intermediate language and nested pattern transformations. Rather than single-node, DMML enables efficient execution on heterogeneous clusters with non-uniform memory and accelerators, like GPUs.

Programming Language Frameworks

This approach proposes the use of *programming language frameworks* that not only offer programmability, but also portability across multiple platforms. An example of such an approach is MapCG [143], a high-level programming model based on the MapReduce framework that supports multi-core CPUs and (NVIDIA) GPUs. As an added bonus to portability, the MapReduce-based programming provides a layer of programmability. Chapel [58] is a language designed as part of

DARPA's High Productivity Computing Systems (HPCS) program and is specifically focused on programmability. The original version of Chapel itself has been extended [255] to support accelerators. A single Chapel program can now offer satisfactory performance portability across CPUs, Intel MIC, and GPUs. Specifically for GPUs, the achieved performance is comparable to the corresponding CUDA implementations, thereby achieving both functional and performance portability. As in MapCG, Chapel as a high-level language increases programmability, while relevant compiler back-ends enable transformations suitable for CPU and GPU and MIC architectures. In [232] the authors extend PetaBricks [27], a high-level programming model that allows the user to define *choices*. An auto-tuner subsequently selects the best for a given case. For the case of heterogeneous architectures the PetaBricks extensions in this work entail OpenCL kernel code generation – when possible – and a run-time system to integrate GPU targets work allocation and associated memory management. Similarly, Cashmere [137] is a programming system for heterogeneous architectures. It utilizes and builds on Satin [279] and Many-Core Levels (MLC) [138] programming models, allowing OpenCL code generation and handling execution across CPUs, Intel MICs, and GPUs, with good scalability and high efficiency. Last, TANGRAM [60] is a programming system (language, compiler, run-time) based on *codelets*, i.e., interchangeable, reusable code snippets, that seeks to ascertain cross-platform performance portability. The language is rich in supporting data parallelism primitives and work decomposition, among others. Based on related experiments TANGRAM delivers over 70% of the performance of existing high-performing libraries (cuBLAS, cuSPARSE, etc.)

Our work in identifying the performance and programmability gap (Section 5.2) complements prior works (e.g., [245]) in that it explores a wider range of optimizations (including the new *shuffle* feature of Kepler GPU architecture and the – then – novel Intel Xeon Phi and associated optimizations) and provides a more thorough analysis of the optimizations and their impact for the n-body class of problems. Finally, as part of directive-based parallel implementations, we contribute an analysis of compiler-hinted parallelization for GPUs using OpenACC and provide direct comparisons between the corresponding optimization levels across *all* three platforms, rather than focusing on the speed-ups and performance gap *within* a single platform. With respect to performance differences across different programming models, our work with the Grid-Based Language and Auto-tuning Framework (GLAF) [5] (Section 5.3) attempts to eliminate the problem by automating code-generation with a set of appropriate optimizations in different programming models (currently OpenMP and OpenCL) from a single program developed in our visual programming framework.

Programming heterogeneous platforms using directive-based extensions is easier than lower-level languages like CUDA or OpenCL. Still, it behooves the programmer to identify and appropriately annotate the parallel regions. In most cases satisfactory performance cannot be achieved, unless the programmer uses appropriate clauses to handle data allocation in an efficient manner. While the above approaches may be acceptable by programmers, they may constitute a burden that non-programmers – like domain scientists – may be reluctant to undertake. Our work (Section 5.3.1) *complements* the category of auto-vectorization and auto-parallellization, by generating code that

[5]The GLAF work originated at Intel by Dr. Ruchira Sasanka

is *more amenable* to auto-vectorizing and auto-parallelizing compilers. The auto-generated code *automatically* takes advantage of extensions like OpenMP and can be extended to exploit language extensions like OpenACC. The important contribution of our work, however, lies on programmability. Specifically, developing code using our research prototype framework (GLAF) enables novice programmers or non-programmers to develop programs in a visual, intuitive way.

Last, our work attempts to address the need for a programming abstraction and framework that is *general* enough to be of use across domains, as problems in engineering and sciences may be composed by multiple different parts that the restrictive nature of auto-tuners may not be able to directly address. *Generality* also refers to multiple target languages and architectures. Grid-based data structures, which lie at the basis of our GLAF framework (Section 5.3.1), have also been the central datatype of languages/extensions (APL [154], NumPy [278]). In GLAF the grid data structure is – among others – meant to support a programming paradigm that resembles the familiar *spreadsheet* workflow (where cells/tables undergo transformations based on formulas/macros). GLAF extends this familiar paradigm in ways to enable complex, general-purpose program development, and addresses the need for performance via parallelism support and other optimizations in automatically generated code. Following related conclusions from prior works [159, 51], we enhance upon existing work by *integrating* the visualization aspect in programming, where data and the operations on data (i.e., code) coexist during the development stage.

Chapter 4

On the Performance of Heterogeneous Platforms

In this chapter we focus on the first of the three P's [106]: *Performance*. As we state in Chapter 1, the proliferation of heterogeneous computing platforms presents the parallel computing community with new challenges. One such challenge entails evaluating the performance of such parallel architectures, and identifying the architectural and compiler/tool/run-time systems infrastructure innovations that ultimately benefit applications. To address this challenge, we propose the need for benchmarks that capture the computation and communication patterns (i.e., dwarfs or motifs) of applications, both present and future.

We introduce OpenDwarfs, a benchmark suite that currently realizes the Berkeley dwarfs in OpenCL, a vendor-agnostic and open-standard computing language for parallel computing. Employing

OpenDwarfs enables us to characterize a diverse set of modern fixed and reconfigurable parallel platforms: multi-core CPUs, discrete and integrated GPUs, Intel Xeon Phi co-processor, as well as a FPGA. We describe the computation and communication patterns exposed by a representative set of (architecture-agnostic) dwarfs, and obtain relevant profiling data and execution information, with the goal of drawing conclusions that highlight the complex interplay between dwarfs' patterns and the underlying hardware architecture of modern parallel platforms.

While our study with architecture-agnostic dwarf implementations achieves the above goal by focusing on the computation and communication patterns, it only sheds light to part of the performance aspect. A thorough study of performance (optimization, evaluation, and characterization) calls for architecture-aware optimizations across a diverse set of platforms and languages/language extensions. To this end, we switch our focus from performance evaluation of architecture-agnostic implementations with respect to dwarf patterns, to architecture-aware optimizations on CPU, GPU, and Intel Xeon Phi (MIC) using the C language (with SIMD/OpenMP/OpenACC extensions), and CUDA. While, in this chapter we focus on a detailed study on performance, in Section 5.2 we discuss the aspects of programmability and performance versus programmability trade-offs, as well as portability.

Recent advances in heterogeneous computing attempt to render FPGAs a major target architecture. While until very recently OpenCL was not officially supported on FPGAs, Altera and Xilinx – the two major FPGA vendors – have now extended the typical hardware design language (HDL) programming model by introducing a design process based on OpenCL-based toolchains that resembles the traditional CPU software development workflow. We present an overview of OpenCL

for Altera FPGAs in Section 2.2.2. In this chapter, we seek to explore the FPGA-specific optimization space and provide insights on the performance obtained via such optimizations and the early versions of the Altera OpenCL (AOCL) compiler.

We conclude this chapter on performance by exploring the trends in heterogeneous computing that may help sustain increasing performance benefits on the road to exascale. Historically, architectural innovation has telescoped the HPC community from the commodity (Beowulf) cluster in a machine room, i.e., a multi-node system with Ethernet interconnect, to a commodity cluster on a chip, i.e., multicore CPU with an ondie interconnect. We project that this "telescoping architecture" will apply more broadly to heterogeneous computing, namely from heterogeneous clusters like Tianhe-2 in a machine room to on a chip. To that end, we present an experimental study that relies on dwarf-based benchmarking and that extends the notion of telescoping architectures to identify the ideal mixture of compute engines (CEs) and the number of such CEs on a chip to create a heterogeneous "cluster on a chip" (CoC). Specifically, we experiment with heterogeneous architectures that contain single or multiple instances of CPUs, GPUs, Intel MICs, and FPGAs to demonstrate their performance efficacy given continuing advances in hardware technology, software, tools, and run-time support.

4.1 On the Performance of Architecture-Agnostic Dwarf-Based Applications

Over the span of the last decade, the computing world has borne witness to a parallel computing revolution, which delivered parallel computing to the masses while doing so at low cost. This democratization was ultimately by necessity as the *power wall* had been reached with respect to processor design and future improvements in computing capability would only be achieved by increasing both the number of and types of processing cores, thus creating a heterogeneous computing environment. The programmer has been presented with a myriad of new computing platforms promising ever-increasing performance. Already existing -albeit at a lower core count- multi-core CPUs, were complemented by many-core GPUs, APUs (accelerated processing units, i.e., a CPU and GPU fused on a single die), various types of co-processors (e.g., Intel Xeon Phi), and even FPGAs. Programming these platforms entails familiarizing oneself with a wide gamut of programming environments, (such as AMD Brook, NVIDIA CUDA, Cilk+) along with optimization strategies strongly tied to the underlying architecture. The aforementioned realizations present the parallel computing community with two challenging problems:

(a) The need of a common means of programming these architectures that obviates the need for learning a number of different parallel programming languages, and

(b) The need of a common means of evaluating this diverse set of parallel architectures.

The former problem was effectively solved through a concerted industry effort that led to a new parallel programming model, i.e., OpenCL. Through a standardization procedure spearheaded by companies including Apple, Intel, AMD, IBM, NVIDIA, and Qualcomm OpenCL emerged as a programming model that would serve heterogenous computing's needs. Other efforts, like SOpenCL [225] and Altera OpenCL [24] enable transforming OpenCL kernels to equivalent synthesizable hardware descriptions, thus facilitating exploitation of FPGAs as hardware accelerators, while obviating the overhead of additional development cost and expertise.

The latter problem cannot be sufficiently addressed by the existing benchmark suites. Such benchmarks suites (e.g., SPEC CPU [132], PARSEC [42]) are often written in a language tied to a particular architecture (e.g., C, C++ or FORTRAN for CPU benchmarking). and porting the benchmarks to another platform would typically mandate re-writing them using the programming model suited for the platform under consideration. Given OpenCL's ability to target a wide range of parallel platforms, one could argue that re-casting these benchmark suites as OpenCL implementations would solve the problem. The additional caveat in simply re-casting these benchmarks as OpenCL implementations is that existing benchmark suites represent collections of overly specific applications that do not address the question of what the best way of expressing a parallel computation is. This impedes innovations in hardware design, which will come as a quid pro quo, only when software idiosyncrasies are taken into account at design and evaluation stages. This is not going to happen unless software requirements are abstracted in a higher level and represented by a set of more meaningful benchmarks.

4.1.1 OpenDwarfs Benchmark Suite

OpenDwarfs is a benchmark suite that comprises 13 of the computation and communication patterns (i.e., *dwarfs*), as defined in [28]. The dwarfs and their corresponding instantiations (i.e., applications) are shown in Table 4.1. The current OpenDwarfs release provides full coverage of the dwarfs, including more stable implementations of the *Finite State Machine* and *Backtrack & Branch and Bound* dwarfs. CSR (*Sparse Linear Algebra* dwarf) and CRC (*Combinational Logic* dwarf) have been extended to allow for a wider range of options, including running with varying work-group sizes or running the main kernel multiple times.

An important departure from previous implementations of OpenDwarfs is related to the *uniformity* of optimization level across all dwarfs. More precisely, none of the dwarfs contains optimizations that would make a specific architecture more favorable than another. Use of shared memory, for instance, in many of the dwarfs in previous OpenDwarfs releases favored GPU architectures. Also, work-group sizes should be left to the OpenCL run-time to select for the underlying architecture, rather than being hard-coded (in which case they may be ideal for a specific architecture, but sub-optimal for another). Such favoritism limits the scope of a benchmark suite, in that it takes away from the general suitability of an architecture with respect to the *computation and communication pattern* intrinsic to a dwarf and rather focuses attention into very architecture-specific and often exotic software optimizations. We claim that architectural design should be guided by the dwarfs on the premise that they form basic, recurring, patterns of computation and communication, and

Table 4.1: Dwarf instantiations in OpenDwarfs

Dwarf	**Dwarf Instantiation**
Dense Linear Algebra	LUD (LU Decomposition)
Sparse Matrix-Vector Matrix Multiplication	CSR (Compressed Sparse-Row Vector Multiplication)
Graph Traversal	BFS (Breadth-First Search)
Spectral Methods	FFT (Fast Fourier Transform)
N-body Methods	GEM (Electrostatic Surface Potential Calculation)
Structured Grid	SRAD (Speckle Reducing Anisotropic Diffusion)
Unstructured Grid	CFD (Computational Fluid Dynamics)
Combinational Logic	CRC (Cyclic Redundancy Check)
Dynamic Programming	NW (Needleman-Wunsch)
Backtrack & Branch and Bound	NQ (N-Queens Solver)
Finite State Machine	TDM (Temporal Data Mining)
Graphical Models	HMM (Hidden Markov Model)
MapReduce	StreamMR

that the ensuing architectures following this design approach would be efficient without the need for the aforementioned optimizations (at least the most complex ones for programmers).

Of course, the above point does not detract from the usefulness of optimized dwarf implementations for specific architectures that may employ each and every software technique available to get the most of the *current* underlying architecture. In fact, we have ourselves been working on providing such optimized implementations for dwarfs on a wide array of CPUs, GPUs and MIC (e.g., N-body methods [180]). The open source nature of OpenDwarfs actively encourages the developers' community to embrace and contribute to this goal, as well.

In the end, optimized and unoptimized implementations of *dwarf* benchmarks are complementary and one would argue essential constituent parts of a complete benchmark suite. We identify three cases that exemplify why the above is a practical reality:

(a) Hardware (CPU, GPU, etc.) vendors are mostly interested in the most optimized implementation for their device, in order to stress their current device's capabilities. When designing a new architecture, however, they need a basic, unoptimized implementation *based on the dwarfs' concept*, so that the workloads are *representative* of broad categories, on which they can subsequently build and develop their design in a hardware-software synergistic approach.

(b) Compiler writers also employ both types of implementations: the unoptimized ones to test their compiler back-end optimizations on and the (manually) optimized ones to compare the efficacy of such compiler optimizations. Once more, the generality of the benchmarks, being based on the dwarfs concept, is of fundamental importance in the generality (and hence success) of new compiler techniques.

(c) Independent parts/organizations (e.g., lists ranking hardware, IT magazines) want a set of benchmarks that is *portable* across devices and in which *all* devices start from the same starting point (i.e., unoptimized implementations) for fairness in comparisons/rankings.

In order to enhance code uniformity, readability and usability for our benchmark suite, we have augmented the OpenDwarfs library of common functions. For example, we have introduced more uniform error checking functionality and messages, while a set of common options can be used to select and initialize the desired OpenCL device type at run-time. CPU, GPU, Intel Xeon Phi and FPGA are the currently available choices. Finally, it retains the previous version's timing infrastructure. The latter offers custom macro definitions, which record, categorize and print timing information of the following types: *data transfer time* (host to device and device to host), *kernel*

execution time, and *total execution time*. The former two are reported both as an aggregate, and in its constituent parts (e.g., total kernel execution time, and time per kernel- for multi-kernel dwarf implementations).

The build system has remained largely the same, except for changes allowing the selection of the Altera OpenCL SDK for FPGA execution, while a test-run *make* target allows for installation verification and execution of the dwarfs using default small test datasets. FPGA support for Altera FPGAs is offered, but currently limited to two of the dwarfs, due to lack of complete support of the OpenCL standard by the Altera OpenCL SDK, which requires certain alterations to the code for successful compilation and full FPGA compatibility [25]. We plan to provide full coverage in upcoming releases. For completeness in the context of this work we use SOpenCL for full Xilinx FPGA OpenCL support.

4.1.2 Experimental Setup

This section presents our experimental setup. First, we present the software setup and methodology used for collecting the results and discuss the hardware used in our experiments.

Software and Experimental Methodology

For benchmarking our target architectures we use OpenDwarfs (as discussed in Section 4.1.1), available for download at *https://github.com/opendwarfs/OpenDwarfs*.

The CPU/GPU/APU software environment consists of 64-bit Debian Linux 7.0 with kernel version 2.6.37, GCC 4.7.2 and AMD APP SDK 2.8. AMD GPU/APU drivers are AMD Catalyst 13.1. Intel Xeon Phi is hosted on a CentOS 6.3 environment with the Intel SDK for OpenCL applications XE 2013. For profiling we use AMD CodeXL 1.3 and Intel Vtune Amplifier XE 2013 for the CPU/GPU/APU and Intel Xeon Phi, respectively. In Table 4.2 we provide details about the subset of dwarf applications used and their input datasets and/or parameters. Kernel execution time and data transfer times are accounted for and measured by use of the corresponding OpenDwarfs timing infrastructure. In turn, the aforementioned infrastructure lies on the OpenCL events (which return timing information as a *cl_ulong* type) to provide accurate timing in nanosecond resolution.

Table 4.2: OpenDwarfs benchmark test parameters/inputs

Benchmark	**Problem Size**
GEM	Input file & parameters: nucleosome 80 1 0.
NW	Two protein sequences of 4096 letters each.
SRAD	2048x2048 FP matrix, 128 iterations.
BFS	Graph: 248,730 nodes and 893,003 edges.
CRC	Input data-stream: 100MB.
CSR	2048^2 x 2048^2 sparse matrix.

Hardware

In order to capture a wide range of parallel architectures, we pick a set of representative device types: a high-end multi-core CPU (AMD Opteron 6272) and a high-performance discrete GPU (AMD Radeon HD 7970). An integrated GPU (AMD Radeon HD 6550D) and a low-powered low-end CPU (A8-3850), both part of a heterogeneous Llano APU system (i.e., CPU and GPU fused on the same die), as well as a newer generation APU system (Trinity) comprising an A10-

5800K and an AMD Radeon HD 7660D integrated GPU. Finally, an Intel Xeon Phi co-processor. Details for each of the aforementioned architectures are given in Table 4.3.

To evaluate OpenDwarfs on FPGAs, we use the Xilinx Virtex-6 LX760 FPGA on a PCIe v2.1 board, which consumes approximately 50 W and contains 118560 logic slices. Each slice includes 4 LUTs and 8 flip-flops. FPGA clock frequency ranges from 150 to 200 MHz for all designs. FPGAs can be reconfigured in various ways, leading to a potentially huge design space. We provide representative alternative hardware implementations with increasing hardware resources for each dwarf, loop unrolling, where applicable (detailed in Table 4.4). These alternative implementations indicate the trade-offs between performance and area on the FPGA, and illustrate the performance scalability with additional hardware (i.e., more accelerator instantiations). Given the FPGA's reconfigurable nature, it is important how the SOpenCL tool automates hardware design, based on the OpenCL code. We discuss such details in more detail in Section 2.2.3. Generating a lower-performing implementation may appear counter-intuitive, however design restrictions, such as energy-efficiency and area requirements (often associated with a target device's cost), may favor a low-performing implementation over a fast, area- and power-demanding one that may only fit in a high-end FPGA.

Table 4.3: Configuration of the target fixed architectures

Model	AMD Opteron 6272	AMD Llano A8-3850	AMD Radeon HD 6550D	AMD A10-5800K	AMD Radeon HD 7660D	AMD Radeon HD 7970	Intel Xeon Phi P1750
Type	CPU	CPU*	Integr. GPU*	CPU*	Integr. GPU*	Discrete GPU	Co-processor
Frequency	2.1 GHz	2.9 GHz	600 MHz	3.8 GHz	800 MHz	925 MHz	1.09 GHz
Cores	16	4	5†	4	6†	32†	61
Threads/core	1	1	5	1	4	4	4
L1/L2/L3 Cache (KB)	16/2048/ 8192‡	64/1024/- (per core)	8/128/- (L1 per CU)	64/2048/- (per 2 cores)	8/128/- (L1 per CU)	16/768/- (L1 per CU)	32/512/- (per core)
SIMD (SP)	4-way	4-way	16-way	8-way	16-way	16-way	16-way
Process	32nm	32nm	32nm	32nm	32nm	32nm	22nm
TDP	115W	100W*	100W*	100W*	100W*	210W	300W
GFLOPS (SP)	134.4	46.4	480	121.6	614.4	3790	2092.8

† Compute Units (CU) ‡ L1: 16KBx16 data shared, L2: 2MBx8 shared, L3: 8MBx2 shared * CPU and GPU fused on the same die, total TDP

Table 4.4: FPGA implementations details

GEM	
FPGA_A(1)	Single accelerator
FPGA_A(1)_LU	Single accelerator, 4-way inner loop unrolling
FPGA_A(12)_LU	Multiple accelerators (12), 4-way inner loop unrolling
NW	
FPGA_A(1)	Single accelerator per OpenCL kernel
FPGA_A(5)_LU	Multiple accelerators (5) per OpenCL kernel, fully unrolled inner loop
SRAD	
FPGA_A(1)	Single accelerator per OpenCL kernel
FPGA_A(5)_LU	Multiple accelerators (5) per OpenCL kernel, fully unrolled inner loop
BFS	
FPGA_A(1)	Single accelerator per OpenCL kernel
CRC	
FPGA_A(1)	Single accelerator
FPGA_A(20)	Multiple accelerators (20)
FPGA_A(20)_DP	Multiple accelerators (20), enhanced data partitioning across BRAMs
CSR	
FPGA_A(1)	Single accelerator
FPGA_A(1)_LU	Single accelerator, fully unrolled inner loop

4.1.3 Results

Here we present our results of running a representative subset of the dwarfs on a wide array of parallel architectures. After we verify functional portability across all platforms, including the FPGA, we characterize the dwarfs and illustrate their utility in guiding architectural innovation, which is one of the main premises of the OpenDwarfs benchmark suite.

N-body Methods: GEM

The n-body class of algorithms refers to those algorithms that are characterized by all-to-all computations within a set of particles (bodies). In the case of GEM, our n-body application, the electrostatic surface potential of a biomolecule is calculated as the sum of charges contributed by all atoms in the biomolecule due to their interaction with a specific surface vertex (two sets of bodies). Listing 4.1 presents the pseudo-code for GEM: for each vertex near the biomolecule surface, all atoms contribute a certain amount of electrostatic potential that is added to the running total.

```
total_potential = 0
for i = 0 to num_of_vertices
  for j = 0 to  num_of_atoms
    Calculate electrostatic potential k between vertex(i), atom(j)
    total_potential += k
  end for
end for
return total_potential
```

Listing 4.1: GEM algorithm

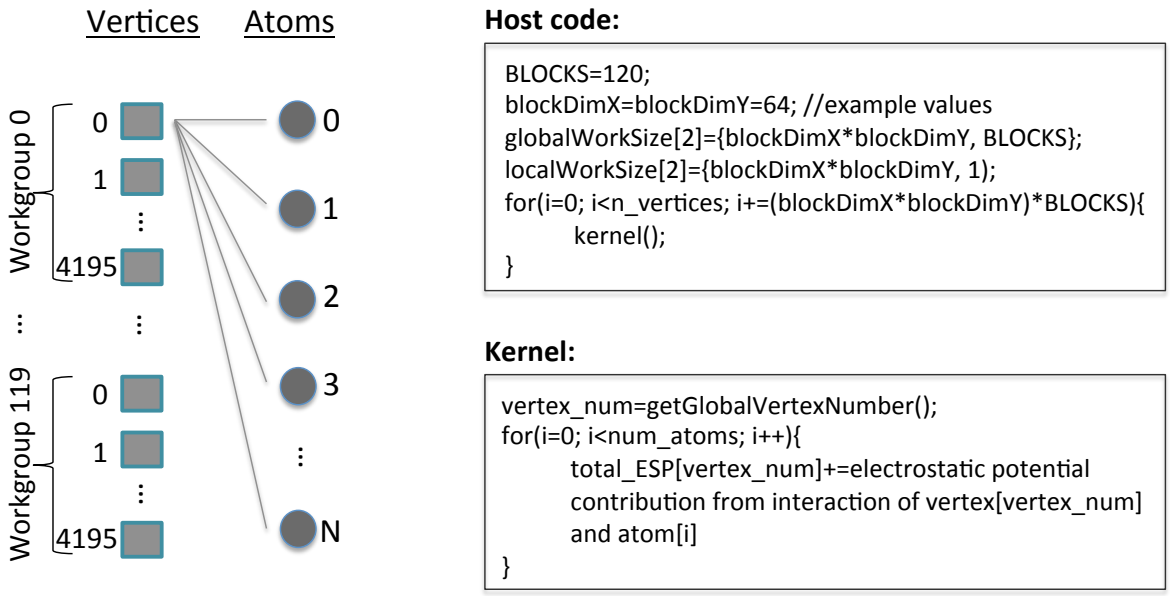


Figure 4.1: Parallel OpenCL implementation of GEM

In Figure 4.1 we illustrate the computation pattern of GEM and present the parallel implementation using OpenCL (host and device parts of the code). Each work-item accumulates the potential at a single vertex due to every atom in the biomolecule. A number of work-groups (*BLOCKS=120* in our example) each having *blockDimX*blockDimY* work-items (*4096* in our example) is launched, until all vertices' potential has been calculated.

GEM's computation pattern is regular, in that the same amount of computation is performed by each work-item in a work-group and no dependencies hinder computation continuity. Total execution time mainly depends on the maximum computation throughput. Computation itself is characterized by floating point (FP) arithmetic, including (typically expensive) division and square root operations that constitute one of the main bottlenecks. Special hardware can provide low latency alternatives of these operations, albeit at the cost of minor accuracy loss that may or may

not be acceptable for certain types of applications. Such fast math implementations are featured in many architectures and typically utilize look-up tables for fast calculations.

With respect to data accesses, atom data is accessed in a serial pattern, simultaneously by all work-items. This facilitates efficient utilization of cache memories available in each architecture. Figure 4.2 and Table 4.3 can assist in pinpointing which architectural features are important for satisfactory GEM performance: good FP performance and sufficient first-level cache. With respect to the former, the Opteron 6272 and A10-5800K CPUs reach about 130 GFLOPS and A8-3850 falls behind by a factor of 2.9, as defined by their number of cores, SIMD capability and core frequency. However, the cache hierarchy between the three CPU architectures is fundamentally different. The Opteron 6272 has 16K of L1 cache per core, which is *shared* among all 16 cores. Given the computation and communication pattern of n-body dwarfs, such types of caches may be an efficient choice. Cache miss rates at this level (L1) are also indicative of the fact. For example, A8-3850 with 64KB of dedicated L1 cache per core is characterized by a 0.55% L1 cache miss rate, with Opteron 6272 at 10.2% and A10-5800K a higher 24.25%. Those data accesses that result in L1 cache misses are mostly served by L2 cache and rarely require expensive RAM memory accesses. Measured L2 cache miss rates are 4.5%, 0.18% and 0%, respectively, reflecting the L2 cache capability of the respective platforms (Table 4.3). Of course, the absolute number of accesses to L2 cache, depend on the previous level's cache misses, so a smaller percentage on a platform tells only part of the story if we plan to compare different platforms to each other. In cases where data accesses follow a predictable pattern, like in GEM, specialized hardware can predict what data is going to be needed and fetch it ahead of time. Such *hardware prefetch* units are available

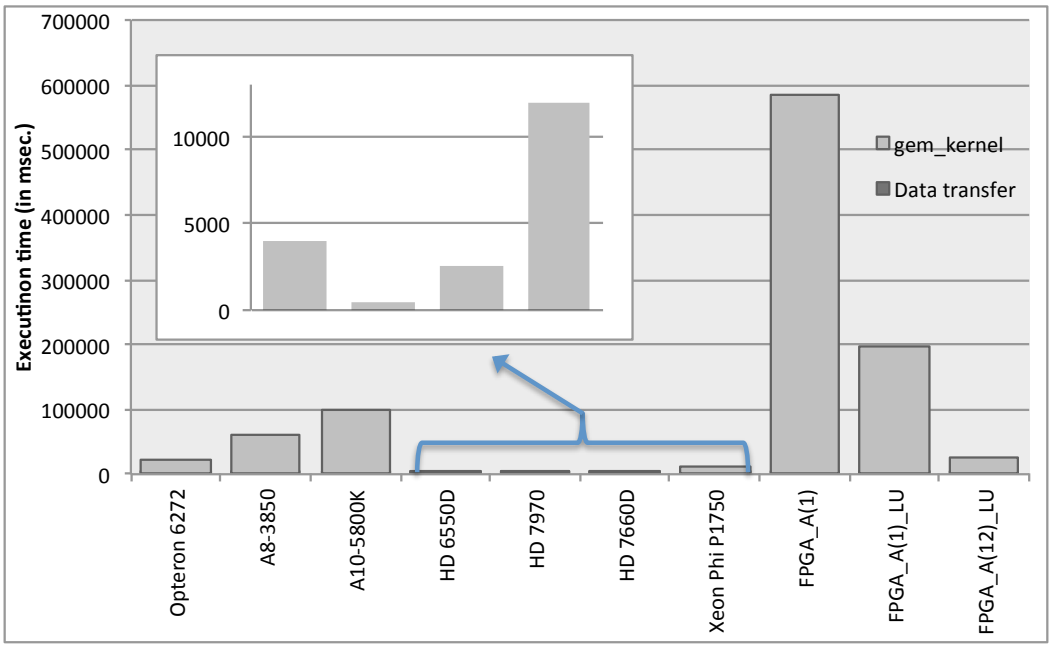


Figure 4.2: GEM performance results

– and of advanced maturity – in multi-core CPUs. This proactive loading of data can take place between the main memory and last level cache (LLC) or between different cache levels. In all three CPU platforms, a large number of prefetch instructions is emitted, as seen through profiling the appropriate counter, which, together with the regular data access patterns, verify the overall low L1 cache miss rates mentioned earlier.

Xeon Phi's execution is characterized by high vectorization intensity or the ratio of vector processing unit (VPU) elements active to the number of VPU instructions executed (12.84, the ideal being 16), which results from regular data access patterns and implies efficient auto-vectorization on behalf of the Intel OpenCL compiler and its *implicit vectorization module*. However, profiling reveals that the estimated latency impact is high indicating that the majority of L1 misses result in misses in L2 cache, too. This signifies the need for optimizations such as data reorganization

and blocking for L2 cache or the introduction of a more advanced hardware prefetch unit in future Xeon Phi editions – currently there is lack of automatic (i.e., hardware) prefetching to L1 cache (only main memory to L2 cache prefetching is supported). Further enhancement of the ring interconnect that allows efficient sharing of the dedicated (per core) L2 cache contents across cores would also assist in attaining better performance for the n-body dwarf. While Xeon Phi, lying between the multi-core CPU and many-core GPU paradigms, achieves good overall performance for this unoptimized and architecture agnostic code implementation, it falls behind its theoretical maximum performance of nearly 2 TFLOPS.

With respect to GPU performance, raw FP performance is one of the deciding factors as well. As a result, the HD 7970 performs the best and is characterized by the best occupancy (70%), compared to 57.14% and 37.5% for HD 7660D and HD 6550D, respectively. In all three cases, cache hit rates are over 97% (reaching 99.96% for HD 7970, corroborating that our conclusions for the CPU cache architectures hold for GPUs, too, for this class of applications (i.e, n-body dwarf). Correspondingly, the measured percentage of memory unit stalls is held at low levels. In fact, the memory unit is kept busy for over 76% of the time for all three GPU architectures, including all extra fetches and writes and taking any cache or memory effects into account.

Although FPGAs are not made for FP performance, SOpenCL produces accelerators whose performance lies between that of CPUs and GPUs. SOpenCL instantiates modules for single-precision FP operations, such as division and square root. Partially unrolling the outer loop executed by each thread four times results in nearly 4-fold speed-up (FPGA_A(1)_LU) compared to the base

accelerator configuration (FPGA_A(1)). Multiple accelerators can be instantiated and process in parallel different vertices on the grid, thus providing even higher speed-up (FPGA_A(12)_LU).

Dynamic Programming: Needleman-Wunsch (NW)

Dynamic programming is a programming method in which a complex problem is solved by decomposition into smaller subproblems. Combining the solutions to the subproblems provides the solution to the original problem. Our dynamic programming dwarf, Needleman-Wunsch, performs protein sequence alignment, i.e., identification of the similarity level between two given strings of amino acids. Listing 4.2 presents the NW algorithm: the potential pairs of sequences are organized in a 2D matrix *M*. The first step of the algorithm, on which we focus in this problem, is concerned with filling the 2D matrix M with scores, from the top left to the bottom right, in a step-wise manner. The first row and column are initialized as shown in the pseudo-code. Subsequently, each elements score in the 2D matrix depends on the values of its northwest, west, and north neighbors and in whether a match, insertion, or deletion operation yields the higher score. *S[i,j]* is the substitution matrix that contains the score of the aminoacids in the (i, j) position, as given from a pre-defined similarity/substitution matrix *S*.

```
Read two sequences A, B
Read substitution matrix S
for i = 0 to length(A)
  M[i,0] = i * gap_penalty
end for
for j = 0 to length(B)
  M[0,j] = j * gap_penalty
end for
for i = 1 to length(A)
  for j = 1 to  length(B)
    Match_score = M[i - 1, j - 1] + S[Ai, Bj]
    Insertion_score = M[i - 1, j] + gap_penalty
    Deletion_score =  M[i, j - 1] + gap_penalty
    M[i, j] = max(Match_score, Insertion_score, Deletion_score)
  end for
end for
```

Listing 4.2: NW algorithm

Figure 4.3 illustrates its computation pattern and two levels of parallelism, along with the OpenCL parallel mapping of the algorithm. Each element of the 2D matrix depends on the values of its west, north and northwest neighbors. This set of dependencies limits available parallelism and enforces a wavefront computation pattern. On the first level, blocks of computation (i.e., OpenCL work-groups) are launched across the anti-diagonal and on the second level, each of the work-group's work-items works on cells on each anti-diagonal. Available parallelism at each stage is variable, starting with a single work-group, increasing as we reach the main anti-diagonal and decreasing again as we reach the bottom right. Parallelism varies within each work-group in a similar way, as shown in the respective figure, where a variable number of work-items work independently in parallel at each anti-diagonal's level. Needleman-Wunsch algorithm imposes significant syn-

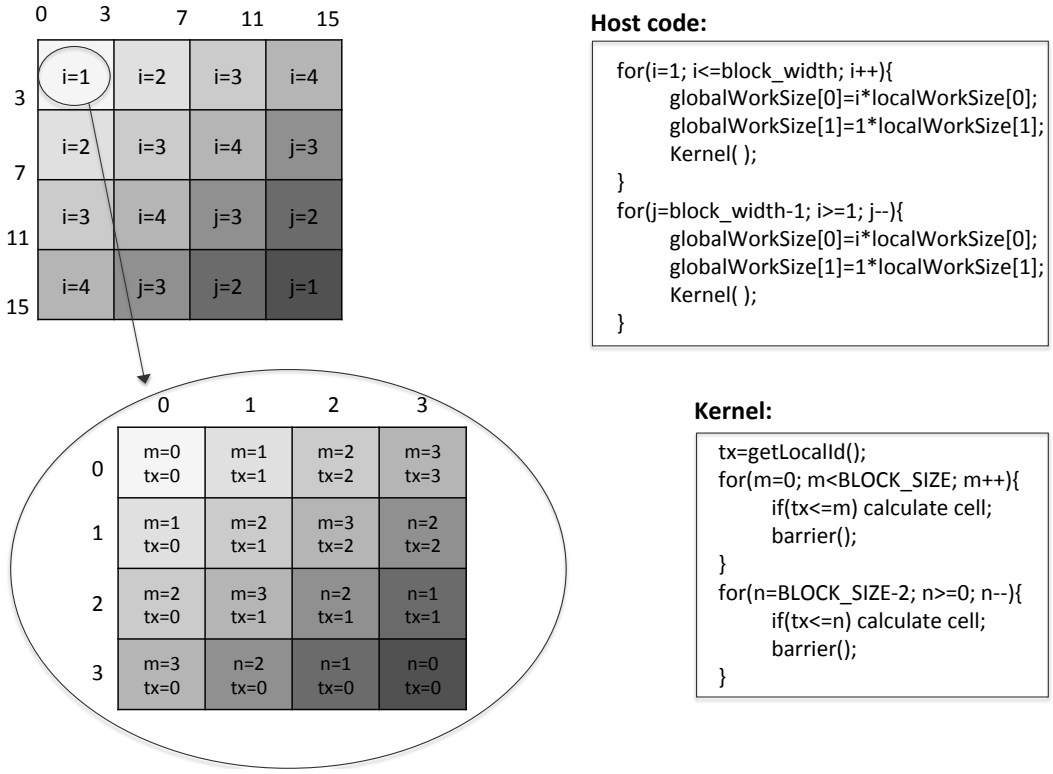


Figure 4.3: Parallel OpenCL implementation of Needleman-Wunsch

chronization overhead (repetitive barrier invocation within the kernel) and requires modest integer performance. Computations for each 2D matrix cell entail calculating an alignment score that depends on the three neighboring entries (west, north, northwest) and a max operation (i.e., nested if statements).

In algorithms like NW that are characterized by inter- and intra-work-group dependencies there are two big considerations. First, the overhead for repetitively launching a kernel (corresponding to inter-work-group synchronization), and second, the cost of the intra- work-group synchronization via *barrier()* or any other synchronization primitives. Introducing system-wide (hardware) barriers would help to solve the former of the problems, while optimization of already existing intra-work-

group synchronization primitives would be beneficial for this kind of applications for the latter case.

Memory accesses follow the same pattern as computation, i.e., for each element the west, north and northwest elements are loaded from the reference matrix. For each anti-diagonal *m* within a work-group (Figure 4.3), the updated data from anti-diagonal *m-1* is used.

Figure 4.4 shows the performance results for NW. As we can observe, GPUs do not perform considerably better than the CPUs. In fact, the Opteron 6272 surpasses all GPUs (and even Xeon Phi), when we only take kernel execution time into account. What needs to be emphasized in the case of algorithms, such as NW, is the variability in the characteristics of each kernel iteration. In Figure 4.5 we observe such variability for metrics like the percentage of the time the arithmetic and logical unit (ALU) is busy, the cache hit rate, and the percentage of time the fetch unit is busy or stalled, on the HD 7660D. Similar behavior is observed in the case of the HD 6550D. Most of these metrics can be observed to be a function of the number of active wavefronts in every kernel launch. For instance, the cache hit rate follows an inverse-U-shaped curve, as do most of the aforementioned metrics. In both cases, occupancy is below 40% (25% for HD 6550D) and ALU packing efficiency barely reaches 50%, which indicates a mediocre job on behalf of the shader compiler in packing scalar and vector instructions as VLIW instructions of the Llano and Trinity integrated GPUs (i.e., HD 6550D and HD 7660D).

As expected, the FPGA performs the best when it comes to integer code, in which case, its performance lies closer to GPUs than to CPUs. Multiple accelerators (five pairs) and fully unrolling the innermost loop deliver higher performance (FPGA_A(5)_LU) than a single pair (FPGA_A(1)) and

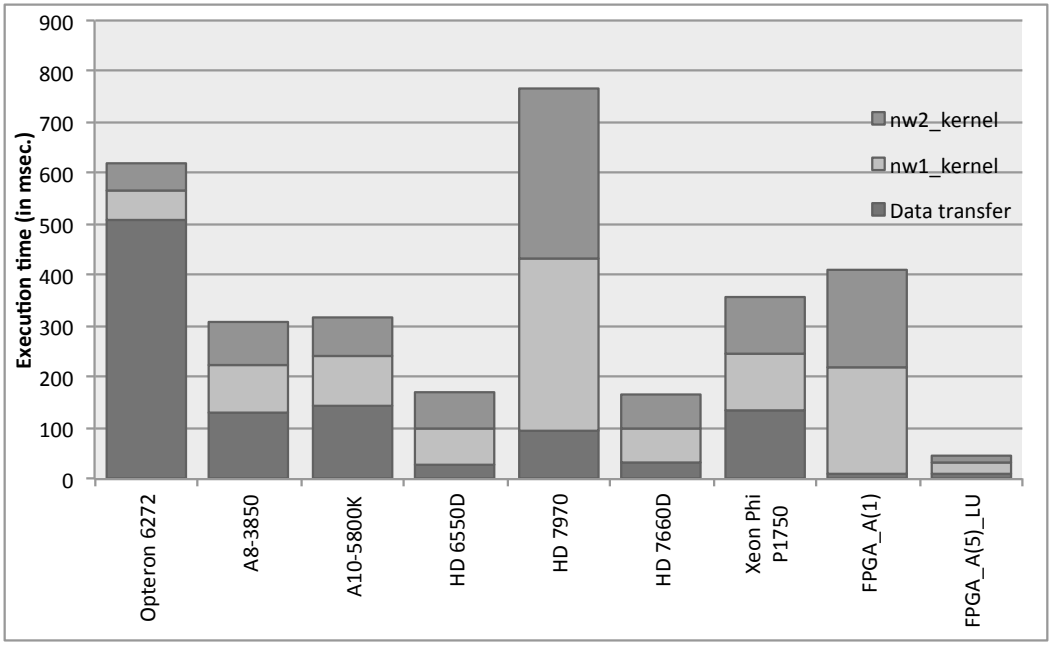


Figure 4.4: NW performance results

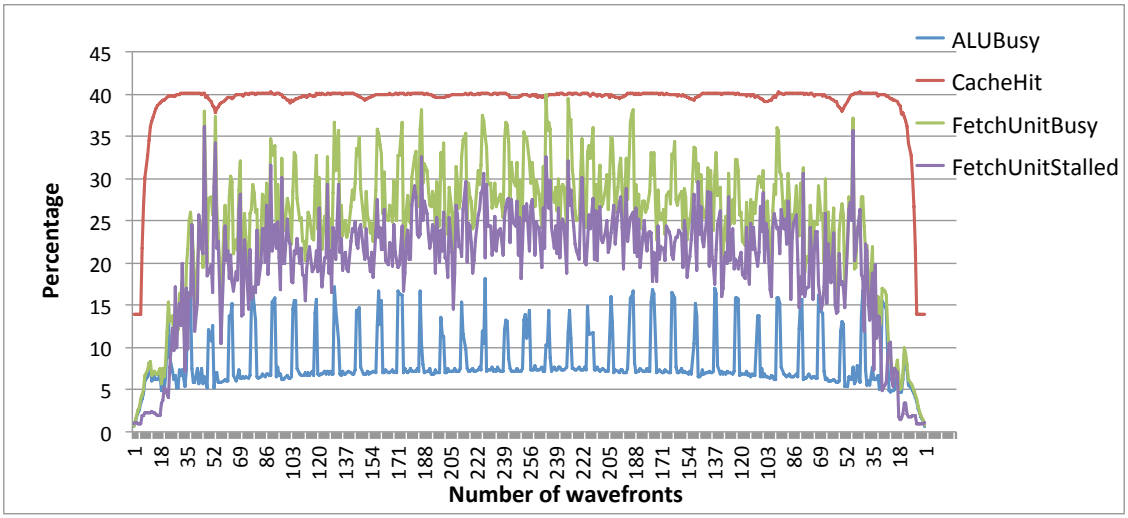


Figure 4.5: NW profiling on HD 7660D

render the FPGA implementation the fastest choice for the dynamic programming dwarf. In the FPGA implementation of NW, the pattern of data fetches favors decoupling of the compute path from the *data fetch & fetch address generation unit*, as well as from the *data store & store address generation unit*. This allows aggressive data prefetching in buffers ahead of the time of the actual data requests.

Structured Grids: Speckle Reducing Anisotropic Diffusion (SRAD)

Structured grids refers to those algorithms in which computation proceeds as a series of grid update steps. It constitutes a separate class of algorithms from unstructured grids, in that the data is arranged in a regular grid of two or more dimensions (typically two-dimensional or three-dimensional). SRAD is a structured grid application that attempts to eliminate speckles (i.e., locally correlated noise) from images, following a partial differential equation approach. Listing 4.3 presents the SRAD algorithm: it consists of two passes over a 2D image (structured grid). The first pass calculates the diffusion coefficient for each pixel of the image. The second pass updates the image using (among others) the diffusion coefficients that correspond to the neighboring pixels.

Figure 4.6 presents a high-level overview of the parallel mapping (using OpenCL) of the SRAD algorithm, without getting into the specific details (parameters, etc.) of the method, and Figure 4.7 shows the performance results. Performance is determined by FP compute power. The computational pattern is characterized by a mix of FP calculations including divisions, additions and multiplications. Many of the computations in both SRAD kernels are in the form: $x = a * b + c * d + e * f + g * e$. These computations can easily be transformed by the com-

piler to multiply-and-add operations. In such cases, special *fused multiply-and-add* units can offer a faster alternative to the typical series of separate multiplication and addition. While such units are already existent, more instances can be beneficial for the structured grids dwarf.

```
Read image I[num_rows, num_cols]
J[num_rows, num_cols] = exp(I[num_rows, num_cols])
for i = 0 to num_rows
  for j = 0 to  num_cols
    dN[i][j] = J[i - 1][j] - J[i][j]
    dS[i][j] = J[i + 1][j] - J[i][j]
    dW[i][j] = J[i][j - 1] - J[i][j]
    dE[i][j] = J[i][j + 1] - j[i][j]
    Calculate various parameters as function of the above
    Calculate diffusion coefficient c[i][j] as function of the above
  end for
end for
for i = 0 to num_rows
  for j = 0 to  num_cols
    cN = c[i - 1][j]
    cS = c[i + 1][j]
    cW = c[i][j - 1]
    cE = c[i][j + 1]
    D = cN * dN[i][j] + cS * dS[i][j] +
        cW * dW[i][j] + cE * dE[i][j]
    J[i][j] =  J[i][j] + 0.25 * lambda * D
  end for
end for
```

Listing 4.3: SRAD algorithm

A series of *if* statements (simple in *kernel1*, nested in *kernel2*) handles boundary conditions and different branches are taken by different work-items, potentially within the same work-group.

Host code:

```
Loop for iter number of iterations{
        calculate statistics for the region of interest
        blockX=columns/BLOCK_SIZE;
        blockY=rows/BLOCK_SIZE;
        localWorkSize[2]={BLOCK_SIZE, BLOCK_SIZE};
        globalWorkSize[2]={blockX*localWorkSize[0],
                                blockY*localWorkSize[1]};
        kernel1();
        kernel2();
}
```

Kernel1:

```
(Each work-item (i,j) works on a 2D table element)
dN[i][j]=J[north][j]-J[i][j];
dS[i][j]=J[south][j]-J[i][j];
dW[i][j]=J[i][west]-J[i][j];
dE[i][j]=J[i][east]-J[i][j];
Calculate various parameters based above
values & initial J[i][[j] value;
Using the above value, calculate diffusion
coefficient c[i][j];
```

Kernel2:

```
 (Each work-item (i,j) works on a 2D table element)
 cN=c[i][j];
 cS=c[north][j];
 cW=c[i][j];
 cE=c[i][east];
 D=cN*dN[i][j]+cS*dS[i][j]+cW*dW[i][j]+cE*dE[i][j];
 J[i][j]=J[i][j]+0.25*lambda*D;
```

Figure 4.6: Parallel OpenCL implementation of SRAD

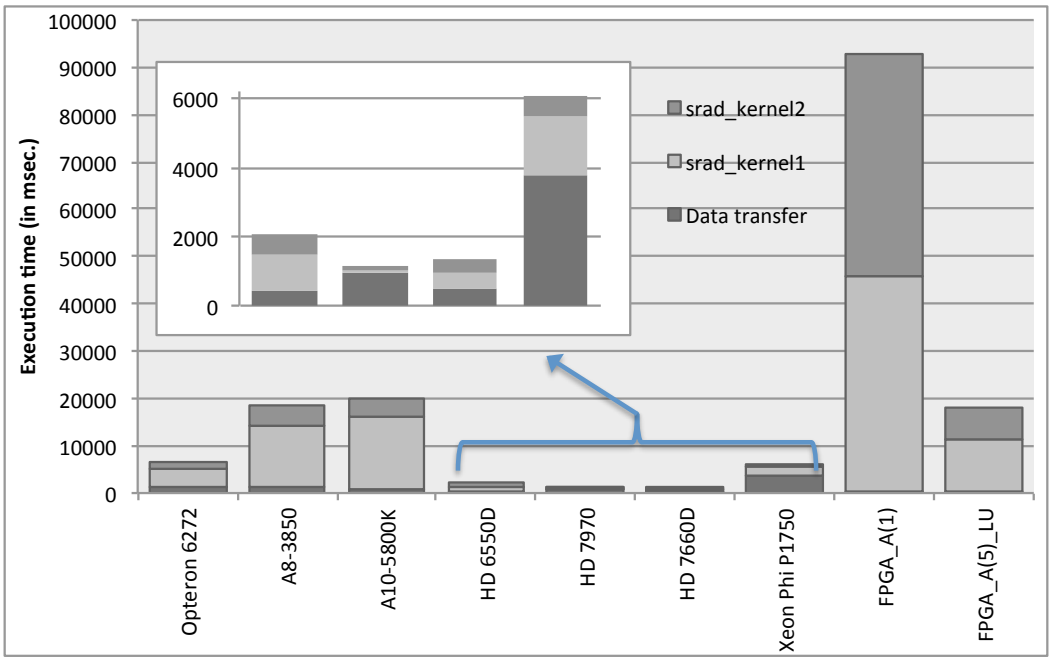


Figure 4.7: SRAD performance results

Since boundaries constitute only a small part of the execution profile, especially for large datasets, these branches do not introduce significant divergence. In the case of CPU and Xeon Phi execution, branch misprediction rate never exceeded 1%, while on the GPUs vector ALU utilization (*VALUUtilization*) remained above 86% indicating a high number of active vector ALU threads in a wave and consequently minimal branch divergence and code serialization.

Following its computational pattern, memory access patterns in SRAD, as in all kinds of stencil computation, are localized and statically determined, an attribute that favors data parallelism. Although the data access pattern is a priori known, non-consecutive data accesses prohibit ideal caching. As in the NW case, where data is accessed in a non-linear pattern, data locality is an issue here, too. Cache hit rates, especially for the GPUs, remain low (e.g., 33% for HD 7970). This leads to the memory unit being stalled for a large percentage of the execution time (e.g., 45%

and 29% on average for HD 7970, for the two OpenCL kernels – srad_kernel1 and srad_kernel2). Correspondingly, the vector and scalar ALU instruction units are busy for a small percentage of the total GPU execution time (about 21% and 5.6% for our example, on the two kernels on HD 7970). All this is highlighted by comparing performance across the three GPUs, and once more, indicates the need for advancements in the memory technology that would make fast, large caches more affordable for computer architects.

On the CPU and Xeon Phi side, large cache lines can afford to host more than one row of the 2D input data (depending on the input sequences' sizes). The huge L3 cache of the Opteron 6272, along with its high core count, make it very efficient in executing this structured grid dwarf. In such algorithms, the balance between cache and compute power distinguishes a good target architecture. Of course, depending on the input data set there are obvious trade-offs, as in the case of GPUs, which despite their poor cache performance are able to hide the latency by performing more computation simultaneously while waiting for the data to be available.

An FPGA implementation with a single pair of accelerators (one accelerator for each OpenCL kernel) offers performance worse even than that of the single-threaded Opteron 6272 execution (FPGA_A(1)). This is attributed mainly to the complex FP operations FPGAs are notoriously inefficient at. Multiple instances of these pairs of accelerators (five pairs in FPGA_A(5)_LU) can process parts of the grid independently, bringing FPGA performance close to that of multicore CPUs. Different work-groups access separate portions of memory, hence multiple accelerators instances access different on-chip memories, keeping accelerators isolated and self-contained.

Graph Traversal: Breadth-First Search (BFS)

Graph traversal algorithms entail traversing a number of graph nodes and examining their characteristics. As a graph traversal application, we select a BFS implementation. Listing 4.4 presents the BFS algorithm. The algorithm start from the root node and visit all the immediate neighbors. Subsequently, for each of these neighbors the corresponding (unvisited) neighbors are inspected. The whole process is repeated until the whole graph is traversed.

```
Mark all nodes as "not visited"
Initialize empty queue Q
Q.enqueue(start node)
while (Q is not empty)
  cur_node = Q.dequeue()
  if (cur_node has not been visited)
    visited[cur_node] = true
    for (all edges <cur_node, neighbor>)
      if (neighbor has not been visited)
        Q.enqueue(neighbor)
      end if
    end for
  end if
end while
```

Listing 4.4: BFS algorithm

BFS's computation pattern can be observed through a simple example (Figure 4.9), along with its parallel OpenCL implementation (host and device side code) in Figure 4.8. The BFS algorithm's computation pattern is characterized by an *imbalanced* workload per kernel launch that depends on the sum of the degrees deg(v_i) of the nodes at each level. For example (Figure 4.9), deg(v_0)=3, so

Host code:

```
maxThreads=numNodes < maxThreads?numNodes : maxThreads;
globalWorkSize=(numNodes/maxThreads)*maxThreads+
                ((numNodes%maxThreads)==0?0:maxThreads);
localWorkSize=max_threads;
node_as_source[]={1,0,0,0,0,0,0};
node_visited[]={1,0,0,0,0,0,0};
update_node_info[]={0,0,0,0,0,0,0};
cost[]={0,0,0,0,0,0,0};
```

```
do{
        stop=1;
        kernel1();
        kernel2();
}while(stop==0);
```

Kernel1:

```
tid=getGlobalId();
if(tid<numNodes and node_as_source[tid]==1){
        node_as_source[tid]=0;;
        for (all neighbors neighb[i] of current node)
                if(!node_visited[neighb[i]]){
                        cost[neighb[i]]=cost[tid]+1;
                        update_node_info[neighb[i]]=1;
                }
}
```

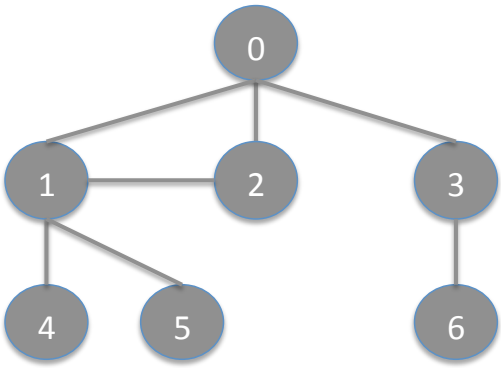
Kernel2:

```
tid=getGlobalId();
if(tid<numNodes and
update_node_info==1){
        node_as_source[tid]=1;
        mark node_visited[tid]=1;
        update_node_info[tid]=0;
        stop=0;
}
```

Figure 4.8: Parallel OpenCL implementation of BFS

only three work-items perform *actual* work in the first invocation of *kernel2*. Subsequently, *kernel1* has three work-items, as well. Second invocation of *kernel2* performs work on three nodes again (deg(v_1) + deg(v_2) + deg(v_3) = 8, but nodes v_0, v_1, v_2 have already been visited, so effective deg(v_1) + deg(v_2) + deg(v_3) = 3). Computation itself is negligible, being reduced to a simple addition with respect to each node's cost.

In Figure 4.10 we show the results obtained by executing BFS across our test platforms. The way the algorithm works might lead to erroneous conclusions, if only occupancy and ALU utilization is taken into account, as in all three GPU cases it is over 95% and 88%, respectively (for both kernels). The problem lies in the fact that *not* all work-items perform useful work, and the fact that the kernels are characterized by reduced compute intensity (Figure 4.9). In such cases, up to



Iter.	Kernel	Thread id (tid)						
		0	1	2	3	4	5	6
1	kernel1	✔	✗	✗	✗	✗	✗	✗
1	kernel2	✗	✔	✔	✔	✗	✗	✗
2	kernel1	✗	✔	✔	✔	✗	✗	✗
2	kernel2	✗	✗	✗	✗	✔	✔	✔
3	kernel1	✗	✗	✗	✗	✔	✔	✔
4	kernel2	✗	✗	✗	✗	✗	✗	✗

Figure 4.9: Example that shows load imbalance of BFS

a certain degree of problem size or for certain problem shapes, the number of compute units or frequency are not of paramount importance and high-end cards, like HD 7970 are about as fast as an integrated GPU (e.g., HD 7660D). The above is highlighted by the hardware performance counters that indicate poor ALU packing (e.g., 36.1% and 38.9% for the two BFS OpenCL kernels, on HD 7660D). Similarly, for HD 7970, the vector ALU is busy only for 5% (approximate value across kernel iterations) of the GPU execution time, even if the number of active vector ALU threads in the wave is high (*VALUUtilization*: 88.8%).

For similar reasons, CPU execution performance is capped on Opteron 6272, which performs only marginally better than A8-3850. It is interesting to see that A10-5800K and even Xeon Phi, with

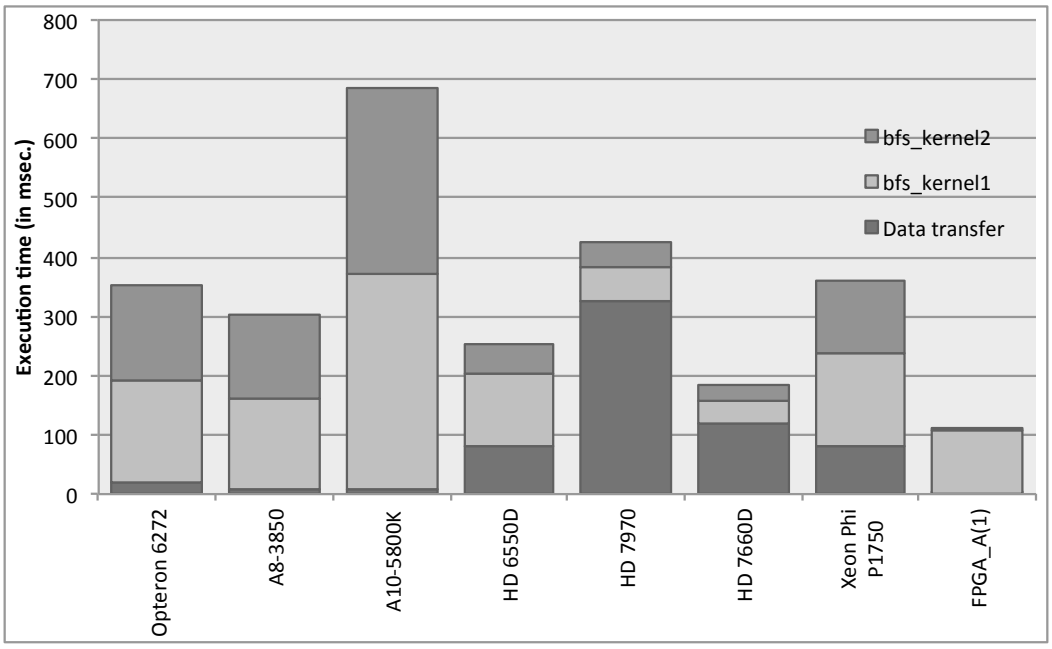


Figure 4.10: BFS performance results

8- and 16-way SIMD are characterized by lack of performance scalability. Why performance of A10-5800K is not *at least* similar to that of A8-3850 could not be pinpointed during profiling. However, in both A10-5800K and Xeon Phi cases, we found that the OpenCL compiler could not take advantage of the 256- and 512-bit wide vector unit, because of the very nature of graph traversal.

With respect to data accesses, BFS exhibits irregular access patterns. Each work-item accesses discontiguous memory locations, depending on the connectivity properties of the graph, i.e, how nodes of the current level being inspected are being connected to other nodes in the graph. Figure 4.9 is not only indicative of the resource utilization (work-items doing useful work), but of the inherent irregularity of memory accesses that depend on run-time assessed multiple levels of indirection, as well. Available caches' size define the cache hit rate, even in these cases, so HD

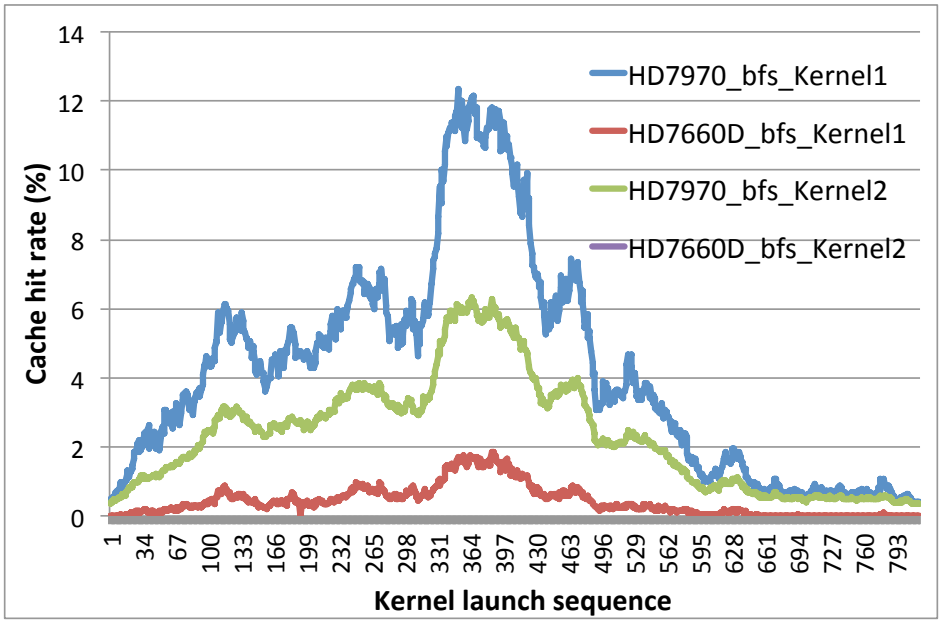


Figure 4.11: BFS cache performance comparison between HD 7970 and HD 7660D

7970, which provides larger amounts of cache memory provides higher cache hit rates compared to the HD 7660D (varying for each kernel iteration, Figure 4.11).

The FPGA implementation of BFS (FPGA_A(1)) is the fastest across all tested platforms. While *kernel1* is not as fast as in the fastest of our GPU platforms, minimal execution time for *kernel2* and data transfer time render it the ideal platform for graph traversal, despite the irregular, dynamic memory access pattern (which causes the input streaming unit to be merged with the data path, eliminating the possibility of aggressive data prefetching). In the SOpenCL-produced FPGA implementation, data for the graph nodes and edges is stored in the on-chip FPGA BRAMs, which are characterized by very fast (single-cycle) latency. By generating multiple memory addresses in every clock cycle, graph nodes can be accessed with minimal latency (provided there are no conflicts to the same BRAM) contributing to overall faster execution times.

Combinational Logic - Cyclic Redundancy Check (CRC)

Cyclic Redundancy Check (CRC) is an error-detecting code designed to detect data errors (e.g., during to network transmission). A polynomial division by a predetermined CRC polynomial is performed on the input data stream and the division remainder constitutes the stream's CRC value. This value is typically added to the end of the transmitted data stream. At the receiver end, a division of the augmented data stream with the (same, pre-determined) polynomial, will yield zero remainder on successful transmission. CRC algorithms that perform at the bit level are rather inefficient and many optimizations have been proposed that operate in larger units, namely 8, 16 or 32 bits. Listing 4.5 presents the basic CRC algorithm working on an 8-bit granularity.

```
Polynomial = 0xEDB88320
length = length of message (in bytes)
prevCRC32 = 0
crc = prevCRC32 XOR 0xFFFFFFFF
while (remaining bytes in message)
  crc = crc XOR current byte of message
  for (j = 0 to 8)
    if (crc & 1)
      crc = (crc >> 1) XOR Polynomial
    else
      crc = crc >> 1
    end if
  end for
end while
return ~crc
```

Listing 4.5: CRC algorithm

The implementation in OpenDwarfs follows a byte-based table-driven approach, where the values of the look-up table can be computed ahead of time and reused for CRC computations. The algorithm we use exploits a multi-level look-up table structure that eliminates the existence of an additional loop, thereby trading-off on-the-fly computation with the need for pre-computation and additional storage. Figure 4.12 shows the parallel OpenCL mapping of CRC and provides a small, yet illustrative example of how the algorithm is implemented in parallel in OpenCL: the input data stream is split in byte-chunked sizes and each OpenCL work-item in a work-group is responsible for performing computation on this particular byte. The final CRC value is computed on the host once all partial results have been computed in the device. Figure 4.13 supplements Figure 4.12 by illustrating how multi-level look-up tables used in the kernel work and their specific values for the example at hand.

Figure 4.14 shows the CRC performance results. CRC, being a representative application of combinational logic algorithms is characterized by abundance of simple logic operations and data parallelism at the byte granularity. Such operations are fast in most architectures, and can be typically implemented as minimal-latency instructions, in comparison to complex instructions (like floating point division) that are split across multiple stages in modern superscalar architectures and introduce a slew of complex dependencies. Given the computational pattern of the CRC algorithm at hand, which is highly parallel, we are not surprized to observe high speed-ups for multi-threaded execution, in all platforms. For instance, in the Opteron 6272 CPU case, we observe a 12.2-fold speed-up over the single-threaded execution. Similarly, Xeon Phi execution for the OpenCL kernel reaches maximum hardware thread utilization, according to our profiling results. The integrated

Host code:

```
localWorkSize=getMaxWorkitemsPerWorkgroup();
globalWorkSize=N_bytes/localWorkSize-N_bytes%localWorkSize;
Kernel();
for(i=0; i<N_bytes;i++){
        crc=crc^crc_loc[i]; //crc_loc[] contains crc for byte i,
                            //calculated on the device.
}
```

Kernel:

```
tid=getGlobalId();
If(tid<N_bytes){
        tmp=in_stream_byte[tid];
        val=N_bytes-tid;
        for(i=0; i<numTables; i++){
                if( (val>>i)%2==1){
                        tmp=table[i][tmp]
                }
        }
        crc_loc[tid]=tmp;
}
```

Example:

S = 0000101100000011
in_stream_byte[] =
{00001011, 00000011}

Work-item 0

tid=0
tmp=00001011(=11)
val=2-0=2
i=0: (condition false)
i=1: tmp=table[1][11]
crc_loc[0]=tmp;

Work-item 1

tid=1
tmp=00000011(=3)
val=2-1=1
i=0: tmp=table[0][3]
i=1: (condition false)
crc_loc[1]=tmp;

Figure 4.12: Parallel OpenCL implementation of CRC and example

GPUs in our experiments, which belong to the same architecture family, exhibit performance that is analogous to their number of *cores* and *threads per core* (as defined in Table 4.3). HD 7970, is a representative GPU of the AMD GCN (Graphics Core Next) architecture and bears fundamental differences to its predecessors, which may affect performance, as we see below.

With respect to the algorithm's underlying communication patterns, memory accesses in CRC are affine functions of a dynamically computed quantity (*tmp*). Specifically, as we see in Figure 4.12, inner-loop, cross-iteration dependencies due to stored state in variable *tmp*, cause input data addresses to the multi-level look-up table to be runtime-dependent. Obviously, this implies lack of cache locality, is detrimental to any prefetching hardware utilization and hence results to poor

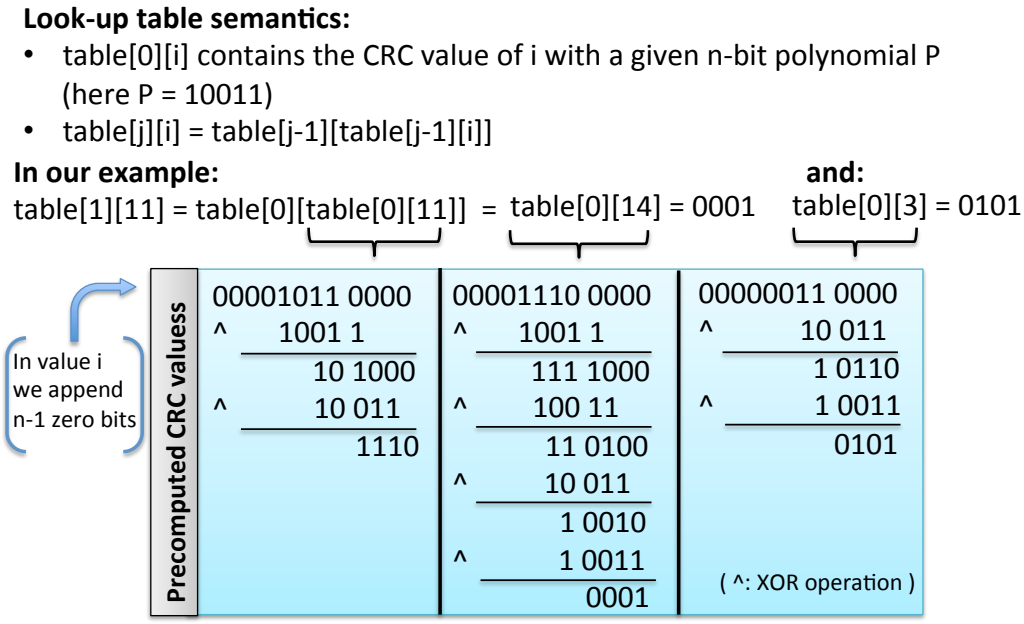


Figure 4.13: CRC look-up table semantics

overall cache behavior. The effect of such cache behavior is highlighted by our findings in profiling runs across our test architectures. All three GPUs suffer from cache hit rates that range from 5.48% to 7.13%. Depending on the CRC size, such precomputed tables may be able to fit into lower level caches. In such cases, more efficient data communication may be achieved, even in the adverse, highly probable case of consecutive data accesses spanning multiple cache lines. CRC is yet another dwarf that benefits from fast cache hierarchies.

Of course, in algorithms like this where operations take place on the byte-level the existence of efficient methods for accessing such data sizes and operating on them is imperative, if one is to fully utilize wider than 8-bit data-path, bus widths, etc. Such an example is SIMD architectures that allow packed operations on collections of different data sizes/types (such as bytes, single

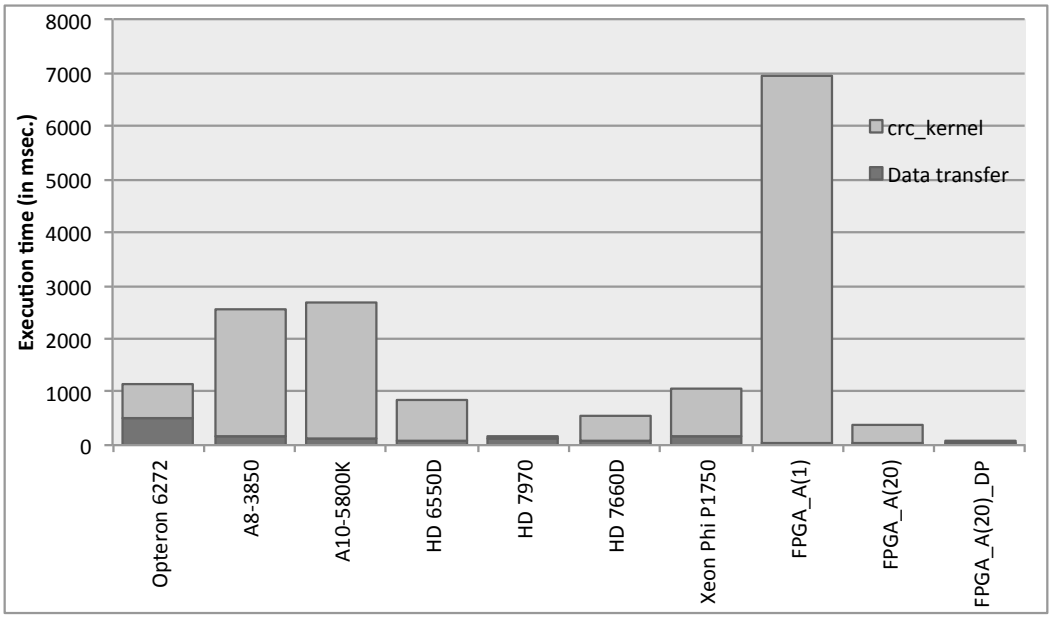


Figure 4.14: CRC performance results

or double precision floating point elements). CPU and GPU architectures follow a semantically similar approach.

Profiling for Xeon Phi corroborates a combination of the above claims. For instance, *vector intensity* is 14.4 close to the ideal value (16). This metric portrays the ratio between the total number of data elements processed by vector instructions and the *total* number of vector instructions. It highlights the vectorizability opportunities of the CRC OpenCL kernel, and helps quantify the success of the Intel OpenCL compiler's vectorization module in producing efficient vector code for the MIC architecture.

L1 compute to data access ratio is a mere 2.45. The ideal value would be close to the calculated vector intensity (14.4). This metric portrays the average number of vector operations per L1 cache access and its low value highlights the irregular, dynamic memory access pattern's toll in caching.

In this case vector operations, even on high-width vector registers will not benefit performance being bounded by the time needed to serve consecutive L1 cache misses.

On the FPGA, the SOpenCL implementation cannot disassociate the module that fetches data (input streaming unit) from the module that performs computations (data path), hence, reducing the opportunity for aggressive prefetching. A Processing Element (PE) is generated for the inner for-loop (FPGA_A(1)). This corresponds to a "single-threaded" FPGA implementation. If multiple FPGA accelerators are instantiated and operate in parallel, the execution time is better than that of the lower-end HD 6550D GPU. The number of accelerators that can "fit" in an FPGA is a direct function of available resources. In our case, up to 20 accelerators can be instantiated in a Virtex-6 LX760 FPGA, each reading one byte per cycle from on-chip BRAM (FPGA_A(20)). The area of accelerator can be reduced after bitwidth optimization. Utilization of fully customized bitwidths results to higher effective bandwidth between BRAM memory and the accelerators, which in turn translates to performance similar to that of HD 7970, with a more favorable performance-per-power ratio (FPGA_A(20)_DP).

Sparse Linear Algebra - Compressed Sparse Row Matrix-Vector Multiplication (CSR)

From an algorithmic standpoint, CSR in OpenDwarfs calculates the sum of each of a matrix's rows' elements, after it is multiplied by a given vector. However, the difference of CSR to the traditional matrix-vector multiplication lies on the fact that the matrix is not stored in its entirety, but rather in a compressed form known as compressed row storage sparse matrix format. This matrix representation is very efficient in terms of storage when the number of non-zero elements

is much smaller than the zero elements. Listing 4.6 presents the CSR algorithm that employs a sparse structured. As can be seen the computation is row-wise with respect to the original matrix represented via the sparse structure.

```
for i = 0 to num_rows - 1
  row_start_in_Ax = Ap[i]
  row_end_in_Ax = Ap[i+1]
  sum = 0
  for j = row_start_in_Ax to row_end_in_Ax
    sum += Ax[j] * X[Aj[j]]
  end for
  Y[i] = sum
end for
```

Listing 4.6: CSR algorithm

Figure 4.15 provides an example of how a "regular" matrix corresponds to a sparse matrix representation. Specifically, only non-zero values are stored in Ax (thus saving space from having to store a large number of zero elements). Alongside, Aj[i] stores the column that corresponds to the same position i of Ax. Ap is of size num_rows+1 and each pair of positions i, i+1 denote the range of values for j where Ax[j] belongs to that row. The pseudocode of CSR and a small, traceable example is depicted in Figure 4.15 and Figure 4.16 shows the performance results for CSR.

As we see in Listing 4.6, sparse matrix-vector multiplication entails a reduction that is performed across each row, in which the results of the multiplication of that row's non-zero elements with the corresponding vector's elements are summed. Such operations' combinations, which are typical in many domains, such as digital signal processing, can benefit from specialized *Fused multiply-add* (FMADD) instructions and hardware implementations thereof. This is yet another example where

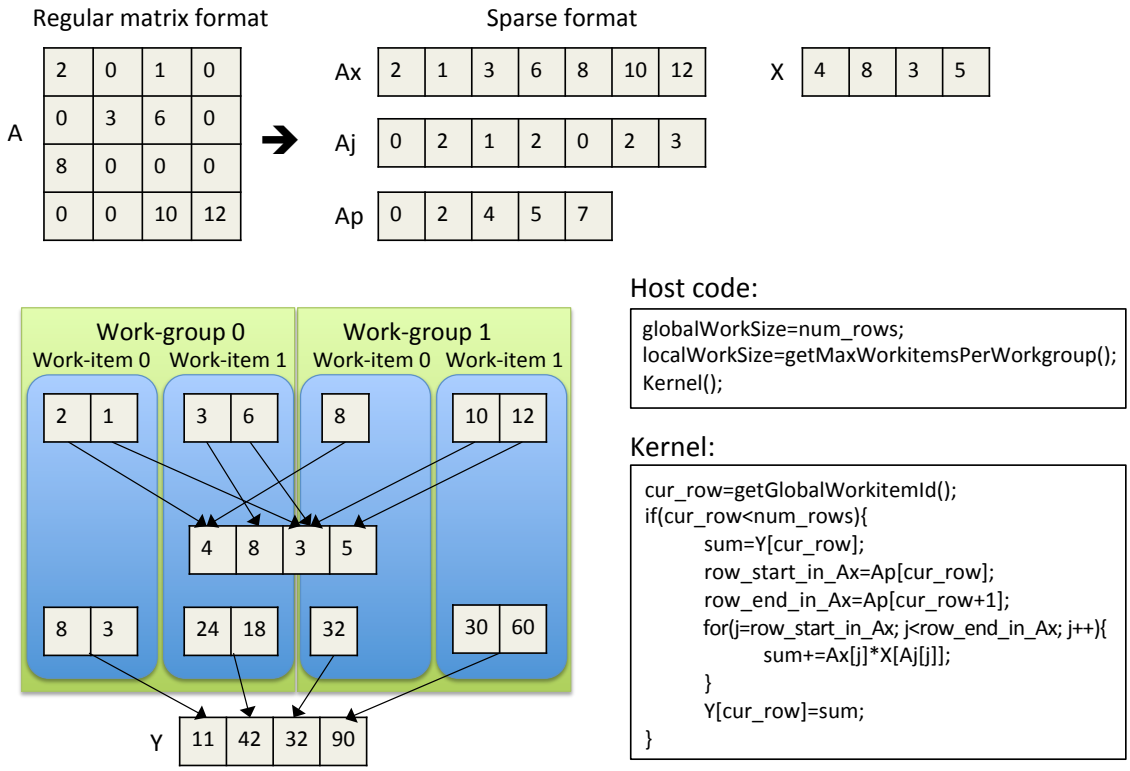


Figure 4.15: Representation and parallel OpenCL implementation of CSR

a typical, recurring combination of operations in a domain is realized in a fast, efficient way in architecture itself. FMADD instructions are available in CPUs, GPUs, and Intel Xeon Phi alike. OpenDwarfs, based on the dwarfs concept that *emphasizes* such recurring patterns, seeks to aid computer architects in this direction.

CSR is memory-latency limited and its speed-up by activating multiple threads on the two CPUs is low (5-fold and 1.8-fold for 16 and 4 threads on the Opteron and Llano CPUs, respectively). While performance in absolute terms is better in HD 7970 and Xeon Phi, its bad scalability is obvious and speed-ups compared to the CPU multithreaded execution are mediocre. As we can

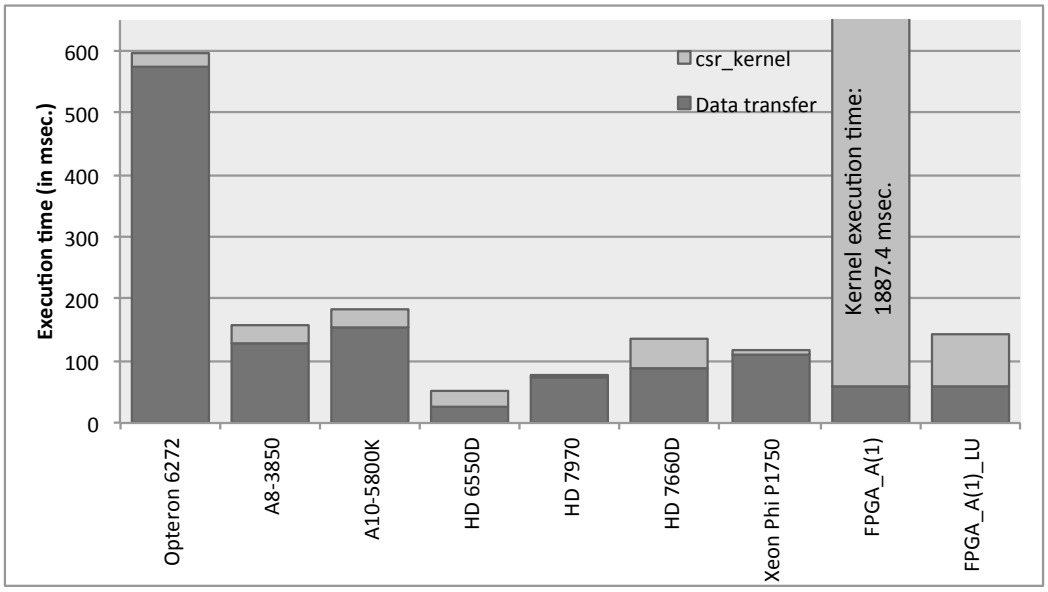


Figure 4.16: CSR performance results

see in Figure 4.15, data parallelism in accessing vector x is based on indexed reads, which limits memory-level parallelism. As with other dwarfs, such runtime-dependent data accesses limit the efficiency of mechanisms like prefetching. Indeed, in contrast to dwarfs like *n-body* the number of prefetch instructions emitted in all three CPUs, as well as in Xeon Phi are very low. Gather-scatter mechanisms, on the other side, are an important architectural addition that alleviates the effects of indirect addressing that are typical in sparse linear algebra. Especially in sparse linear algebra applications, the problem is aggravated from the large distance between consecutive elements within a row's operations (due to the high number of - conceptual - zero elements in the sparse matrix) and elements across rows (depending on the parallelization level/approach, e.g., multithreading, vectorization). In these cases, cache locality is barely existent and larger caches may only prove of limited value. Overall cache misses are less in Opteron 6272 that employs a larger L2 cache and an L3 cache, compared to the rest of the CPUs. On the GPU side, we have similar observa-

tions: HD 7970 13.27% cache hit rate, followed by 4.3% and 3.93% in HD 7660D and HD 6550D, respectively. The memory unit is busy (*MemUnitBusy* and *FetchUnitBusy* counters for HD 7990 and HD 6550D/HD 7660D) for most of the kernel execution time (reaching 99% in all the GPU cases). Any cache or memory effects are taken into account and the above indicates the algorithm in GPUs is fetch-bound. *VALUBusy* and *ALUBusy* counters indicate a reciprocal trend of low ALU utilization, ranging from 3-6%. Even during this time, ALU vector packing efficiency, especially n Llano/Trinity is in the low 30%, which indicates ALU dependency chains prevent full utilization. The case is not much different in Xeon Phi, where the ring interconnect traffic becomes a serious bottleneck, as L1 and L2 caches are shared across the 61 cores.

In the FPGA implementation of sparse matrix-vector multiplication, cross-iteration dependence due to $y[row]$ causes tunnel buffers to be used to store $y[row]$ values. Tunnels are generated wherever a load instruction has a read-after-write dependency with another store instruction with constant cross-iteration distance larger than or equal to one (FPGA_A(1)). Allowing OpenCL to fully unroll the inner loop dramatically improves FPGA performance by almost 23-fold because it reduces iteration interval (II) from 8 down to 2 (FPGA_A(1)_LU).

4.2 On the Performance of Manually Optimized Dwarf-Based Applications: GEM, an N-Body Dwarf

In Section 4.1 we presented the performance evaluation of a subset of dwarf applications across heterogeneous platforms using a common programming language, i.e., OpenCL, and discussed the interdependence of the underlying architectural characteristics and the computation and communication patterns embodied by architecture-agnostic dwarf implementations. Our insights with respect to the above may be of use to any of the three interest target groups we outline in the beginning of this dissertation: computer architects, programmers, and compiler/tool writers.

In this section, we keep our focus on performance, but we approach and evaluate it from a different standpoint. Specifically, we steer the discussion from architecture-agnostic implementations to implementations that span the optimization search-space from the architecture-agnostic to the architecture-aware in an attempt to obtain the best performance on each target platform. Achieving the best performance possible entails broadening the programming languages and paradigms search-space. In the context of the above, our study includes C and CUDA, the OpenMP and OpenACC programming extensions, as well as manual vectorization intrinsics (SIMD). In terms of our case-study application, we select to use GEM [17], an instantiation of the n-body dwarf that addresses the problem of molecular modeling via electrostatic surface potential.

We perform a characterization of architecture-aware optimizations across heterogeneous platforms, and across languages and programming techniques. Our systematic approach to optimization delivers implementations with speed-ups of $194.98\times$, $885.18\times$, and $1020.88\times$ on the CPU, Xeon

Phi, and GPU, respectively, over the naïve serial version. Our study can provide useful insights mainly to programmers who can identify the best platform and language combination for applications that fall under the n-body dwarf category. However, it can also benefit computer architects, since our characterization of different optimization levels takes into account architectural features (e.g., caches, hardware prefetching, special instructions) that can benefit performance. Similar insights can benefit compiler/tool writers that can potentially incorporate specific optimizations into compilers, tools, and run-time systems.

4.2.1 Molecular Modeling via Electrostatic Surface Potential (ESP)

Molecular modeling refers to the mathematical models that seek to describe the behavior and properties of biological molecules and the corresponding computational techniques. An important part of molecular modeling simulation in areas like materials science, computational chemistry, and rational drug design is the calculation of electrostatic surface potential (ESP) in support of locating bonding sites and other features.

The computational pattern of GEM [17] is an all-to-all n-body interaction between points near the molecular surface and the atoms within the biomolecule. The overall result of ESP calculation, i.e. the electrostatic map of a biomolecule, provides useful information about its function. The long-range nature of electrostatic interactions results in a computationally intense workload of order $O(nm)$ where n is the number of atoms and m is the number of surface points. Many approximation methods have been proposed over the numerical solutions of the Poisson-Boltzmann

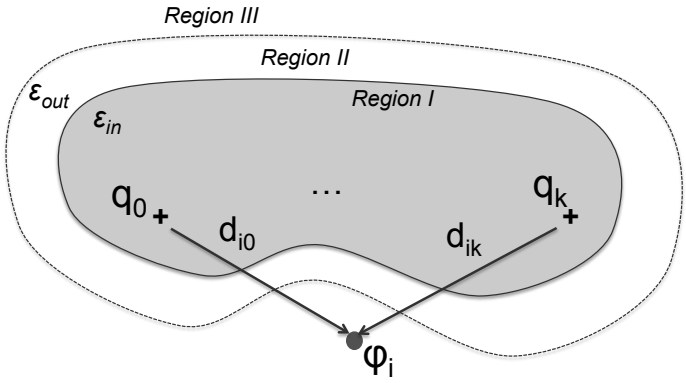
equation [34] that constitute the core of traditional ESP calculation algorithms. One such method is employed by GEM, the ESP application we study in this chapter. Figure 4.17 depicts a biomolecule and the parameters that enter the equation that provides the electrostatic potential due to a point charge at any point of the solvent near the molecular surface.

In GEM, the biomolecule is divided in three distinct regions, and separate functional forms of the electrostatic potential ϕ_i apply for each.[1] The electrostatic potential at each point near the surface (*vertex*) is the sum of electrostatic potentials contributed by each single point charge to that point. Similarly, the sum of potentials at all surface points define the total electrostatic potential of the system. The above computation and communication pattern classifies GEM as an n-body dwarf [28] with the subtle difference that it performs all-pair computations between two sets (versus one). Thus, we expect many of our conclusions regarding performance and programmability of GEM to apply to most n-body applications.

4.2.2 Evaluated Platforms

In the context of optimizing GEM, we evaluate three distinct parallel platforms and their attendant programming models. The baseline multi-core CPU is represented by Intel's Sandy Bridge x86-64 CPUs, specifically two Xeon model E5-2680s, and uses the C language and the Intel compiler suite with Intel OpenMP directives for parallelism. Our CPU platform is indicative of a standard server node with cache-coherent, moderate-latency NUMA memory; large well-tuned caches; and all the niceties of traditional "fat" CPU cores.

[1] Details for each of the regions and the corresponding functional forms are given in [116].



$$\varphi_i = \sum_{j=0}^{j=k} \frac{q_j}{\varepsilon_{out}} \frac{1}{(1 + a(\varepsilon_{in}/\varepsilon_{out}))} \left(\frac{(1+a)}{d_{ij}} - \frac{a(1 - (\varepsilon_{in}/\epsilon_{out})}{r} \right)$$

Figure 4.17: Electrostatic potential interactions between a molecular surface point and atom charges within the molecule

Moving to the Intel Xeon Phi, much of the CPU architecture is preserved. The instruction set is highly similar to that of the x86-64 CPUs and can be natively programmed by the same interfaces. In fact, our evaluations in this work use the same libraries and compilers at all phases for both the CPU and Xeon Phi. Even so, the Xeon Phi differs substantially at the architectural level. The Xeon Phi uses multiple banks of high-throughput but high-latency graphics memory and offers 512-bit SIMD units and four thread contexts on each core, double the width offered by AVX and double the thread contexts on the Sandy Bridge CPUs. Thus, the Xeon Phi architecture shifts the compute/memory ratio to favor throughput rather than latency-centric computing. In the same vein, the cores are comparatively simple in-order cores with only minimal prefetching support.

Finally, the GPU, represented by an NVIDIA K20c, eliminates the cache-coherent memory offered by the other platforms. Otherwise, the GPU is architecturally more similar to the Xeon Phi. Both use graphics memory and wide SIMD units to offer high throughput and many thread contexts

to mitigate the effects of latency. The difference is in the programming model. Since GPUs are SIMD engines, GPU programming models such as CUDA have no concept of running a single thread with scalar mathematics. Instead, the programming model assumes that many threads will execute every instruction in SIMD fashion. While in the other platforms, SIMD support is either compiled in or added with intrinsics; in CUDA/OpenCL, SIMD is the standard state of affairs, and single-threading must be produced manually.

4.2.3 Algorithm Mapping to Heterogeneous Platforms

Exploiting a parallel computing architecture starts with identifying the parts of the algorithm that can execute in parallel. Some algorithms are amenable to parallelism, while others are characterized by complex dependencies that make parallelization an onerous task. Even though the calculation of electrostatic surface potential (ESP) in GEM belongs to the first category, its mapping and optimization onto different parallel computing devices still presents a gamut of challenges of varying difficulty, particularly with respect to programmability.

Figure 4.18a shows how we expose multi-level parallelism on the Sandy Bridge CPU (SNB) and Xeon Phi co-processor (XP). We present these devices together because they share a similar architecture and programming model, and therefore, the same basic parallel algorithmic mapping and optimization. Two sets of mapping and optimization effectively automate parallelization with minimal programming effort: (1) directive-hinted loop parallelization and (2) auto-vectorization. For the first, all available threads in SNB and XP are assigned a portion of the total surface points (ver-

tices) on which ESP needs to be calculated. This can be trivially achieved by using the directive-hinted loop parallelization of OpenMP. For brevity, we refer to this as *auto-parallelization*. For the second, the ESP contributed by each atom to a given surface point is calculated in a data-parallel manner (i.e., 8 or 16 atoms simultaneously) using vector arithmetic. This corresponds to 8- or 16-way single-precision, floating-point operations using the available 256- or 512-bit wide vector registers and execution units on SNB and XP, respectively. This optimization can be achieved by simply setting an appropriate compiler flag for *auto-vectorization* and can be quite efficient, as presented in Section 4.2.5.

Figure 4.18b shows how we map GEM onto the K20 GPU (K20). Due to the abundant number of threads that GPU architectures offer, we allocate one surface-point vertex per thread and have each thread calculate the sum of all the contributions of atom charges. In this case, when the programmer uses the CUDA programming model, (s)he needs to define the kernel configuration, i.e., number of threads per block and number of blocks, and allocate memory space on the GPU and transfer data between the host CPU and the GPU, as needed. With the arrival of *OpenACC*, a variant of the aforementioned OpenMP, all of the above is automated via OpenACC directives. As a related (but different) analog to auto-vectorization, the PGI compiler for OpenACC also allows for automatic optimization by using the *fast math compiler flag*.

The above discussion addresses the high-level parallel mapping of the algorithm to the underlying hardware and the (mostly) automated optimizations that provide an initial speed-up over the serial implementation. To extract more significant performance gains from the hardware, we need to

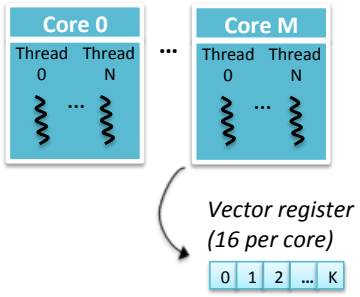
apply a series of optimizations, some that are applicable to *all three platforms* and some that are *platform-specific*. A detailed discussion of such optimizations is presented in Section 4.2.4.

4.2.4 Optimization

In this section we discuss the optimizations applied across each platform. Most of the optimizations presented are beneficial for all platforms under consideration. Removal of conditional statements and flattening of data structures have been applied to the serial CPU version we use as a baseline. We explicitly mention when an optimization only applies on a subset of the target platforms.

Vectorization and multithreading: Since GEM is a data-parallel n-body code, each output potential can be calculated independently of all others. This state is commonly referred to as "embarrassingly parallel," and makes the first and most important optimization the use of parallelism to divide the workload across as many compute resources as possible. On the CPU and Xeon Phi, we use all thread contexts across all cores, as well as all SIMD lanes wherever possible. We use hand-tuned AVX/MIC vector code to pack and operate on 8/16 atoms at a time for a given vertex. GPUs, featuring an abundance of thread contexts, allow mapping the potential calculation for each given vertex to a separate GPU thread.

Removal of conditional statements: Conditional branches incur execution time overhead on all three platforms, despite efficient branch prediction on the CPU. Since the conditionals in the parallel portion of GEM are all pre-determined, rather than diverging on a per-vertex basis, all of them can be hoisted out to a single conditional nest used to choose a final computational function



M cores:
8 (SNB), 60 (KNC)

N threads per core:
2 (SNB), 4 (KNC)

K (floating point elements per vector register):
8 (SNB), 16 (KNC)

Basic Mapping of the Algorithm to the Sandy Bridge CPU (SNB) / MIC Co-processor (KNC):

Main computational loop

```
for(i=0; i<numOfVertices; i+=numOfCores*numThreadsPerCore){
        Launch (in parallel) function on all threads at every core that
        calculates the electrostatic surface potential at
        verticesPerThread vertices
}
```

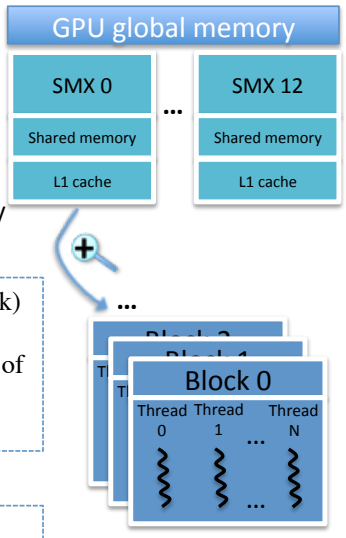
Function (ESP calculation for *numOfCores*numThreadsPerCore/ numOfVertices* vertices)

```
for(i=0; i<verticesPerThread; i++){
        for(j=0; j<numOfAtoms; j+=K){
                Load K atoms (coordinates/charge) into vector register
                Calculate potential contribution of K atoms in
                parallel using vector instructions
        }
}
```

(a) Sandy Bridge CPU and Xeon Phi co-processor

Basic Mapping of the Algorithm to the GPU:

A. Allocate GPU global memory for:
   - Coordinates of surface vertices
   - Coordinates and charge of each atom in the molecule
B. Copy above data from host (CPU) to device (GPU) memory
C. Start main computational loop on host (see below)
D. Copy results (calculated potential) from device to host memory



Main computational loop (CPU)

```
for(i=0; i<numOfVertices; i+=numOfBlocks*numThreadsPerBlock)
{
        Launch GPU kernel on numOfBlocks blocks each consisting of
        numThreadsPerBlock (performed in parallel in all SMXs)
}
```

GPU kernel: main computational loop

```
for(i=0; i<numOfAtoms; i++){
        Calculate potential contribution of current atom and add it to the
        total electrostatic potential of current vertex
}
```

Each thread calculates electrostatic surface potential at one vertex

(b) Kepler K20 GPU

Figure 4.18: Mapping of GEM algorithm

with no conditionals in it. This saves us both dynamic instructions as well as potential branch mis-predictions on all devices at the cost of having several replicated versions of the function expressing each necessary code path.

Flattening of data structures: Laying out data as an array of structures (AoS) can seriously impact vector code performance. The AoS layout is a major cause of misaligned (in CPUs) and non-coalesced (in GPUs) memory accesses. More importantly, AoS complicates the mapping of data to vector units. For example, given a structure of two ints A and B, the AoS layout intersperses As with Bs, forcing at least two vectorized gather loads to load a vector register. If, on the other hand, the As are in one array and Bs in another, only a single load is required. In GEM, we transform the AoS used to store the coordinates and charge of surface points (*vertices*) and atoms, into multiple arrays, each containing a single component (e.g., charge) of the structure.

Approximate reciprocal instructions: Floating-point division and square root are high-latency operations that stall the pipelines of the CPU and Xeon Phi devices as well as working in a lower-width mode on the CPU. In order to avoid as many of them as possible we replace them with their low-latency approximate reciprocal counterparts. These instructions have much lower latency and make use of look-up tables to calculate the result. Their drawback is their reduced accuracy. On the Sandy Bridge CPU they are accurate to the 12 most significant bits of the mantissa. We tackle the reduced accuracy problem by using an iteration of the Newton-Raphson (NR) method, which increases accuracy to a minimum of 23 of 24 bits for single precision numbers. More details about this method can be found in [152]. On Xeon Phi, the corresponding reciprocal instructions natively provide accuracy of 23 of 24 bits of the mantissa. On K20, the corresponding *__frsqrt_rn()* intrinsic

we use is fully IEEE-compliant. For the two latter cases, we do not need to apply Newton-Raphson, keeping the number of instructions lower than in the CPU (Table 4.6). In any case, the root mean squared error (RMSE) of calculated potential values for all implementations against the original serial version's output does not exceed 0.000084.

Outer loop unrolling: Each of the m iterations of the outer loop of an n-body problem entails computation against a set of n bodies of the inner loop. In ESP calculation this corresponds to surface points (vertices) and atoms, respectively. As a result, each iteration of the outer loop requires n memory loads (all atoms that contribute to the potential of a given vertex). By "unrolling" the outer loop by a factor k (i.e, calculating potential at k vertices at a time), we reduce the innermost loop's atom loads by the same factor.

Cache blocking and software prefetching: Converting arrays of structures to multiple arrays enhances spatial locality and cache use efficiency in all three platforms. Moreover, the algorithm's regular memory access patterns facilitate hardware data prefetching in our multi-core platforms. However, the large number of atoms in the innermost loop leads to eviction of relevant atom data from the lower level caches, before they are fully reused. To alleviate this problem we apply cache blocking, where each thread loops over all its assigned surface points and calculates potential contribution for blocks of atoms at a time. Block size is theoretically calculated, based on the data size accessed with each iteration and cache details, and experimentally tuned and verified. Finally, the programmer can assist the hardware prefetcher by emitting the prefetch intrinsic with appropriate prefetch distance as a parameter.

Shuffling method: This optimization is not applicable to all kinds of algorithms, but is especially useful in specific n-body problems, as in our case. In this method (Figure 4.19), we change the default computation pattern, where each (same) vertex point is loaded at all positions of a vector register and loops over all atoms. Instead, we load *N distinct* vertices at a time in a vector register, load *N* atoms in another vector register (where *N* is 8 for AVX) and then shuffle (i.e., rotate in a wrap-around fashion) the data elements of the latter, using the corresponding shuffle intrinsics. This way we can obtain all the possible combinations of vertices and atoms, as defined by the algorithm's all-to-all computation pattern, with a reduced number of vector loads. In addition to CPUs and Xeon Phi, Kepler architecture has introduced a shuffle instruction that achieves similar functionality in the context of a thread warp.

8-byte shared memory access (in Kepler): Kepler GPU architecture features 32 shared memory banks, 8-bytes wide each, with a corresponding bandwidth of 8 bytes/bank/clock per streaming multiprocessor (SMX). Default mode defines 4-byte access to support backward compatibility and similar bank-conflict behavior with Fermi, a behavior that leads to sub-optimal bandwidth for certain access patterns. To exploit the Kepler-supported 8-byte access mode, the programmer needs to use the appropriate CUDA function and then, in the case of GEM, transform floating point (FP) type variables to *float2* type variables in a suitable manner (e.g., six FP ones to three float2).

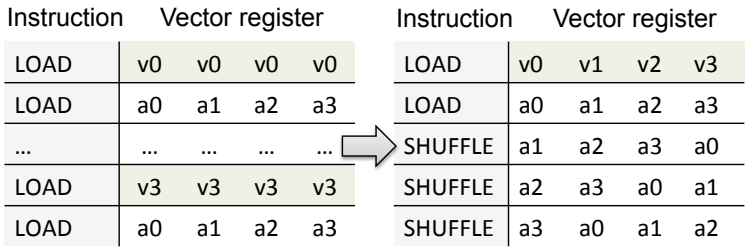


Figure 4.19: Shuffling optimization

4.2.5 Results and Discussion

In this section we describe the experimental setup and discuss the effects of the application of optimizations on each platform in terms of performance. At the end of this section, we discuss the best optimization methods set for each platform and perform a cross-platform comparison of the corresponding implementations. Finally, we discuss their efficiency with respect to the theoretical peak throughput, as experimentally derived by means of assembly code inspection and taking into account each platform's specifications.

Experimental Setup

We evaluate our GEM [17] implementations across various optimization levels on three multi- and many-core platforms: a Sandy Bridge CPU (SNB), Xeon Phi co-processor (XP), and Kepler K20 GPU (K20), as noted in Table 4.5. Results for all structures we experimented with, which comprised a different number of vertices and atoms, show similar trends, which is characteristic of n-body methods once the workloads are big enough to saturate available computation units on each platform. For brevity, we only present results for the *tobacco ring virus capsid (1A6C)* biomolecular structure, which requires ESP calculation between 593,615 surface points and 476,040 atoms.

Table 4.5: Architectural parameters

Model (Architecture)	Intel E5-2680 (Sandy Bridge)	Intel Xeon Phi P1750 (MIC)	NVIDIA K20c (Kepler)
Frequency Cores Threads/core	2.7 GHz 16 (8/socket) 2	1.09 GHz 61 4	706 MHz 13 SMXs 16 (blocks/SMX)
SIMD (SP) GFLOPS (SP)	8-way 691.2	16-way 2092.8	192-way 3524.35
Mem. BW	102.4 GB/s	320 GB/s	208 GB/s
L1/L2/L3 cache (KB)	32/256/20480 (L1,L2 per core)	32/512/- (L1,L2 per core)	48(max)/1280/- (L1 per block)
Power	260W	300W	225W
Compiler	ICC 13.0	ICC 13.0	NVCC 5.0

In all experimental runs, we use the full parallelization available on the platform 32 threads on SNB, 240 threads on XP (leaving one core for the system software) and 1024 threads per block on the GPU with enough blocks to cover the workload (theoretically achieving 100% occupancy according to the NVIDIA occupancy calculator). Our results are all reported based on the runtime of the computational kernel, setup and data transfer costs are not included. While these costs are important, our focus in this work is the effectiveness of each level of optimization on each platform, which is independent of the data transfer costs and unaffected by them.

Performance Progression by Architecture

In Figure 4.20 we present speed-up over the reference single core implementation. Optimizations are cumulative as we move to the right for each platform, unless otherwise noted and the *Manually vectorized* results include the approximate reciprocal instructions. Below we conduct a quantitative analysis of the effect of each optimization series on each platform.

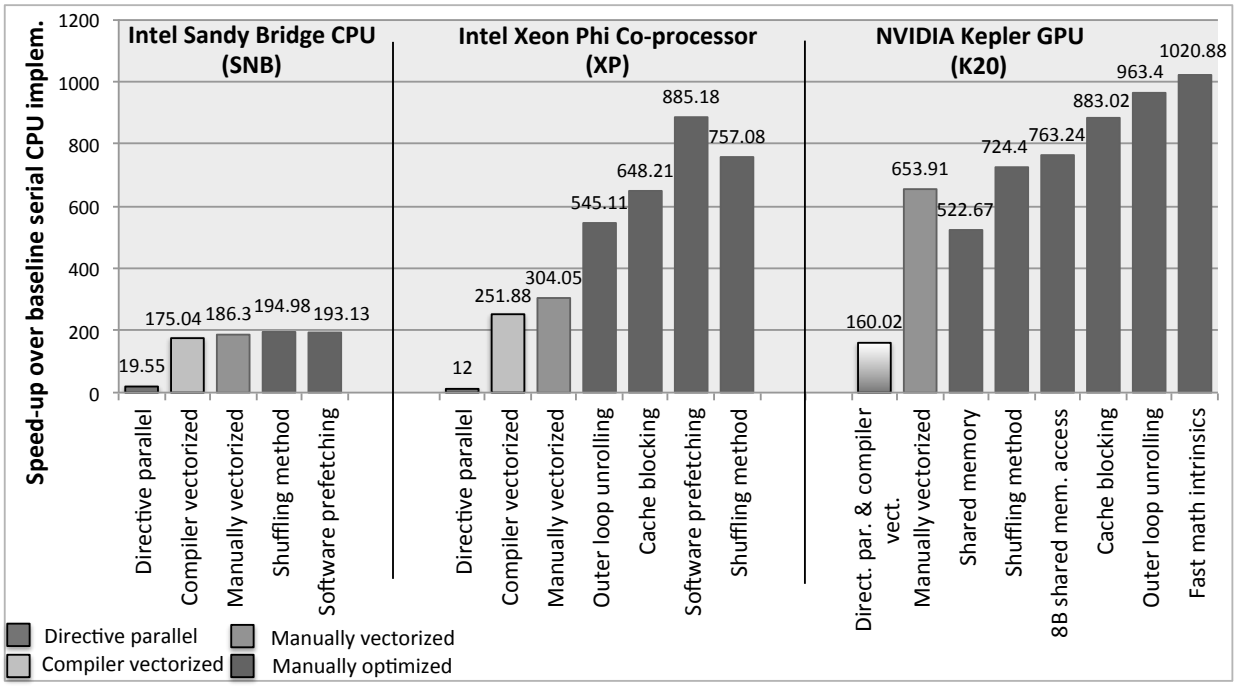


Figure 4.20: Step by step optimizations

Sandy Bridge CPU (SNB): Looking at the performance of directive-based parallelization and compiler-assisted vectorization, we observe that the compiler proves very efficient, when compared to the manually vectorized implementation that makes explicit use of vector intrinsics. This is the case both when we use accurate and approximate versions of the intrinsics and the corresponding compiler flags. On SNB we observed, however, that when using the approximate reciprocal intrinsics (included in *Manually vectorized*), we get even better performance compared to using compiler flags for approximate division and square root (*Compiler vectorized*). The reason is apparent in assembly code level. Intel Compiler is not able to perform the algebraic changes we manually make to accommodate the fast reciprocal square root and division intrinsics. As such, the optimized code only uses approximate fast division and fast square root. Even in this case, we obtain performance improvement against using the regular division and square root. The main

reason is that the execution unit used for division and square root is still 128-bit and 256-bit packed division/square root is broken down into two 128-bit operations. Using the shuffle method, as described in Section 4.2.4, we achieve an extra 4.7% improvement. This optimization reduces the number of vertex loads by a factor equal to the SIMD vector width divided by the size of the data type we are using (e.g., 8 for AVX and float data type). It also helps reducing the number of times atoms' coordinates are loaded, by the same factor. See Figure 4.19 for a visual explanation. While the number of loads are reduced by a factor of 16 the achieved speed-up is much less impressive, as the loads occur at worst case in our SNB's large (20MB) L3 cache, which is fast by itself and even faster when combined with efficient hardware prefetching to this and lower -and faster- cache levels. Finally, using software prefetching does not offer any significant performance benefits in the case of multi-core CPUs. The reason behind this behavior is the advanced hardware prefetching capabilities of modern multi-core CPUs. For algorithms with regular memory accesses the hardware prefetcher can efficiently move data between the main memory and L2 or between L2 and L1 caches ahead of time based on previous access patterns.

Xeon Phi (XP): In contrast to SNB, we observe that using vector intrinsics on XP is slightly better than compiler vectorized code (1.20x). As in SNB, use of fast reciprocal math prevents bottlenecks in the corresponding units observed in the exact division/square root cases. The performance gap after multi-threading and vectorization have been applied has to be filled by manual code optimizations. In the case of architectures, such as Intel MIC, where there is a lack of large L3 cache, techniques that make efficient use of the available cache hierarchy are of great importance. One such technique, whose effectiveness on that aspect is algorithm-specific, is outer loop unrolling. In

our case, where the outer loop's vertices loop over the same set of atoms, unrolling the outer loop by a factor of two instantly reduces memory accesses by a factor of two and increases performance by 1.79x.

Cache blocking techniques enhance cache usage and result in a 1.19x speed-up versus not using them. Software prefetching instructions, when added on top of the earlier optimizations, yield an additional 1.37x improvement. As opposed to SNB, software prefetching is important in XP. One reason is that in XP, the hardware prefetcher proactively loads data between memory and L2 cache, but not from L2 to L1. This gap can be filled by blocking for L1 cache or software prefetching.

Last, we should mention that the shuffling method we used for SNB is a technique worth trying on XP, as well. As a matter of fact, shuffling by itself reduces atom coordinates' loads by a factor of 16 (i.e., SIMD-width). However, using shuffling with the optimizations we mentioned earlier results in a slight slowdown, as it mainly contributes unnecessary overhead, since outer loop unrolling, cache blocking and software prefetching address the expensive main memory transfers in an efficient way.

Kepler K20 GPU (K20): For K20 the naïve CUDA version performs 4.08x faster than the one produced by using OpenACC directives. This is as far as we can get by directive-based programming or naïve CUDA programming and compiler directives. To get anywhere beyond this performance we need to resort to lower-level optimizations. Ensuring coalesced global memory accesses is one of the first optimizations one has to consider on the GPU but since ours already are, optimization efforts should be geared towards utilizing shared memory for data that are accessed by all threads in a block, such as the atom coordinates and charge. We should note that even when not using shared

memory, the regular memory access patterns facilitate caching. As a matter of fact, our shared memory implementation boosts the number of registers used and along with shared memory limitations leads to reduced occupancy and performance, with respect to the preceding implementation. Adding the shuffling method to a shared memory implementation boosts performance to 724.4x over the single core CPU implementation, 10% faster than the preceding implementation. For K20 using one float2, instead of 2 float variables, allows successive 8-byte words allocation in successive banks and increased 8-byte wide shared memory access. This results in a 1.054x speed-up. Blocking, which proved to increase performance on SNB and XP, is beneficial for K20 as well. In particular it accounts for an extra 1.16x. Finally, we perform similar algebraic changes as the ones we described for SNB and XP and make use of the fast reciprocal square root and division CUDA intrinsics, together with fused multiply and add instructions. An extra 6% performance gain is achieved by this optimization, leading us to the fastest of our K20 implementations at 1020.88x over the baseline.

Intrinsics and Approximation

Last, but not least, we leave the discussion of further optimizing the code using special intrinsics for fast approximate versions of instructions such as square root, division, fused multiply-and-add. Compilers, such as Intel Compiler (icc) and NVIDIA CUDA Compiler (nvcc) provide flags for automatic detection of the regular instructions (or combinations thereof- such as multiplications followed by an addition). Indeed, inspecting the assembly/PTX code respectively, we verify that both icc and nvcc make use of the corresponding fast instructions (given the algebraic changes

mentioned in Section 4.2.4). We should note that without performing these algebraic changes none of the compilers were able to automatically perform all the aforementioned optimizations, which is a field of further research. It is also worth mentioning that nvcc provided more efficient code with minimal use of manual intrinsics and the `-use_fast_math` parameter than our fully intrinsic based version. Given the nature of the problem, there must be a manual set which would behave as well, but the compiler does better than most. On the other hand, manually adding the intrinsics under consideration on the XP implementation drastically improved performance.

On the surface it might sound as though the NVIDIA CUDA compiler is performing more advanced conversion of instructions than those in the Intel compiler. The truth is somewhat more complicated. Since the CUDA programming model is implicitly vectorized, it does not require intrinsics to specify the intended width of instructions. In practice it just assumes all instructions are of width 32 and masks off the extra. On the other hand, the standard programming model used on the XP is serial and must be explicitly vectorized. Once intrinsics are used to ensure the correct vector width, it appears that they are not converted by the compiler even though it would have the right given the supplied options. Since intrinsics are meant to be a way to directly insert a particular instruction, it makes sense that the compiler does not change it, but it restricts the compiler from performing a potentially important set of transformations on those instructions. Adding explicit vectorization to the programming model without intrinsics, either through directives such as the `simd` directive in OpenMP 4.0 or through a language extension like CUDA, should solve this issue.

Performance Efficiency

By examining the assembly/PTX code of the best performing implementations for each device, we count the number of floating point (FP) instructions in the algorithm's critical region (innermost loop)– the larger number of single precision floating point operations in the SNB version is due to the accuracy correction approach applied with the addition of reciprocal divisions and square roots. Taking into account the potential overlapping of instructions on different units along with their cycle time, we calculate the expected number of cycles per iteration of that code region. From these numbers we can infer the expected vector instructions per cycle (IPC) and, given each platform's clock frequency, the maximum theoretical throughput in GFLOPS ignoring memory load costs. Subsequently, we calculate the achieved throughput and efficiency as the ratio of achieved to ideal performance for the particular algorithm and instruction mix, as shown in Table 4.6.

These results show a different side to the application performance than is portrayed either by performance, as in Figure 4.20 (or by the percentage of achieved performance as we would see in Figure 5.5 in the next chapter). Specifically, while K20 is the best performing overall, and despite the optimization effort expended on it, it remains at only 54.34% of theoretical peak performance. In principle that means that we should be able to get nearly *double* the performance we on that architecture. In practice our application is running with full occupancy and the most optimized instruction mix, shared memory behavior and instruction mix we have found. On the other hand, both the SNB and XP parts achieve greater than 80% of peak performance. This trend in perfor-

Table 4.6: Achieved performance over theoretical peak

	SNB	XP	K20
SP FP vector ops	52	35	32
Cycles/iteration	36	25	23
Ideal vector IPC	1.444	1.4	1.391
Theoretical peak throughput (GFLOPS)	499.2	1465	2451.7
Achieved throughput (GFLOPS)	413.8	1265.1	1332.3
Achieved efficiency	82.89%	86.35%	54.34%

mance efficiency has been noted before between CPUs and GPUs, but we find it telling that the XP achieves not just good efficiency, but higher than SNB in this case.

4.3 On the Performance of OpenCL as a Programming Method for FPGAs: a Preliminary Study with Altera OpenCL

Programming FPGAs in a way to extract the high performance reconfigurable computing has the potential to offer over fixed architectures has been an intrinsically arduous task that requires extensive knowledge of hardware design languages (HDLs), such as Verilog or VHDL, and excruciatingly low-level hardware details. In an attempt to render FPGAs more accessible, Altera and Xilinx extended the typical hardware design language (HDL) programming model by introducing a design process, based on OpenCL-based toolchains, that resembles the traditional CPU software development workflow. This OpenCL-based model (which we discuss in detail in Section 2.2.2) facilitates design, prototyping and implementation by moving towards a much higher level of abstraction, when compared to the intrinsically low-level nature of HDLs and obviating the require-

ment of HDL knowledge and other parts of FPGA design. However, while an OpenCL program written for another target platform (e.g., GPU) may run in a fairly straightforward way on an FPGA (*functional portability*), the acquired performance will be sub-par (lack of *performance portability*). We discuss the programmability and portability aspects of FPGA programming in more detail as part of our work towards enhancing them, in Chapter 5. Here, we focus on the peculiarities of OpenCL programming for Altera FPGA targets. While the level of low-level details needed for HDL programming is drastically minimized, the programmer still has to keep in mind certain unique characteristics of the underlying hardware (e.g., certain boards may lack hard-IP for floating point arithmetic). More importantly, one should be aware of how the Altera Offline Compiler (AOC) for OpenCL translates the OpenCL code to efficient hardware circuitry, and how certain programming choices affect FPGA *resource utilization*. Given the novelty of OpenCL programming for FPGAs, this is one of the first studies to address the above issues.

In this work, we conduct a preliminary evaluation of OpenCL as a language for programming reconfigurable target architectures, i.e., FPGAs. Similar to our study of an n-body code across a CPU, GPU, and Intel Xeon Phi in Section 4.2, we explore the limits of programming an FPGA using Altera OpenCL in the context of the same n-body application, thus concluding a thourough performance evaluation for this dwarf across all major heterogeneous architectures using manual optimizations. We experiment with kernel vectorization, compute-unit replication, evaluation of the efficacy of Altera OpenCL compiler optimizations, as well as algorithmic refactoring for FPGAs.

4.3.1 Experimental Setup

Hardware/Software: For the experiments in this chapter we use the *Bittware S5-PCIe-HQ* board (S5PHQ-D8) shipped in the form of a PCIe card (Figure 4.21). At its core lies a high-performance Altera Stratix V GS FPGA and 16 GB DDR3 SDRAM. The FPGA card is installed in a Linux-based (Debian, kernel v3.2.46) machine with an *Intel E5-2697* (Ivybridge) CPU and 64 GB RAM. We use the Altera OpenCL SDK (v14.2) to compile the device-side code (OpenCL kernels), and gcc v4.8.2 for the host-side code.

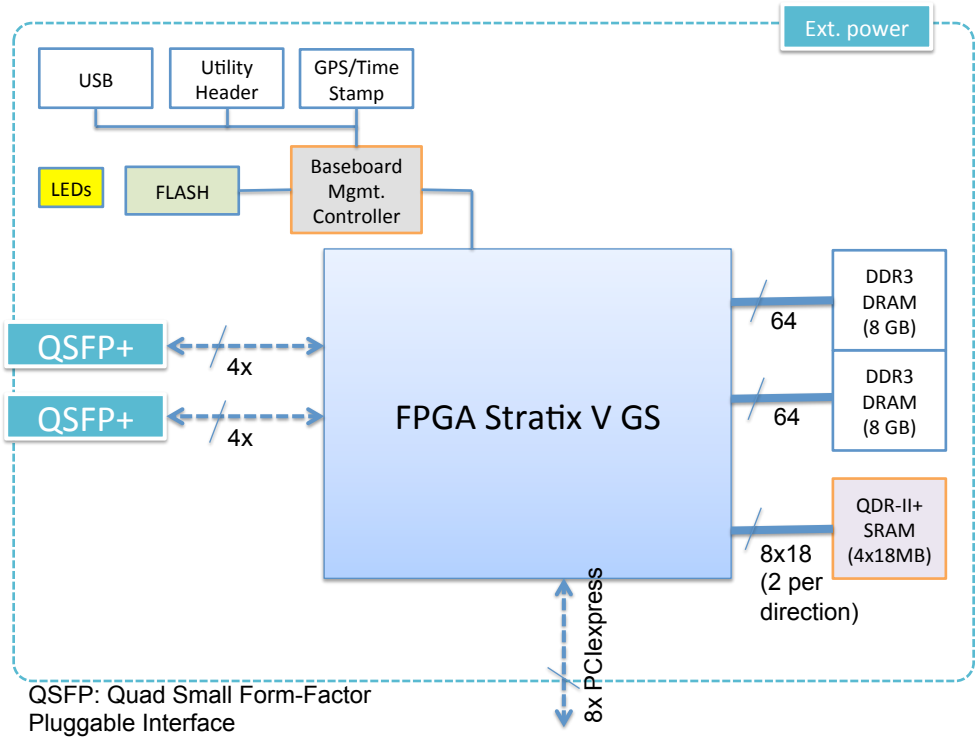


Figure 4.21: Bittware S5-PCIe-HQ architectural diagram

Benchmark: To study OpenCL as a programming method for FPGAs we use GEM, the same application we use to study the performance across the parallel architectures in Section 4.2, and

the same input dataset. GEM is an algorithm that describes the all-to-all n-body interactions in a biomolecule as they occur between points near the molecular surface and atoms within. We discuss molecular modeling via electrostatic surface potential and GEM in detail in Section 4.2.1.

4.3.2 FPGA Optimizations: Results and Insights

To improve the performance of OpenCL kernels on FPGAs, we can exploit different parallelism levels: task, data (SIMD vectorization) and pipeline parallelism. We can minimize memory accesses by controlling data movement across the memory hierarchy levels, and coalescing memory accesses. Since FPGAs have limited hardware resources and memory bandwidth, it is imperative that we analyze different combinations of these optimization techniques to identify the best and generate the most efficient (performance, resource utilization) hardware design for all dwarfs. In the context of this work, we start exploring the large FPGA-oriented optimization space and attempt to provide some preliminary insights. Table 4.7 shows the options we implement and evaluate in the context of this study.

Use of Restrict/Const Keywords and Kernel Vectorization

An optimization strongly suggested by Altera [25] is use of the *restrict* keyword for kernel arguments that are guaranteed to not alias (i.e., point to the same memory location). Using *restrict* allows more efficient designs in terms of performance by eliminating unecessary assumed memory dependencies. Although a side effect of such an optimization could be lower resource utilization,

we find that this is not the case in our application. Cases IMP2 and IMP4 (Figure 4.22) highlight the difference (1.31 times higher utilization with *restrict*) across two otherwise identical implementations. Performance-wise, IMP4 is 3.94 times faster and this stems from the vast majority of memory accesses resulting in cache hits. Conversely, IMP2 is characterized by sub-optimal memory accesses that result in cache misses and pipeline stalls (about 80% of the time). While their base performance (no kernel vectorization or compute unit replication) favors the *restrict* implementation, there are certain trade-offs that need to be considered for applications where kernel vectorization or compute unit replication is beneficial (e.g., highly parallel, regular, compute-bound applications). In such cases, being able to fit a design with higher vectorization or more compute units might surpass the benefits from using *restrict* for more efficient memory accesses (e.g., when resource utilization without *restrict* is enough to fit kernel vectorization of width 16, where the corresponding with *restrict* can only fit 8). As far as *const* keyword is concerned we observe no difference neither in resource utilization, nor in execution time. One reason may be that the compiler successfully identifies the constant parameters as such (no write on these memory locations and no aliasing is ensured).

Compiler Resource-Driven Optimizations

In compilation with resource-driven optimization the compiler applies a set of heuristics and estimates resource utilization and throughput given a number of kernel attributes, like loop unroll factor, kernel vectorization, number of compute units. This process should not be always expected

Table 4.7: Features of GEM kernel implementations

Implem.	Refact.	Restrict	Constant	SIMD	CU	Unroll
IMP1				1	1	1
IMP2			✓	1	1	1
IMP3	✓		✓	1	1	1
IMP4		✓	✓	1	1	1
IMP5			✓	1	1	4
IMP6			✓	8	1	1
IMP7	✓		✓	16	1	1
IMP8		✓	✓	8	1	1

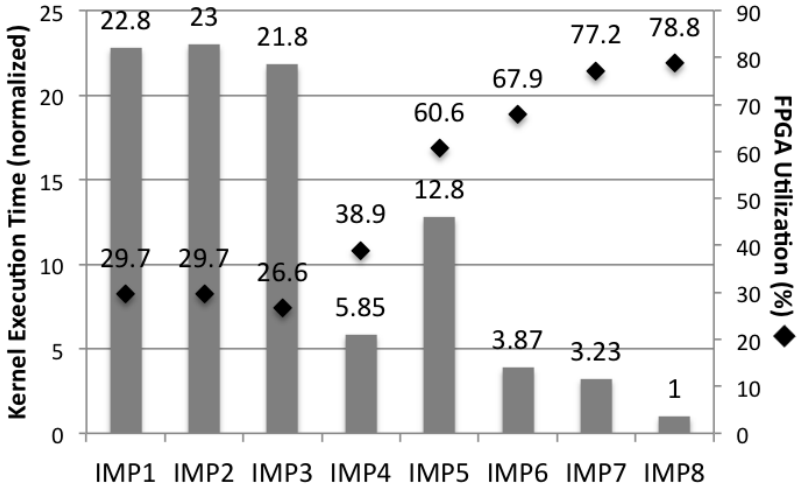


Figure 4.22: Optimized GEM kernel implementations

to provide the best implementation. In our example application, we identify at least one case where manual choice of kernel vectorization width surpasses (by 3.33x) the compiler-selected attributes (*pragma unroll 4*) (IMP6, IMP5 in Figure 4.22). Profiling the kernel, we find that IMP6 benefits from coalesced memory accesses, while memory accesses in IMP5 result in costly pipeline stalls. Also, bandwidth efficiency is higher (more than double) in IMP6 (i.e., more of the data acquired from the global memory system is actually used by the kernel). Altera discusses the inherent limitations of static resource-driven optimizations in their optimization guide [25]. Developers should consider the aforementioned limitations when compiling using the resource-driven optimization option.

Algorithmic Refactoring

A given algorithm implementation may solve an actual problem, but this does not mean that a set implementation is appropriate for every platform (e.g., CPU, GPU, FPGA). A different implementation for solving the same problem, i.e., produce the same output given the same input, may be necessary. While this may not be intuitive, or even applicable for all cases, certain algorithmic restructuring can prove very beneficial. To illustrate the above, we apply basic algorithmic refactoring in our example application. Specifically, we remove the complex conditional statements for different cases encapsulated in a single kernel, and tailor the kernel to the problem at hand. This provides a two-fold benefit, as shown going from IMP2 to IMP3: (a) better resource utilization (in our examples the refactored algorithm requires about 10% less FPGA resources, and (b) better performance (i.e., 5% faster). More importantly, better resource utilization may allow wider SIMD

or more compute units to fit in a given board. For example, in refactoring from IMP6 to IMP7, the reduced resource utilization of the refactored algorithm allows a SIMD length of 16, whereas the original algorithm only accomodated up to 8 (logical elements being the limiting factor). This translates to a 1.22-fold faster execution of the refactored over the original.

Resource Utilization Estimation

We find that the resource utilization estimation (when compiling with *-c* flag) is very accurate and can be safely trusted by developers. The benefit of being able to trust AOCs resource utilization estimation is that it can be performed without building hardware, and as such one can avoid spending hours compiling a design only to find that it cannot fit on a given board's resources. Specifically, for our example application and all the compiled implementations, the estimate captured the resource utilization to the closest round number.

4.4 Enhancing Performance via Heterogeneous Architectures: an Architectural Approach

Moore's Law has fueled processor advances for half a century now. As per Dennard's law [90], transistor scaling has reduced transistor dimensions by 30% with every generation and area shrinkage by 50%. This doubling of transistor density has allowed microarchitectural innovations that provide further performance increases (roughly 40%, as per Pollack's Rule [234]) and the de-

sign of multi-level cache hierarchies to address the increasing gap between processor and memory speeds. While these trends have held true of single-core designs for decades; limitations in transistor scaling, prohibitive heat dissipation levels, and power/energy-related constraints have mandated a switch to multicore in the mid-2000s. Looking ahead, large-scale parallelism with *heterogeneous* cores now appears to be on the path to exascale computing.

The trend of heterogeneity is not new; it is a recurring pattern observed in chip design. Heterogeneity was first introduced in the form of discrete devices that were later *unified* on-chip in a subsequent technology iteration. For instance, specialized floating-point or other co-processors (e.g., encryption or signal processing), which complemented early single-core CPUs, eventually relocated onto the CPU die itself. By the mid-2000s, the multi-core CPU era effectively telescoped the homogeneous compute cluster from a machine room to a *homogeneous cluster on a chip*. By the late 2000s, this same telescoping trend saw discrete GPUs that were out on the PCIe interface move onto the CPU die to create a fused CPU+GPU die called an *accelerated processing unit* (APU). Discrete FPGAs have also been used as co-processors to CPUs. With Intel's recent acquisition of Altera, a key FPGA vendor, Intel will telescope previous "discrete CPU+FPGA" offerings into a fused on-package integration with the Purley platform and then a fused on-die integration with the Tinsley platform.

Currently, a typical supercomputer node may exploit the synergistic (parallel) performance gains of architectures that are heterogeneous in nature, such as multi-core CPUs, GPUs, Intel MICs (Xeon Phi), or even FPGAs. These constituent elements are interconnected via PCIe at the intra-node level and over high-speed interconnects (e.g., Infiniband) at the inter-node level. Following the

above examples on the recurring, a la *déjà vu*, trend of *architectural unification* that has defined the chip evolution, we project that it may only be a matter of years before the notion of telescoping architectures that led from a "homogeneous cluster in a machine room" to a "homogeneous, multicore-CPU cluster on a chip" is extended to create a *heterogeneous cluster on a chip* (CoC).

In light of this trend and given the variety of homogeneous architectures (i.e., CPU, GPU, Intel MIC, FPGA), the research questions that arise are as follows:

1. What is the ideal mixture of compute engines (CEs) and number of such CEs that will fuel telescoping architectures and enable the transformation of a heterogeneous cluster in a machine room to a heterogeneous CoC?

2. What methodology should we use to answer the above question in a systematic and generalizable way?

To address these questions, we study heterogeneous CoC and discuss the roadblocks that need to be addressed before such a concept is materialized at the intra-node level. Such a preliminary study is imperative before it can be expanded to the inter-node level, where new sets of challenges arise.

We summarize our contributions below:

- A systematic and generalizable methodology towards identifying future trends and catalyzing exploration into the heterogeneous architecture space via *real hardware* and the use of *dwarf-based (or motif-based) benchmarking*.

- Application of the above methodology to quantify the performance benefits of different instantiations of a *cluster on a chip* (CoC), each employing single or multiple instances of CPUs, GPUs, Intel MICs, and FPGAs.

4.4.1 Architectural Unification in the History of Computing

In this work we examine the trends towards architectural unification as we move from a heterogeneous cluster in a machine room to a heterogeneous cluster on a chip. To put this trend in perspective it is useful to examine how the unification trend has manifested throughout computing history. Moore's law has allowed doubling the transistor density in a chip, thus facilitating heterogeneity at various levels and at the same time on-chip replication of homogeneous resources.

As far as *heterogeneous coupling* is concerned, there are various notable examples. At the early days of computing, floating-point arithmetic was implemented in software (e.g., on Intel 8086). The Intel 8087 was the first math co-processor for the 8086 line that allowed fast, hardware implementation of floating-point instructions. Nowadays, all modern processors include an (integrated) floating-point unit (FPU) and dedicated floating-point registers on-chip.

Graphics processing units (GPUs), in a similar fashion, originated as separate devices (graphics co-processors) with dedicated graphics memory. While still widely available in a discrete form factor, *integrated graphics processors* (IGPs) are the norm in many cases, especially in the laptop market. For example, Intel HD Graphics [114] in Ivy Bridge and Intel Iris Graphics [127] in Haswell CPUs are IGPs on the same package or die as the CPU and utilize a portion of the computer system RAM.

Similarly, in 2011, AMD introduced Llano [47], the first generation of *accelerated processing units* (APUs), which combined a CPU and GPU on the same die.

FPGAs have also been used to accelerate computations in many domains (e.g., bioinformatics [261], finance [213]). FPGAs connect to a host platform typically via the PCIe interface or Ethernet. However, there are system-on-chip (SoC) implementations, where a CPU is embedded in an FPGA board (e.g., big.LITTLE by ARM [120], Cyclone V SoC with ARM Cortex-A9 [7]). To this end, Intel is also introducing their Xeon and FPGA accelerator platform that incorporates an FPGA module attached to the processor via a Quick Path Interconnect (QPI), After Altera's acquisition by Intel, it is expected that in the next year, we will see a FPGA fabric as part of the package or die.

As far as *replication of homogeneous resources* is concerned, the CPU and GPU cases are indicative examples. In the former case, single-core CPUs initially got connected over a network to form compute clusters. When manufacturing technology allowed, multiple cores (dual-, quad-, octa-cores, etc.) fitted on the same chip. Later, with Intel Many Integrated Core (MIC) [74] architecture, tens of (simpler) cores (60 or 61) became a reality. The next generation of Intel MIC (code-named *Knight's Hill*) will be bootable, obviating the need for a host processor needed in current generation's MICs. Similarly, GPUs include an ever-increasing number of compute units (or *CUDA cores* in NVIDIA terminology) and more advanced architectural features.

Both above trends, in isolation and, more so, combined, hint towards the concept of *supercomputing on a chip*.

4.4.2 Methodology

In this section, we present our methodology in addressing the research questions set forth at the start of Section 4.4. We start with identifying the search space for candidate *Cluster on a Chip* (CoC) platforms. Then (Sections 4.4.2 and 4.4.2) we describe in detail how we evaluate the performance of each CoC candidate. Section 4.4.2 discusses assumptions we make in the process.

Cluster on a Chip (CoC)

By *Cluster on a Chip* (CoC) we refer to the combination of discrete types and numbers of compute engines (CEs). This conglomerate of CEs constitutes the expected result of the unification trend that we describe in Section 4.4.1. Such a hypothetical platform may not be feasible under current technology and manufacturing constraints. We discuss our assumptions in Section 4.4.2. CEs under consideration include a general-purpose CPU, a high-performance discrete GPU, an Intel Xeon Phi (MIC) co-processor, and an FPGA. Table 4.8 shows the detailed characteristics of each platform.

Given these CEs we construct hypothetical CoCs by creating combinations thereof (also allowing multiple CE instances). In order to restrict the CoC search space, we enforce an (arbitrary) *chip area constraint*. Table 4.9 shows how we define a *chip area base unit* (BU), based on the number of transistors of each device and the process technology used in each to approximate the chip area size of each CE, assuming that transistors for all four types of CEs are laid out in the two-dimensional space. A BU measures the relative chip area using the calculated CPU chip area as a baseline.

Table 4.8: Configuration of the target fixed architectures

Model	AMD Opteron 6272 (CPU)	AMD Radeon HD 7970 (GPU)	Intel Xeon Phi P1750 (MIC)
Type	CPU	Discrete GPU	Co-processor
Frequency	2.1 GHz	925 MHz	1.09 GHz
Cores	16	32†	61
Threads/core	1	4	4
L1/L2/L3 Cache (KB)	16/2048/ 8192‡	16/768/- (L1 per CU)	32/512/- (per core)
SIMD (SP)	4-way	16-way	16-way
Process	32nm	32nm	22nm
TDP	115W	210W	300W
GFLOPS (SP)	134.4	3790	2092.8

† Compute Units (CU) ‡ L1: 16KBx16 data shared, L2: 2MBx8 shared, L3: 8MBx2 shared

Table 4.9: Defining Base Unit (BU) for chip area size

Type	**Model**	**Transis-tors (bil.)**	**Process (nm)**	**Base Units**
CPU	AMD Opteron 6272	2.4	32	1
GPU	AMD Radeon HD 7970	4.3	28	1.372
MIC	Intel Xeon Phi P1750	5	22	0.985
FPGA	Xilinx Virtex-6 LX760	5.8	40	3.778

For example, given the values for number of transistors and process technology in Table 4.9 for CPU and GPU, the BUs of a GPU are $(\frac{28}{32})^2 * \frac{4.3}{2.4} = 1.372$. Following the above, the restriction we enforce is that all CoCs we consider have a chip size area that is equal or less than the aggregate area of one CPU, plus one GPU, plus one Intel Xeon Phi, plus one FPGA. Based on Table 4.9 (*Base Units* column) this adds up to 7.135 BUs. According to this constraint there are 100 possible CoCs.

Performance Evaluation

For evaluating the CoC candidates (Section 4.4.2) we use the OpenDwarfs benchmark suite (Section 4.1.1. Specifically, we employ a subset of the dwarf instantiations: GEM, NW (Needleman-Wunsch), SRAD (Speckle-Reducing Anisotropic Diffusion), BFS (Breadth-first Search), CRC (Cyclic Redundancy Check), and CSR (Compressed Sparse-Row Matrix-Vector Multiplication). The dwarf categories are shown in Table 4.10 together with the dwarf instantiations listed above and their input parameters/datasets. Using these dwarfs we create a large number of *synthetic benchmarks*. These synthetic benchmarks contain all possible combinations of four, five, and six dwarfs. This allows for the creation of benchmarks that cover a sufficiently large number of potential real-world applications. This stems from the fact that dwarfs, by definition, represent computation and communication patterns and real-world applications are largely composed of such patterns (dwarfs) that can be *temporally or spatially* distributed across a set of CEs (examples of dwarf composition of five ParLab applications and seven general application areas are given in [29]). Along these lines, in each synthetic benchmark we assume the latter (spatial) distribution of constituent parts, i.e., dwarfs within an application are independent and can run in parallel in a form of *request-level parallelism* (similar assumptions are used in similar works, e.g., [75]). Each dwarf runs in parallel by itself on the specific CE it is scheduled on. The way each dwarf of a benchmark is scheduled to each part of a CoC is described in Section 4.4.2. The above methodology allows us to draw broader conclusions that are better representative of real-world workloads.

Table 4.10: OpenDwarfs benchmark test parameters/inputs

Dwarf	Algor.	Problem Size
N-body methods	GEM	Input file: nucleosome
Dynamic programming	NW	Two 4096-letter protein sequences
Structured grids	SRAD	2048x2048 FP, 128 iterations
Graph traversal	BFS	248,730 nodes, 893,003 edges
Combinational logic	CRC	Input data-stream: 100MB
Sparse linear algebra	CSR	2048^2 x 2048^2 sparse matrix

Table 4.11: Execution time (in msec) of dwarf benchmarks

	GEM	NW	SRAD	BFS	CRC	CSR
CPU	21592	112	5093	331	672	22
GPU	401	672	232	96	19	4
MIC	11871	222	2298	278	881	5
FPGA	25345	35	17651	105	24	83

For our results we use the kernel execution times shown in Table 4.11, which were obtained in [175] by executing the OpenCL-based OpenDwarfs on the hardware shown in Table 4.8. Implementation details and performance evaluation of each dwarf separately on each of the different architectures (CPU, GPU, Intel MIC, FPGA) is also done in [175] and beyond the scope of this work. In this work we focus on the *overall* performance benefits of using CoCs. For this reason, the results of synthetic benchmarks that contain all possible combinations of four, five, and six dwarfs (among the total six used) are averaged (*4-mers, 5-mers, 6-mers*). This ensures that no specific dwarf can disproportionally distort the high-level insights. As we would see in Section 4.4.3 the trends observed in the results are similar, irrespective of the number of constituent dwarfs, which is indicative of the generality of our methodology.

Scheduling

In Section 4.4.2 we describe the performance evaluation methodology with respect to the workloads. In this section, we discuss the details of scheduling each constituent part of our synthetic benchmarks on the disparate parts of CoC architectures. Our scheduling methods assume *oracle* prediction (as, e.g., in [199] or [75]), i.e., we assume a priori perfect knowledge of execution times of each dwarf on each type of CE within a CoC. Note that this work is not focused on scheduling, but in providing an early evaluation of the upper performance bound of CoC architectures, hence the oracle assumption.

Specifically, in our schedule we execute each dwarf on the *best available* CE (i.e., not necessarily on the fastest one, which would correspond to the local minimum). This schedule is feasible given use of an off-line scheduling algorithm, where we know the execution time of each dwarf, or a performance prediction model. Such schedules may allow late start for a given dwarf, if such a schedule leads to a *globally optimal* solution. We explore cases with multiple instances of CEs (given space constraints as described in Section 4.4.2), and we allow multiple CEs to be active simultaneously.

Assumptions

We start our study with a simplified model that incorporates certain assumptions. Our goal is to provide an initial systematic and broad study of the CoC concept based on existing hardware and to rationalize the benefits of a conceptual architecture like it. While a CoC may be unfeasible given

the current state of the practice, we envision it can materialize in the (near) future, once further advances in chip manufacturing and other areas are made. Even in the presence of the stated assumptions, like negligible inter-CE interference, our study projects a conservative upper bound on performance (a la the Roofline model [285], for example) and provides useful insights about the future of heterogeneous computing. Similar approaches with respect to assumptions are made in related work, like [181, 75, 199]. We discuss such relevant required technology and system-level advances in Section 4.4.4.

Data transfers: The architectures, as discrete devices, that comprise a CoC would require data transfers between a CPU host and the corresponding accelerator in a real OpenCL execution scenario. Such data transfers could in fact be potential bottlenecks also limiting maximum parallelism [79]. In the context of this work we deem them negligible, expecting a broader adoption of the *unified memory* trend (*a la* CPU and GPU in the context of an APU system).

Prediction/scheduling: In Section 4.4.2 we discuss scheduling. When multiple CEs, and especially CEs of different types, are available, it is essential that scheduling of workloads be done in an efficient manner, taking advantage of the specific characteristics of each CE type. This would require a form of pattern/application signature recognition. Such scheduling is non-trivial and adds an extra overhead on top of the actual execution time of a given workload. Within the confines of our study, we assume perfect prediction, given off-line scheduling and zero scheduling overhead.

Power: One of the main premises of CoCs, besides performance scaling, is power efficiency and energy savings by assigning workloads to the most appropriate CE type (as opposed to a generic *fat core*). The actual power budget of an architecture that consists of one (or more) of

each of a CPU, MIC, GPU and FPGA could be prohibitive without employing advanced power management techniques. Our study focuses on the performance aspect of the CoC concept and indirectly addresses the thermal power density aspect by only allowing sufficient heterogeneous computing elements to fit within a constrained die area. Ensuring that CoCs are power-efficient and a detailed study of power is an important future research avenue beyond the scope of this work.

Parallelism within applications: Our synthetic benchmarks consist of applications that fall under the dwarfs classification, for reasons we discuss in Section 4.4.2. For the purposes of our work, we assume that workloads are composed of independently parallelizable parts (i.e., the constituent dwarfs of each synthetic benchmark) following a request-level parallelism paradigm.

4.4.3 Results

In this section we present the findings of our experiments that are based on the methodology outlined in Section 4.4.2. The main variables in our experiments include:

1. **Synthetic benchmarks** (i.e., benchmarks comprising different combinations of dwarf N-mers).

2. **Cluster on a Chip (CoC) instantiations** (i.e., different combinations of types and numbers of CEs under consideration).

Despite the magnitude of the potential options search space, we attempt to provide relevant experiments and group the results we obtain in such ways so as to allow us to identify certain trends in a clear way and draw useful insights with respect to the following research questions:

1. For a widely-varying set of dwarf combinations forming synthetic benchmarks what are the best combinations of CEs to form a CoC across the potential CoC spectrum?

2. What are the trade-offs between the CoCs' chip area and performance? What are the most efficient CoCs based on a performance per chip area metric?

3. What is the expected benefit of fusing an FPGA in three specific CoC instantiations and what are the CE usage trends with different combinations of dwarfs in smaller synthetic benchmarks?

Answering the above questions enables us to draw useful conclusions related to the main research question of *what is the best combination and number of CEs within a CoC* for a varied set of algorithmic patterns (as classified by the *dwarfs* concept).

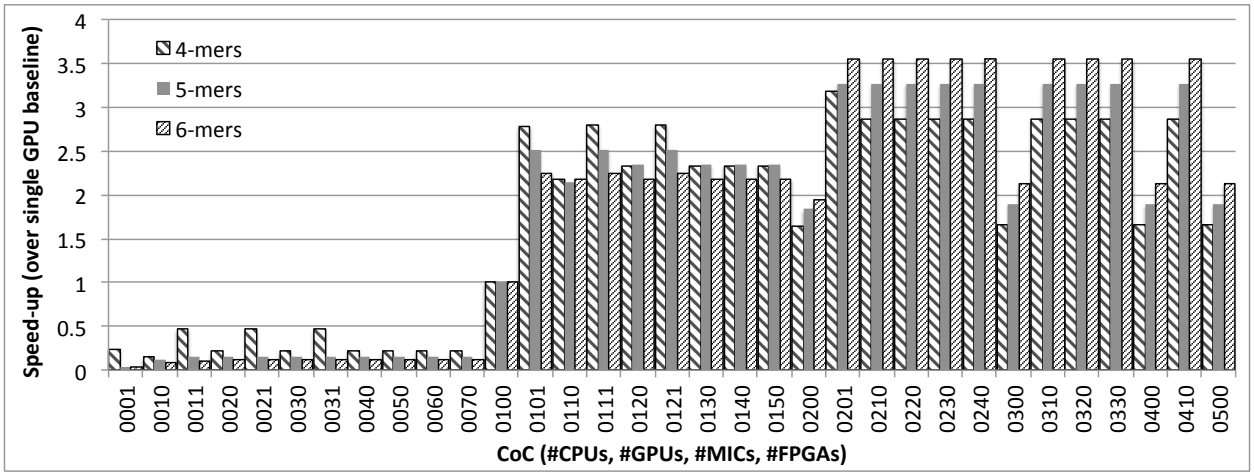
Performance of CoC Instantiations

Figure 4.23 presents the results for all possible CoC instantiations and for the average performance across all three categories of synthetic benchmarks built and scheduled as described in detail in Section 4.4.2. Performance (execution time) is presented as the speed-up over a CoC that contains a single GPU. In this case, separate dwarfs within a synthetic benchmark are executed one after

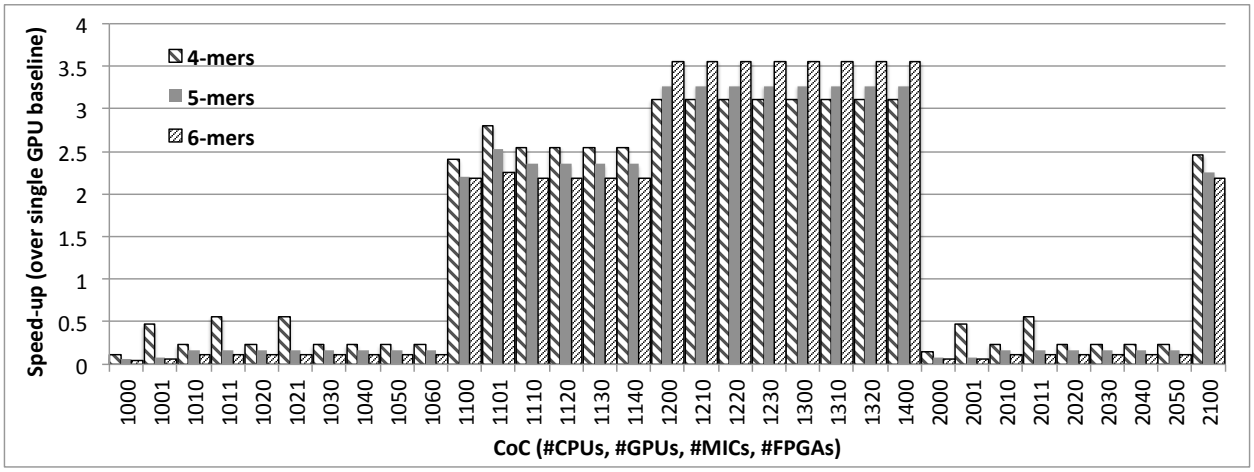
the other on the single GPU, but each dwarf itself is executed in parallel within the GPU. Each CoC is identified by the number of its constituent CEs (number of CPUs, GPUs, MICs, FPGAs). For example, *1210* corresponds to a CoC that includes one CPU, two GPUs, one Intel MIC and zero FPGAs. Note that in the OpenCL paradigm a *host* (CPU) is required alongside the *device* (accelerator). A "traditional" CPU, a soft- or hard-core CPU within a FPGA or a MIC core can serve as both an OpenCL host and device. While for now a discrete GPU cannot serve as a host, this may not be the case in the future.

First, on a high-level we observe that for the majority of CoCs the achieved performance is irrespective of the number of dwarfs contained in a synthetic benchmark (*4-mers, 5-mers, 6-mers*). The cases where the observed performance is higher for the 4-mers case than in 5- or 6-mers are generally these where the number of CEs in a CoC is less than four. In these cases, one or two dwarfs in the 5- or 6-mer synthetic benchmarks need to run sequentially (with respect to the other dwarfs) on one of the available CEs, thereby increasing the overall execution time. The fact that the general performance trends remain the same, irrespective of the number of dwarfs in our synthetic benchmarks, indicates the suitability of our benchmarking methodology (Section 4.4.2) for our purpose of evaluating next-generation heterogeneous architectures, in the form of CoCs.

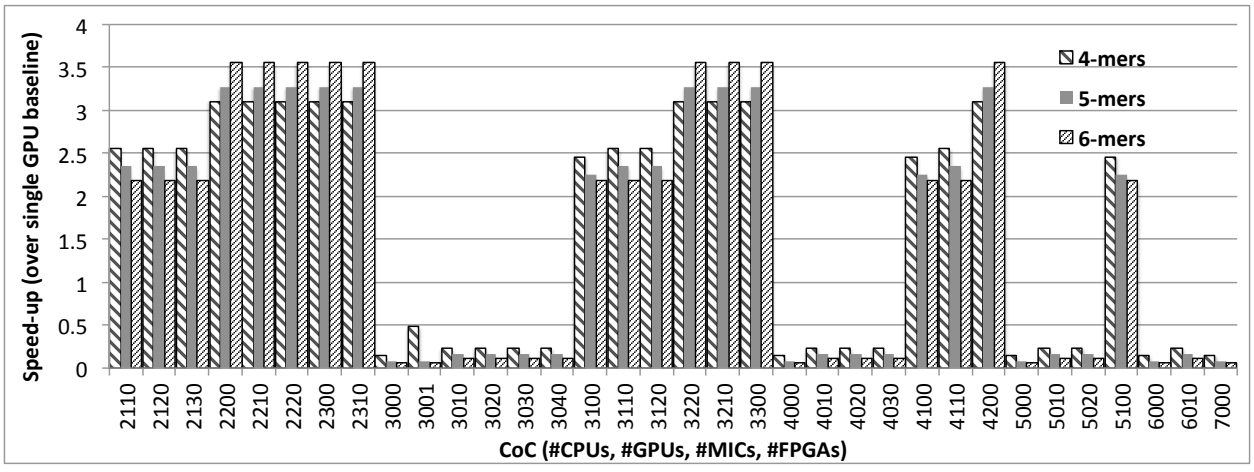
Without loss of generality, we focus on the 6-mer case and provide a more detailed analysis of the observed results. Without a GPU, relative performance (i.e., speed-up over common baseline) is limited in the 0.033 to 0.12 range. A GPU is indeed an indispensable CE in any CoC. Note that a CoC with two GPUs (0200 in graph) provides a 1.952-fold speed-up over a CoC with one GPU. This may seem counter-intuitive at first (expecting a 2-fold speed-up), but one needs to remember



(a) Combinations 1-33



(b) Combinations 34-66



(c) Combinations 67-100

Figure 4.23: Performance of Clusters on a Chip (CoC): Speed-up over single-GPU baseline for all possible combinations (100) restricted by the maximum number of Base Units (BUs), for three classes of synthetic benchmarks

that the execution time of different dwarfs within a synthetic benchmark vary and the way they can be scheduled onto the two GPUs (i.e., as a whole) may lead to such schedules. For similar reasons, CoCs with three, four or five GPUs only (0300, 0400, 0500) exhibit the same performance (2.123-fold speed-up). Maximum performance is capped by the longest-running dwarf within the synthetic benchmarks. So if chip area (and consequently power) is a concern, a CoC with three GPUs is better than one with four or five. Or one may deem the 8.7% performance increase between using two and three (or four or five) GPUs negligible and elect a CoC with two GPUs only. Similar observations can be gleaned from Figure 4.23 for CoCs entailing other CEs, like MIC (e.g., cases 1210, 1220, 1230). The best performance (3.555) can be obtained using different combinations of CEs within a CoC. Again, these combinations exhibit varying chip area requirements. The above observations provide useful insights with respect to the performance per area ratio that we discuss further in Section 4.4.3.

Performance vs. Area Trade-offs

In this Section we attempt to provide another view of our experimental results focusing on the performance versus area trade-offs. Figure 4.24 presents the speed-up over a single GPU baseline with respect to the chip area required for 300 points (100 CoCs times three different set of experiments to include all possible 4-, 5-, and 6-mers). While the 300 points are not annotated, due to their large number, our purpose is to provide a high-level view of the trade-offs entailed, and which we only briefly discuss in Section 4.4.3. Specifically, one can observe three large clusters for each of the three experiments. The higher the number of dwarfs in each of the three experiments, the

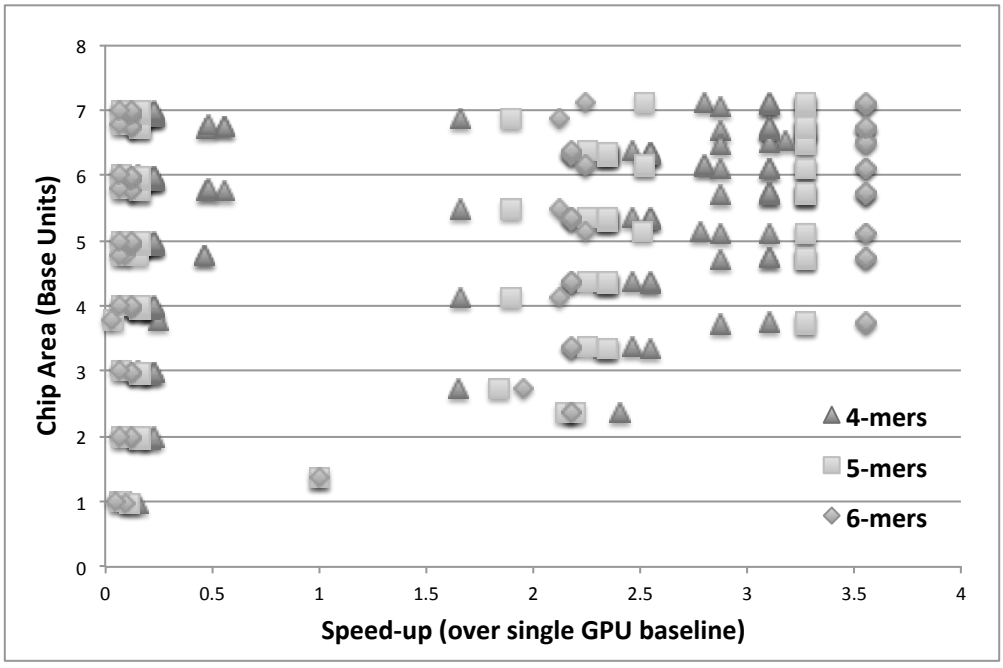


Figure 4.24: Performance vs. area results

more concentrated the clusters are along the horizontal axis. For example, in the 6-mer synthetic benchmark experiments, most points fall on or near the 0.12, 2.18, 3.56 mark (x-axis). Focusing on the 3.56-fold speed-up case, there are 26 CoCs with varying chip area requirements (a lot of them overlapping in Figure 4.24). The best performance (3.56) with the least area (3.73 BUs) within this performance point is achieved on a CoC with two GPUs and one MIC. The same performance with the second best area is a CoC with two GPUs and one CPU (chip area of 3.74 BUs). The following best one (chip area of 4.7 BUs) comprises two GPUs and two MICs. The above observations are indicative of two cases: a) *different combinations of CEs in CoCs can achieve the same performance*, and b) *more CEs are not always beneficial*. In the former case, the choice of one CoC over the other may make more sense for non-technical reasons, too. For example, a 1-CPU + 2-GPUs CoC is a cheaper choice over a 1-MIC + 2-GPUs CoC (assuming the cost of a combination of CEs

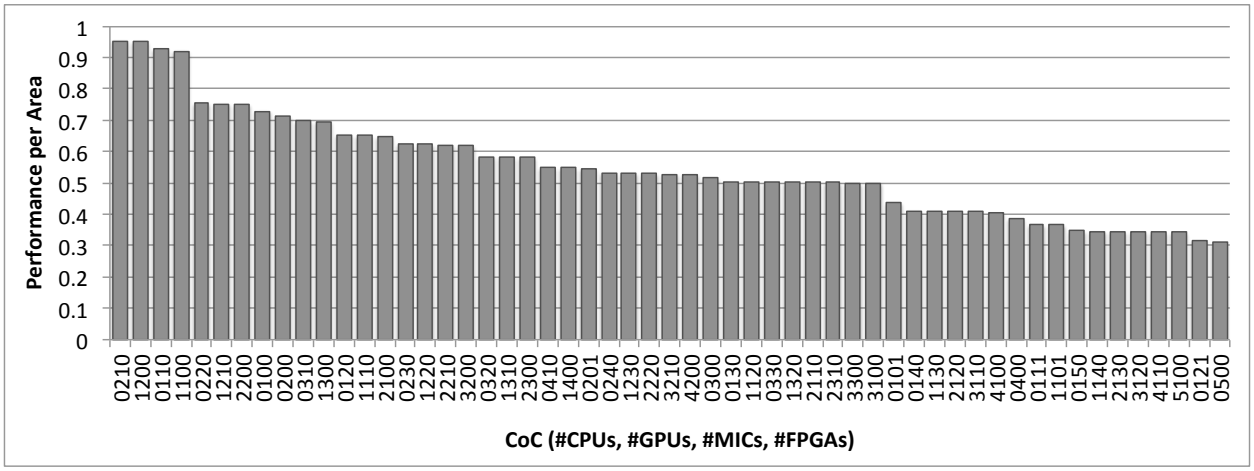


Figure 4.25: Performance per area results

within a CoC is the same as the CE separately). Also, there are power implications depending on the CE selection (e.g., MIC TDP is generally higher than CPU TDP). In the latter case the higher number of CEs (especially of the same type, as discussed above) does not provide any benefit.

Minimizing chip area while keeping performance steady is important, due to manufacturing costs, cooling requirements, power, etc. As such, it is important to evaluate CoCs' performance with respect to occupied chip area. To do so we introduce a *performance per area* metric. We present the results in Figure 4.25 in descending order (the higher the better). This provides insight with respect to what combinations of CEs within a CoC provide the best performance per chip real estate. As in Section 4.4.3, we focus on the 6-mer synthetic benchmarks experiments. We show the CoCs that achieve a value of performance per area over 0.3 (all the rest CoCs score less than 0.06 and are not shown). As can be seen in Figure 4.25 the two best CoCs with respect to the performance per area metric are the ones we discussed in the previous paragraph (0210, 1200). The next best ones with a performance per area value over 0.9 (and still close to the maximum)

incorporate one GPU and one MIC (0110), and one CPU and one GPU (1100), respectively. Notice that while these two CoCs are efficient with respect to our target metric, they are not optimal in terms of performance alone. This accentuates the trade-offs entailed and the fact that the choice of a CoC depends on the design goals. The most efficient CoCs, both from a performance per area perspective (Figure 4.25) and performance alone (Figure 4.23), contain at least one GPU. Intel MIC is also sufficiently present across the top performance per area CoCs, but always in combination with GPU(s).

Case Study: FPGAs in CoCs

In Section 4.4.1 we discuss practical examples of telescoping architectures, such as fusing CPUs and GPUs in the form of an Accelerated Processor Unit (APU). Also, we hint towards the unification of reconfigurable fabric (i.e., *FPGA*) with CPU on-chip, in future implementations. The latter is projected to be of particular importance in *datacenter computing* (e.g., search-engines), because of the FPGA's high-performance for certain algorithms, and its high energy efficiency. This Section discusses the performance benefits of CoCs that include an FPGA CE. In contrast to Section 4.4.3, here we focus on simpler CoCs that may be more viable in the shorter term. As such, we focus on three sample cases, that is CoCs that include: a) one CPU and one FPGA, b) one CPU, one GPU and one FPGA, c) one CPU, one GPU, one MIC and one FPGA. As far as workloads are concerned, we use synthetic benchmarks that comprise dwarf kernels from the OpenDwarfs subset discussed in Section 4.4.2 minus GEM (to keep number of combinations tractable). However, in these experiments we focus on synthetic benchmarks that are composed of three dwarfs

(i.e., 3-mers); specifically all combinations of three dwarfs possible out of the five dwarfs: NW, SRAD, BFS, CRC, CSR. Table 4.12 shows the results of the above experiments. For each of the three sample cases, we list the number of times dwarfs from each synthetic benchmark are run on the CPU, GPU, MIC or FPGA. Also, we indicate the speed-up obtained with the CoC at hand compared to the same configuration of CEs without the FPGA. For instance, in the 1-CPU + 1-FPGA case, and for the NW/SRAD/BFS synthetic benchmark, the CPU is used for one of the three constituent dwarfs of that benchmark and the FPGA for two. Also, the indicated speed-up is the performance obtained using this CoC compared to a CPU-only equipped CoC. Although we do not show the details of where dwarfs of each synthetic benchmark are scheduled, we provide details where necessary below.

In the first case (Table 4.12a), we examine the case where we incorporate an FPGA with a CPU on a chip. This would be conceptually similar to Intel's announced Xeon CPU + FPGA chip. Apparently, the FPGA is utilized on average twice as much as the CPU in the CoC (1.1 vs. 1.9). The performance obtained compared to a CPU-only CoC (practically a typical CPU) ranges from a meager 1.09- to a considerable 13.51-fold increase. In the average case, we observe a 4.01-fold speed-up. Specifically, as expected, all ten cases exhibit performance improvement. However, speed-up in six out of ten lies below 1.20x. These are the cases where the SRAD dwarf is present in the synthetic benchmark. Even for the best schedule, where the CPU is chosen for SRAD (3.47x than the FPGA), and the FPGA is used for the (faster) execution of the remaining two dwarfs in each case, the actual execution time of SRAD compared to the other two dwarfs dominates total execution time. While not shown in Table 4.12, a CPU + GPU CoC exceeds a CPU + FPGA CoC

Table 4.12: CoCs with FPGA: CE utilization and speed-up vs. corresponding CoC without FPGA

	1 CPU + 1 FPGA		
Synthetic Dwarf	**#CPU**	**#FPGA**	**Speed-up w/o FPGA**
NW / SRAD / BFS	1	2	1.09
NW / SRAD / CRC	1	2	1.15
NW / SRAD / CSR	1	2	1.03
NW / BFS / CRC	1	2	8.59
NW / BFS / CSR	2	1	3.46
NW / CRC / CSR	1	2	13.51
SRAD / BFS / CRC	1	2	1.2
SRAD / BFS / CSR	1	2	1.07
SRAD / CRC / CSR	1	2	1.14
BFS / CRC / CSR	1	2	7.9
Average:	**1.1**	**1.9**	**4.01**

(a)

	1 CPU + 1 GPU + 1 FPGA			
Synthetic Dwarf	**#CPU**	**#GPU**	**#FPGA**	**Speed-up w/o FPGA**
NW / SRAD / BFS	1	1	1	1.41
NW / SRAD / CRC	1	1	1	1.08
NW / SRAD / CSR	2	1	0	1
NW / BFS / CRC	0	1	2	1.2
NW / BFS / CSR	1	1	1	1.16
NW / CRC / CSR	1	1	1	3.2
SRAD / BFS / CRC	0	1	2	1.43
SRAD / BFS / CSR	1	1	1	1.42
SRAD / CRC / CSR	1	1	1	1.08
BFS / CRC / CSR	1	1	1	1.2
Average:	**0.9**	**1**	**1.1**	**1.42**

(b)

	1 CPU + 1 GPU + 1 MIC + 1 FPGA				
Synthetic Dwarf	**#CPU**	**#GPU**	**#MIC**	**#FPGA**	**Speed-up w/o FPGA**
NW / SRAD / BFS	1	1	0	1	1.2
NW / SRAD / CRC	1	1	0	1	1.08
NW / SRAD / CSR	2	1	0	0	1
NW / BFS / CRC	0	1	0	2	1.2
NW / BFS / CSR	1	1	0	1	1.16
NW / CRC / CSR	1	1	0	1	3.2
SRAD / BFS / CRC	0	1	0	2	1.2
SRAD / BFS / CSR	1	1	0	1	1.2
SRAD / CRC / CSR	1	1	0	1	1.08
BFS / CRC / CSR	1	1	0	1	1.2
Average:	**0.9**	**1**	**0**	**1.1**	**1.35**

(c)

performance for the workloads under consideration, for nine out of ten cases. Specifically, all six synthetic benchmarks that contain SRAD are considerably slower on the CPU + FPGA CoC. Of the four remaining ones the CPU + GPU CoC is faster in three out of four cases (by 1.12-1.20x), whereas in one case (NW/CRC/CSR) the CPU + FPGA CoC is faster by 1.88x. In that case, NW runs 3.2x faster on the FPGA, CRC runs 27.35x faster on the FPGA, and CSR runs 3.77x faster on the CPU. Since all three dwarfs can run in parallel and we have one CPU and one FPGA in the CoC, NW and CRC run one after the other on the FPGA while CSR runs on the CPU. The above shows how important heterogeneity (or hardware customization) is to cover special cases that may appear and completely nullify any expected performance gains.

The second case (Table 4.12b) studies a CoC that contains an FPGA on a chip together with CPU and GPU CEs. That would correspond to an APU chip, as available today, with the addition of an FPGA on that same chip package. As we observe despite the presence of a GPU, the FPGA is still utilized at least once in nine out of ten cases. That makes it obvious that the addition of an FPGA on chip would be beneficial for performance, if added alongside a CPU or a CPU and GPU. Even though the benefits of adding an FPGA to an APU chip are not as high as doing so on a CPU-only, they still provide an average of 1.42-fold performance increase. In our example synthetic benchmarks, the major benefit comes from scheduling SRAD on the GPU. As we mentioned the actual execution time for SRAD (even on the GPU) is comparably bigger than the rest of the dwarfs. This may mean that the scheduler picks the second fastest CE for a dwarf, as long as SRAD is assigned to the GPU. For example, in the NW/SRAD/BFS case, NW is assigned to the CPU and BFS to the FPGA, although these CEs are the second best for these dwarfs. As one can

see in the CPU + FPGA and CPU + GPU + FPGA cases, adding an extra CE is always better. Even in the case a CE may not be the most suitable (i.e., fastest) for a given dwarf, it provides more scheduling possibilities that can lead to a better overall performance. While this may be worse with respect to power, clever power-saving techniques could counter such effects (e.g., switching off CEs when not used, etc.)

In the third case (Table 4.12c), we consider the case of placing an FPGA together with a CPU, GPU, and MIC CEs. This experiment provides useful insights, that also tie back to Section 4.4.3. Specifically, it is an example of a case where an extra CE, which also occupies useful chip real estate, does not contribute any performance benefit. As we see, the MIC is not used at all across the synthetic benchmarks search space. In fact, the statistics reduce to the previous case (CPU + GPU + FPGA CoC). While the number of constituent workloads within our synthetic benchmarks (three) is less than the available CEs (four), it is still interesting to see that MIC is not used over one of the other CEs in *any* of the synthetic benchmarks. Similarly, comparing the CPU + GPU + MIC + FPGA CoC to the corresponding one without the FPGA, we identify only three cases where adding the FPGA benefits performance (the rest remains the same). These are the cases where the BFS dwarf is present and now scheduled on the FPGA, instead of MIC.

4.4.4 Discussion

Telescoping architectures in the form of CoCs bears many similarities with similar works and can provide performance and power benefits. In this work, we focus on a first-order approach on the

performance aspect of CoCs, identifying an upper bound on attainable performance, under certain assumptions. Telescoping architectures in the context of heterogeneous computing (i.e., CoCs) is a non-trivial effort. In order for CoCs to find practical implementation a number of issues needs to be addressed, some of which warrant a whole research area by themselves. The proposed methodology, based on dwarf-based synthetic benchmarks, can be used in such future studies that focus on issues like scheduling, power/energy-aware optimizations, memory hierarchies, network-on-chip. Also, it can serve as a starting point for refined approaches for performance modeling/prediction. Borkar and Chien discuss challenges regarding the future of microprocessors, many of which are relevant to a CoC, in more detail in [46]. Here, we provide a high-level overview of three issues related to our assumptions in Section 4.4.2.

Programming CoCs

One of the obstacles in democratizing heterogeneous computing has been learning a disparate set of programming languages and optimization techniques. Eventually, key hardware vendors introduced the OpenCL standard that allows programming CPUs, GPUs, Intel MICs and even FPGAs. The "write once, run anywhere" concept of OpenCL renders it ideal for a heterogeneous cluster on a chip. While OpenCL, as is, covers the *functional portability* aspect (i.e., correct results across architectures), there yet remains the issue of *performance portability* (i.e., equally fast performance across architectures, given the same code). It is expected that compiler technology advances, in tandem with auto-tuners, and optimized/customized libraries will assist towards bridging the performance portability gap across heterogeneous architectures.

Identifying Patterns/Signatures and Scheduling

Scheduling dwarfs within an application to the CEs of a CoC in an efficient way requires identifying the dwarfs within that application. Research efforts include using performance counters to dynamically identify the best program phase to core matching [37, 181, 121, 125] or signature-driven approaches [252, 259, 55] and other scheduling techniques/run-time systems for workloads with various levels of parallelism (e.g., task-based, loop-based) on heterogeneous resources [173, 246, 264]. Efficient recognition of dwarfs as constituent parts of real applications is challenging because applications may be composed of multiple dwarfs in a non-trivial way. Related work could be expanded to leverage CoC architectures and dwarf-based workloads.

Hardware-Related Issues: Chip Integration, Data Transfers and Memory Unification, Power

For CoCs to materialize there is a need for major advances on the hardware level, specifically in the area of System on Chip (SoC) or System in Package (SiP) [194]. The latter is on the focus of ITRS [10], where complex, 3D SiP architectures are proposed on the road to *heterogeneous integration*. In both cases assembly, packaging, and most importantly interconnect of separate parts of a die (or chips in a SiP) will play a very important role in the fruition of CoC architectures. Issues like energy efficiency and energy proportionality are also of paramount importance. An interesting discussion on interconnects of future multi-processors, as well as power-related concerns, is given in [46]. We expect that the trend towards unifying the memory space and collocating heterogeneous

cores on the same die (e.g., in the APU case) will continue allowing *more* and *different* instances of compute engines to perform synergistically within the same chip or package.

4.5 Conclusion

In this chapter we examined the performance aspect of heterogeneous architectures. Specifically, we introduced dwarf-based benchmarking for heterogeneous computing via OpenDwarfs, a collection of dwarf implementations in OpenCL. We verified functional portability of dwarfs across a multitude of parallel architectures and characterized a subset's performance with respect to specific architectural features. Computation and communication patterns of these dwarfs led to diversified execution behaviors, thus corroborating the suitability of the dwarf concept as a means to characterize computer architectures. Based on dwarfs' underlying patterns and profiling we provided insights tying specific architectural features of different parallel architectures to such patterns exposed by the dwarfs.

Focusing our efforts on one of the dwarfs (n-body), we extended our performance-oriented study into multiple dimensions to include multiple optimizations, in different languages, and varying target architectures. We specifically attempted to shed light on the impact of optimizations that find application across heterogeneous platforms and evaluated their effect with respect to the underlying architectures. Of particular importance is our work on using OpenCL for FPGA programming. OpenCL support for FPGAs is fairly nascent and thus detailed performance studies on a variety of applications are needed. We presented a preliminary exploration of the FPGA OpenCL opti-

mization search space (to include kernel vectorization, kernel refactoring, loop unrolling, etc.) and assessed the resulting performance. Our study accentuated the need of manual optimizations that are tightly coupled with the platform used, as well as the algorithmic patterns found in a specific application. This holds true in more "traditional" architectures (such as CPU, GPU, Intel MIC), and even more so in the case of reconfigurable architectures (i.e., FPGA). While we focused on a single dwarf, similar studies like the above can provide insights on the rest of the dwarfs. Ultimately, such insights as those obtained from our study can be a useful aid in a programmer's search for the best combination of language, optimizations, and compute platform for their application. While our study so far focuses on performance, in Chapter 5 we discuss the aspects of programmability and portability, and ways to bridge the gaps between performance, programmability and portability via use of tools.

We concluded this chapter, by proposing a means of identifying trends regarding the future of heterogeneous architectures based on the dwarfs concept. Using our proposed methodology we were able to conduct a broad, yet rapid exploration of the heterogeneous architecture space via *real hardware* and extending the notion of *telescoping architectures*. This notion of telescoping allows us to envision, in a form of déjà vu, that similar advances that led from a commodity (heterogeneous) cluster in a machine room to a commodity cluster on a chip, will be repeated, now in the context of heterogeneity. To this end, we experimented with heterogeneous architectures that comprise multiple types and instances of CPUs, GPUs, Intel MICs, and FPGAs and attempted to provide an early study on the performance benefits of such heterogeneous clusters on a chip (CoC). We found that CoCs exhibit not only performance benefits overall, as expected, but also interesting

characteristics with respect to their performance and constituent compute engines (CEs), as well as with respect to specific workloads.

Chapter 5

On the Programmability and Portability of Heterogeneous Platforms

General-purpose computing on an ever-broadening array of parallel devices has led to an increasingly complex and multi-dimensional landscape with respect to programmability and performance optimization, as well as portability. The growing diversity of parallel architectures presents many challenges to an audience that spans expert programmers through domain scientists with minimal programming knowledge. Such challenges entail, among others, device selection, programming languages, and level of investment in optimization. All of these choices influence the balance between programmability and performance. At the same time, portability, i.e., the ability to run a program across different computing platforms remains an important issue.

In this chapter, we first seek to identify the gap between programmability and portability in heterogeneous platforms. To this end we employ GEM, the n-body, molecular-modeling application we use in our performance-oriented study in Section 4.2.1. We characterize programmability achievable across a range of architecture-specific optimizations for multi- and many-core platforms – specifically, an Intel Sandy Bridge CPU, Intel Xeon Phi co-processor, and NVIDIA Kepler K20 GPU. Additionally, we characterize the incremental optimization of the code from naïve serial to fully hand-tuned on each platform through four distinct phases of increasing complexity and expose the strengths and weaknesses of the main programming models offered on each platform. Our findings reveal that the widespread adoption of parallel platforms, languages, as well as optimization methods tailored for each combination of platform and language makes it a non-trivial task to come up with an efficient - and easy to program - algorithmic implementation.

To address the above performance, programmability and portability issues we propose realizing a programming abstraction and implementing it within an integrated development framework. Specifically, we present GLAF — a grid-based language and auto-parallelizing, auto-tuning framework. Its key elements are its intuitive visual programming interface, which attempts to render expressing and validating an algorithm easier for domain experts, and its ability to automatically generate efficient serial and parallel Fortran and C code, as well as OpenCL, including potentially beneficial code modifications (e.g., with respect to data layout). We find that the above features assist novice programmers to avoid common programming pitfalls and provide fast implementations. At the basis of GLAF design, development, and testing decisions lie dwarfs from the OpenDwarfs benchmark suite, among other applications. Dwarfs have been used throughout the development,

testing and debugging, and most importantly in identifying appropriate optimizations of generalized applicability in GLAF's auto-tuning back-end. Last, a broader study of GLAF's functionalities and effectiveness as a tool for domain scientists is conducted in the context of an application of interest to NASA.

5.1 A High-Level Discussion on Programmability and Portability

By the term *programmability* we refer to the perceived level of difficulty for a programmer to "translate" an algorithm of interest into a computer program. *Portability*, on the other hand, refers to the ability to run a given program across multiple types of devices and can be divided to *functional* and *performance* portability. The former holds true for programs that generate the same output, given the same input, across platforms, while the latter implies that the same program maintains equal performance levels across platforms (taking into account the intrinsic performance capabilities of the underlying hardware).

As we mention in Chapter 1, programmability is interrelated to the concept of performance (i.e., the *performance vs. programmability gap*), as well as portability. All *three Ps* [106], in fact, entail intrinsic trade-offs. For example, higher levels of programmability typically correspond to higher levels of portability, but lower levels of performance. On the contrary, lower programmability typically implies higher performance and lower portability. Of course, there may be exceptions as

compiler technology, tools and run-time systems advance. OpenCL, for instance, is considered a difficult programming language, yet comes with the intrinsic advantage of functional, and – to a certain degree – performance portability.

Figure 5.1 shows an example on programmability for a simple matrix multiplication algorithm across four programming languages, three of which (OpenACC, CUDA, OpenCL) can be used to program heterogeneous devices, like the GPU. The CUDA and OpenCL cases are intentionally shrunk to give a perspective on the number of lines of code and difficulty of programming in them, even for a simple program, as in our example. As we move from the left to the right, programmability decreases. While OpenACC requires a single directive (*#pragma acc* line), CUDA and OpenCL mandate much more programming effort. Specifically, both CUDA and OpenCL require the user to manually expose parallel computation and handle data decomposition, allocation, and transfers across a host CPU and the GPU. OpenCL further requires initialization of OpenCL objects (*platform*, *program*, *command queues* etc.) that require a large number of lines of code.

We use the above simple example above to provide a first-order feel on the aspect of programmability. As we saw in Chapter 4, programming heterogeneous targets and obtaining acceptable performance requires parallel programming knowledge, as well as familiarity with the details of the underlying architecture. In Section 5.2 we provide a detailed study on programmability and portability based on our experiments with a larger-scale program. This perceived lack of programmability of heterogeneous platforms is one of the reasons that hinder wider adoption of high-performance computing. Indicative of this fact is Figure 5.2, derived from a study by the Council on Competitiveness and the University of Southern California. It portrays the fact that a high num-

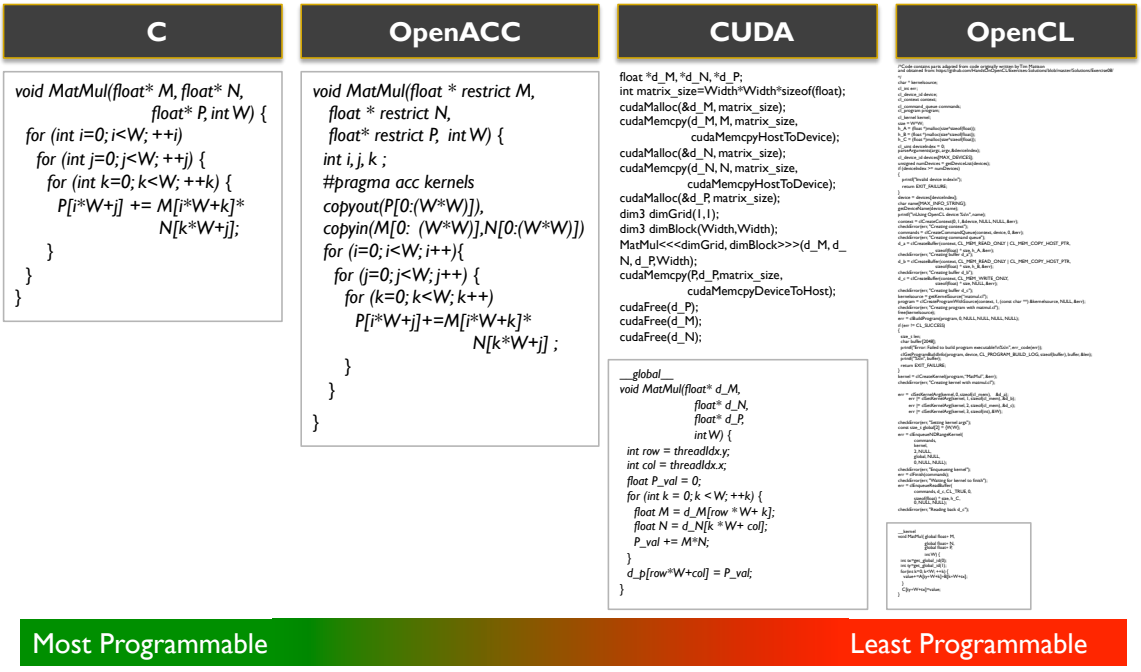


Figure 5.1: Programmability example: Matrix multiplication

ber of users use computing devices with a small number of processors and memory size, and utilize them to carry out simple jobs. As we move further along the horizontal axis of processor count, memory size, and job complexity, the number of users declines dramatically. This discrepancy in utilization of high-performance computing, termed "the missing middle", identifies a *national productivity opportunity*. Addressing this missing middle (that includes among others small and middle companies and institutions) remains a grand challenge. In Chapter 5.3 we propose one way towards addressing the needs of the missing middle by introducing a high-level abstraction and programming framework to automate performance, programmability and portability.

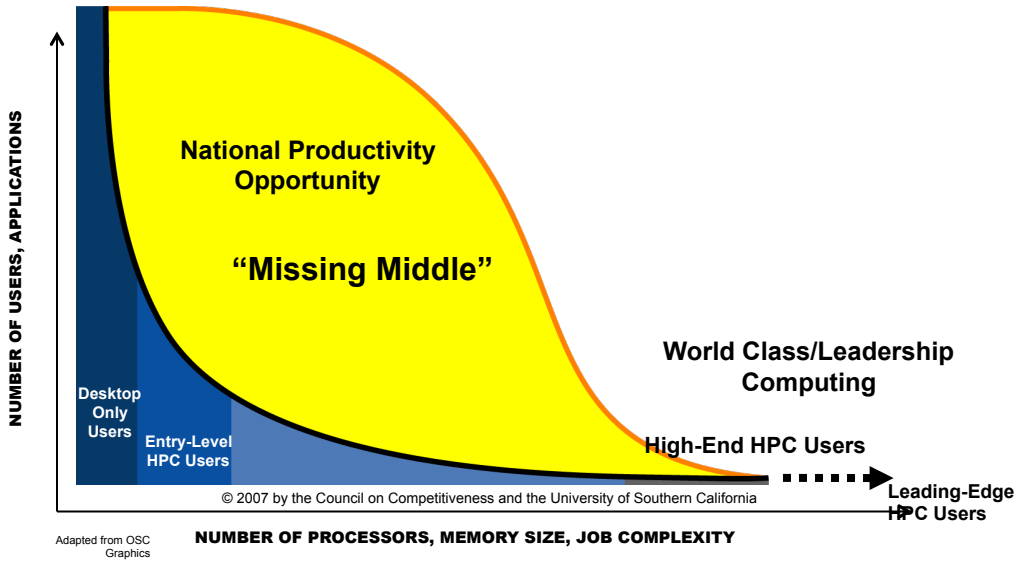


Figure 5.2: The "missing middle" in high-performance computing [119]

5.2 On the Programmability and Portability: A Case-Study with GEM

Many application areas, including finance, life sciences, physics, and manufacturing, have begun to use computational co-processors such as graphics processing units (GPUs), field programmable gate arrays (FPGAs), digital signal processors (DSPs), and even customized application-specific integrated circuits (ASICs) to achieve substantial gains in performance per watt and performance per dollar over traditional CPU implementations. Each of these solutions require programmers to adopt a different programming mindset than the typical, and well-studied, multi-core programming paradigm. This shift in mindset decreases the perceived programmability of these devices, and in turn, increases the cost to optimize and maintain code. While many scientists and industrial programmers possess a working knowledge of basic programming concepts, they typically

lack expertise in parallel programming. Programmability of a parallel platform is consequently a deciding factor in its adoption by such audiences.

Each platform is attempting to bridge the gap between performance and programmability in its own way. The Intel Xeon Phi co-processor attempts to ease programmability by offering a standard Linux environment on the device, which can be programmed with standard multi-core programming techniques. GPU and compiler vendors seek to increase the programmability of GPUs via extensions to familiar CPU interfaces, such as the development of the OpenACC directives to provide OpenMP-like functionality for fundamentally non-CPU architectures. In each case, there are highly programmable but imprecise and, comparatively, low performance interfaces as well as extremely difficult but high performance interfaces. The push and pull between programmability and performance comes down to a balance between cost and benefit, that is how much performance you can get and for how much effort.

In this chapter, we characterize the programmability and performance of multi- and many-core processors across a range of optimization levels, starting from naïve serial CPU code and extending to fully optimized CPU, Xeon Phi, and GPU code. Rather than skipping directly to the fully hand-tuned optimized versions for each target platform, we realize multiple versions of our molecular modeling code (i.e., GEM [17]) at different levels of optimization, ranging from the most programmable to the least and correspondingly from the worst performing to the best. We seek to address programmability both from a qualitative and quantitative standpoint. In order to provide a quantitative metric for programmability, or more generally, code complexity, we use the classical *source lines of code* (SLOC) metric, as well as *cyclomatic complexity* (CC). We also include code

examples and discussion to provide a more qualitative "feel" for the programming models in terms of portability, readability, and maintainability. Our contributions are as follows:

- An analysis of the trade-off between performance and programmability across various levels of optimization on CPU, Xeon Phi, and GPU.
- A characterization of the portability of architecture-aware optimizations across architectures.
- An evaluation of the effectiveness of directive-based parallelism along with compiler-assisted vectorization versus hand-tuned alternatives.

5.2.1 Measuring Code Complexity

Measures of code complexity can provide useful insights for the programmability of a program (written in a certain language for a certain architecture and with a number of potential code optimizations). Below, we discuss two of the more commonly used code complexity measures: source lines of code and cyclomatic complexity.

Source Lines of Code (SLOC)

One fairly straightforward and "easy" to calculate metric of programmability is the number of source lines of code (SLOC). Intuitively, one expects that more lines of code correspond to higher program complexity and, accordingly, decreased programmability. SLOC can prove to be a useful first-order approach for measuring programmability. However, it may fall short on providing qual-

itative insights on how complex it may be for a programmer to write each line (i.e., "not all lines were created equal"). Similarly, one may condense more code within a single line or lay it out in a more spaced-out manner (e.g., brackets of an if/else statement in separate lines versus in the same line with the if/else keywords). The last problem is largely addressed when comparing code that follows the same programming conventions (as we do in our case). Also, blank lines and comment lines should not be counted, as this could unfairly skew results.

Cyclomatic Complexity (CC)

Cyclomatic complexity [207] (commonly referred to as *McCabes cyclomatic complexity* from T. McCabe who introduced it) is one of the metrics that can implicitly help quantify the programmability of an algorithm. Cyclomatic complexity indicates the number of linearly independent paths in a certain program. Formally, this can be inferred by referring to the control flow graph of the program into consideration; a control flow graph is a directed graph where groups of instructions, represented as a node, can be independently executed and nodes connected via an edge represent the control flow of the program. For a given program (formulation is similar for a single function in a program) cyclomatic complexity is defined as:

$$M = E - N + 2,$$

where E = number of edges, and N = number of nodes in the graph.

From a practical standpoint, McCabe has showed that cyclomatic complexity can be calculated as:

$$M = K + L + 1,$$

where K = number of binary conditional statements, and L = number of conditional loop statements in a program (or function of a program).

We need to stress that cyclomatic complexity is a graph-based metric that is characteristic of the structural properties of a program. As such, the programming language used (e.g., C versus Fortran), or many optimizations do not necessarily affect this metric by themselves, unless they lead to modification of the algorithm and its ensuing control flow. An example of the latter would be loop unrolling; in this case L would decrease. It is important to note that the cyclomatic complexity and SLOC metrics are not necessarily correlated. For instance, in the loop unrolling example cyclomatic complexity would decrease by reducing the number of loops, but at the expense of code size (source lines of code) that increases by the unroll factor. Intuitively, though, conditional statements and loops introduce multiple code paths and arguably contribute to code complexity more than simple instructions do.

5.2.2 Experimental Setup

Our experimental setup for evaluating programmability and portability of heterogeneous architectures is identical to the setup in Section 4.2, where focus was on performance. Accordingly,

Table 5.1: Architectural parameters

Model (Architecture)	Intel E5-2680 (Sandy Bridge)	Intel Xeon Phi P1750 (MIC)	NVIDIA K20c (Kepler)
Frequency Cores Threads/core	2.7 GHz 16 (8/socket) 2	1.09 GHz 61 4	706 MHz 13 SMXs 16 (blocks/SMX)
SIMD (SP) GFLOPS (SP)	8-way 691.2	16-way 2092.8	192-way 3524.35
Mem. BW	102.4 GB/s	320 GB/s	208 GB/s
L1/L2/L3 cache (KB)	32/256/20480 (L1,L2 per core)	32/512/- (L1,L2 per core)	48(max)/1280/- (L1 per block)
Power	260W	300W	225W
Compiler	ICC 13.0	ICC 13.0	NVCC 5.0

we evaluate the same GEM [17] implementations across various optimization levels on the same three multi- and many-core platforms: a Sandy Bridge CPU (SNB), Xeon Phi co-processor (XP), and Kepler K20 GPU (K20), as noted in Table 5.1 (replicated from Section 4.2.2 for the reader's convenience). Results presented in Section 5.2.4 correspond to the *tobacco ring virus capsid (1A6C)* biomolecular structure, which entails calculating the electrostatic surface potential between 593,615 surface points and 476,040 atoms.

5.2.3 Optimization Levels and Programmability

Figure 5.3 provides a high-level overview of the optimization levels that we evaluate, starting from the original serial implementation and concluding with the manually hand-tuned implementation for each of the three platforms. We describe each method and evaluate its programmability aspects via the metrics discussed in Section 5.2.1. At the same time, we provide a qualitative perspective of programmability thereof.

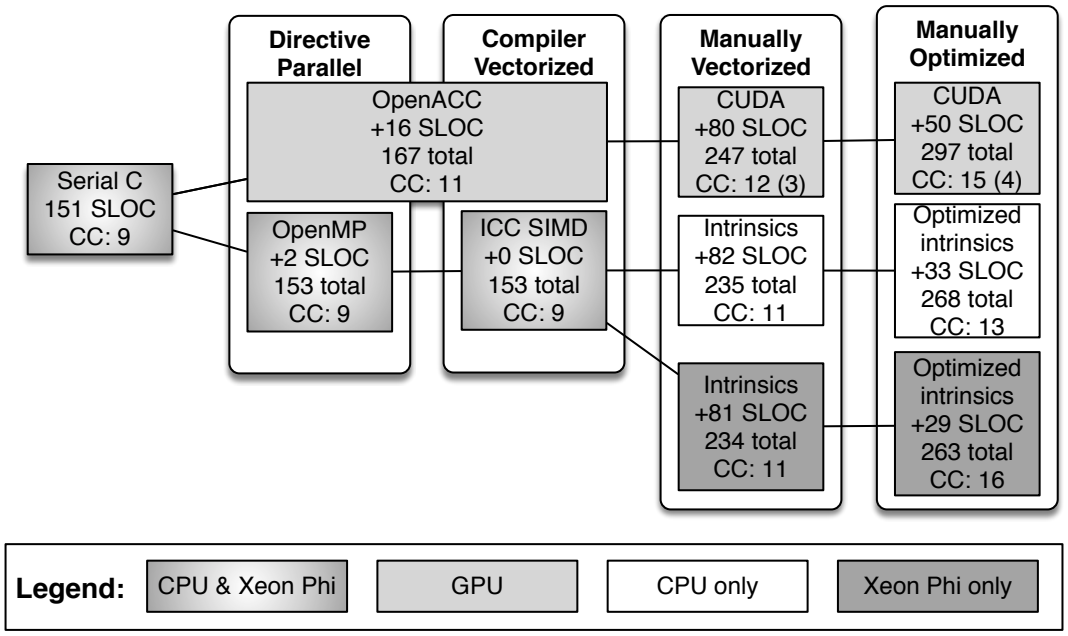


Figure 5.3: The progression and instantiation of each level of optimization on each architecture with the number of Source Lines of Code (SLOC) and Cyclomatic Complexity (CC) used in each implementation

Directive-based parallelization: The first set of implementations uses OpenMP and exploits the compiler support for the CPU and Xeon Phi to provide hinted multithreading as well as OpenACC, a variant of OpenMP for the GPU, which we will discuss in further detail below. Using OpenMP, the programmer can exploit all cores in a compatible device with the sole inclusion of the OpenMP library and the appropriate OpenMP directive on each section that should execute in parallel (hence the addition of two extra lines of code compared to the Serial implementation). This straightforward approach also facilitates code portability. With the computational kernel of the application unchanged, the same source can be compiled to serial code or run on systems with any number of cores. While OpenACC at first appears to provide an equivalent approach for GPUs, it is inherently *both* multi-threaded and vectorized, hence its spanning of both *directive parallel* and *compiler vectorized*. As shown in Figure 5.3, the OpenMP code retains the original cyclomatic

complexity value, since no changes take place in the code beyond placing the appropriate OpenMP directive.

Compiler-assisted vectorization: Modern compilers can transform scalar arithmetic to vector arithmetic for regular algorithms and loops. This set of implementations makes use of this compiler feature available in the Intel compiler for multi-core CPUs and Xeon Phi co-processor and in the PGI compiler suite for GPUs via the native vectorization that comes with compiling OpenACC for GPUs. The approach of the latter bears many similarities with OpenMP and the Intel compiler's offload model, combined with the newly released OpenMP SIMD directives.

As with the Intel compiler, various parameters/hints can be used to tune OpenACC regions for better performance. For example, OpenACC defines clauses to tune the division of loop nests across parallel blocks and threads, (*gang* and *vector* parameters) and control the independence, or lack thereof, of iterations in a given loop. However, OpenACC initialization routines and data movement directives require an additional 16 lines of code and increase the cyclomatic complexity from 9 to 11. In contrast, for the CPU or Xeon Phi, *no extra lines of code are needed* and hence cyclomatic complexity remains the same; only the appropriate setting of compiler flags is needed. In all cases the serial compute code is retained entirely in its original form and can be compiled to that serial version without alteration.

Manual vectorization: Explicit use of SIMD intrinsics offers far greater control over the vectorization of any given algorithm. Therefore, it can be worth abandoning automatic cross-architecture compatibility and manually vectorizing the code. This is the phase where the CPU and Xeon Phi codes diverge. While they each employ similar vector intrinsics, they have different vector widths,

and thus must use different registers and different sets of intrinsics. For this phase, the CPU and Xeon Phi require an additional 82 and 81 SLOC, respectively. Despite the code differentiation in terms of intrinsics used, the general algorithm remains the same (especially the number of loops and conditional statements), so the cyclomatic complexity stays constant among the two. This is not the case compared to the previous optimization level, from which we observe an increase of 2 units in cyclomatic complexity. For the GPU, the corresponding approach uses the CUDA programming model directly, which implicitly specifies all computations as vector operations and requires significant setup and data-management code to be added. These operations require 80 additional SLOC, nearly the same number needed for transitioning between the corresponding optimization levels for the CPU and Xeon Phi. While the cyclomatic complexity only increases by one unit with respect to the host code, the CUDA kernel function introduces another level of complexity, albeit small by itself (3). It should be noted, however, that the SIMT paradigm of GPU computing with CUDA eliminates the need for explicit loops and "artificially" hides cyclomatic complexity. Therefore, in evaluating cyclomatic complexity of the host side code one should take into account that the host side code starts with two less loops (the ones "translated" to the CUDA kernel).

This level of optimization imposes extra intellectual burden on the programmer, specifically "thinking in parallel is required." For the CPU and Xeon Phi, the process is quite similar, as we see in Figures 5.4b and 5.4c. Each employ compiler intrinsic functions to explicitly specify the vector operations to use. To CPU optimization veterans, this may look familiar, but otherwise it obscures the intent of the code significantly. Alternatively, Figure 5.4a shows the line of code as it is in serial

```
float sum2=(1.f/d_int−1.f/d_ext)/(one_plus_a_b∗A);
```

(a)

```
__m512 sum2_vect=_mm512_div_ps (
    _mm512_sub_ps (
        _mm512_div_ps (ONE,D_INT),
            _mm512_div_ps (ONE,D_EXT)
        ),
    _mm512_mul_ps (ONE_PLUS_A_B,A)
);
```

(b)

```
__m256 sum2_vect=_mm256_div_ps (
    _mm256_sub_ps (
        _mm256_div_ps (ONE,D_INT),
            _mm256_div_ps (ONE,D_EXT)
        ),
    _mm256_mul_ps (ONE_PLUS_A_B,A)
);
```

(c)

Figure 5.4: (a) Scalar/CUDA code (b) Vector intrinsics code for Xeon Phi (c) Vector intrinsics code for Sandy Bridge CPU

C, OpenMP, OpenACC, or CUDA, the computation remains visually the same. The CUDA version does the same thing as the explicit vector instructions in Figures 5.4b and 5.4c, but it preserves the readability of the original. The burden on the GPU is mostly in the setup required to call GPU kernels, but it leaves the computational kernel largely unchanged. The CPU and Xeon Phi are the reverse at this level, requiring virtually no setup but a great deal of changes in the computational kernel.

Manually optimized code: For the final set of implementations, we provide an extra level of optimizations for each of the evaluation platforms by explicitly using blocking/tiling, shuffling, and explicitly altering the specific hardware instructions used to target faster execution, such as fused multiply-add (FMA) and approximate reciprocal division/square root. More details about the best set of optimizations for each platform and a detailed description are given in Section 4.2.4.

In CUDA, the applicable optimization techniques are quite different than those used in typical x86 code. The optimization search space itself is bigger as well, with minor changes in the code

severely affecting performance (e.g., data structures, memory coalescing, and efficient use of the memory hierarchy). The required programming effort is hence not realistically reflected by complexity measures, since deciding on the right combination of optimizations is a strenuous process by itself. The measures of complexity, still partially reflect the programmability challenges, with a 20% SLOC increase, and an associated increase in cyclomatic complexity in both the host and device code. In optimizing for the Intel MIC architecture, details about the underlying architecture (e.g., memory hierarchy and interconnect details) are essential, but Intel MIC's resemblance to traditional multi-core CPU architectures implies similarity in the optimizations, most of which parallel programmers are already familiar with. As a result of the above the number of SLOC among the manually optimized implementations in CPU and Xeon Phi remain almost the same. However, the cyclomatic complexity increases considerably more in Xeon Phi versus the CPU and with respect to the manually vectorized implementation (e.g., addition of extra loops related to the L2 blocking optimization in both CPU and Xeon Phi, and L1 blocking required in the case of Xeon Phi only).

5.2.4 Performance Impact

So far, we have discussed the programming effort required for each optimization level and provided the number of SLOC and CC as a rough quantitative measure. In Figure 5.5, we show how close to the best achieved performance for each platform we get with each optimization level. We observe that different levels of optimization help reach best performance at a different rate, depending on the platform. In the case of the CPU, directive parallel and compiler-assisted vectorization help

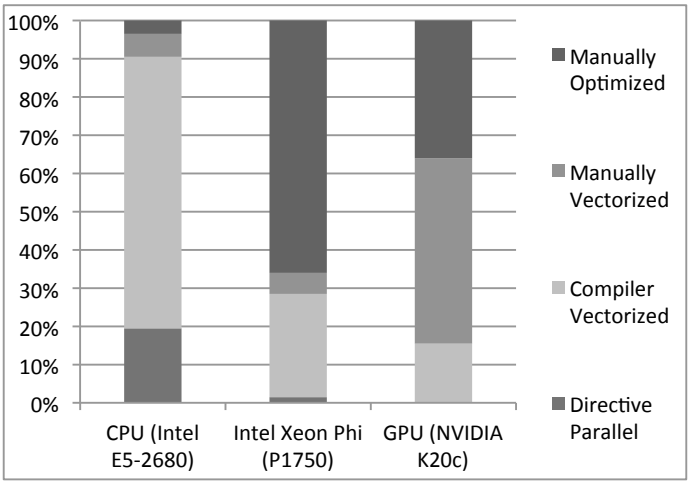


Figure 5.5: The percentage of best achieved performance achieved with each level of optimization

attain over 90% of the best achieved CPU performance. Auto SIMD, in particular, accounts for 71.05% of the achieved performance. Considering the above, the programmer can rely on the extensive and highly mature CPU compiler infrastructure along with OpenMP and still attain high performance.

On the other hand, manual optimizations are quite important for both the GPU and Xeon Phi. OpenACC is a relatively new standard and its correspondingly young compilers can only take performance so far. Explicit use of CUDA or very careful tuning is essential for achieving acceptable performance. Manual optimizations are required to fill the last 35.9% of the gap between the performance attained through use of naïve CUDA code and the best achieved performance.

Finally, in the Xeon Phi case, using CPU code directly on the device in the first two levels delivers extraordinary programmability, but also a very low percentage of the best possible performance. Auto-vectorization makes a significant difference, but even with that and manual vectorization, our implementation only reached 35% of the best performance that we achieved overall. Manual

optimizations are the most important (65.7% of overall performance) for Intel MIC, due to its high sensitivity to caching behavior. We discuss these optimizations and their effect on performance in detail in Sections 4.2.4 and 4.2.5.

We note that the above conclusions refer to the case of n-body class problems or, more generally, data-parallel workloads with an emphasis on floating-point arithmetic. Different problem classes might benefit less from directive-based multithreading or compiler-assisted vectorization due to irregular data access patterns or complex dependencies. In such cases, manual SIMD and more complex threading using appropriate synchronization constructs (e.g., semaphores and barriers) would be of paramount importance.

5.3 On Bridging the Performance, Programmability and Portability Gap of Heterogeneous Platforms

The ongoing parallel revolution has democratized parallel computing by making unprecedented amounts of computational power accessible to an ever increasing part of the scientific community. In contrast to what used to be the norm, more scientists, engineers, researchers (herein collectively referred to as *domain experts*) have access to at least commodity multi-core CPUs or single-node accelerator-based heterogeneous systems.

While powerful hardware is readily available, the ability to exploit it at its fullest and ignite scientific breakthrough at a proportionate level remains on the ground. One prime reason is that

programming itself remains a prerogative of the few. Many domain experts possess rudimentary knowledge of at least one programming or scripting language that allows them to verify functional correctness of their algorithms, but the vast majority practically ignores *parallel* programming, an issue exacerbated by the number of parallel programming abstractions and languages. As such, domain experts typically have to resort to programming experts in order to have their algorithms coded, or optimized for *performance*. This process introduces communication overhead, errors and barriers, including the need for the programmer to obtain domain-specific knowledge and vice versa.

The question we attempt to address in this chapter is: "Can we realize a *programming abstraction* and implement a *development framework* for domain experts that adresses the aforementioned issues, i.e., an approach that provides a balance between performance/programmability, and renders this audience active participants of the parallelism era?"

We claim that such an abstraction should ideally be: (a) automatically parallelizable, optimizable and tunable to desired target hardware, yet platform agnostic, (b) intuitive, familiar, with minimalistic syntax, yet general, scalable and powerful enough to express real-world problems, (c) data-visual and interactive, in that code, data structures and data itself are visible simultaneously, thus facilitating algorithmic expression, understanding, and debugging, (d) able to (implicitly) integrate with existing legacy code (code used today in many sciences dates back to the early 70s).

In this work, we propose **GLAF**, a visual code generation and auto-tuning framework for single-node parallel computing systems, which aspires to steer programming by domain experts with *minimal* or *basic* programming knowledge towards the general directions discussed above.

5.3.1 GLAF Framework

Graphical User Interface (GUI)

Programming using GLAF differs from programming using a typical programming language (e.g., C or Java). In the latter users write code in a free textual format using the keyboard and express the algorithm using the appropriate language constructs. In GLAF, typing is kept at a minimum (e.g., naming grid variables) and programming is based on an intuitive point-and-click *visual interface*. GLAF's GUI is implemented in the form of a web page (Figure 5.6) that is written in a combination of HTML5 and JavaScript. This code drives the web interface (buttons, forms, images, menus, etc.) that facilitates GLAF code development and is responsible for populating appropriate environment variables (or *internal objects*, modeled after JavaScript functions). Modeling of such objects (Section 5.3.1) lies at the basis of code generation and parallelism analysis algorithms.

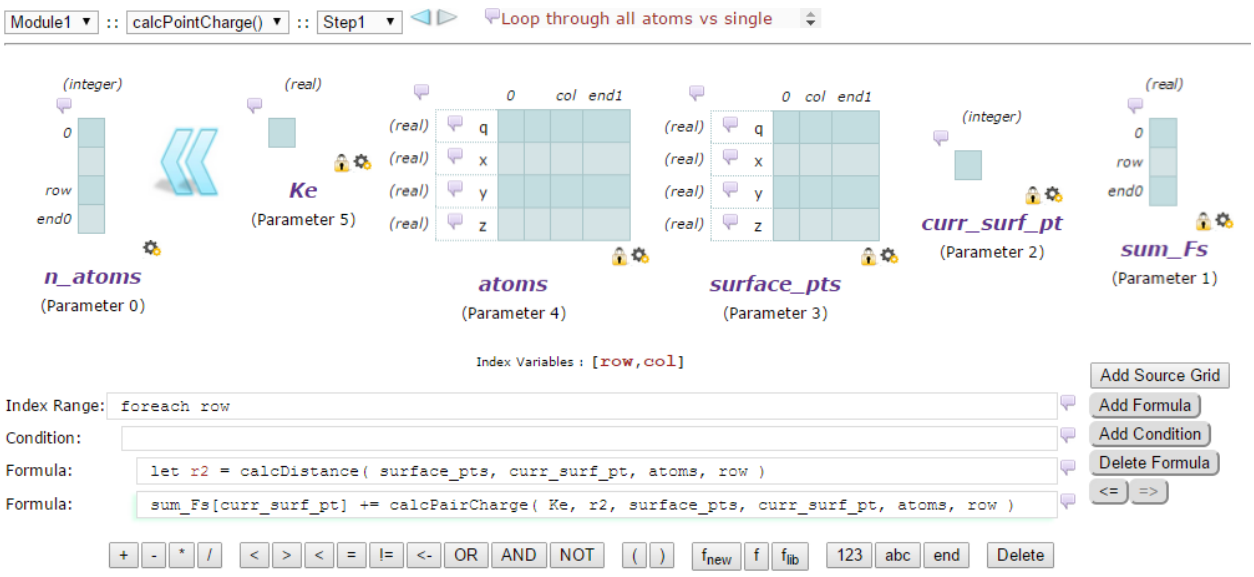


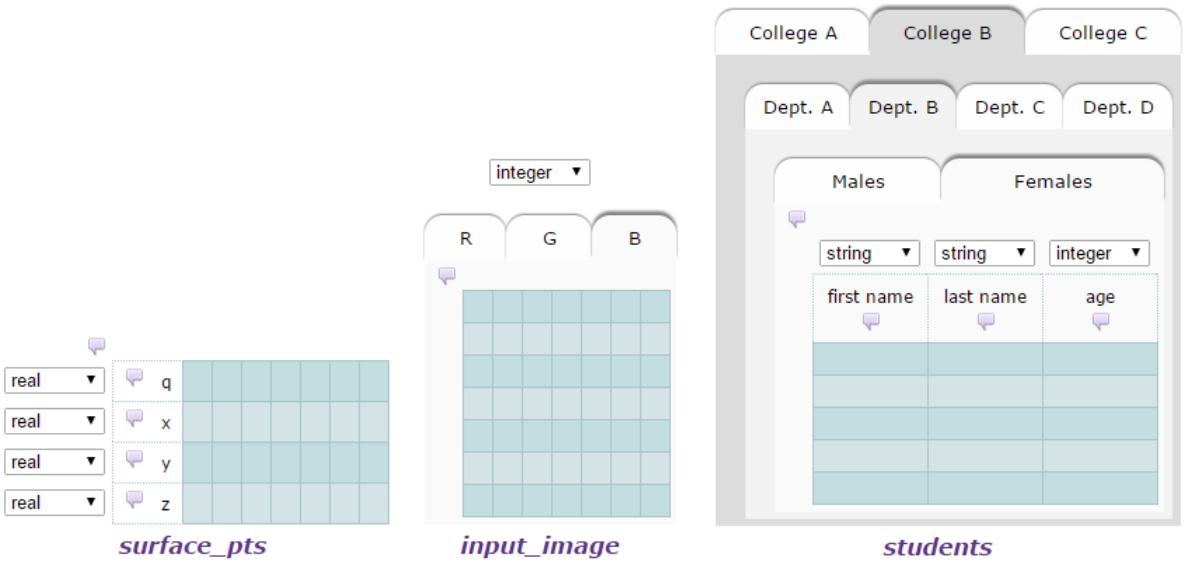
Figure 5.6: GLAF user interface: a GLAF step (code boxes are *automatically* filled through a point-and-click interface)

GLAF as a Programming Language

Data Structures

GLAF variables are based upon the concept of *grids*. Grids are simple, yet powerful data structures that can be used to represent a variety of real-world problems. A scalar variable is a 0D grid with one element and a 1D array is a 1D grid with multiple elements. Similarly, we can generalize for higher-dimensional data structures. Grids of this type may contain a single data type and be indexed by corresponding *index variables*. Allowing dimension(s) to have *titles* we can represent *tables*, in which case we may use a combination of titles and indices to address a specific grid cell. More complex structures can be described using the grid abstraction, by use of *multiple* data types across one of the dimensions with titles. Such grids can represent what would be a *struct* in C. Examples of grid declaration in GLAF are shown in Figure 5.7.

The *grid abstraction* has been used as the basis of programming languages or language extensions in the past [154, 278] due to certain advantages over using multiple, distinct data structures. In practical terms, the grid abstraction is *general* and *scalable* enough to model many real-world problems. A mathematical relation (a mapping from a domain to a range) that is discrete and finite can be represented with a grid (e.g., trees, graphs, databases). For example, a graph can be represented by using an adjacency matrix, a tree data structure can be substituted by matrices indicating the parent-child relationships and a sparse matrix can be represented in the compressed sparse row format (CSR). More importantly, the *regularity* of the grid abstraction allows for a uniform internal representation (Section 5.3.1) that in turn renders code generation and many types



(a) Declaration of a struct. (b) Declaration of an RGB image. (c) Declaration of a simple database.

Figure 5.7: Examples of grid declaration in GLAF

of transformations and optimizations, including parallelism analysis, more straightforward. In terms of programming, coding an algorithm using the grid abstraction urges programmers to think in terms of relationships as opposed to exact data layout. For instance, representing a tree requires considering the higher-level relationships within the data (e.g., *parent_of*, *child_of*), instead of the exact tree structure and how it is coded on the lower language-level , e.g., using pointers in C and how to follow them. All things considered, the grid is a *familiar* abstraction (e.g., images, matrices, spreadsheets) to programmers and non-programmers, alike, makes visualizing certain data structures easy and inspecting results more intuitive. In some cases, however, and despite the aforementioned advantages, visualization-wise the original data structures may be more intuitive. Visualization-related issues are further discussed in Section 5.3.2, along with the rest of GLAF's capabilities.

Programming Constructs

In Figure 5.6 we see an instance of a GLAF step. A *step* is the basic building block of a GLAF program and represents a step of computation, whereby data from input grid(s) flows after undergoing computation to an output grid. It may include a *loop* over zero, one or more dimensions. Such loops can be typical *foreach* loops (in the *start:end:step* format), or *forever* loops. Furthermore, a step may include *conditional statements*. Appropriate buttons insert the corresponding conditional keyword when a condition box is clicked on, and boxes are *indented* accordingly to make the order (and potential nesting) of conditionals clear. A *formula statement* can include grid cells, arithmetic/logical operations on them, as well as user-defined or library function calls (i.e., predefined sets of useful functions, like typical *Math* or *I/O* library functions). A GLAF program may include multiple steps, which belong into one (main) or more (user defined) functions. Functions, finally, can be grouped into GLAF *modules*.

Internal Representation

All constituent GLAF elements have a corresponding internal representation in JavaScript. As the user develops an algorithm using GUI buttons and keyboard, event listeners activate appropriate JavaScript functions to populate JavaScript objects modeled after *constructor functions* (in support of an object-oriented programming JavaScript). These are used in creating and navigating the GUI screens, in code generation, parallelism analysis, etc. They can also provide a preliminary error checking substrate (e.g., disallows declaring the same grid name twice) before code is generated,

thus minimizing multiple compiler errors at the last development stage. A representative selection of the most important internal objects is outlined below:

(a) **Expression Type object:** defines the type of an *expression object* (e.g., grid, function call, string, number, conditional statement), (b) **Function object**: models a function and contains information about grids declared in a function, its arguments, its return value, etc. A function object encapsulates an array of *step objects* containing information about its steps, (c) **Step object:** models a step and contains information about all grids used in it, as well as information about the code in this step contained in arrays of box *expression objects*, (d) **Expression object:** models a line of code, can represent a *formula, a loop, or a conditional statement* and be a single object or comprise more expression objects in a tree-like structure, (e) **Grid object:** models a grid object according to the guidelines set in Section 5.3.1 (name, number of dimensions and their sizes, etc.) A simplified example of a grid object and parts of its internal representation is shown in Figure 5.9.

For parallelism analysis the above objects are used in a single pass to formulate an *additional collection* of objects that store information about *scalar* and *non-scalar* grids (Figure 5.8). Finally, for non-parallelizable steps, detailed information is stored in appropriate objects that may be used in a feedback functionality. Such objects collect information about the name of grid, its type (scalar/non-scalar), error type (e.g., RAW dependency), and the name of function(s), steps and box numbers (“line of code”).

IndVarsWrittenInStep[M][F][S]: 3D structure, elements correspond to a step (in a given module/function) containing info on the index variables iterated over on this step's loop.
FuncsFromCaller[M][F][S]: 3D structure, elements correspond to a step (in a given module/function) containing info (like name, ID, arguments per each call within step) about each function called from within this step.
NonScalarGridsInFunc[M][F]: 2D structure, elements correspond to a function (in a given module) containing info (like name, written/read, indices written/read for each dimension) about non-scalar grid arguments of function.
NonScalarGridInstances[M][F][S]: 3D structure, elements correspond to a step (in a given module/function) containing info (like name, written/read, indices written/read per dimension) on non-scalar grids written/read in step.
ScalarGridInstances[M][F][S]: 3D structure, elements correspond to a step (in a given module/function) containing info on scalar grids written/read in this step.

Figure 5.8: Internal representation objects for parallelism analysis back-end

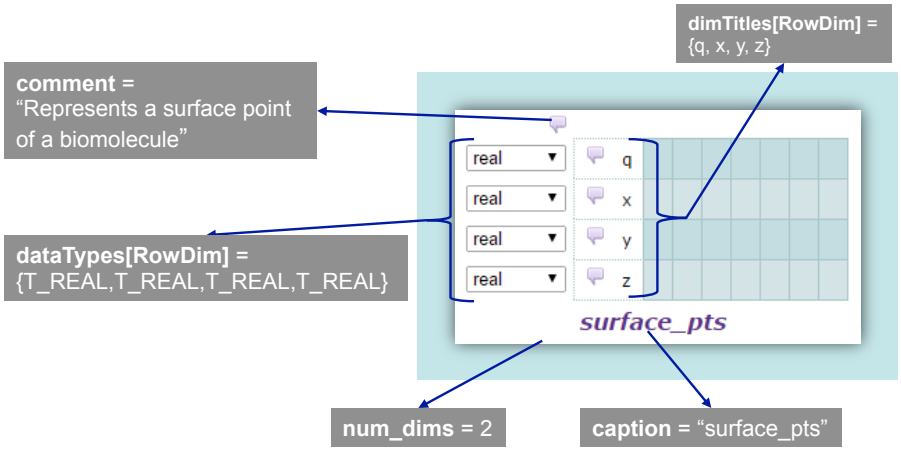


Figure 5.9: Example (simplified) of grid object internal representation

5.3.2 Capabilities

Code Generation

To support GLAF data visualization within the GUI, JavaScript code generation is *de-facto* supported. However, to broaden GLAF's utility we have implemented support of code generation for *Fortran*, *C*, and *OpenCL*, languages that have been extensively used in technical/scientific computing. Users need *not* write a single line of Fortran-, C-, or OpenCL-specific code, since GLAF follows a paradigm of *writing an algorithm in one language (GLAF) and getting source code in many*. Much of the complexity and peculiarities of languages are concealed from the user, thus facilitating code development for domain experts.

Code generated for a target language by GLAF falls under *two* broad categories, i.e., *serial* and *parallel*. The former is a one-on-one mapping of the GLAF programming constructs, as represented internally, to the target language's constructs (Figure 5.11), while the latter can follow one of two different approaches: (a) for Fortran and C parallel versions, parallelizable regions are decoreted with appropriate *OpenMP directives* so that code can run on any device supporting OpenMP (e.g., multi-core CPUs, Intel Xeon Phi), (b) for OpenCL, appropriate OpenCL kernels are generated (parallel by design) along with the corresponding host-side code. Detecting parallelism is a task undertaken by the *GLAF parallelization analysis back-end* (Section 5.3.2).

C and Fortran code generation: Figure 5.12 shows the algorithm for generating C/Fortran code via GLAF, and Figure 5.10 shows generated C code for the parallel implementation of a GLAF step (Figure 5.6) (Fortran look is similar). One can notice the *readability* of the automatically-

```
int ft_calcPointCharge(int *ft_n_atoms, double *ft_sum_Fs,
int ft_curr_surf_pt, struct TYP_surface_pts typvar_surface_pts,
struct TYP_surface_pts typvar_atoms, double ft_Ke) {
    int fun_curr_surf_pt;
    double fun_Ke;
    int ft_ReturnValue;
    int ft_row;
    int ft_end0;
    int atoms_q=0, atoms_x=1, atoms_y=2, atoms_z=3;
    int surface_pts_q=0, surface_pts_x=1, surface_pts_y=2,
        surface_pts_z=3;
    double ft_r2;
    fun_curr_surf_pt = ft_curr_surf_pt;
    fun_Ke = ft_Ke;
    // Loop through all atoms vs single surf pt
    ft_end0 = 4-1;
    double tmp_sum_Fs=ft_sum_Fs[fun_curr_surf_pt];
    #pragma omp parallel for collapse(1) private(ft_r2) \
    reduction(+: tmp_sum_Fs)
    for (ft_row = 0; ft_row <= ft_end0; ft_row += 1) {
        // Calculating the distance between a surface point
        // and each atom
        ft_r2 = ft_calcDistance(typvar_surface_pts,
                fun_curr_surf_pt,typvar_atoms,ft_row);
        // Add current pairs charge to total
        tmp_sum_Fs = tmp_sum_Fs + ft_calcPairCharge(fun_Ke,
                    ft_r2,typvar_surface_pts,fun_curr_surf_pt,
                    typvar_atoms,ft_row);
    }

    ft_sum_Fs[fun_curr_surf_pt]=tmp_sum_Fs;
}
```

Figure 5.10: Example of automatically generated C code

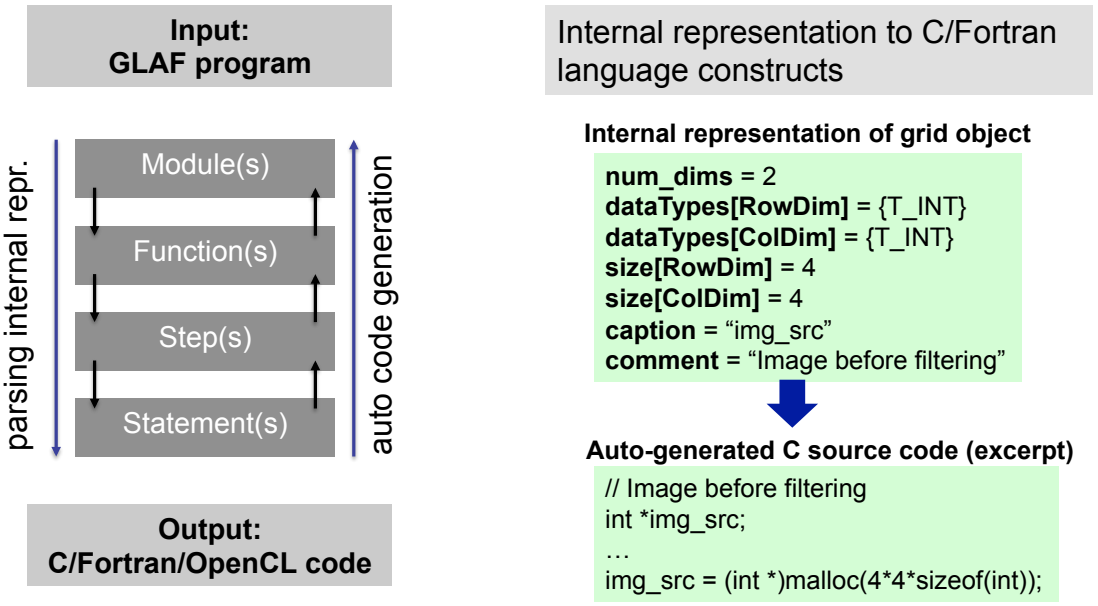


Figure 5.11: Overview of the automatic code generation process

generated code: code is *indented* and contains appropriate *comments*, as inserted by the user using the GUI. Depending on the user, the above features may vary on the importance scale: a non-programmer domain expert may never need to see the actual code. On the other hand, users that know programming (although *not necessarily* parallel), may want to inspect the code, apply further optimizations, or write a GLAF module, auto-tune taking into account legacy code's requirements (e.g., data types and layout of a function's arguments) and use the resulting parallel generated code as part of that larger unoptimized or serial legacy code (as we showcase in Section 5.4).

OpenCL code generation: Starting from the same high-level GLAF program, OpenCL code generation enables executing code on targets that support OpenCL. While C and Fortran code (with OpenMP directives) can already address the multi-core CPU and Intel MIC cases, OpenCL provides yet another opportunity. More importantly though, OpenCL now enables users to exploit the power of *GPU* and *FPGA* computing. Following, we illustrate some of the main concepts of

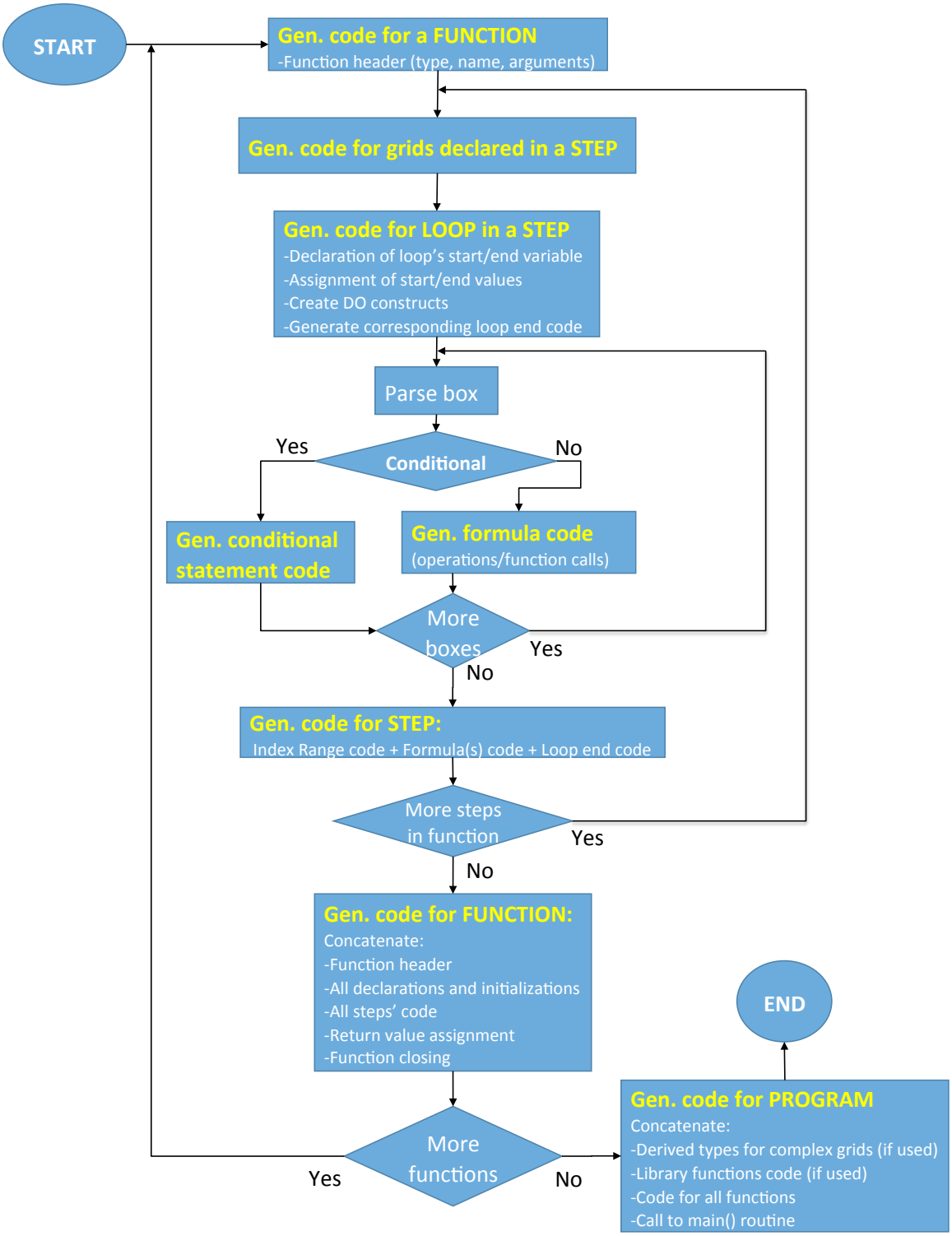


Figure 5.12: Algorithm for automatic code generation

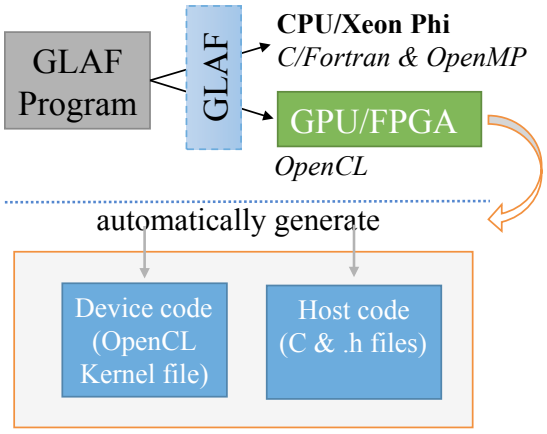


Figure 5.13: GLAF OpenCL code generation

OpenCL code generation by showing how a simple example developed in GLAF (Figure 5.14) is translated automatically to OpenCL (Figure 5.15). Overall, the automatic OpenCL code generation process results in three files (Figure 5.13): a host code (.c) file, a file (.cl) that contains all the device code (kernels) and a header file (.h) that contains the OpenCL boilerplate initialization/finalization code and auxiliary functions.

(a) **Main host and device code generation:** For OpenCL code generation purposes, parallelism is exposed at the *GLAF step* level. Each step that contains a loop undergoes loop-level parallelism analysis, is identified as non-parallel or parallel, and serial or parallel code (OpenMP or OpenCL) is generated, accordingly. Specifically, a check-box at each step (bottom of Figure 5.14) allows the user to select OpenCL over OpenMP. The reason for this choice is that a GLAF step can range from simple to more complex (with function calls, multiple statements, conditionals). Therefore, it may be faster for a simple parallel step (e.g., few lines of code, small array initialization) to run in the host (CPU) with OpenMP than in the device (GPU or

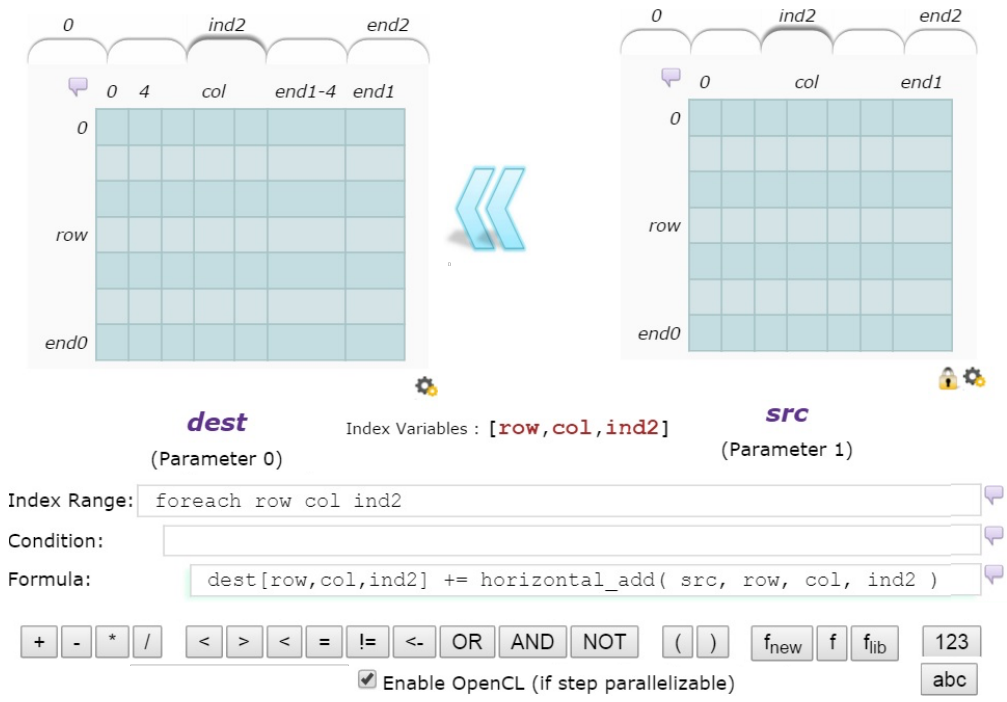


Figure 5.14: Example GLAF program for explaining OpenCL code generation

GLAF step code (as filled automatically using the GUI)
(parallelism identified at the granularity of a GLAF step)

```
foreach row col=dest(4:end1-4:1) ind2
    dest[row,col,ind2] += horizontal_add( src, row, col, ind2);
```

b

a

a Generation of OpenCL kernel code for GLAF steps that are identified to be parallelizable.

```
__kernel void ft_st_3dfd_1_dev(__global double * restrict ft_dest,
                               int ft_grid_dimen,
                               __global double * restrict ft_src,
                             * __global int* restrict index_dat) {
1  int ft_row;
2  int ft_col;
3  int ft_ind2;
4  double res_horizontal_add_device;

5  ft_row= get_global_id(0);
6* ft_col= get_global_id(1)+index_dat[2];
7  ft_ind2= get_global_id(2);

8  ft_dest[ft_ind2*ft_grid_dimen*ft_grid_dimen + ft_row*
           ft_grid_dimen + ft_col] +=
9  ft_horizontal_add_device(ft_src,ft_grid_dimen, ft_row, ft_col,
                            ft_ind2);
}
```

.cl file

b Generation of appropriate OpenCL host code for parallelizable step (transformation of loop into an OpenCL kernel call to the kernel generated for the step).

```
int ft_st_3dfd(cl_mem *ft_dest_dev, double *ft_dest, int ft_grid_dimen,
               cl_mem *ft_src_dev, double *ft_src) {
   (...)
1  ft_end0 = ft_grid_dimen-1;
2  ft_end1 = ft_grid_dimen-1;
3  ft_end2 = ft_grid_dimen-1;
4* index_data[2] = 4;
5  //Converting parallel loop to NDRange dimensions.
6  size_t globalWorkSize_s0[3] = {ft_end0+1, ft_end1-4-4+1, ft_end2+1};
7  //Creating kernel (generated in .cl file from loop's body).
8  kernel_cl = clCreateKernel(program, "ft_st_3dfd_1_dev", &error);
9  assert(error == CL_SUCCESS);
10 //Data transfers.
11* clEnqueueWriteBuffer(commands, index_data_dev, CL_TRUE, 0,
                         sizeof(int)*9, (void *)index_data, 0,NULL, NULL);
12 h2d_tran(ft_dest, sizeof(float)*ft_grid_dimen*ft_grid_dimen*
           ft_grid_dimen);
13 h2d_tran(ft_src, sizeof(float)*ft_grid_dimen*ft_grid_dimen*
           ft_grid_dimen);
14 //Setting kernel arguments (based on parallel step's grids).
15 clSetKernelArg(kernel_cl, 0, sizeof(cl_mem), ft_dest_dev);
16 clSetKernelArg(kernel_cl, 1, sizeof(int), (void* )&ft_grid_dimen);
17 clSetKernelArg(kernel_cl, 2, sizeof(cl_mem), ft_src_dev);
18* clSetKernelArg(kernel_cl, 3, sizeof(cl_mem), &index_data_dev);
19 //Enqueuing OpenCL kernel for execution.
20 clEnqueueNDRangeKernel(commands, kernel_cl, 3, NULL,
                          globalWorkSize_s0, NULL, 0, NULL, NULL);
21 clFinish(commands);
   (...)
}
```

.c file

Figure 5.15: GLAF OpenCL auto-generated code

FPGA) with OpenCL (e.g., if the OpenCL kernel call overheads cannot be amortized over a large computation).

For exposing parallelism via OpenCL (for steps identified as loop-independent) the step's body is converted to an OpenCL kernel (kernel code generation is discussed later). A special function in the code generation back-end is responsible for parsing the loop's internal representation and converting the loop to the kernel's *NDRange*. An NDRange represents the global work size partitioning across each of a maximum of three dimensions (OpenCL standard's limitation). This happens as follows (Figure 5.15): each loop index (e.g., *row, col, ind2*) represents a global dimension in the NDRange. The value of each global dimension corresponds to the number of loop's iterations across this dimension, taking into account the loop's boundaries and step. The generated NDRange in our example is shown in Figure 5.15, *line 6* of .c file.

The parallel step's body is replaced with a kernel call (clEnqueueNDRangeKernel()) to the appropriately named and automatically generated OpenCL kernel in the separate .cl file (e.g., Figure 5.15, *line 20* of .c file). The kernel call is preceded by a series of calls (clSetKernelArg() - Figure 5.15, *lines 15-18* of .c file) for setting the appropriate kernel arguments (OpenCL cl_mem buffers or scalar variables). The above functionality is achieved by another specialized code generation function, whose role is to parse the GLAF internal representation of input and output grids present in the current step and construct the calls with the appropriate arguments (kernel name, argument number, size, variable name). The kernel call is followed by a clFinish() call (including automatic error-checking) that also functions as a synchronization point that enforces coherence between the memory contents of the host and device.

As far as device code is concerned (all included in the generated .cl file), there are two types of functions generated: a) the *kernel functions* that correspond to the parallel steps of a GLAF program, and b) the *device functions* that are functions called from within a kernel function. The parameters to these functions are in the form of *__global* pointers (in the case of dynamically allocated memory) or normal scalar variables. Data parallelism in kernel functions (according to the OpenCL SPMD paradigm) is achieved by accessing different memory locations based on the combination of work-group/work-item IDs (Figure 5.15, *lines 5-7* of .cl file). Obtaining the index for each dimension takes into account the *start* and *step* values for each index variable of the original loop that is being encapsulated in the OpenCL kernel (star-annotated lines in Figure 5.15).

(b) **Host/device memory considerations:** In OpenCL, when a host and device (e.g., CPU and FPGA) have separate memory address spaces, memory needs to be allocated in both. GLAF OpenCL code generation back-end parses a step's grid objects internal representation (which includes name, size, data type, etc.) and automatically generates appropriate code for declaring and allocating space for cl_mem device-side buffers that correspond to the host-side ones and passes them as needed to the kernel code.

As execution alternates between host and device data is transferred between the two. GLAF OpenCL code generation back-end generates the code (in the form of wrapper functions - e.g., *h2d_tran()*, Figure 5.15, *lines 12-13* of .c file) that is responsible for host-to-device and device-to-host data transfers, ensuring data coherence across host and device memory spaces, and elimination of redundant data transfers. For instance, if a device-side buffer is written in the

device, but not used in host code that follows until the next kernel execution, then the buffer's contents are neither transferred to the host (redundant), nor subsequently transferred from the host to device (coherence violation). The above scheme is implemented by tracking allocation and read/write accesses to the non-scalar grids in host and device code and performing appropriate checks at run-time before each GLAF step.

(c) **Data linearization:** Disjoint memory spaces in an OpenCL execution scenario (e.g., CPU host/GPU or FPGA device) impose inherent limitations to passing structs that include pointer elements to a kernel. Such structs cannot be passed as such, since a CPU address space address is unusable in an FPGA context. This kind of structs needs to undergo linearization/marshaling. This effectively means that a struct declaration needs to be expanded as multiple declarations of arrays/pointers of the respective type on the host-side and corresponding declarations need to take place for the device-side (cl_mem buffers). As in the general case, initialization of the latter may be needed. GLAF extends the existing data layout transformations (structures of arrays to/from arrays of structures) to include breaking down a struct to multiple (stand-alone) arrays. All the above procedures, as well as parameter passing and struct element accesses are automatically handled by GLAF.

(d) **Boilerplate OpenCL code:** Every OpenCL program requires an initialization procedure that selects the OpenCL platform and a specific OpenCL device of that platform, and that initializes various OpenCL objects (program, command queue(s), etc.) This procedure, while standard, requires many lines of code, including appropriate error handling, and is non-intuitive for novice programmers. Similarly, at the end certain actions need to take place (like *releasing*

program objects, command queues, etc.) GLAF obviates the need for programmers to familiarize themselves with the aforementioned procedure and OpenCL objects, as a corresponding function in code generation back-end automatically generates the boilerplate code, including the necessary OpenCL objects that are then used in the appropriate places within the rest of the code, as needed. Last, GLAF provides wrapper functions for memory allocation enforcing the alignment requirements of Altera OpenCL, as well as data transfers.

Auto-Parallelization

An important part of GLAF is its *parallelism analysis back-end*. Parallelism is analyzed at the granularity of a GLAF step and the result of this analysis is two-fold: (a) *information about the loop index variables whose loops are parallelizable*, (b) *information about the reasons parallelization is not possible*. Our analysis and resulting code transformations act in a complementary manner to a compiler providing code more favorable to compiler optimizations and do not claim to surpass its well-established capabilities. However, despite the comparatively simpler nature of our analysis, there are certain reasons for performing it on this level (Section 5.3.6).

For (a), a form of cross-iteration loop dependence analysis is carried out in a pre-code-generation pass to identify dependencies within a GLAF step. High-level pseudocode is given in Figure 5.16. This analysis includes: *non-scalar* grid variables (includes grids passed in functions called from within a step), and *scalar* grid variables (for which we build a control flow graph that is subsequently used to identify dependencies). Information collected in this stage are given in Section 5.3.1. Relevant parallelism information is used in parallel code generation and displayed

```
Function findParallelismInProgram()
For each module M
  For each function F of M
    For each non-scalar grid argument G
      Record name of G in NonScalarGridsInFunc[M][F]
    Call recordGridsAndIndicesOfStepBoxes(M, F)
Call findParallelismInFunction(MainModule, MainFunc)

Function findParallelismInFunction(M: <module ID>, F: <function ID>)
For each step S of F of M
  Parse Index Range box,record index variables in IndVarsWrittenInStep[M][F][S]
  Call buildCFG() to build control flow graph used in scalar grid dep. analysis
  For each box B after the Index Range box
    If B is of type Mask Statement (if/elseif)
      Add non-scalar/scalar grids in Mask to NonScalarGridInstances[M][F][S]
    Else if B is a Formula of the form G = RHS_expr
      Call addGrid(G) and parseRightHandSide(RHS_expr)
    Else if B is a formula of the form LET N = RHS_expr,
      Call parseRightHandSide(RHS_expr)
    Else if B is a stand-alone function call of the form func(args)
      Add func(arg) info to FuncsFromCaller[M][F][S]
  // Do cross-iteration dependency analysis among non-scalar grids, in two steps:
  // (i) For each function in FuncsFromCaller[M][F][S], detect and record iteration
  // dependencies among its arguments
  Call doDependAnalForNonScalarGridsPassedToAllFuncs(M, F, S)
  // (ii) For each grid in this step, find iteration dependencies ignoring any function
  // calls (since they were processed in (i))
  Call doDependAnalForNonScalarGrids(M, F, S)
  Call analyzeConstantIndices(M, F, S)
  Call doScalarDependAnalysis(CFG)
  Call estimateOverallParallelization(M, F, S)

Function addGrid(G: <grid to be added>)
If G scalar and entry for G does not exist in ScalarGridInstances[M][F][S], add entry E
for G to ScalarGridInstances[M][F][S] or if G non-scalar do so for
NonScalarGridInstances[M][F][S]
Initialize/update E to indicate whether G is read/written and record attributes of G (e.g.,
box id, box type where G is found if scalar, or dimension read/written and the specific
index (location) from/to which each dimension is read/written if non-scalar)

Function addNonScalarForFunc(NS: <non-scalar grid to be added>, M: <module ID>,
F: <function ID>)
If NS is in NonScalarGridsInFunc[M][F], update information of corresponding element
E named after NS as read/written in NonScalarGridsInFunc[M][F] and store current
instance (i.e., index) for each of the dimensions
```

(a)

Figure 5.16: Parallelism back-end pseudocode (related internal objects used are described in Figure 5.8)

```
Function doDependAnalForNonScalarGrids (M: <module ID>, F: <function ID>,
S: <step ID>)
gridsInstances = NonScalarGridInstances  or NonScalarGridsInFunc  (depending if
called from MainFunc or not)
For all elements E of gridsInstances[M][F][S] whose grid G is written
   For each dimension D of E
      For all combinations of write instances I1,I2 in D
         If I1,I2 write locations are different and contain index variable I from
         IndVarsWrittenInStep[M][F][S], mark I as non-parallelizable in G
      For each combination of write & read instances I1, I2
         If I1,I2 write, read locations are different contain index variable I from
         IndVarsWrittenInStep[M][F][S], mark I as non-parallelizable in G

Function parseRightHandSide(RHS_expr: <expression appearing as right-hand side of
assignment)
For each element E in RHS_expr, call addGrid(G) if E is a grid G or add F call's
information to FuncsFromCaller[M][F][S], if E is a function call

Function recordGridsAndIndicesOfStepBoxes(M: <module ID>, F: <function ID>)
For each step S in function F, record (in NonScalarGridsInFunc[M][F]) each grid G
that appears in boxes of S, if G is an incoming argument to F. For each function-call
foo(args) in S that passes G as an argument, record  foo and its relevant info in
FuncsFromCaller[M][f], and call this method recursively.

Function doDependAnalForNonScalarGridsPassedToAllFuncs(M: <module ID>,
F: <function ID>, S: <step ID>)
For each function-call FC in FuncsFromCaller[M][F][S] and for each non-scalar
grid G passed as an argument in FC's call
   Find corresponding name G of FC's argument GC in F's argument list
   If G present and read-only in S
      If GC read-only in FC, do nothing
      Else if GC is written or both read and written in FC
         Analyze all valid pairs of indices across all dimensions D that G is read in
         F and GC written (or written and read) in FC and if indices differ and
         contain index variable I from IndVarsWritenInStep[M][F][S], stop checking
         for I and mark I on G as non-parallelizable
   Else if G present and written (or written and read) in S and written or read (or
   written and read) in FC
      Analyze all valid pairs of indices across all dimensions D that G is written (or
      written and read) in FC and written (or written and read) in FC and if indices
      differ and contain index variable I from IndVarsWrittenInStep[M][F][S], stop
      checking for I and mark I on G as non-parallelizable
   Else if G not present in current step of F
      Call nonScalarGridDependencyAnalysis(MC, FC, SC)
      Call analyzeConstantIndices(MC, FC, SC) for all steps SC of FC of MC
```

(b)

Figure 5.16: Parallelism back-end pseudocode (related internal objects used are described in Figure 5.8) (cont.)

```
Function estimateOverallParallelization (M: <module ID>, F: <function ID>,
S: <step ID>)
Check and mark if parallelism is broken for each index variable I in
IndVarsWrittenInStep[M][F][S], because of:
    ■ start/end/ step scalar variable for I being written in S
    ■ a scalar grid  (using info from scalar grid dependency analysis)
    ■ a non-scalar grid (using info from non-scalar grid dependency analysis)
For each function called from S of F of M, call findParallelismInFunction(M, F)

Function analyzeConstantIndices(M: <module ID>, F: <function ID>, S: <step ID>)
gridsInstances = NonScalarGridInstances or NonScalarGridsInFunc (depending if
called from MainFunc or not)
For each G in gridsInstances[M][F][S] that is written
   Check the indices of the instances where G is written and identify the cases where
   a dimension's index is always a constant as non-parallelizable

Function doScalarDependAnalysis(CFG: <control flow graph for a step>)
For each scalar grid occurrence X that is written at least once
   Push CFG root node in stack
   While stack not empty
      N = pop last element from stack
      If X is written in current node N
         If there is a read of X in N
            Identify write after read condition, reduction, break
         Else do nothing, break
      Else if X is not written in N
         If there is a read of X in N
            Identify write after read condition, reduction, break
         Else push each child of N to stack
    If reduction found, mark as a reduction clause
```

(c)

Figure 5.16: Parallelism back-end pseudocode (related internal objects used are described in Figure 5.8) (cont.)

in a *parallelism meter* on each step's header indicating the index variable name and number of parallelizable iterations. For (b), each dependency is recorded and GLAF can provide feedback about module name(s), function name(s), step(s) and line(s), as well as a description of the reason parallelization fails.

Auto-Tuning

Figure 5.17 shows the *auto-tuning menu screen*. It includes options regarding *source code, binaries, compilation and execution scripts generation*, as well as *execution times presentation* for the resulting implementations. At the **first level** the user selects the *target platform*. This option identifies the appropriate compilation flags to be passed to the generated platform-specific auto-tuning script. In the future, this option will allow *platform-specific* optimizations. The **second level** of auto-tuning concerns the *target language* (*Fortran*, *C*, or *OpenCL*). The **third level** offers three different potential basic implementations *within* a selected language and platform: (a) *serial:* serial code (C and Fortran only) as automatically generated by GLAF, (b) *GLAF-parallelized:* parallel implementation of the code in OpenCL, or in C/Fortran with appropriate OpenMP directives that precede parallelizable code sections, as identified by the auto-parallelization back-end, (c) *compiler-parallelized:* with this option serial code in (a) is to be compiled with the auto-parallel compiler flag (*-parallel*) - this flag is not used in (b) where the code includes explicit OpenMP pragmas and is compiled with the *-openmp* flag. The **fourth level** expands our choices in code generation. Options at this level can be *combined*. The first option showcases the flexibility in *expressing data structures*, and creates implementations where grids are expressed as either *Struc-*

tures of Arrays (SoA) or *Arrays of Structures* (AoS), if applicable (i.e., in the form of Figure 5.7a). Figure 5.18 shows code automatically generated in C for declaring, initializing and using such data structures. The second option, tunes the level of *loop collapsing* for nested parallelizable loops and the third concerns *loop interchange*, when applicable (do not apply in OpenCL code generation). Particular options from this category are more suitable for specific algorithmic patterns and their effect is reflected in the execution time of an application written in a specific way. We discuss such concerns in more detail in Section 5.3.4, where we present results forrepresentative cases. Users who are familiar with the options' meaning and implications may select to prune the auto-tuning space by deselecting certain options. Non-programmers may simply let GLAF generate code using all applicable permutations and present the best combination via experimental evaluation (and corresponding code). Figure 5.19 depicts the"write once - run anywhere" concept embedded in GLAF.

On selecting OpenCL for FPGA as a target language, an extra set of FPGA-specific optimizations are at the disposal of the user. Due to their importance, we describe them in more detail separately.

Generation of Optimized Code for Altera FPGAs

GLAF OpenCL code generation back-end performs certain FPGA-specific code optimizations and renders code amenable for further optimizations by the Altera Offline Compiler (AOC). Here, we describe such optimizations, while in the Section 5.3.5 we provide examples and discussthe effect of the most important ones. As with OpenCL code generation itself, generating code for the

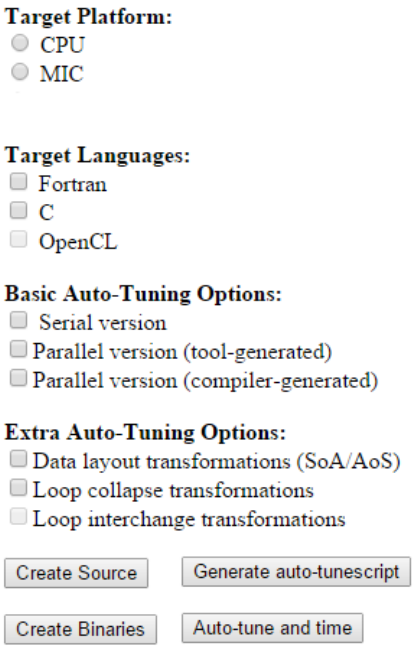


Figure 5.17: Auto-tuning options page

```
struct TYP_surface_pts {
  float dim0;
  float dim1;
  float dim2;
  float dim3;
};
// Surface points outside the biomolecule
struct TYP surface pts *typvar surface pts;
// Initialization of array of structures (AoS)
typvar_surface_pts = (struct TYP_surface_pts *)malloc(
                     sizeof(struct TYP_surface_pts)*3);
// Value assignment
typvar_surface_pts[ft_col].dim0 = ft_col * 3;
```

(a) Code for AoS

```
struct TYP_surface_pts {
  float *dim0;
  float *dim1;
  float *dim2;
  float *dim3;
};
// Surface points outside the biomolecule
struct TYP surface pts typvar surface pts;
// Allocation of each array within the struct (SoA)
typvar_surface_pts.dim0=(float *)malloc(sizeof(float )*3);
// Value assignment
typvar_surface_pts.dim0[ft_col] = ft_col * 3;
```

(b) Code for SoA

Figure 5.18: Generated code for declaration and accessing different data layouts

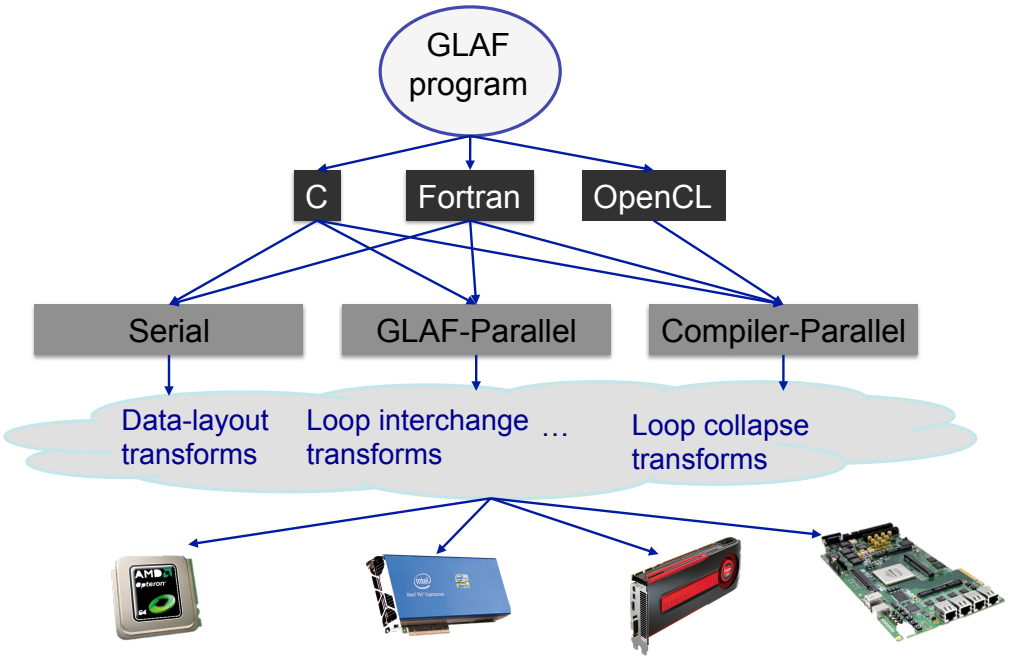


Figure 5.19: High-level overview of the "write once - run anywhere" concept in GLAF

optimizations described below is achieved by translating the appropriate information in the GLAF internal representation to the appropriate OpenCL language constructs and code.

Single Work-Item Kernels: In Section 5.3.2, we describe how GLAF generates OpenCL device code in the form of *NDRange kernel* (i.e., multiple work-groups and work-items). NDRange kernels represent an appropriate way of programming (data) parallel code regions in OpenCL for exploiting the multi/many-core capabilities of devices like CPUs and GPUs. In FPGAs, however, constructing a kernel as a *single work-item kernel* (equivalent to an OpenCL *task*) may be a more appropriate paradigm and yield better performance in some cases. Instead of assigning loop iterations to separate work-items, this method utilizes *loop pipelining*. Single work-item kernels are specially suited for task-parallel algorithms, or algorithms with data dependencies and synchronization, where NDRange is not applicable or sub-optimal. However, as we observe in

Section 5.3.5, they can also benefit highly parallelalgorithms, also by enabling further optimizations that are not applicable (or beneficial) otherwise. GLAF is able to automatically generate OpenCL code in the single work-item kernel paradigm if the user has selected the corresponding option in the optimizations menu. In this case, as opposed to the NDRange case (Figure 5.15), the generated host code defines a single-dimension, single-item *globalWorkSize[]* array. On the device code the kernel contains the loop itself (i.e., the parallel loop that was previously converted to NDRange) and the *get_global_id()* calls are not generated. On compilation of code generated in the above format with AOC, the compiler (AOC) identifies the kernel as a single work-item kernel and attempts to infer a loop pipeline.

Initiation Interval Reduction: Single work-item kernel compilation with AOC yields an optimization report that informs whether pipelined execution was inferred, what the *initiation interval* (II) between successive loop iterations is and, if possible, the reason. In GLAF OpenCL generated code single work-item execution for parallelizable loops leads to successful pipeline inference. In certain cases, however, II may be high. While, the range of such cases is too broad to conclusively address, we show how GLAF can automate *loop relaxation* [25]. These optimizations can drastically reduce or eliminate II for *reduction* operations within a single work-item kernel. An example is shown in Figure 5.20. GLAF parallelism analysis back-end identifies reductions and stores the reduction variable and operation in its internal representation. After compiling code, parsing the optimization report reveals the II value and whether it is reduction-induced. In this case, GLAF can automatically generate *M* copies of the reduction variable (line 1, Figure 5.20b) by generating the declaration (by appending the *_copies[M]* to the reduction variable name). Subsequently, it

generates code (lines 2-4) for initialization of this variable according to the reduction operation (e.g., zero for addition). The main computational loop now is transformed to a temporary variable in which we store the reduction operation on the last reduction variable copy (line 6), a loop that shifts all copies by one position (lines 7-10), and code for storing the temporary variable to the first copy (line 11). Finally, code for reduction on the reduction variable copies is generated outside the reduction loop (lines 13-16) and the result is assigned to the original reduction variable (line 17). This method relaxes the dependencies and reduces II. In general, it can be observed from the example in Figure 5.20 that the code that is auto-generated follows a specific template. The key change according to a specific problem lies on substituting A[i] with the corresponding computation of the reduction at hand. For example, in one of our examples (time-domain FIR filtering) A[i] is substituted with the convolution between input elements and the filter. The number of copies M is the important factor in reducing II. Different values can be attempted manually (by changing a simple *#define*) – the whole process has the potential to be trivially automated through a script in a feedback loop with AOC compilation and optimization report parsing.

Shift Register Inference: *Sliding window* computation [21] is a common pattern (e.g., filters) that can benefit from a single work-item kernel design. This pattern includes a loop that accesses a fixed number of contiguous locations in an array shifted by one position per iteration. Such sliding-window memory access patterns can benefit from using *shift registers*. For the AOC to infer a shift register implementation code has to be written in a certain, counter-intuitive from a software development standpoint, way. The resulting implementation is very similar to the method used for enhancing II in single work-item kernels: declaration and initialization of the shift register to a zero

```
1  float sum = 0;
2  for (i = 0; i < N; i++) {
3        sum += A[i];
4  }
5  result[idx] = sum;
```

(a) Original Code.

```
1  float sum_copies[M];

2  for (i = 0; i < M; i++) {
3        sum_copies[i] = 0;
4  }

5  for (i = 0; i < N; i++) {

6        float cur = sum_copies[M-1] + A[i];

7        #pragma unroll M-1
8        for (j = M-1; j > 0; j--) {
9              sum_copies[j] = sum_copies[j-1];
10       }

11       sum_copies[0] = cur;

12 }

13 #pragma unroll M
14 for (i = 0; i < M; i++) {
15       sum += sum_copies[i];
16 }

17 result[idx] = sum;
```

(b) Code after II Reduction.

Figure 5.20: Initiation Interval (II) optimization

value, a fully unrolled shifting loop that includes shifting contents across neighboring elements except to the first (or last) that gets its value from the original input array. Last step entails replacing the original input array accesses with shift register accesses. Since sliding-window algorithms have a fixed number of iterations (usually small) the above optimization is coupled with full loop unrolling. What is challenging, and currently limiting its practical implementation within GLAF, is automatically identifying sliding-window patterns in applications. We see two examples in more detail in Section 5.3.5.

Kernel Vectorization (SIMD) / Multiple Compute Units (CU): *Kernel vectorization* enables work-items (in NDRange kernels) to execute in a SIMD-like fashion, thus increasing throughput. A desirable potential side effect of kernel vectorization is static memory coalescing automatically performed by AOC. *Compute Unit replication* helps achieving higher throughput by generating multiple copies of a CU for a kernel, but increases global memory traffic. Generally, between the two, kernel vectorization is more efficient resource-usage-wise but the trade-offs may not always be straightforward. GLAF OpenCL code generation back-end can generate multiple code implementations to be compiled and evaluated. Specifically, the GLAF OpenCL code generation back-end generates code that annotates a kernel with the corresponding attributes (*__attribute__*): *num_simd_work_items(N)* for kernel vectorization with vectors of length *N*, and *num_compute_units(N)* for *N* compute units. One limitation of kernel vectorization is that *N* must evenly divide the available work (or *work-group size*).

Restrict Clause: Visual programming via GLAF entirely hides the concept of pointers from users and *aliasing* issues are de facto not applicable. As such, all pointers in auto-generated code are fur-

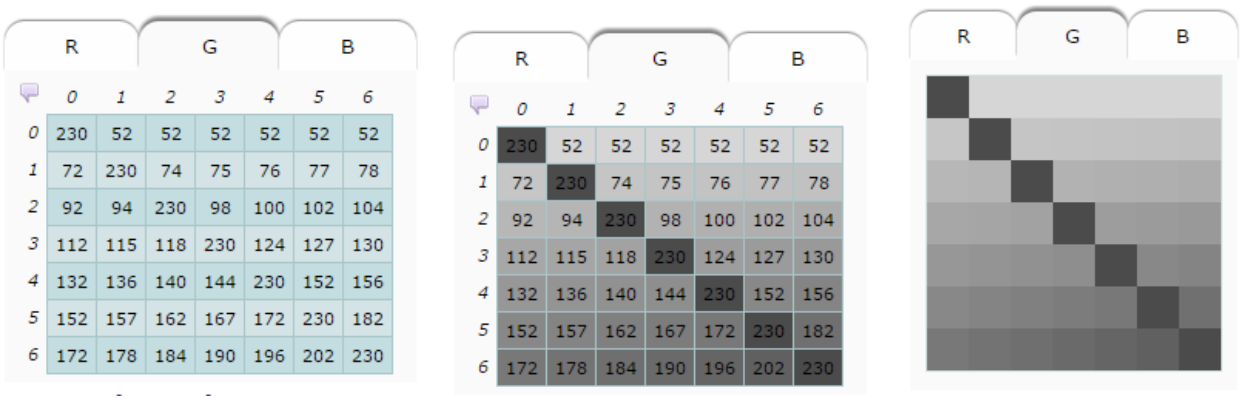
ther annotated with the *restrict* keyword in the function header (see kernel header in Figure 5.15). This eliminates unnecessary assumed memory dependencies and leads to more efficient designs, in terms of area and performance.

Constant cache memory: Declaring kernel pointers for data that are read-only throughout kernel execution as *__constant* enables loading into an on-chip cache optimized for hit performance. Constant memory is particularly useful for high-bandwidth table look-ups. In GLAF OpenCL we can keep track of read-only grids in a kernel (by conservatively analyzing the read/write locations in the code, using the GLAF internal representation of the code - information already obtained during parallelism analysis). If data can fit in cache (detectable for static grid sizes by inspecting the data type and dimension sizes elements of the grid object) the generated code for the corresponding kernel pointers is annotated as *__constant*. This method has the inherent limitation that grids with dynamic sizes (i.e., unknown at compile time) cannot take advantage of this optimization.

Memory Alignment: Aligned memory allocation of the host-side buffers enables direct memory access (DMA) transfers that can be considerably faster than data transfers between the host CPU and FPGA from/to unaligned memory. GLAF OpenCL auto-generated code ensures that all memory allocations follow the board-specific alignment requirements. Specifically, instead of the default *malloc()* call in GLAF, GLAF generates *alignedMalloc()* calls in the host code and the implementation of this function in the header file (.h) that is effectively a wrapper of *posix_memalign()*.

Visualization

Data visualization facilitates understanding the algorithms being developed, as well as revealing bugs, in an *intuitive, visual way*. Currently, GLAF supports *two* ways of visualizing data. By selecting the *"Show Data"* menu item in a step, data for each grid cell are calculated up to that step by evaluating automatically generated JavaScript code on-the-fly and using the internal representation of data structures, which is by design in JavaScript (Section 5.3.1). The user can navigate in multi-dimensional grids by clicking on the appropriate dimension. For grids whose dimensions are larger than the screen space, appropriate arrows enable navigating within grid contents. The *"Colorize"* menu option paints each grid cell on the greyscale color spectrum according to the magnitude of the corresponding cell value, together with its value. Finally, the *"Image Map"* option, a straightforwad extension of the previous two, contains only color (i.e., no data values). Techniques based on color make it easy to spot *outlier* values or specific *patterns* in relative problems, especially in the presence of huge amounts of data, thus facilitating result observation and interpretation. They are especially useful in the image processing field, as well. Figure 5.21 shows three simple examples of data visualization functionalities. In the future, more *complex* visualization schemes can be built upon the unified and regular internal representation of the grid abstraction by use of specialized visualization libraries for frequently-used data structures. For example, sparse matrices in CSR format could be visualized automatically as single (sparse) matrices, rather than the collection of the helping 1D arrays that represent such matrices in CSR format. Similarly, graphs could be visualized as such, while internally represented via adjacency matrices.



(a) Values only. (b) Values and colorization. (c) Image map.

Figure 5.21: Examples of data visualization methods in GLAF

5.3.3 Example Applications

To illustrate the features of GLAF (code generation, parallelism analysis, auto-tuning) and assess its utility as a high-level programming environment in diverse situations, we present our experiences and results for a set of example applications that span four different scientific domains (imaging, physics, bioinformatics, signal processing) that exhibit various types of parallelism, computation and memory access patterns. Using some of the above example applications we illustrate how easy it is for novice programmers using typical programming languages to fall into *coding pitfalls*, their effect in performance, and how GLAF addresses these problems in an automated way thus providing a fair trade-off between performance and programmability, while ensuring portability, too. Similarly, some of the applications are specifically chosen to showcase certain features of the GLAF Altera FPGA OpenCL-specific optimization opportunities and their effect in performance.

3D Finite Difference Calculation (3DFD)

Finite difference calculation represents structured grids algorithms, where computation proceeds as a series of grid update steps. In the 3D case data is arranged in a regular 3D grid. For each step iteration each point is updated as a function of its neighboring points' values across each of the 3 dimensions. In our example each point for step $n + 1$ is the value at step n plus the sum of 8 neighboring points across each dimension (Listing 5.1). We define a source and destination grid, which are used interchangeably at each computation iteration. In 3DFD, we have ample parallelism within a step iteration: within each dimension summation of all 8 neighboring values for each point can be seen as a reduction and on the outer level (3 nested *for* loops) each grid cell update can be calculated independently.

for number of steps/iterations S
 for each $point[i, j, k]$
 (incl. vertical boundary condition)
  $sum[i, j, k] += 8pt\ stencil\ across\ horizontal\ dim$
 for each $point[i, j, k]$
 (incl. horizontal boundary condition)
  $sum[i, j, k] += 8pt\ stencil\ across\ vertical\ dim$
 for each $point[i, j, k]$
 (incl. 3rd dim. boundary condition)
  $sum[i, j, k] += 8pt\ stencil\ across\ 3rd\ dim$

Listing 5.1: 3DFD pseudocode

Electrostatic Surface Potential Calculation (NB)

In NB we calculate the electrostatic potential at a collection of points on the surface of a biomolecule. The potential at each point is the sum of charges contributed by its interaction with all atoms within the biomolecule (Listing 5.2). Each surface point and atom is represented as a structure containing information about its electric charge and its coordinates in the 3D space (Figure 5.7a). In NB we have *two levels* of parallelization opportunities for the two nested loops described above (the latter being a *reduction*). Computations are *independent* for each surface point/surface point-atom pair.

for all surface points $sp[i]$
 for all atoms $at[j]$
  $sq\_dist = (x[i] - x[j])^2 + (y[i] - y[j])^2 + (z[i] - z[j])^2$
  $sum\_esp[i] += K_e * (q[i] * q[j])/sq\_dist$

Listing 5.2: NB pseudocode

Sequence Search (SS)

Sequence search is a fundamental process in bioinformatics and computational biology. Informally, it can be defined as finding similarities between a query sequence (i.e., an unknown string of DNA bases or aminoacids) and a subject sequence (i.e., a sequence in the database that may have known origin and functionality). Identifying similarities between DNA or protein sequences serves as a proxy for detecting similarities in function and structure between sequences, thus providing clues on the evolutionary history and origin of unknown sequences, for example. Other uses include phylogenetic profiling and identifying members of gene families. In SS we have parallelism available at two levels; (a) every n-mer (where *n* is the number of elements in the search sequence)

of the reference sequence can be compared in parallel with the search sequence, (b) within each reference sequence n-mer and search sequence, element-level pair-wise scoring can take place in parallel.

for each position i of reference sequence ref_seq
  for each j position fo search sequence $search\_seq$
    $pair\_score[i] += scoring\_matrix[search\_seq[j],\ ref\_seq[i+j]]$

Listing 5.3: SS pseudocode

Time-Domain Finite Impulse Response (FIR) Filter (FF)

The FIR filter is the basic building block in many larger algorithms in the area of signal processing (video, communications, etc.) Time-domain FIR filter represents a sliding-window type of algorithm that, as its name denotes, implements a set of *M* FIR filters, each with *K* filter coefficients. A filters output is calculated as the convolution of its coefficients (complex number that consists of a real and an imaginary part) and the input vector. In terms of parallelism available, there are two opportunities, that correspond to each of the for loops. The outermost loop is parallelizable, as there are no cross-loop iteration dependencies. Within the second loop there are two sum reduction operations.

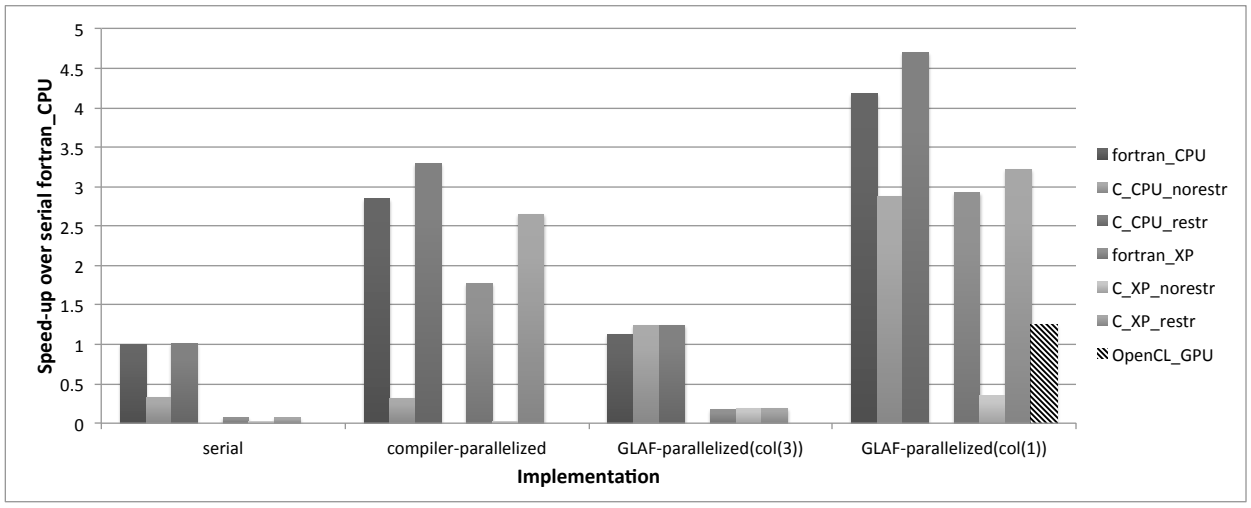
for each element i of (combined) input ($inp\_real[]$ and $inp\_img[]$)
  for each element j of filter ($flt\_real[]$ and $flt\_img[]$)
    $out\_real[i] += inp\_real[i-j+flt\_length-1] * flt\_real[j] - inp\_img[i-j+flt\_length-1] * flt\_img[j]$
    $out\_img[i] += inp\_real[i-j+flt\_length-1] * flt\_img[j] - inp\_img[i-j+flt\_length-1] * flt\_real[j]$

Listing 5.4: FF pseudocode

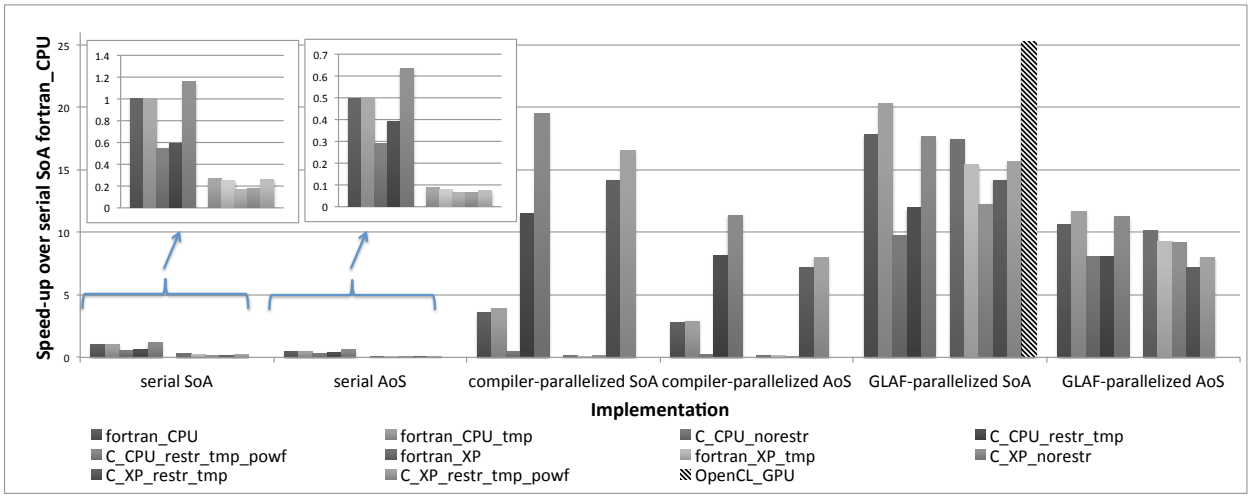
5.3.4 Evaluation: Fixed Target Architectures

In this section, we evaluate code automatically generated by GLAF for fixed target architectures, that is the CPU Intel Xeon Phi and GPU. Specifically, our experiments are run on: (a) a dual-socket **Intel Xeon E5-2697 CPU** (each with 12 cores at 2.6GHz, 256KB of L2 and 30MB of L3 cache) , (b) an **Intel Xeon Phi (XP) 7120** co-processor (61 cores at 1.238GHz, 30.5MB cache), and (c) an **NVIDIA K20c** GPU (13 Streaming Multiprocessors at 706MHz). We used the Intel set of compilers (ifort for Fortran, icc for C, in Intel Composer XE 2015 v.15.0.1), Intel Vtune Performance Analyzer and compiler optimization/parallelization reports to identify performance issues. For the OpenCL code we use the NVIDIA SDK's OpenCL 1.2 CUDA 8.0.46.

In our example applications, we generate *serial*, *compiler-parallelized* and *GLAF-parallelized*, implementations (see Chapter 5.3.2 for details) for both Fortran and C, with appropriate optimizations applied by the auto-tuning back-end. OpenCL is de facto parallel, so we only provide *GLAF-parallelized* implementations (serial and compiler-parallelized implementations are not applicable). For explicit performance comparisons graphs (Figure 5.22) for our experiments show speed-up (y-axis) over a single baseline implementation (Fortran CPU serial) for the most relevant implementations (x-axis) as created by the auto-tuning framework. The serial baseline code highly resembles code that a novice programmer would write (e.g., Figure 5.10).



(a) 3DFD results



(b) NB results

Figure 5.22: Performance results for the example applications developed, auto-tuned by the GLAF framework

3D Finite Difference Calculation (3DFD)

For the 3DFD experiment we run the algorithm for 20 steps and a 512x512x512 input grid. There are no *data layout* concerns (i.e., SoA, AoS) in 3DFD, however, *loop collapsing* and *loop interchange* auto-tuning options are applicable in C and Fortran, but not in OpenCL. For the former, we present results for *collapse(1)* (i.e., no collapsing), and *collapse(3)*. For the latter, we auto-generate the fastest loop ordering, according to the target language's way of arranging multi-dimensional arrays in memory (*row-major* for C, *column-major* for Fortran).

The best performance is obtained using the C_CPU_restr GLAF-parallelized(col(1)) implementation (4.7x), followed by the corresponding implementation in Fortran (4.17x). Despite this performance difference *across* languages, *within* each language and platform, C achieves *better* speed-up (over serial) than Fortran (e.g., 38.37x for compiler-parallelized C_XP_restr versus 24.64x Fortran_XP). Moreover, a non-negligible 1.49x performance gain (on XP, compiler-parallelized implementation) can be obtained by switching from Fortran to C. GLAF renders this trivial by its ability to automatically generate code in both languages. OpenCL auto-generated code for GPU only provides a mediocre speed-up of 1.26x. The reason for the above lies in two factors. First, the nature of the algorithm requires cross-step synchronization. Due to the lack of explicit device-wide synchronization in the GPU, multiple kernel launches are required. This introduces a significant overhead in the GPU, which is aggravated by the fact that the computation is carried out in three separate kernels (one for each of the three dimensions). Second, the unique memory hierarchy of the GPUs that lacks large hardware-controlled caches, combined with the specific memory access

patterns of two of the three kernels (i.e., beyond the kernel that computes along the horizontal dimension) proves non-ideal (accesses are served by global memory and cannot be coalesced or efficiently cached). In this case, use of the GPU shared memory would be an important manual optimization (or part of an optimizing GPU back-end in a future version of GLAF).

In Figure 5.22a, we observe that C_CPU_restr and C_XP_restr perform overall better than their *norestr* counterparts in all implementations. *Norestr* corresponds to an earlier stage of C code generation in GLAF. In the case of C *norestr* implementations, we observe that compiler-parallelized code fails to provide any speed-up versus the corresponding serial. C enforces strict aliasing rules, and as such the compiler acts conservatively by not parallelizing the three nested loops (Listing 5.1) and assuming the existence of dependence between the pointer variables for each grid in a function call. To indicate no aliasing a C programmer would need to use the *restrict* keyword and *-restrict* compiler flag. In Fortran, grids (i.e., Fortran arrays) are always passed by reference and are assumed to not alias by default. Strict aliasing rules, when unnecessarily enforced, do *not* only affect auto-parallelization, but vectorization, too. Namely, it seriously hinders serial C performance, as well. Note the huge difference (*restr* vs. *norestr*) in XP (16.04x), as opposed to 3.07x in CPU, owing to the wider vector units in XP compared to the CPU. In the case of OpenCL, the compiler-parallelized case where the *restr* and *norestr* cases are important is not applicable, as OpenCL explicitly exposes parallelism. Novice programmers are typically *not* aware of issues like aliasing and related solutions. This is another point in favor of automatic code generation by GLAF, doing so in multiple languages, as well as GLAF's simplicity in not allowing aliasing by

default. As such, the above remedies can be taken care of by the framework at code generation and compilation script creation time.

In some cases with C and Fortran, where nested loops are parallelizable, it may make sense to collapse them. A novice programmer with rudimentary OpenMP knowledge and lack of architecture knowledge could have coded these nested loops with nested OpenMP pragmas, spawning an increasing number of threads with the associated overheads, or with a *collapse(3)* OpenMP clause. Using *collapse(1)* (i.e., parallelizing only across one dimension) allows the compiler to generate *vector code* for the remaining two (parallelizable with *unit stride*) loops. This increases performance as a function of the vector unit's width. As such the relevant performance gain between those two options is higher (16.8x) in the case of Xeon Phi (512-bit wide vector units), than in the lower (3.7x) CPU case (256-bit AVX) in *both* Fortran and C implementations.

Electrostatic Surface Potential Calculation (NB)

For NB we use a problem size of 128000 surface points and 256000 atoms. NB illustrates the utility of GLAF's *data layout transformations*. *Structure-of-arrays* and *array-of-structures* implementations can be automatically generated by GLAF for the surface point and atom grids (Figure 5.18).

The best overall performance for NB is obtained using the OpenCL GPU implementation, which is about 25% faster than the second best. NB, as an n-body dwarf, provides abundant parallelism in a regular form that is amenable for parallelization using the GPU (as we also see in Section 4.2.5). For the x86-based cases of CPU and Xeon Phi, the fast performance is obtained by the Fortran

GLAF-parallelized SoA implementation in the CPU, followed by the corresponding C CPU implementation. The fact that the language and platform of choice (OpenCL and GPU) for NB is *different* than that of 3DFD emphasizes the utility of automatic code generation in multiple target languages/platforms. In many instances Xeon Phi execution proves slower than the corresponding CPU C and Fortran implementations. We expect to be able to achieve better speed-ups for the Xeon Phi case, and even better for the GPU, once GLAF provides MIC and GPU architecture-aware optimizations, as well as specific hints to the compiler (e.g., alignment attributes that facilitate more efficient vectorization, aligned loads, cache miss reduction in Xeon Phi).

As in the case of 3DFD we observe compiler-parallelized C implementations (*norestr*) to initially exhibit subpar performance, equal to the serial. According to the compiler parallelization report the reason for failing to parallelize the outer loop was "insufficient work". This was an erroneous assessment by the compiler, given the available amount of work. In fact, the 2013 (v.13.1.3) version of icc *did* parallelize the same loop when the number of loop iterations was larger than the number of (logical) processors. The latest compiler was eventually able to automatically parallelize this loop once we performed (via GLAF) another optimization that the compiler was not performing itself. In particular (*restr_tmp*), we introduce a temporary variable in the nested loop of Listing 5.2 in place of the *sum_[i]* array, which further allows the loop to be identified as a reduction (C does not otherwise allow reductions on dynamically allocated arrays). GLAF applies the same code transformation for Fortran.

NB is an illustrative case for the utility of *data layout* transformations. Results highlight how a bad choice for structs declaration (AoS here) account for twice worse performance. SoA layouts

are amenable to efficient vectorization and offer the possibility for unmasked *unit stride* loads, as opposed to more expensive *strided* loads, and *gather* operations employed with AoS. Profiling data across our implementations validate AoS detrimental effect, especially on cache, as CPU and XP speed-ups denote. Especially for the serial fast implementations performance degradation ranges between 1.52 to 2x (C, Fortran in CPU) and 2.89 to 3.15x (C, Fortran in XP). In SoA cases, all vector loads in Fortran are aligned, whereas in C the majority is unaligned, resulting in higher load latencies. In both SoA and AoS CPU cases C is better in its compiler-parallelized and worse in the GLAF-parallelized implementations, while in XP it far surpasses the corresponding Fortran compiler-parallelized implementations and performs similarly for the GLAF-parallelized. In the case of GPU execution, OpenCL code generation back-end automatically converts code in the SoA format (more specifically it generates multiple arrays, as discussed in Section 5.3.2). While GLAF provides the answer to what the best implementation is, we can imagine the pitfalls a novice user would fall into if he were to write a serial version in the "wrong" language and/or in the "wrong" struct format (i.e., failure of compiler auto-parallelization).

Finally, C implementations with *powf* suffix (Figure 5.22b) highlight how a simple oversight can affect performance. In particular, our code generation back-end initially produced a *pow()* function call for the corresponding function in GLAF. While in Fortran the exponentiation intrinsic function (**) is overloaded with the appropriate version according to the arguments, in C *pow()* emits the *double* version. Ignoring calling appropriate versions of a function in C is common among non-programmers. Use of *powf()* instead of *pow()* increased C serial CPU performance by 1.94x for the SoA and 1.61x for the AoS case. Argument type detection is inherent within the frame-

work (Section 5.3.1) and detection of such cases of potential unwanted performance degradation is important in steering users away from such pitfalls.

5.3.5 Evaluation: Reconfigurable Target Architectures

For the FPGA implementations we use a *Bittware S5-PCIe-HQ* board (S5PHQ-D8) that comes in the form of a half-length PCIe x8 card. It is based around a high-performance Altera Stratix V GS FPGA and contains 16 GB of DDR3 SDRAM arranged in two 64-bit banks. The OpenCL kernel codes were compiled using the Altera OpenCL SDK v14.2 tool-chain. For the CPU reference base-line implementations we use a *Intel E5-2697* (Ivybridge) with 12 cores (24 threads), clocked at 2.7GHz, AVX support and 30MB of L3 cache. The CPU (parallel) implementations (in C with OpenMP directives), as well as the host-side code of the OpenCL implementations, were compiled using gcc v.4.8.2 and run on a Debian host (kernel 3.2.46) with 64GB RAM.

Figures 5.23a-5.23c show the **execution time** of OpenCL kernel implementations normalized to the corresponding execution time of the OpenMP-parallel CPU implementation. Both the CPU and FPGA implementations are generated by GLAF to ensure a certain level of fairness of comparisons. We also show the FPGA **resource utilization** to obtain insights on the effect of various optimizations on it and trade-offs between resource utilization and performance. The characteristics of alternative implementations for each example application are outlined in Table 5.2. Some implementations are shown for the purpose of quantifying the effect of certain optimizations, while most are implementations automatically generated by GLAF. Our experiments focus on the FPGA and

Table 5.2: Kernel implementations

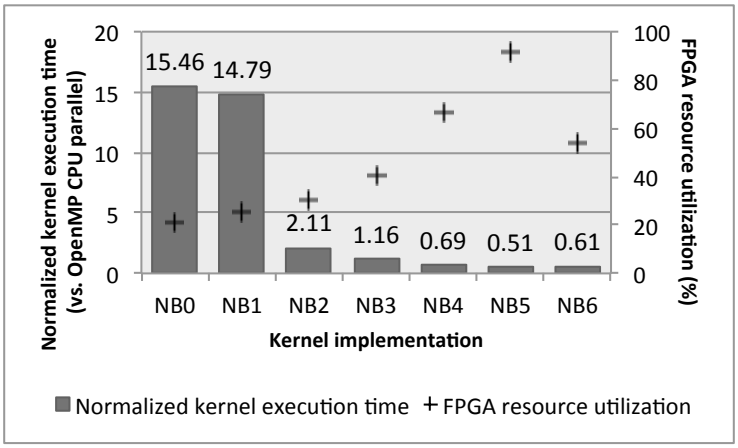
Implem.	Type	CUs	SIMD	Const. mem.	Kernel Freq.
NB0	NDR	1	1	N	268.95
NB1	SWI	1	1	N	280.58
NB2	NDR	1	8	N	244.91
NB3	NDR	1	16	N	223.01
NB4	NDR	2	16	N	190.73
NB5	NDR	3	16	N	193.19
NB6†	NDR	1	1	N	215.33
SS0	NDR	4	8	N	183.95
SS1	NDR	2	16	N	184.16
SS2	NDR	1	1	Y	144.3
SS3	NDR	6	16	Y	153.04
SS4⋆	NDR	1	8	N	186.7
SS5*	SWI	1	1	Y	118.35
FF0	NDR	4	16	Y	183.89
FF1	SWI	1	1	Y	262.61
FF2	SWI	10	1	Y	195.65
FF3‡	SWI	1	1	Y	191.97
FF4‡	SWI	10	1	Y	170.12
FF5*	SWI	1	1	Y	188.46

† Resource-driven optimized ‡ Initiation interval reduction
* Shift register inference ⋆ Inner loop unrolling
NDR: NDRange **SWI:** Single work-item

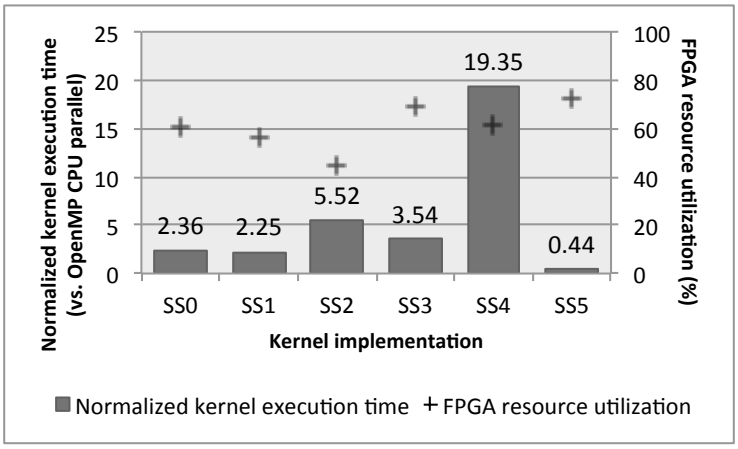
performance improvements as achieved through optimizations automatically applied by GLAF. Normalization of performance with respect to multi-core CPU OpenMP performance is used as a reference, but further cross-language analysis (e.g., HDL on FPGA) is beyond the scope of this work.

Electrostatic Surface Potential Calculation (NB)

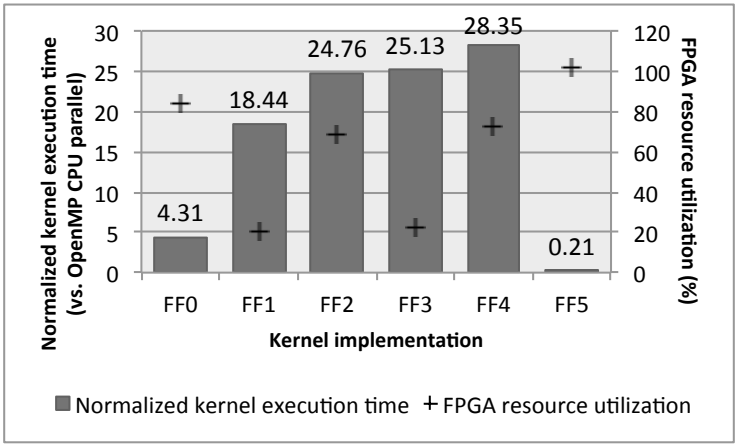
NB is a highly parallel application and provides insight about the optimizations of kernel vectorization (SIMD), compute unit (CU) replication, NDRange (*NDR*) versus single work-item (*SWI*), and the effectiveness or resource-driven optimizations by the Altera Offline Compiler (AOC).



(a) Electrostatic Surface Potential Calculation (NB).



(b) Sequence Search (SS).



(c) TDFIR (FF).

Figure 5.23: Results: Execution time and FPGA resource utilization (lower is better)

As seen in Figure 5.23a, increasing SIMD lanes from 1 to 8 (NB0, NB2) and doubling SIMD from 8 to 16 (NB2, NB3) yields a 7.32- and 1.81-fold speed-up, respectively. With each doubling in SIMD vector length resource utilization increases by about 1.35x. Memory access patterns of NB make it amenable for kernel vectorization, as observed in the profiling data: memory accesses are coalesced and result to cache hits in over 99% of the time minimizing memory-related pipeline stalls below 4%. Increased SIMD length only leads to increase of the datapath of a CU (all SIMD lanes share control logic). CU replication case differs, as seen in NB3, NB4, NB5 (SIMD length kept constant): performance increases by a 1.68x when increasing the number of CUs from one to two, while tripling the number of CUs yields a 2.27x increase in performance. Increasing the number of CUs comes at the expense of increased global memory bandwidth across CUs and each doubling of CUs leads to resource utilization almost doubling, too.

Comparing NDRange and single work-item kernel, we observe (NB0, NB1) that performance and resource utilization are almost similar. This is expected, since both NDR and SWI kernels have no kernel vectorization or CU replication and are expressed via pipeline parallelism in FPGA hardware. In fact, the execution time ratio (t_{NB0}/t_{NB1}) equals the inverse kernel frequencies ratio (f_{NB1}/f_{NB0}). As we see in other example applications (SS, FF), SWI can be beneficial over NDR after applying further optimizations that are not applicable in the NDR paradigm.

Last, NB5 and NB6 provide insight on the effectiveness of resource-driven optimization by AOC. Specifically, in NB6 we use this feature: AOC compiles a kernel with attributes (SIMD length, number of CUs, loop unrolling) based on estimated throughput derived using heuristics. In NB6, AOC identifies loop unrolling (by a factor of 32) to be the most beneficial optimization. Our (brute-

force) choice (SIMD 16, CU 3), which provides a 1.2x speed-up over resource-driven optimization, indicates certain limitations of the latter.

Sequence Search (SS)

SS serves as an example of the trade-offs in combinations of SIMD length, number of CUs and loop unrolling, use of constant memory, as well as shift register inference in single work-item implementations.

With respect to SIMD, CU and loop unrolling, we compare versions SS0, SS1, SS4 (Figure 5.23b) that yield an overall 32-way parallelism (e.g., SIMD 16 with 2 CUs, or SIMD 8 with 4 CUs). Using wider SIMD (SS1) requires less hardware resources than SS0 and SS4 and provides speed-up over SS0, as expected for similar reasons with NB (e.g., more efficient hardware, coalescing). In SS4, enforcing 4-way loop unrolling together with SIMD only illustrates that careless combination of SIMD and loop unrolling without taking memory access paterns into account can be detrimental for performance (high cache misses and lengthy pipeline stalls).

For small search sequences, when applicable (as in our case), use of constant cache memory may present a considerable advantage for an FPGA design, that is better resource utilization that may allow wider SIMD or more CUs to fit in an FPGA (e.g., SS0 and SS1 versus SS3, Table 5.2). In SS, due to the parallelization scheme and memory access pattern (i.e., each thread accesses contiguous parts of the reference sequence array shifted by one position) we may have an unfavorable partitioning of the problem to CUs. While the details of scheduling are transparent to the programmer,

bandwidth efficiency (i.e., percentage of data acquired from global memory system that the kernel actually uses) is indicative of such an unfavorable partitioning (about 84% in SS3, 95% in SS0 and 99% in SS1). Notice that the bandwidth efficiency increases as the number of CUs decreases.

SS5 illustrates the SWI optimization, which pertains to the sliding-window access pattern of the reference sequence array. This access pattern is ideal for the shift register inference optimization, in which the OpenCL code follows the guidelines (static size, full unrolling) that allow AOC to generate a shift register structure that is placed into block RAM and is considerably faster than global memory accesses. It is worth noting that without the constant memory optimization the FPGA hardware resources would not suffice for full loop unrolling (and hence successful shift register inference). Also, a search sequence longer than 128 bases would lead to insufficient hardware resources. The fact that the search sequence needs to be small and statically determined to allow the shift register optimization is an inherent limitation of programming hardware. Finally, we observe that the clock frequency of the single work-item implementation is the lowest in this set, but in no way affects the end performance.

Time-domain FIR filter (FF)

Some of the optimizations discussed in Section 5.3.2 and found in NB and SS are also relevant for FF. For example, all shown FF implementations utilize constant cache memory for the filter coefficients. Notably, FF serves as an example where observed results can be counter-intuitive.

Here, the fastest SWI implementation, with shift register inference and loop unrolling (FF5) is 20.5 times faster than the fastest NDR one we were able to compile. FF highlights yet again the importance of the above optimizations in sliding-window type of algorithms. Notice that FF5, barely fits the FPGA for our 128-tap filter example. Without use of constant memory – and its lighter resource consumption, FF5 would not be possible. It is also worth noting that without the shift register inference optimization full loop unrolling cannot be applied at all, due to resource restrictions. The above observations highlight the importance of applying FPGA-specific optimizations *in concert*, rather in isolation only. GLAF is useful in this respect, in that it automates code generation with multiple combinations of optimizations that the user can evaluate.

Conversely, optimizations that may be beneficial for a specific computation pattern can be detrimental for another. While in NB we observe the positive effect of wider SIMD and more CUs, in FF this is not the case (FF1 is 1.34x faster than FF2, despite having 1/10 of the CUs). Accordingly, higher resource utilization does not necessarily imply better performance. Last, the initiation interval (II) reduction optimization in FF3 (reduces II from 8 cycles to 1) yields worse performance than FF1. Despite the reduction in the number of cycles between iterations in this particular case the resulting clock frequency for the design is 1.36 times slower than FF1 (as is the speed-up of FF1 over FF3).

5.3.6 Discussion

In this discussion section we address some of the potential questions and concerns the reader may have.

"A good programmer would definitely be able to get better performance." While we achieve good speed-ups and our auto-tuning choices include typical optimizations (and will include more in the future), we do not expect to obtain *ninja-programmer* results at this stage. A hand-tuned implementation by an experienced programmer can likely beat GLAF (in fact any similar tool), because it has the inherent limitation (or feature, depending on one's perspective) of requiring certain generality of applicability. Moreover, GLAF generates code and attempts to optimize the *specific* algorithm the user develops (using the GUI). While an experienced user is able to perform *algorithmic refactoring* for a certain platform, this is something that tools cannot currently do (machine learning approaches aspire to do so in the future). The gap between "ninja" performance and "good" performance is relatively small compared to the effort it requires [245]. Reaching such performance levels would require knowledge of the algorithm, which in turn would require complex manual hints on the part of the user, which we try to avoid. Our focus is on hitting a balance between performance and programmability. GLAF's ability to generate code in more than one languages (C, Fortran, OpenCL) is a very useful feature, nonetheless. A language's features and/or available compilers may be more suitable/perform better for a given algorithm "out of the box" or using our auto-tuning back-end, than a "ninja-optimized" implementation in a language that is not suitable. In most cases, "ninjas" are typically such in a certain language and GLAF-

generated code in their language of expertise can constitute a starting point saving them precious development time to the end solution.

"Compilers can auto-parallelize and optimize code. Why perform parallelism analysis and transformations within the tool?" It is important to stress that GLAF is not a compiler per se. It performs parallelism analysis and compiler-like optimization at code generation level, but at the end all the above implementations are *still* compiled using an actual compiler. The way code is developed with GLAF and the way code is automatically generated render it more favorable to compiler optimizations. In fact, as we see in Section 5.3.4,a compiler would fail to auto-parallelize certain implementations that a novice programmer, like most domain experts, could have written. Parallelism analysis at the language level provides a substrate of information that makes it easier to provide better feedback - and potentially advice - when auto-parallelization fails. While compiler parallelization reports may provide such information, it is not always clear to non-programmers, or may not be useful if it refers to parallelization attempts to code that has already undergone optimizations and does not correspond to what the user originally wrote. Finally, most widely-used programming languages were not written with auto-parallelization in mind.

"If for an application the compiler auto-parallel implementation is faster, then GLAF is useless." One should not view GLAF as an optimizing compiler and evaluate it as such. We don't see compiler auto-parallelization as an opponent that we try to beat. Achieving decent speed-up is undoubtedly one of GLAF's main purposes. This is why we need to further develop the auto-tuning part of the framework. But, ultimately GLAF is an all-encompassing *development framework* and should be evaluated as such. The compiler auto-parallel implementation may not always be the

fastest, as seen in our results, but, in any case, the *serial* code that constitutes the input to the compiler to be auto-parallelized is *still* produced by GLAF. And it is the way even serial code is generated that makes it more amenable to parallelization, by taking into account common pitfalls, as discussed in the Results section, in which code manually written by domain experts could have failed to be efficiently parallelized/vectorized.

"It seems that GLAF does a lot of things but is best at none." GLAF is not intended to be, and cannot be, the best in all aspects it tries to address. It is a *general-purpose* framework that introduces a different development approach for domain experts and provides a solid foundation with its basic building blocks that can be further developed. We make some clear, conscious design choices, given our target audience, and as such programmability for this audience is one of the top concerns. We intend to keep the language as simple as possible and parallelism-agnostic, to the degree we feel novice or non-programmer domain experts are comfortable with. Further specialization, however, can be incrementally added by use of libraries, as is the case with other languages and frameworks.

5.4 A GLAF Case Study with NASA

As we have mentioned before, the current and – even more so – future parallel computing landscape is destined to be *heterogeneous* in nature if we are to expect further performance increases within a practical power envelope. Beyond traditional multi-core CPUs, accelerator/co-processor architectures like the GPGPU, Intel MIC, or even the FPGA, have made their way into compute

nodes exposing huge potential for parallel execution. In order to democratize parallel, heterogeneous computing to audiences with minimal or no programming experience (especially *parallel*), like domain scientists, we have introduced GLAF (Section 5.3.1), a grid-based language and auto-tuning framework that seeks to narrow the gaps among performance, programmability and portability.

The main use-case of GLAF that we have showcased so far entails developing a program from scratch, i.e., the main function, user functions, and associated libraries used by said program are all developed using GLAF itself. Developing a program wholly within GLAF enables the user to take advantage of all features GLAF has to offer. However, the above approach disregards an important use-case, namely one where *one or more* functions of a larger program are desired to be implemented in GLAF (e.g., to take advantage of auto-parallelization and cross-platform code generation), but *not* the whole program. In fact, this is a typical case in high-performance/accelerator-based computing; parallel, computationally heavy code segments are offloaded to the accelerator (e.g., GPU), while the rest code is executed serially on the CPU. Interoperability, in a *plug and play* fashion, of GLAF-generated code with existing code was not originally accommodated by GLAF, but has been listed since the onset as one of the desirable features of our programming framework.

On a different note, the design, development and evaluation of GLAF have revolved around smaller-scale benchmark applications that exhibit diverse behaviors with respect to their computation and communication pattern, as well as types and levels of available parallelism. While the above has served as a reasonable and diversified test infrastructure for various functionalities and the grid

abstraction at the core of GLAF programming, we have not had the opportunity to evaluate GLAF in its entirety with a larger-scale application.

In light of the above, in this chapter we present our work with a real-world application of interest to NASA, called *Synoptic SARB*. Synoptic SARB is one of eight subsystems in the software ecosystem that supports the NASA CERES program. We describe both the CERES program, and Synoptic SARB in more detail in Section 5.4.1.

With this case study, we seek to extend the GLAF graphical user interface front-end, as well as the code generation back-end, to facilitate expressing existing data structures found in real-world applications and enable code generation whose output can seamlessly integrate with pre-existing code. Then, we evaluate GLAF in its entirety by stressing its back-ends (including the novel additions for ease of integration) via a larger-scale, real-world case-study that spans beyond previous small benchmark size and scope. We prove that the grid abstraction can represent all the data needed in the context of our case-study application, and that the resulting code is syntactically correct and functionally equivalent with the original. With respect to performance, the GLAF parallel auto-generated version performs up to 1.41x faster than the original serial implementation.

5.4.1 Background

The NASA Cloud and Earth's Radiant Energy System (CERES) Program and Associated Data Collection

The National Aeronautics and Space Administration (NASA) is a pioneer in the exploration of space, spearheading an array of far-reaching programs with global impact. One such umbrella-program is the *Earth Observing System* (EOS) [14]. In the context of this work we focus on the *Cloud and the Earth's Radiant Energy System* (CERES) [13], a key subprogram of EOS, specifically responsible for providing a better understanding of the complex interdependencies between the clouds and the energy cycle, as well as their effect in the global climate change. As part of EOS, CERES is conducive to enhancing our knowledge about the Earth's climate system and associated climate prediction models.

Satellites are an integral part of many NASA's missions, like the above. They fly in orbit around the Earth gathering (and relaying) invaluable information about the atmosphere, oceans, land surface, etc. Specifically, satellites that are part of the EOS, like EOS-Terra and EOS-Aqua, fly over the earth at a height of about 5000 miles and in a sinusoidal track. Terra, Aqua – and all satellites for that matter – are equipped with various instruments used in carrying out measurements of diverse kinds. A key instrument in the CERES program of EOS is the same-named (*CERES*) instrument. The CERES instrument is capable of measuring the energy levels at the top of the atmosphere, and providing estimates about the energy levels in the atmosphere and surface of the Earth. In practice, the CERES instrument is largely a radiometer, i.e., a device that measures the radiant flux

of electromagnetic radiation. It has three channels: a shortwave channel, a total channel, and an infrared window channel. Except for the above, it collects cloud and other atmospheric data that are relevant to its purposes.

Surface and Atmospheric Radiation Budget (SARB) Software System: Making Sense of CERES Data

The key software system that supports the CERES program is called *SARB*, which stands for *Surface and Atmospheric Radiation Budget*, and includes a number of subsystems. Each subsystem ingests certain inputs and provides outputs that may be archived "as is" or used as inputs in a subsequent subsystem. In the context of this work, our focus is on Subsystem 7.2: *Synoptic SARB*. Synoptic SARB computes the longwave, shortwave, and window channel vertical flux profiles that span from the surface of the Earth to the top of the atmosphere. Put simply, it calculates the energy exchange between the Sun, the Earth's atmosphere, clouds and surface, and the space (what "goes in" versus what "goes out").

In the context of Synoptic SARB the Earth is split into a number of zones, parallel to the equator. Computation for each zone can take place independently from each other, but *within* each zone processing is serialized according to the time (*synoptic hour*) for which the data being processed was acquired via the CERES and other measurement collection instruments. Still, within a synoptic hour various computations can run in parallel in a finer-grained resolution. The time it takes for each zone to complete is proportionate to its size (zones closer to the equator are naturally larger than zones near the poles). Inter-zone parallelism is the current way Synoptic SARB exploits

multiple CPU cores using MPI in the context of *Sun Grid Engine* (SGE). Intra-zone parallelism is not exploited, at all; any opportunities for parallel execution of loops, via multithreading or vectorization are left unexploited and available compute resources can thus be underutilized.

5.4.2 Extensions to GLAF

In Section 5.3 we refer to enabling GLAF auto-generated code *interoperability* with existing code as one of our goals towards addressing the needs of domain scientists. Given the large amounts of legacy code, much of which has been incrementally written over a span of decades, it is imperative that we provide a method for domain scientists to re-write *parts* of such code in a way to accommodate parallel, heterogeneous computing. Our proposed approach entails exploiting GLAF for automatic code generation and transparent integration of such auto-generated code segments with existing code.

Below, we provide an overview of the extensions the code generation back-end requires to accommodate the code integratability requirement. The enhancements and changes described below target the FORTRAN graphical user interface changes and code generation part, but some can be straightforward or trivially ported to C code generation, as needed.

Enabling Use of Existing Variables from Imported Modules

Real-world codes include variables that may be defined in external included files (*modules* in Fortran). To access such variables in GLAF and use them in various GLAF steps, as needed,

we have to create the corresponding grid in the GLAF *Global Scope* and mark them as belonging to an *existing module*. For grids that belong to an existing module the user needs to provide the name of the module the grid belongs to. This information is subsequently used in code generation. Specifically, variables that belong to the above category do not need to be re-defined in the body of the function where they are used. However, the code generation back-end needs to generate code for using the appropriate Fortran module.

Enabling Use of Variables in COMMON Named Blocks

Common blocks are a Fortran 77 language construct that defines a block of memory that can be shared among different program units (e.g., functions, subroutines). While this eliminates the need for passing variables that belong in common blocks as arguments, it can create more complex and less maintainable code. Despite the fact that common blocks are considered a bad programming practice, maintaining backwards compatibility and enabling integration of GLAF-generated code with a larger existing code is essential. The way variables that belong to common blocks are defined in GLAF is similar to variables that belong to existing imported modules. That is, the grid that corresponds to such a variable is created in the GLAF Global Space and selected as belonging to a common block, in which case the user is prompted to enter the name of the common block. In code generation, the appropriate language structure reflects the common block; the variables identified as belonging to the same common block are grouped and defined (i.e., grid type and grid name), and the appropriate *"COMMON* $< name >$ *var1, var2, ..."* code is generated afterwards.

Enabling Use of Subroutines

The original GLAF version models all Fortran subprograms as functions and generates code accordingly. To support the case the user desires code generation of subprograms in FORTRAN as subroutines (i.e., subprograms that do not return a value, as opposed to functions), we provide the corresponding functionality. While the same functionality can be achieved using functions, enabling code generation for subroutines is imperative for supporting interoperability/integration with existing code. On selecting a return value of *void* data type at the header of a GLAF function (i.e., effectively no return value), code for subroutine is auto-generated together with a "*CALL*" subroutine call at the caller site/step.

Enabling Use of Existing Elements of TYPE Variables

Support for existing elements of TYPE variables (i.e., the analogous to elements of a C struct in Fortran) is a subcase of using existing variables from imported modules. As for any existing grid, the user needs to declare it as such in the grid definition GUI screen in the Global Scope, and provide the name of the module in which it belongs. Furthermore, he needs to specify that it is part of an existing TYPE variable, in which case he is prompted for the TYPE variable name. On the code generation side, any use of the above element of a TYPE variable, is appropriately generated with the TYPE variable name prefix (e.g., an element *charge* that belongs to a TYPE variable named *atom* would be generated as *atom%charge*).

Enabling Additional Library Functions

GLAF supports a number of libraries and associated library functions. Such functions correspond to frequently used operations (e.g., GLAF supports a large number of the C/Fortran math library). Using a library function (via the GUI), leads to code generation in the supported languages. Libraries are an extensible part of GLAF, which can be adapted to domain-specific needs. In the case of this project, we extended support for the ABS(), ALOG(), and SUM() functions, used in FORTRAN.

5.4.3 Results

In this section, we present: a) the details of implementing Synoptic SARB using GLAF and our findings with respect to the functional correctness of the code, and, b) the performance results of the automatically generated parallel code and associated insights.

Implementing Synoptic SARB Using GLAF

The subroutines of interest are part of the *fulib* library, a library that provides an implementation of the *Fu-Liou Radiative Transfer Model* [109], the model used in Synoptic SARB to model the energy transfer (in the form of electromagnetic radiation) between and across the earth and the top of the atmosphere. The Fu-Liou model takes into account absorption, emission and scattering of the radiation. Energy can be lost due to absorption, gained by emission, while redistribution can occur by scattering. Specifically, the algorithmic implementation of Fu-Liou in Synoptic SARB ad-

dresses the two sub-cases of radiation in the longwave and shortwave spectrum (as can be gleaned from the names of the corresponding subroutines in Table 5.3).

The identified subroutines of interest originally span about 700 source lines of code (break-down per subroutine presented in Table 5.3). This number does not account for lines of code that correspond to data types and variables from imported modules (mainly related to the Fu-Liou radiative transfer model's input and output variables and custom data types whose part some of them are). Due to the nature of the computations, and the fact that these subroutines are part of a much larger code-base with lots of dependencies, we are able to exercise multiple aspects of GLAF auto-parallelization and code generation, including aspects of code integration not accommodated or tested before.

For evaluating correctness of the code, we generate a wrapper function that calls the GLAF auto-generated subroutines and provides sample values for the required inputs. The imported modules, from which the auto-generated code uses existing variables and custom data types, are used "as is". We then conduct a step-by-step unit testing of the code, and a code-wide side-by-side comparison of the results from the execution using the GLAF auto-generated subroutines, against the results from executing the original code. We repeat this process for both the serial and parallel versions

Table 5.3: Subroutines implemented using GLAF

Subroutine name	**SLOC**
lw_spectral_integration	75
longwave_entropy_model	422
sw_spectral_integration	50
shortwave_entropy_model	13
entropy_interface	46
adjust2	38

of the auto-generated code and conclude that the auto-generated code is functionally equivalent to the original code. For the parallel version of the auto-generated code, as an additional inspection step, we manually verify the correctness of the OpenMP directives and associated clauses used.

Our experiment with implementing the computationally intensive parts of Synoptic SARB shows that: (a) the grid abstraction on which GLAF is built is generic enough to accommodate real-world applications, and, (b) GLAF is now more robust and can successfully generate correct serial and parallel code that is interoperable with existing code.

Performance Evaluation

In this section we present and discuss the performance of the parallel code for Synoptic SARB, as automatically generated by GLAF in Fortran (the original code-base that the GLAF-generated code needs to be integrated with is in Fortran, too). The code (implemented as described in Section 5.4.3) was compiled with gfortran (v.4.9.2) at the -O3 optimization level and run on a Linux-based machine (Debian Linux 8.6, kernel v.3.16) with an Intel Core i5-2400 CPU (four cores clocked at 3.10 GHz).

Figure 5.24 shows the results of performance evaluation across different implementations of Synoptic SARB. Specifically, it shows the speed-up of the GLAF serial implementation (*GLAF serial*) and incrementally optimized GLAF parallel (four threads) implementations (*GLAF-parallel v0-v3*) versus the original serial Synoptic SARB implementation (*original serial*). The details of each implementation are given in Table 5.4.

Table 5.4: Synoptic SARB implementations

Implementation	Description
original serial	Original serial implementation
GLAF serial	Serial implementation generated by GLAF
GLAF-parallel v0	Parallel implementation generated by GLAF with OMP directives in all applicable loops
GLAF-parallel v1	GLAF-parallel v0 with removed OMP directives from initializations to zero or with single value assignments (loads)
GLAF-parallel v2	GLAF-parallel v1 with removed OMP directives from simple single loops
GLAF-parallel v3	GLAF-parallel v2 with removed OMP directives from simple double loops

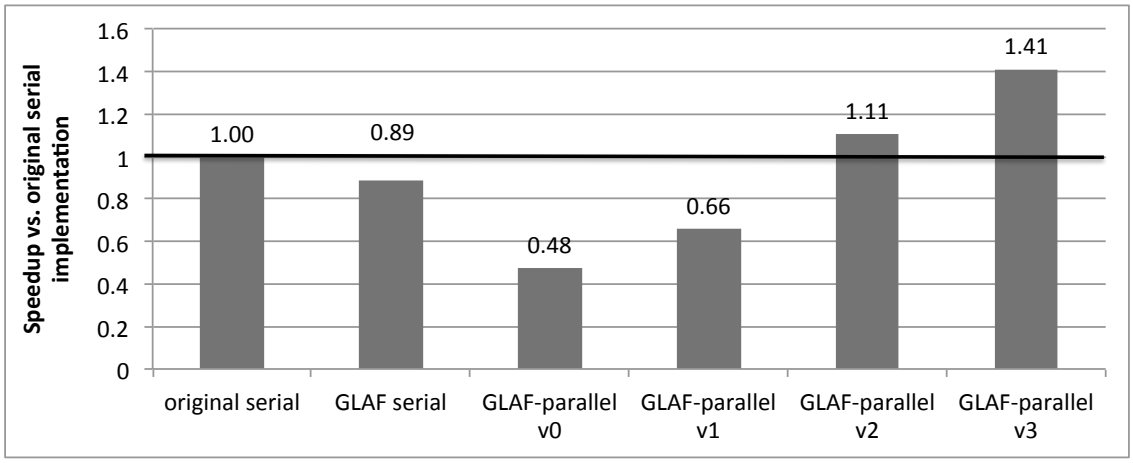


Figure 5.24: Performance results: Speed-up of GLAF-generated versions versus the original serial implementation of Synoptic SARB

First, we observe that the GLAF generated code in its serial implementation (*GLAF serial*) performs slightly worse than the original serial implementation (*original serial*). In our experience with GLAF we have observed cases like this, as well as cases where the GLAF auto-generated serial implementation outperforms the original serial. The GLAF GUI programming enforces an implicit structure in a program, whereby any loops within a step except at the outermost level (e.g., within an *if* statement) need to be implemented as a new function. If the functions are not inlined, there is a certain calling overhead that depending on the algorithm trace may negatively affect execution time. Also, certain compiler optimizations may be difficult/impossible when code spans multiple functions. In the opposite case, smaller functions can be automatically inlined by the compiler, and the implicitly enforced structure can help with certain function-level compiler optimizations. In any case, the potential for such (small) performance deterioration is expected to be outweighted by the parallelism benefits of GLAF auto-generated code.

GLAF-parallel v0 in Figure 5.24 corresponds to the implementation that contains the parallel code generated by GLAF. As we describe in Table 5.4, this includes OpenMP directives that surround all loops that the parallelism detection back-end has identified as parallelizable. This implementation performs about 50% slower than the original serial implementation, highlighting the disadvantage of a "one-size fits all" approach when it comes to applying OpenMP directives to eligible loops. Currently, GLAF does not contain a means of evaluating whether a loop is better off without OpenMP directives. As future work, we suggest the incorporation of a performance prediction/-modeling back-end that will guide the auto-code generation in a more intelligent way (e.g., selecting SIMD directives, instead of OpenMP, or neither). We discuss this issue further in Section 6.2.

In the cases *GLAF-parallel v0* to *GLAF-parallel v3* we incrementally remove OpenMP directives from three distinct cases of loops and provide insights on performance, thereby highlighting potential automation of such removal in future work.

GLAF-parallel v1 is based on *GLAF-parallel v0* with the difference that we have manually removed OpenMP parallelization directives from two types of loops: a) initialization of arrays (grids) to zero, and, b) initialization of arrays with a single value loaded from another array. These two are typical cases where the compiler can apply optimizations that outperform thread-level parallelism (and its associated overheads). For instance, initializing an array to zero can be done via *memset* emitted by the compiler, as an optimization, for eligible loops. Alternatively, SIMD operations can be used for loading and assigning values from an input array. This can be observed in the performance results: speed-up increases from 0.48 (*GLAF-parallel v0*) to 0.66 (*GLAF-parallel v1*).

In *GLAF-parallel v2* we proceed with removal of further OpenMP directives from otherwise parallelizable parts of the code. Specifically, we remove OpenMP directives from all remaining single loops of the code. This contains loops with one-line assignments that contain mathematical operations, few lines (two to four) of similar assignments, as well as loops that contain reductions (and that have been identified as such by GLAF auto-parallelization back-end). In these cases, we identified (as above, via inspection of the compiler optimization reports and/or generated assembly code) that the compiler emits SIMD instructions or proceeds with loop unrolling (when the number of loop iterations is low). As with the previous cases, the overhead of thread-level parallelism is not justified over data-level parallelism (SIMD) or instruction-level parallelism (ILP). Hence,

removing the OpenMP directives from the corresponding loops and allowing the compiler to apply its own optimizations increases the speed-up over the original serial implementation to 1.11-fold.

Last, in *GLAF-parallel v3* we remove OpenMP directives from double-nested loops that contain one or a few statements without including any control structure (if/else statements). Again, observation of the compiler optimization reports reveals that the compiler can identify the loops as parallel and applies SIMD optimizations or loop unrolling. Effectively, this leaves the GLAF auto-generated parallel code with OpenMP directives in two large loops in the *longwave_entropy_model* subroutine. The compiler fails to identify these loops as parallel, hence the performance of the GLAF code that includes the appropriate OpenMP directives (*"OMP PARALLEL FOR"* with the necessary *"PRIVATE"* clause) outperforms the original serial implementation by 1.41x.

As we mention at the start of Section 5.4.3, the parallel implementations shown correspond to execution with four threads. Experimentation with varying number of threads (up to the maximum of 8 threads for our test CPU) showed that four threads provides the optimal performance. Figure 5.25 shows the performance of the fastest implementation (*GLAF-parallel v3*) as the speed-up over the *GLAF-serial* implementation. The parallel version that uses one thread presents a minor slow-down (0.92-fold over *GLAF-serial*) due to the OpenMP run-time associated overhead (which is present, despite using a single thread). Two and four threads yield speed-ups of 1.24- and 1.59-fold, respectively, while adding more threads (e.g., 8) yields diminishing returns (0.7-fold). For the latter, one should consider the fact that our test CPU has a maximum of four physical cores (up to 8 logical cores with hyper-threading). Also, the double-nested loops that are decorated with OpenMP directives by GLAF consist of a total of *2 x 60 = 120* iterations (since GLAF gener-

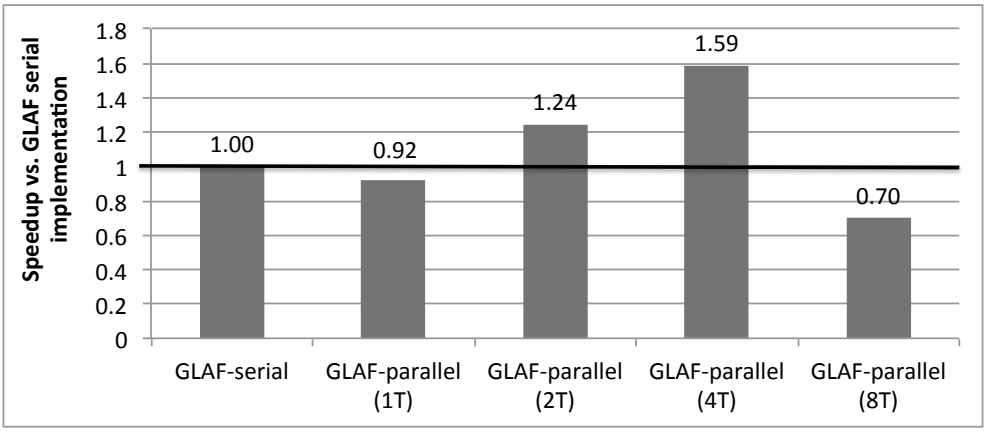


Figure 5.25: Parallel scalability: Speed-up of fastest GLAF-generated version (*GLAF-parallel v3*) with varying number of threads (*T*) versus GLAF serial implementation of Synoptic SARB

ates a *"COLLAPSE(2)"* clause). This is a small number (also considering the complexity of the loop code), hence more threads entail overhead (OpenMP run-time, memory coherence, etc.) that cannot be amortized.

Overall, we find that GLAF auto-generated parallel code (with appropriate selection of which parallelized loops to keep) performs 1.41 times faster than the original serial implementation. While this may appear underwhelming for a multi-threaded execution with four threads, it does not reflect on the capabilities of GLAF itself. Rather, performance owing to parallelization is limited by the workload itself (in the original algorithm) as described above (i.e., small number of iterations in parallel loops). Also, serial parts of the algorithm, between the parallel section can limit the maximum parallelism (*Amdahl's Law*). As far as other opportunities for parallelism are concerned, most loops prove to be amenable to automatic optimizations by the compiler (mainly in the form of SIMD or loop unrolling optimizations). The amount of optimizations the compiler can perform depend on the algorithm itself (in other examples in this Chapter we see GLAF parallel code to far

outperform the compiler's optimizations). In any case, as we mention in Section 5.3.2, GLAF does not seek to compete against the compiler, but rather to complement it in an attempt to provide the best achievable performance for domain scientists.

5.5 Conclusion

In this chapter we addressed the issues of programmability and portability. First, we presented a discussion on the programmability of GEM across three multi- and many-core architectures: Intel Sandy Bridge multi-core CPU, Intel Xeon Phi (MIC) co-processor, and the NVIDIA Kepler K20 GPU. From a programmability perspective, the CPU programming model is arguably the most programmable, mainly due to its widespread adoption. Xeon Phi, which follows the same programming paradigm, benefits from allowing programmers to leverage their existing knowledge and expertise. On the other hand, GPU programming for high performance demands the use of native code, such as CUDA. This entails familiarizing oneself with a different mindset of parallel programming, a pertinent array of platform-specific optimization techniques, as well as taking care of mundane details, such as data transfers and kernel configuration, but while preserving more of the original computational code in its original state. Which is more programmable remains a matter of opinion, but the GPU and Xeon Phi devices require higher levels of optimization to reach their performance potential. New compilers and tools are needed to bridge this gap, and bring high-performance acceler- ated computing into the reach of the automatic optimizations.

To this end, we proposed GLAF, an all-encompassing code development environment for non-programmers or novice programmers, like the majority of *domain experts*. Its key elements are its *intuitive* visual programming interface that aspires to make it easy for them to express and validate their algorithms and its ability to *automatically* generate efficient serial and parallel C and Fortran, as well as OpenCL code. We showed illustrative example applications and presented results that corroborate that GLAF and its associated "*easily* develop-once and run-everywhere" programming paradigm can help alleviate the performance, programmability, and portability gaps. With a high-level programming framework like GLAF, domain scientists can not only exploit traditional multi-core CPU architectures, but also accelerators, like Intel MICs, GPUs, and even FPGAs. GPU computing is of paramount importance and, as such, so is further facilitating entry of access via GLAF OpenCL support. FPGAs are predicted to be necessary on the path to exascale computing, but adding reconfigurable computing to the existing mix of CPUs, GPUs and Intel MICs further perplexes the current status quo. Hence, providing support for FPGA computing via Altera FPGAs and a novel FPGA-targeted back-end that supports FPGA-specific optimizations has a great potential to further democratize FPGA computing and position FPGAs as a competitive solution in the heterogeneous landscape.

We showed how automatically generating multiple versions of code (derived from the *same* GLAF code) in different languages and using different data layouts and optimizations can facilitate obtaining the best performing code. This systematic code generation and auto-tuning approach extends beyond standard optimizations and tuning that typically target a single language and is especially useful for novice programmers, where certain mistakes in a language may result in code that the

compiler fails to auto-parallelize. Our findings reveal that the traditional coding paradigm, where a single implementation is written, can be sub-optimal for novice (or even average) programmers. Rather, GLAF allows multiple starting points (analogous to different seeds in a state-space search algorithm) for different optimizations, which lead to overall better performance (analogous to the global as opposed to local minimum). We concluded this chapter with a case study that puts GLAF into practice in the context a large application of interest to NASA. Our overall experience indicated that the grid high-level abstraction at the core of GLAF and GLAF as a programming framework are capable of handling real-world, larger-scale, practical scenarios.

Chapter 6

Summary and Future Work

This chapter summarizes the work presented in this dissertation and concludes with related future research directions.

6.1 Summary

This dissertation seeks to provide a better understanding of the intertwined aspects of *performance*, *programmability*, and *portability* in the increasingly important, fast-progressing, and research-challenging area of *heterogeneous computing*.

To this end we conduct a multi-faceted study towards providing answers to three main research questions, under the assumption of heterogeneity:

(1) What are the performance implications of architectural characteristics of modern heterogeneous architectures and how can we systematically study them? How can we extend such a methodology to delineate trends that may shape the future of heterogeneous computing?

(2) What are the trade-offs between performance, programmability, and portability in light of a perplexed ecosystem of programming languages, compilers, tools, and optimization techniques?

(3) How and to what degree can we secure acceptable performance at a much increased programmability, as well as functional and performance portability via the use of tools and frameworks?

In answering the **first question**, we build upon prior work with the *OpenDwarfs* benchmark suite. The contributed enhancements in the suite itself allow us to use this new version of OpenDwarfs to draw conclusions regarding performance across a wider range of target platforms, now to include Intel Xeon Phi and FPGAs. Additionally, our work towards ensuring a uniform level of optimization in the benchmarks facilitates fairness in cross-platform performance comparisons. Dwarf-based benchmarking, i.e., based on computation and communication patterns, can be a more representative way of characterizing heterogeneous platforms, as opposed to largely arbitrarily chosen applications. Using an n-body dwarf we present a multi-dimensional performance study that spans architectures, languages, and optimization levels. This work is among the very first to evaluate the Intel MIC architecture (Knights Corner), the NVIDIA Kepler architecture (K20), and OpenCL with associated optimizations as a programming language for the FPGA. Extending the concept

of dwarf-based benchmarking, we introduce the notion of telescoping architectures and propose a state-of-the-practice methodology for evaluating conceptual heterogeneous architectures.

With respect to the **second question**, we extend our performance study using an n-body dwarf to include programmability and portability. The scope of this study encompasses representative platforms (including a multi-core CPU, Intel MIC, and a GPU), different programming languages (including C with OpenMP, OpenACC, CUDA, and CPU SIMD intrinsics), and varying optimization levels (stretching from compiler optimizations to manual code optimizations). From a programmability perspective, we find that programming in the CPU is the most programmable, not least because of the maturity of the programming tools (e.g., GNU tool-chain). The similarity of the programming model for Intel MIC to that of traditional multi-core programming facilitates leveraging existing knowledge and expertise transfer. However, the programming tools available are not yet as mature as in the case of the CPU domain, hence automated methods for parallelization, vectorization, and other compiler optimizations fall short. Last, the GPU proves to be the least programmable platform, as it mandates use of CUDA to extract the best performance, requiring the programmer to adopt a wholly parallel mindset, which is fundamentally different from the traditional CPU (sequential) programmind model.

Last, with regards to the **third question**, and based on our insights from answering the first two, we propose a high-level programming abstraction based on *grids* that can automatically generate parallel code, optimized for certain platforms, in multiple target languages. We incorporate this grid abstraction as the cornerstone of GLAF, an all-encompassing visual code development environment for domain experts with minimal or no parallel programming knowledge. GLAF seeks to

offer a means of enhancing programmability and portability of parallel, heterogeneous computing, while ensuring sufficient performance. We show the generality of the proposed abstraction and validate the features of GLAF via a number of benchmark applications, as well as a larger-scale NASA application. Our findings reveal that the traditional programming paradigm, where a single implementation is developed, can be sub-optimal for novice programmers. GLAF allows multiple starting points for different optimizations (analogous to different seeds in a state-space search algorithm) that can lead to overall better performance (analogous to the global versus the local minimum).

6.2 Future Work

In this dissertation we attempted to provide a holistic view and thorough coverage of performance, programmability, and portability aspects of heterogeneous computing, given their combined overall importance for the future of computing. Given the intrinsic breadth of each of the above areas, there inevitably exist further challenges and ensuing areas of future research interest. We present a brief overview of such future work in the sections below.

6.2.1 OpenDwarfs

OpenDwarfs is an open-source project [16] and, combined with its robust build system, extensible API and ease of use, aims at being utilized and further extended by the research community. Future work includes:

- **Extending the OpenDwarfs benchmark suite:** OpenDwarfs can benefit by incorporating features such as: (a) input dataset generation for dwarfs that enables exercising multiple code paths and stressing different subsystems, (b) automated result verification functionality. More importantly (and accordingly challenging), OpenDwarfs is in need of a means of genericizing each of the dwarfs, i.e., an attempt to abstract them on a higher level (it has been argued that some dwarf implementations may be considered too application-specific, thus defying the very purpose of use as "patterns").

- **Extending breadth of evaluated architectures:** Given the ongoing introduction of novel architectures (e.g., automata processor), older architectures that only recently started officially supporting OpenCL (e.g., Altera, Xilinx FPGAs), or new devices within an platform family (e.g., different generations of GPUs within and across vendors, or new generations of Intel MICs), there is an accompanying need for characterizing the performance thereof. Further options for more thorough evaluation include considering different vendors' OpenCL runtimes, experimenting with varying size and/or shape of input datasets, and considering the power consumption profiles with respect to dwarf execution.

- **Architecture-aware optimizations:** Based on the breadth of available architectures, as discussed above, there is a corresponding breadth of optimizations for each dwarf that tailor performance to each target architecture (e.g., shared memory optimizations for GPU, data-transfer optimizations in the case of APUs, or unique FPGA-specific OpenCL optimizations).

6.2.2 Telescoping Architectures

With our work on *Telescoping Architectures* we seek to provide a systematic and generalizable methodology for identifying future trends in heterogeneous computing via the use of dwarf-based benchmarking. Moreover, we show how to apply the proposed methodology in the context of what we term *Cluster on a Chip* (CoC) that may contain a combination of CPU, GPU, Intel MIC and FPGA *compute engines* (CEs).

Our study intentionally focuses on breadth, i.e., a broader design space exploration, as far as CoCs are concerned. We also attempt to shed light on cases of CoCs that contain FPGAs, since unifying CPUs and FPGAs on package – and later on-die – is an upcoming trend. As such, there is plenty of space for refinements, and further studies in the heterogeneous domain (e.g., scheduling, power/energy-aware optimizations, memory hierarchies, network-on-chip, and so on for highly heterogeneous architectures). Notably, the proposed methodology is characterized by two important features that facilite this:

- *Portability and practicality:* It is based on an open-source benchmark suite (OpenDwarfs), so the methodology can be easily replicated. Also, since the reference benchmark suite is developed in OpenCL, it can seamlessly run across a wide range of target platforms.

- *Extensibility:* It can be further extended, by design, to include more dwarfs or more target platforms under consideration in the future.

In Section 4.4.2 we discuss the assumptions we make in our proposed methodology and discuss limitations of our approach. Relaxing some or most of these assumptions and addressing certain limitations is a key aspect of future work. Our approach provides a first-order study on the problem, and cycle-accurate simulation of specific prospective CoCs falls in the other end of the spectrum; a middle-ground approach could give better insights and help approximate the future realities of heterogeneous computing. Below, we propose further opportunities for future work:

- **Analytical model for "Telescoping Architectures":** Despite the advantages of the real-world, experimental approach we present in our work, it is important to conduct further research on relaxing some of the working assumptions and provide a more realistic *analytical model*. Such a model will not only focus on performance (around which our current work is centered), but also in power and energy. The former should capture issues like inter-CE interference and data transfers among CEs (which in turn touches on considerations such as NoC, off-chip memory bandwidth, efficient intra node and inter node data sharing). The latter should more thoroughly cover power/energy aspects of CoCs and associated constraints (e.g., integrating multiple CEs on-chip can substantially increase the system's power density, potentially leading thermal issues).

- **Extending the "Telescoping Architectures" evaluation methodology:** A first step towards extending our methodology entails: (a) including a broader set of workloads (i.e., more dwarfs for building synthetic benchmarks), (b) more distinct architectures and representative devices from each architecture, and (c) more scheduling techniques for allocating work across compute engines in a CoC. The latter may include exploiting parallelism on

the dwarf-level *across* CEs within a CoC (in this work we schedule *whole* dwarfs from a synthetic benchmark in separate CEs). To harness the full potential of CoCs, it is important to study the performance benefits when optimized implementations for dwarfs in a synthetic benchmark are available for *each* CE, along with appropriate performance modeling and auto-selection techniques. Last, from a software perspective, all the above functionalities should be incorporated into a *CoC evaluation framework*, in which the user can select among a set of options (e.g., target platforms, synthetic benchmarks composition) and execute/evaluate CoCs – based on available hardware and an analytical model that takes into account issues discussed above (e.g., modeling data transfers) – in real-time.

- **Extending Pollack's Rule to heterogeneous computing:** A thorough statistical analysis on the relationship between die area and performance can potentially identify trends or relationships in the heterogeneous domain, in a similar way that Pollack's Rule [234] does for homogeneous multi-core computing.

6.2.3 GLAF

Our GLAF prototype incorporates core functionalities that form the *substrate* for an extensible set of capabilities. Given the breadth of the problem that GLAF seeks to address (as outlined and motivated in Section 5.3) there are various limitations in the original prototype, which accordingly serve as potential branches worth pursuing further as future work. We provide an outline below:

- **Intra-node enhancements:** Currently, GLAF provides a set of alternative code-generation options, beyond the selection of the output language. This includes among others data layout arrangements that can be beneficial, especially in the cases of CPU and Intel Xeon Phi, and certain optimizations in auto-generated OpenCL code when targeting Altera FPGAs. There is a broad set of back-end optimizations that can be beneficial for GPU targets, as well as the latest Intel MIC generation (*Knight's Landing*). Further refinement of the FPGA-specific OpenCL optimizations with GLAF will help to cover a broader set of applications in the general case. Identifying whether certain optimizations can/need to be applied can follow approaches similar to those of auto-tuning frameworks, domain-specific languages, and compiler frameworks. Alternatively, frequently used libraries can be built-in in GLAF. The user can easily use them via a high-level API and back-end code generation can provide optimized (pre-designed) platform-specific implementations. Last, extending the supported target languages (e.g., OpenACC/OpenMP 4.0 for accelerators) can provide broader coverage and more starting points in searching for the optimal combination of language, target platform, and platform-specific optimizations.

- **Inter-node enhancements:** GLAF currently only provides intra-node functionality. That is, all generated code targets a single architecture/platform – CPU, GPU, MIC, or FPGA – *within* a compute node. While this is of great value in and by itself, being able to offer enhanced programmability for cluster computing would provide added value to the tool. To this end, the code generation back-end could be extended to accomodate automatic code generation for splitting computation and appropriate MPI calls (send, receive, gather, etc.)

While this is not a trivial endeavor – and a "one size fits all" approach may not be feasible – certain cases could be accommodated.

- **Smart auto-tuning:** Currently GLAF is capable of generating multiple code versions, as selected by the user at the code generation menu (e.g., language, data layout optimizations, target platform). Then the user has to run the automatically generated script to execute and time *all* the generated versions to find out which one performs the best for the problem at hand (algorithm, input data-set/parameters). Along the same lines, but within a single program instance's scope, auto-parallelization of a step takes place irrespective of whether a given step benefits from running in parallel or not. In certain cases (e.g., parallelizing small loops with OpenMP or as OpenCL kernels) the performance difference may range from minimal to very high. In a more complex scenario, certain loops may benefit from OpenMP parallelization, whereas others may benefit from running as an OpenCL kernel on an accelerator. Taking the above concept even further, execution of certain parallel steps may be split *across* platforms for better resource utilization. Given the number of *combinations* of possible code generation choices the implementation search space can be intractable for practical purposes, and especially in the case of FPGA programming where compilation is slow (in the order of hours). Combining the existing back-ends of GLAF with a performance modeling back-end could help prune the implementation search-space. Such pruning can be beneficial in all three aforementioned levels; estimating the best mix of assigning step loops to target architectures and/or executing the parallel or serial versions.

- **User interface/interaction enhancements:** GLAF seeks to provide a friendly and easy-to-use programming environment. As such, the user interface can be further enhanced to facilitate GLAF program development. While we have had GLAF tested and empirically evaluated by a number of people (among which students in an "Accessible Parallel Programming" graduate-level class at Virginia Tech), further formal studies with a large number of participants need to be conducted. With respect to the provided visualization methods, we have received feedback arguing that certain data structures cannot be easily understood as grids (e.g., graphs). To this end, appropriate libraries can be developed to visualize graphs as such (based on the internal GLAF grid-based representation) and vice-versa. One last, important enhancement, is the introduction of an interactive hint system. Currently, GLAF provides a visual cue to the user as to whether each step is parallelizable or not (based on the analysis performed by the auto-parallelization back-end). Internally, GLAF retains information about the reason parallelism in a loop is "broken". Such information could be relayed to the user along with hints on potentially manually resolving any dependencies (if the process cannot be automated). The aforementioned parallelization hint system could be appropriately designed to be of use in parallel computing education.

- **Beyond the GUI:** From its inception, GLAF was proposed as a visual programming alternative to text-based programming languages. The reasoning behind this has been the fact that collocating the data *together* with the code would make programming easier and more intuitive for non-programmers, like many domain scientists. Additionally, it was designed to be a click-based graphical interface for the very same reasons. Some programming en-

vironments targeted at younger ages, and mainly focused on teaching programming to kids, like Scratch [239], SNAP [111], Parallel SNAP [105], follow a picture-based programming approach. An interesting endeavor would entail combining the front-end and associated features of such environments with the back-ends and associated features of GLAF. On a different path, it has been proposed that certain groups of people, not originally targeted by GLAF, such as seasoned programmers, may indeed prefer a text-based language as an input over a graphical interface, while retaining all the benefits of the GLAF back-ends. To this end a GLAF language may formally be defined, and accompanied by an appropriate parser that can generate the GLAF internal representation that the back-ends expect as input. Extending the above notion of a GLAF text-based input, it has been requested that a source-to-source translator from commonly used languages (e.g., Fortran) to the GLAF internal representation be implemented. From that point onward (i.e., once we have the GLAF internal representation available) we can exploit the rest functionalities of GLAF. Both the above approaches pose certain challenges (design and implementation choices, or even generality and feasibility).

Bibliography

[1] clFFT. http://clmathlibraries.github.io/clFFT/.

[2] Codesign at Lawrence Livermore National Laboratory. https://codesign.llnl.gov/proxy-apps.php.

[3] cuBLAS: NVIDIA Developer. https://developer.nvidia.com/cublas.

[4] CUDA Math Library. https://developer.nvidia.com/cuda-math-library.

[5] cuFFT: NVIDIA Developer. https://developer.nvidia.com/cufft.

[6] CUSP, NVIDIA Developer. https://developer.nvidia.com/cusp.

[7] Cyclone-V SoC. https://www.altera.com/products/soc/portfolio/cyclone-v-soc/overview.html.

[8] Intel Integrated Performance Primitives. https://software.intel.com/en-us/intel-ipp.

[9] Intel Math Kernel Library. https://software.intel.com/en-us/intel-mkl.

[10] ITRS Public Website. http://www.itrs2.net/.

[11] LANL Proxy Applications. http://www.lanl.gov/projects/codesign/proxy-apps/lanl/index.php.

[12] Mantevo. https://mantevo.org/packages/.

[13] NASA CERES: Clouds and the Earth's Radiant Energy System Information and Data. https://ceres.larc.nasa.gov.

[14] NASA's Earth Observing System. https://eospso.nasa.gov.

[15] NVIDIA Performance Primitives. https://developer.nvidia.com/npp.

[16] OpenDwarfs Benchmark Suite. https://github.com/vtsynergy/OpenDwarfs.

[17] Software: Dr. Alexey Onufriev. http://people.cs.vt.edu/ onufriev/software.php.

[18] Thrust: Parallel Algorithms Library. `https://thrust.github.io`.

[19] A. Munshi, Editor. *The OpenCL Specification. Version: 1.0*. Khronos OpenCL Working Group, 2009.

[20] V. Adhinarayanan and W. Feng. An Automated Framework for Characterizing and Subsetting GPGPU Workloads. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016.

[21] C. C. Aggarwal. *Data Streams: Models and Algorithms*, volume 31. Springer Science & Business Media, 2007.

[22] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan. CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2015.

[23] A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, W. Feng, K. R. Bisset, and R. Thakur. MPI-ACC: An Integrated and Extensible Approach to Data Movement in Accelerator-Based Systems. In *International Conference on High Performance Computing and Communication International Conference on Embedded Software and Systems (HPCC-ICESS)*, 2012.

[24] Altera Corporation. *Implementing FPGA Design with the OpenCL Standard*, 2.0 edition, 2012.

[25] Altera Corporation. *Altera SDK for OpenCL: Programming Guide*, 2013.

[26] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. *STAPL: An Adaptive, Generic Parallel C++ Library*, pages 193–208. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[27] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[28] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006.

[29] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A View of the Parallel Computing Landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[30] D. F. Bacon, R. Rabbah, and S. Shukla. FPGA Programming for the Masses. *Communincations ACM*, 56(4):56–63, Apr. 2013.

[31] D. A. Bader, V. Kanade, and K. Madduri. SWARM: A Parallel Programming Framework for Multicore Processors. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2007.

[32] S. S. Baghsorkhi, N. Vasudevan, and Y. Wu. FlexVec: Auto-Vectorization for Irregular Loops. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[33] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The NAS Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[34] N. A. Baker. Poisson-Boltzmann Methods for Biomolecular Electrostatics. *Methods in Enzymology*, 383:94–118, 2004.

[35] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *International Conference on Supercomputing (ICS)*, 2008.

[36] S. Beamer, K. Asanović, and D. Patterson. Direction-Optimizing Breadth-First Search. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[37] M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *ACM International Conference on Computing Frontiers (CF)*, 2006.

[38] N. Bell and M. Garland. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2009.

[39] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for CUDA. *GPU computing gems Jade edition*, 2:359–371, 2011.

[40] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: A Programming Model for the Cell BE Architecture. In *ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2006.

[41] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2008.

[42] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[43] N. Birkbeck, J. Levesque, and J. N. Amaral. A Dimension Abstraction Approach to Vectorization in Matlab. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2007.

[44] F. Bodin and S. Bihan. Heterogeneous Multicore Parallel Programming for Graphics Processing Units. *Sci. Program.*, 17(4):325–336, Dec. 2009.

[45] S. Borkar. Thousand Core Chips: A Technology Perspective. In *Design Automation Conference (DAC)*, 2007.

[46] S. Borkar and A. A. Chien. The Future of Microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[47] A. Branover, D. Foley, and M. Steinman. AMD Fusion APU: Llano. *IEEE Micro*, 32(2):28–37, March 2012.

[48] K. J. Brown, H. Lee, T. Rompf, A. K. Sujeeth, C. De Sa, C. Aberger, and K. Olukotun. Have Abstraction and Eat Performance, Too: Optimized Heterogeneous Computing with Parallel Patterns. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2016.

[49] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *ACM SIGGRAPH 2004 Papers*, 2004.

[50] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguad, and J. Labarta. Productive Programming of GPU Clusters with OmpSs. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2012.

[51] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato. It's Alive! Continuous Feedback in UI Programming. *SIGPLAN Not.*, 48(6):95–104, June 2013.

[52] D. Buttlar and J. Farrell. *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly Media, 1996.

[53] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi. Productivity analysis of the UPC language. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2004.

[54] Z. Cao, H. Tang, Q. Li, B. Li, F. Chen, K. Wang, X. An, and N. Sun. Design of HPC Node with Heterogeneous Processors. In *IEEE International Conference on Cluster Computing (CLUSTER)*, 2011.

[55] L. Carrington, M. M. Tikir, C. Olschanowsky, M. Laurenzano, J. Peraza, A. Snavely, and S. Poole. An Idiom-finding Tool for Increasing Productivity of Accelerators. In *ACM International Conference on Supercomputing (ICS)*, 2011.

[56] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: The New Adventures of Old X10. In *International Conference on Principles and Practice of Programming in Java (PPPJ)*, 2011.

[57] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A Domain-Specific Approach to Heterogeneous Parallelism. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011.

[58] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *Int. Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[59] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc. Optimizing and Tuning the Fast Multipole Method for State-of-the-Art Multicore Architectures. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010.

[60] L.-W. Chang, I. El Hajj, H.-S. Kim, J. Gómez-Luna, A. Dakkak, and W.-m. Hwu. A Programming System for Future Proofing Performance Critical Libraries. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2016.

[61] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

[62] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. Pannotia: Understanding Irregular GPGPU Graph Applications. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2013.

[63] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[64] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron. A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads. In *IEEE International Symposium onWorkload Characterization (IISWC)*, 2010.

[65] D. Chen and D. Singh. Invited paper: Using OpenCL to Evaluate the Efficiency of CPUs, GPUs and FPGAs for Information Filtering. In *International Conference on Field Programmable Logic and Applications (FPL)*, 2012.

[66] L. Chen, L. Liu, S. Tang, L. Huang, Z. Jing, S. Xu, D. Zhang, and B. Shou. *Unified Parallel C for GPU Clusters: Language Extensions and Compiler Implementation*, pages 151–165. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[67] R. Cheveresan, M. Ramsay, C. Feucht, and I. Sharapov. Characteristics of Workloads Used in High Performance and Technical Computing. In *ACM International Conference on Supercomputing (ICS)*, 2007.

[68] A. A. Chien, A. Snavely, and M. Gahagan. 10x10: A General-purpose Architectural Approach to Heterogeneity and Energy Efficiency. *Procedia Computer Science*, 4:1987 – 1996, 2011.

[69] A. A. Chien, T. Thanh-Hoang, D. Vasudevan, Y. Fang, and A. Shambayati. 10x10: A Case Study in Highly-Programmable and Energy-Efficient Heterogeneous Federated Architecture. *SIGARCH Comput. Archit. News*, 43(3):2–9, December 2015.

[70] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer. Diderot: A Parallel DSL for Image Analysis and Visualization. *SIGPLAN Not.*, 47(6):111–120, June 2012.

[71] J. W. Choi, A. Singh, and R. W. Vuduc. Model-Driven Autotuning of Sparse Matrix-Vector Multiply on GPUs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[72] I. Christadler, G. Erbacci, and A. D. Simpson. *Facing the Multicore - Challenge II: Aspects of New Paradigms and Technologies in Parallel Computing*, chapter Performance and Productivity of New Programming Languages, pages 24–35. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[73] M. Christen, O. Schenk, and H. Burkhart. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2011.

[74] G. Chrysos. Intel Xeon Phi Coprocessor (Codename Knights Corner). In *Hot Chips Symposium*, 2012.

[75] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

[76] R. Clint Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1):3–35, 2001.

[77] U. Consortium. UPC language specifications v1.2. *Lawrence Berkeley National Laboratory*, 2005.

[78] D. Cunningham, R. Bordawekar, and V. Saraswat. GPU Programming in a High Level Language: Compiling X10 to CUDA. In *ACM SIGPLAN X10 Workshop (X10)*, pages 8:1–8:10, 2011.

[79] M. Daga, A. M. Aji, and W. Feng. On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. In *Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*, 2011.

[80] M. Daga and W. Feng. Multi-Dimensional Characterization of Electrostatic Surface Potential Computation on Graphics Processors. *BMC Bioinformatics*, 13:1–12, 2012.

[81] M. Daga, Z. S. Tschirhart, and C. Freitag. Exploring Parallel Programming Models for Heterogeneous Computing Systems. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2015.

[82] L. Dagum and R. Menon. OpenMP: an Industry Standard API for Shared-Memory Programming. *IEEE Computational Science Engineering*, 5(1):46–55, 1998.

[83] S. Dalton, S. Baxter, D. Merrill, L. Olson, and M. Garland. Optimizing Sparse Matrix Operations on GPUs Using Merge Path. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2015.

[84] 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 *Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, 2010.

[85] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil Computation Optimization and Auto-Tuning on State-of-the-aAt Multicore Architectures. In *ACM/IEEE International Conference on Supercomputing (SC)*, 2008.

[86] A. Davidson and J. Owens. Toward Techniques for Auto-tuning GPU Algorithms. In *Applied Parallel and Scientific Computing*, volume 7134 of *Lecture Notes in Computer Science*, pages 110–119. Springer Berlin Heidelberg, 2012.

[87] J. Davison de St.Germain, J. McCorquodale, S. Parker, and C. Johnson. Uintah: a Massively Parallel Problem Solving Environment. In *International Symposium on High-Performance Distributed Computing (HPDC)*, 2000.

[88] P. de Oliveira Castro, Y. Kashnikov, C. Akel, M. Popov, and W. Jalby. Fine-Grained Benchmark Subsetting for System Selection. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2014.

[89] C. del Mundo and W. Feng. Towards a Performance-portable FFT Library for Heterogeneous Computing. In *ACM International Conference on Computing Frontiers (CF)*, 2014.

[90] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, October 1974.

[91] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[92] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A Dynamic Optimization Framework for Bulk-synchronous Applications in Heterogeneous Systems. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[93] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur. Hybrid Parallel Programming with MPI and Unified Parallel C. In *ACM International Conference on Computing Frontiers (CF)*, 2010.

[94] 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 Comput.*, 38(8):391–407, Aug. 2012.

[95] J. Duato, A. J. Pea, F. Silla, R. Mayo, and E. S. Quintana-Ort. rCUDA: Reducing the Number of GPU-Based Accelerators in High Performance Clusters. In *International Conference on High Performance Computing and Simulation (HPCS)*, 2010.

[96] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a High-level Language for GPUs: (via Language Support for Architectures and Compilers). In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[97] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a High-level Language for GPUs: (via Language Support for Architectures and Compilers). In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[98] A. Dubey, K. Antypas, M. K. Ganapathy, L. B. Reid, K. Riley, D. Sheeler, A. Siegel, and K. Weide. Extensible Component-Based Architecture for FLASH, a Massively Parallel, Multiphysics Simulation Code. *Parallel Computing*, 35(10):512–522, 2009.

[99] A. Duran, E. Ayugade, R. M. Badia, J. Labarta, L. Matrinell, X. Martorell, and J. Planas. OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[100] 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*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[101] EEMBC Benchmarks Collection. http://www.eembc.org/benchmark/products.php.

[102] V. K. Elangovan, R. M. Badia, and E. A. Parra. *Languages and Compilers for Parallel Computing: 25th International Workshop, LCPC 2012, Tokyo, Japan, September 11-13, 2012, Revised Selected Papers*, chapter OmpSs-OpenCL Programming Model for Heterogeneous Systems, pages 96–111. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[103] J. Enmyren and C. W. Kessler. SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In *International Workshop on High-level Parallel Programming and Applications (HLPP)*, 2010.

[104] J. Fang, A. L. Varbanescu, and H. Sips. A Comprehensive Performance Comparison of CUDA and OpenCL. In *International Conference on Parallel Processing (ICPP)*, 2011.

[105] A. Feng and W. Feng. Parallel Programming with Pictures in a Snap! In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016.

[106] W. Feng, W. Dally, M. Houston, T. Mattson, F. Petrini, and S. Wallach. Panel: On the Three P's of Heterogeneous Computing: Performance, Power and Programmability. In *ACM/IEEE International Conference on High-Performance Computing, Networking, Storage, and Analysis (SC)*, 2010.

[107] W. Feng, H. Lin, T. Scogland, and J. Zhang. OpenCL and the 13 Dwarfs: A Work in Progress. In *ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2012.

[108] M. Frigo and S. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[109] Q. Fu and K. N. Liou. Parameterization of the Radiative Properties of Cirrus Clouds. *Journal of the Atmospheric Sciences*, 50(13):2008–2025, 1993.

[110] S. Gao and J. Chritz. Characterization of OpenCL on a Scalable FPGA Architecture. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2014.

[111] D. Garcia, L. Segars, and J. Paley. Snap! (Build Your Own Blocks): Tutorial Presentation. *J. Comput. Sci. Coll.*, 27(4):120–121, Apr. 2012.

[112] M. Gardner, P. Sathre, W. Feng, and G. Martinez. Characterizing the Challenges and Evaluating the Efficacy of a CUDA-to-OpenCL Translator. *Parallel Computing*, October 2013.

[113] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally. Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[114] V. George, T. Piazza, and H. Jiang. Technology Insight: Intel® Next Generation Microarchitecture Codename Ivy Bridge. In *Intel Developer Forum*, 2011.

[115] T. Goodale, G. Allen, G. Lanfermann, J. Masso, T. Radke, E. Seidel, and J. Shalf. The Cactus Framework and Toolkit: Design and Applications. In *High Performance Computing for Computational Science - VECPAR 2002*, volume 2565 of *Lecture Notes in Computer Science*, pages 197–227. Springer Berlin Heidelberg, 2003.

[116] J. C. Gordon, A. T. Fenley, and A. Onufriev. An Analytical Approach to Computing Biomolecular Electrostatic Potential. II. Validation and Applications. *The Journal of Chemical Physics*, 129(7), Aug. 2008.

[117] N. Goswami, R. Shankar, M. Joshi, and T. Li. Exploring GPGPU Workloads: Characterization Methodology, Analysis and Microarchitecture Evaluation Implications. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2010.

[118] I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer. LibWater: Heterogeneous Distributed Computing Made Easy. In *ACM International Conference on Supercomputing (ICS)*, 2013.

[119] R. Graybill. Relevance of Computing Beyond Desktop in Today's Challenging Economic Times, October 2008. USC Information Sciences Institute and Council on Competitiveness.

[120] P. Greenhalgh. Big.little Processing with ARM Cortex-A15 & Cortex-A7. *ARM White Paper*, pages 1–8, 2011.

[121] T. Grosser and T. Hoefler. Polly-ACC Transparent Compilation to Heterogeneous Hardware. In *ACM International Conference on Supercomputing (ICS)*, 2016.

[122] A. Guha, Y. Zhang, R. ur Rasool, and A. A. Chien. Systematic Evaluation of Workload Clustering for Extremely Energy-efficient Architectures. *SIGARCH Comput. Archit. News*, 41(2):22–29, May 2013.

[123] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng. Twin Peaks: A Software Platform for Heterogeneous Computing on General-purpose and Graphics Processors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[124] J. Guo, G. Bikshandi, B. B. Fraguela, M. J. Garzaran, and D. Padua. Programming with Tiles. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[125] A. Gupta, L. V. Kalé, D. S. Milojicic, P. Faraboschi, R. Kaufmann, V. March, F. Gioachin, C. H. Suen, and B.-S. Lee. Exploring the Performance and Mapping of HPC Applications to Platforms in the Cloud. In *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2012.

[126] A. Haidar, J. Dongarra, K. Kabir, M. Gates, P. Luszczek, S. Tomov, and Y. Jia. HPC Programming on Intel Many-Integrated-Core Hardware with MAGMA Port to Xeon Phi. *Scientific Programming*, 23, 01-2015 2015.

[127] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The Fourth-Generation Intel Core Processor. *IEEE Micro*, 34(2):6–20, March 2014.

[128] T. D. Han and T. S. Abdelrahman. hiCUDA: A High-level Directive-based Language for GPU Programming. In *ACM International Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, 2009.

[129] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson. How Do Scientists Develop and Use Scientific Software? In *ICSE Workshop on Software Engineering for Computational Science and Engineering (SECSE)*, 2009.

[130] J. He, A. E. Snavely, R. F. V. d. Wijngaart, and M. A. Frumkin. Automatic Recognition of Performance Idioms in Scientific Applications. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2011.

[131] A. E. Helal, P. Sathre, and W. Feng. MetaMorph: A Library Framework for Interoperable Kernels on Multi- and Many-core Clusters. In *IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[132] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture News*, 34(4), 2006.

[133] O. Hernandez, W. Ding, B. Chapman, C. Kartsaklis, R. Sankaran, and R. Graham. *Experiences with High-Level Programming Directives for Porting Applications to GPUs*, pages 96–107. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[134] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An Overview of the Trilinos Project. *ACM Trans. Math. Softw.*, 31(3):397–423, Sept. 2005.

[135] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving Performance via Mini-Applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[136] J. Hestness, S. W. Keckler, and D. A. Wood. GPU Computing Pipeline Inefficiencies and Optimization Opportunities in Heterogeneous CPU-GPU Processors. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2015.

[137] P. Hijma, C. J. H. Jacobs, R. V. v. Nieuwpoort, and H. E. Bal. Cashmere: Heterogeneous Many-Core Computing. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2015.

[138] P. Hijma, R. V. van Nieuwpoort, C. J. H. Jacobs, and H. E. Bal. Stepwise-Refinement for Performance: a Methodology for Many-Core Programming. *Concurrency and Computation: Practice and Experience*, 27(17):4515–4554, 2015. cpe.3416.

[139] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, July 2008.

[140] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, and M. V. Zelkowitz. Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers. In *ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2005.

[141] T. Hoefler and R. Belli. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses when Reporting Performance Results. In *ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[142] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-Performance Code Generation for Stencil Computations on GPU Architectures. In *ACM International Conference on Supercomputing (ICS)*, 2012.

[143] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. MapCG: Writing Parallel Program Portable Between CPU and GPU. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[144] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[145] R. Hornung and J. Keasler. The RAJA Portability Layer: Overview and Status. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2014.

[146] K. Hou, H. Wang, and W. Feng. ASPaS: A Framework for Automatic SIMDization of Parallel Sorting on x86-based Many-core Processors. In *ACM International Conference on Supercomputing (ICS)*, 2015.

[147] CUDALink. https://reference.wolfram.com/language/CUDALink/guide/CUDALink.html.

[148] OpenCLLink. https://reference.wolfram.com/language/OpenCLLink/guide/OpenCLLink.html.

[149] Parallel Computing Toolbox. https://www.mathworks.com/products/parallel-computing/.

[150] S. Huang, A. Hormati, D. Bacon, and R. Rabbah. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In *ECOOP 2008 - Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 76–103. Springer, 2008.

[151] N. Instruments. LabVIEW FPGA Module. `http:http://www.ni.com/labview/fpga/`.

[152] Intel. AP-803: Increasing the Accuracy of the Results from the Reciprocal and Reciprocal Square Root Instructions using the Newton-Raphson Method. 1999.

[153] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):186–197, June 2007.

[154] K. E. Iverson. A Programming Language. In *ACM Spring Joint Computer Conference*, 1962.

[155] N. Jain, A. Bhatele, J. S. Yeom, M. F. Adams, F. Miniati, C. Mei, and L. V. Kale. Charm++ and MPI: Combining the Best of Both Worlds. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2015.

[156] W. Jia, K. A. Shaw, and M. Martonosi. Characterizing and Improving the Use of Demand-fetched Caches in GPUs. In *ACM International Conference on Supercomputing (ICS)*, 2012.

[157] W. Jia, K. A. Shaw, and M. Martonosi. Starchart: Hardware and Software Optimization Using Recursive Partitioning Regression Trees. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[158] X. Jiao, M. T. Campbell, and M. T. Heath. Roccom: An Object-Oriented, Data-Centric Software Integration Framework for Multiphysics Simulations. In *ACM International Conference on Supercomputing (ICS)*, 2003.

[159] D. A. Joiner, P. Gray, T. Murphy, and C. Peck. Teaching Parallel Computing to Science Faculty: Best Practices and Common Pitfalls. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2006.

[160] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John. Measuring Benchmark Similarity Using Inherent Program Characteristics. *IEEE Trans. Comput.*, 55(6):769–782, June 2006.

[161] G. Juckeland, W. Brantley, S. Chandrasekaran, B. Chapman, S. Che, M. Colgrove, H. Feng, A. Grund, R. Henschel, W.-M. W. Hwu, H. Li, M. S. Müller, W. E. Nagel, M. Perminov, P. Shelepugin, K. Skadron, J. Stratton, A. Titov, K. Wang, M. Waveren, B. Whitney, S. Wienke, R. Xu, and K. Kumaran. *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers*, chapter SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance, pages 46–67. Springer International Publishing, Cham, 2015.

[162] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, July 2005.

[163] L. Kalé, R. Skeel, M. Bh, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten. NAMD2: Greater Scalability for Parallel Molecular Dynamics. *Journal of Computational Physics*, 151:283–312, 1999.

[164] L. V. Kale and S. Krishnan. *CHARM++: a Portable Concurrent Object-Oriented System Based on C++*, volume 28. ACM, 1993.

[165] N. Kapre and S. Bayliss. Survey of Domain-Specific Languages for FPGA Computing. In *International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

[166] K. Keutzer, B. L. Massingill, T. G. Mattson, and B. A. Sanders. A Design Pattern Language for Engineering (Parallel) Software: Merging the PLPP and OPL Projects. In *International Workshop on Parallel Programming Patterns (ParaPLoP)*, 2010.

[167] Keysight. SystemVue. `http://www.keysight.com/en/pc-1297131/systemvue-electronic-system-level-esl-design-software`.

[168] J. Kim, T. T. Dao, J. Jung, J. Joo, and J. Lee. Bridging OpenCL and CUDA: A Comparative Analysis and Translation. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[169] J. Kim, S. Lee, and J. S. Vetter. IMPACC: A Tightly Integrated MPI+OpenACC Framework Exploiting Shared Memory Parallelism. In *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2016.

[170] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *ACM International Conference on Supercomputing (ICS)*, 2012.

[171] P. M. Kogge and T. J. Dysart. Using the TOP500 to Trace and Project Technology and Architecture Trends. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[172] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. When Polyhedral Transformations Meet SIMD Code Generation. *SIGPLAN Not.*, 48(6):127–138, June 2013.

[173] D. Koufaty, D. Reddy, and S. Hahn. Bias Scheduling in Heterogeneous Multi-core Architectures. In *European Conference on Computer Systems (EuroSys)*, 2010.

[174] K. Krommydas and W. Feng. Telescoping Architectures: Evaluating Next-Generation Heterogeneous Computing. In *IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2016.

[175] K. Krommydas, W. Feng, C. D. Antonopoulos, and N. Bellas. Opendwarfs: Characterization of dwarf-based benchmarks on fixed and reconfigurable architectures. *Journal of Signal Processing Systems*, pages 1–20, 2015.

[176] K. Krommydas, W. Feng, M. Owaida, C. Antonopoulos, and N. Bellas. On the Characterization of OpenCL Dwarfs on Fixed and Reconfigurable Platforms. In *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 153–160, 2014.

[177] K. Krommydas, A. E. Helal, A. Verma, and W. Feng. Bridging the Performance-Programmability Gap for FPGAs via OpenCL: A Case Study with OpenDwarfs. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016.

[178] K. Krommydas, R. Sasanka, and W. Feng. GLAF: A Visual Programming and Auto-tuning Framework for Parallel Computing. In *International Conference on Parallel Processing (ICPP)*, 2015.

[179] K. Krommydas, R. Sasanka, and W. Feng. Bridging the FPGA Programmability-Portability Gap via Automatic OpenCL Code Generation and Tuning. In *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2016.

[180] K. Krommydas, T. Scogland, and W. Feng. On the Programmability and Performance of Heterogeneous Platforms. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2013.

[181] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *International Symposium on Computer Architecture (ISCA)*, 2004.

[182] D. M. Kunzman and L. V. Kalé. Towards a Framework for Abstracting Accelerators in Parallel Applications: Experience with Cell. In *ACM/IEEE International Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2009.

[183] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis Transformation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2004.

[184] K. Lee, H. Lin, and W. Feng. Performance Characterization of Data-intensive Kernels on AMD Fusion Architectures. *Computer Science - Research and Development*, 28(2-3), 2013.

[185] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu. Improving GPGPU Resource Utilization through Alternative Thread Block Scheduling. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[186] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[187] S. Lee, J. Kim, and J. S. Vetter. OpenACC to FPGA: A Framework for Directive-Based High-Performance Reconfigurable Computing. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2016.

[188] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2009.

[189] S. Lee and J. S. Vetter. Early Evaluation of Directive-based GPU Programming Models for Productive Exascale Computing. In *ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[190] S. Lee and J. S. Vetter. OpenARC: Open Accelerator Research Compiler for Directive-Based, Efficient Heterogeneous Computing. In *International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, 2014.

[191] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU Myth: an Evaluation of Throughput Computing on CPU and GPU. In *International Symposium on Computer Architecture (ISCA)*, 2010.

[192] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The ALPBench Benchmark Suite for Complex Multimedia Applications. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2005.

[193] Y. Li, Y. Zhang, H. Jia, G. Long, and K. Wang. Automatic FFT Performance Tuning on OpenCL GPUs. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2011.

[194] S. K. Lim. Physical design for 3D system on package. *IEEE Design Test of Computers*, 22(6):532–539, November 2005.

[195] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: A Programming Model for Heterogeneous Multi-Core Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[196] H. Liu and H. H. Huang. Enterprise: Breadth-First Graph Traversal on GPUs. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[197] J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero. Swing Modulo Scheduling: A Lifetime-Sensitive Approach. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2006.

[198] M. G. Lopez, J. Young, J. S. Meredith, P. C. Roth, M. Horton, and J. S. Vetter. Examining Recent Many-core Architectures and Programming Models Using SHOC. In *International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems (PMBS)*, 2015.

[199] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke. Composite Cores: Pushing Heterogeneity Into a Core. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[200] Y. Luo, G. Tan, Z. Mo, and N. Sun. FAST: A Fast Stencil Autotuning Framework Based On An Optimal-Solution Space Model. In *ACM International Conference on Supercomputing (ICS)*, 2015.

[201] D. Majeti and V. Sarkar. Heterogeneous Habanero-C (H2C): A Portable Programming Model for Heterogeneous Processors. In *IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, 2015.

[202] J. Makino and H. Daisaka. GRAPE-8: An Accelerator for Gravitational N-Body Simulation with 20.5Gflops/W Performance. In *ACM/IEEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[203] A. Mametjanov, D. Lowell, C. C. Ma, and B. Norris. Autotuning Stencil-Based Computations on GPUs. In *IEEE International Conference on Cluster Computing (CLUSTER)*, 2012.

[204] M. Martineau, S. McIntosh-Smith, M. Boulton, and W. Gaudin. An Evaluation of Emerging Many-Core Parallel Programming Models. In *International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*, 2016.

[205] G. Martinez, M. Gardner, and W. Feng. CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2011.

[206] MathWorks. Matlab HDL Coder. `http://www.mathworks.com/products/hdl-coder/`.

[207] T. J. McCabe. A Complexity Measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[208] W. Meeus, K. Van Beeck, T. Goedem, J. Meel, and D. Stroobandt. An Overview of Today's High-Level Synthesis Tools. *Design Automation for Embedded Systems*, 16(3):31–51, 2012.

[209] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh. Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-Stacked and Off-Package Memories. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[210] J. Milthorpe, V. Ganesh, A. P. Rendell, and D. Grove. X10 as a Parallel Language for Scientific Computation: Practice and Experience. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2011.

[211] Mirabilis. VisualSim. `http://mirabilisdesign.com/new/visualsim/`.

[212] G. Misra, N. Kurkure, A. Das, M. Valmiki, S. Das, and A. Gupta. Evaluation of Rodinia Codes on Intel Xeon Phi. In *International Conference on Intelligent Systems, Modelling and Simulation*, 2013.

[213] V. M. Morales, P.-H. Horrein, A. Baghdadi, E. Hochapfel, and S. Vaton. Energy-Efficient FPGA Implementation for Binomial Option Pricing Using OpenCL. In *Conference on Design, Automation & Test in Europe (DATE)*, 2014.

[214] P. Mougin and S. Ducasse. OOPAL: Integrating Array Programming in Object-oriented Programming. In *ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications (OOPSLA)*, 2003.

[215] D. Mustafa and R. Eigenmann. Portable Section-Level Tuning of Compiler Parallelized Applications. In *ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[216] M. Nakao, J. Lee, T. Boku, and M. Sato. Productivity and Performance of Global-View Programming with XcalableMP PGAS Language. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2012.

[217] T. Narumi, K. Yasuoka, M. Taiji, F. Zerbetto, and S. Höfinger. Fast Calculation of Electrostatic Potentials on the GPU or the ASIC MD-GRAPE-3. *The Computer Journal*, 54(7).

[218] NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html.

[219] R. Nath, S. Tomov, T. T. Dong, and J. Dongarra. Optimizing Symmetric Dense Matrix-Vector Multiplication on GPUs. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[220] G. Ndu, J. Navaridas, and M. Luján. CHO: Towards a Benchmark Suite for OpenCL FPGA Accelerators. In *International Workshop on OpenCL (IWOCL)*, 2015.

[221] R. Nikhil. Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications. In *ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, 2004.

[222] A. Nukada, K. Sato, and S. Matsuoka. Scalable Multi-GPU 3-D FFT for TSUBAME 2.0 Supercomputer. In *ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[223] K. Oka, W. Jia, M. Martonosi, and K. Inoue. Characterization and Cross-Platform Analysis of High-Throughput Accelerators. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015.

[224] S. Olivier, J. Prins, J. Derby, and K. Vu. Porting the GROMACS Molecular Dynamics Code to the Cell Processor. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2007.

[225] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos. Synthesis of Platform Architectures from OpenCL Programs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2011.

[226] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil. Fast and Efficient Automatic Memory Management for GPUs Using Compiler-assisted Runtime Coherence Scheme. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[227] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu. FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs. In *IEEE Symposium on Application Specific Processors (SASP)*, 2009.

[228] M. Paredes, G. Riley, and M. Luján. Breadth-First Search Vectorization on the Intel Xeon Phi. In *ACM International Conference on Computing Frontiers (CF)*, 2016.

[229] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. A. Jarvis. Exploring SIMD for Molecular Dynamics, Using Intel Xeon Processors and Intel Xeon Phi Coprocessors. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2013.

[230] A. Phansalkar, A. Joshi, and L. K. John. Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite. In *International Symposium on Computer Architecture (ISCA)*, 2007.

[231] C. Pheatt. Intel Threading Building Blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[232] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe. Portable Performance on Heterogeneous Architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[233] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical Task-Based Programming With StarSs. *International Journal on High Performance Computing Applications*, 23(3):284–299, Aug. 2009.

[234] F. J. Pollack. New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies (Keynote Address). In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 1999.

[235] P. Prabhu, T. B. Jablin, A. Raman, Y. Zhang, J. Huang, H. Kim, N. P. Johnson, F. Liu, S. Ghosh, S. Beard, T. Oh, M. Zoufaly, D. Walker, and D. I. August. A Survey of the Practice of Computational Science. In *State of the Practice Reports - ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[236] A. Prasad, J. Anantpur, and R. Govindarajan. Automatic Compilation of MATLAB Programs for Synergistic Execution on Heterogeneous Processors. *SIGPLAN Not.*, 46(6):152–163, June 2011.

[237] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Xiao, and D. Burger. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2014.

[238] N. Ravi, Y. Yang, T. Bao, and S. Chakradhar. Apricot: An Optimizing Compiler and Productivity Tool for x86-compatible Many-Core Coprocessors. In *ACM International Conference on Supercomputing (ICS)*, 2012.

[239] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: Programming for All. *Commun. ACM*, 52(11):60–67, Nov. 2009.

[240] J. V. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikant, M. D. Tholburn, et al. POOMA: A Framework for Scientific Simulations on Parallel Architectures. *Parallel Programming in C+*, pages 547–588, 1996.

[241] A. D. Robison. Composable Parallel Patterns with Intel Cilk Plus. *Computing in Science and Engineering*, 15(2):66–71, 2013.

[242] C. Rodrigues, T. Jablin, A. Dakkak, and W.-M. Hwu. Triolet: A Programming System That Unifies Algorithmic Skeleton Interfaces for High-Performance Cluster Computing. *SIGPLAN Not.*, 49(8):247–258, Feb. 2014.

[243] C. I. Rodrigues, D. J. Hardy, J. E. Stone, K. Schulten, and W.-M. W. Hwu. GPU Acceleration of Cutoff Pair Potentials for Molecular Modeling Applications. In *ACM International Conference on Computing Frontiers (CF)*, 2008.

[244] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[245] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications? In *International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 2012.

[246] T. Scogland, W. Feng, B. Rountree, and B. de Supinski. CoreTSAR: Core Task-Size Adapting Runtime. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–1, 2014.

[247] T. R. W. Scogland and W. Feng. Runtime Adaptation for Autonomic Heterogeneous Computing. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2014.

[248] S. Seo, G. Jo, and J. Lee. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2011.

[249] S. O. Settle. High-Performance Dynamic Programming on FPGAs with OpenCL. In *IEEE High Performance Extreme Computing Conference (HPEC)*, 2013.

[250] D. E. Shaw, J. P. Grossman, J. A. Bank, B. Batson, J. A. Butts, J. C. Chao, M. M. Deneroff, R. O. Dror, A. Even, C. H. Fenton, A. Forte, J. Gagliardo, G. Gill, B. Greskamp, C. R. Ho, D. J. Ierardi, L. Iserovich, J. S. Kuskin, R. H. Larson, T. Layman, L.-S. Lee, A. K. Lerer, C. Li, D. Killebrew, K. M. Mackenzie, S. Y.-H. Mok, M. A. Moraes, R. Mueller, L. J. Nociolo, J. L. Peticolas, T. Quan, D. Ramot, J. K. Salmon, D. P. Scarpazza, U. Ben Schafer, N. Siddique, C. W. Snyder, J. Spengler, P. T. P. Tang, M. Theobald, H. Toma, B. Towles, B. Vitale, S. C. Wang, and C. Young. Anton 2: Raising the Bar for Performance and Programmability in a Special-Purpose Molecular Dynamics Supercomputer. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[251] C.-Y. Shei, P. Ratnalikar, and A. Chauhan. Automating GPU Computing in MATLAB. In *ACM International Conference on Supercomputing (ICS)*, 2011.

[252] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. HASS: A Scheduler for Heterogeneous Multicore Systems. *SIGOPS Oper. Syst. Rev.*, 43(2):66–75, Apr. 2009.

[253] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu. Performance Gaps between OpenMP and OpenCL for Multi-core CPUs. In *International Conference on Parallel Processing Workshops (ICPPW)*, 2012.

[254] G. Shi and V. Kindratenko. Implementation of NAMD Molecular Dynamics Non-Bonded Force-Field on the Cell Broadband Engine Processor. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, April 2008.

[255] A. Sidelnik, S. Maleki, B. L. Chamberlain, M. J. Garzar'n, and D. Padua. Performance Portability with the Chapel Language. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2012.

[256] S. E. Sim, S. Easterbrook, and R. C. Holt. Using Benchmarking to Advance Research: a Challenge to Software Engineering. In *International Conference on Software Engineering (ICSE '03)*, 2003.

[257] D. P. Singh, T. S. Czajkowski, and A. Ling. Harnessing the Power of FPGAs Using Altera's OpenCL Compiler. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2013.

[258] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken. Regent: A High-productivity Programming Language for HPC with Logical Regions. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[259] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A Framework for Performance Modeling and Prediction. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2002.

[260] M. Snir. Programming Models for High-Performance Computing. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2013.

[261] E. Sotiriades and A. Dollas. A General Reconfigurable Architecture for the BLAST Algorithm. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 48(3):189–208, September 2007.

[262] K. L. Spafford, J. S. Meredith, S. Lee, D. Li, P. C. Roth, and J. S. Vetter. The Tradeoffs of Fused Memory Hierarchies in Heterogeneous Computing Architectures. In *ACM International Conference on Computing Frontiers (CF)*, 2012.

[263] SPEC Benchmarks Collection. https://www.spec.org/benchmarks.html.

[264] S. Sridharan, G. Gupta, and G. S. Sohi. Adaptive, Efficient, Parallel Execution of Parallel Programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[265] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, 2011.

[266] J. E. Stone, D. J. Hardy, I. S. Ufimtsev, and K. Schulten. GPU-Accelerated Molecular Modeling Coming of Age. *Journal of Molecular Graphics & Modelling*, 29(2):116–125, Sept. 2010.

[267] J. A. Stratton, N. Anssari, C. Rodrigues, I. J. Sung, N. Obeid, L. Chang, G. D. Liu, and W. m. Hwu. Optimization and Architecture Effects on GPU Computing Workload Performance. In *Innovative Parallel Computing Conference (InPar)*, 2012.

[268] J. A. Stratton, H.-S. Kim, T. B. Jablin, and W.-M. W. Hwu. Performance Portability in Accelerated Parallel Kernels. Technical Report IMPACT-13-01, University of Illinois at Urbana-Champaign, Urbana, May 2013.

[269] 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. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, 2012.

[270] J. A. Stratton, S. S. Stone, and W.-m. W. Hwu. *MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs*, pages 16–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[271] A. Stromme, R. Carlson, and T. Newhall. Chestnut: A GPU Programming Language for Non-Experts. In *International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*, 2012.

[272] B.-Y. Su and K. Keutzer. clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs. In *ACM International Conference on Supercomputing (ICS)*, 2012.

[273] W. T. Tang, R. Zhao, M. Lu, Y. Liang, H. P. Huynh, X. Li, and R. S. M. Goh. Optimizing and Auto-Tuning Scale-Free Sparse Matrix-Vector Multiplication on Intel Xeon Phi. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015.

[274] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A Framework for Adaptive Algorithm Selection in STAPL. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005.

[275] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense Linear Algebra Solvers for Multicore with GPU Accelerators. In *IEEE International Symposium on Distributed and Parallel Processing (IPDPS)*, Atlanta, GA, 2010.

[276] Top 500 Supercomputer Sites. http://www.top500.org.

[277] D. Unat, X. Cai, and S. B. Baden. Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C. In *ACM International Conference on Supercomputing (ICS)*, 2011.

[278] S. van der Walt, S. Colbert, and G. Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science Engineering*, 13(2):22–30, 2011.

[279] R. V. Van Nieuwpoort, G. Wrzesińska, C. J. H. Jacobs, and H. E. Bal. Satin: A High-level and Efficient Grid Programming Model. *ACM Trans. Program. Lang. Syst.*, 32(3):9:1–9:39, Mar. 2010.

[280] C. Wang, S. Chandrasekaran, and B. Chapman. cusFFT: A High-Performance Sparse Fast Fourier Transform Algorithm on GPUs. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2016.

[281] M. Wang and M. Parashar. Object-Oriented Stream Programming using Aspects. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010.

[282] S. Wienke, P. Springer, C. Terboven, and D. an Mey. OpenACC: First Experiences with Real-World Applications. In *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012.

[283] S. Wienke, C. Terboven, J. C. Beyer, and M. S. Müller. *International Conference Euro-Par 2014 Parallel Processing*, chapter A Pattern-Based Comparison of OpenACC and OpenMP for Accelerator Computing. 2014.

[284] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2007.

[285] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.

[286] M. Wolfe. Implementing the PGI Accelerator Model. In *International Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, 2010.

[287] Y. Wu, Y. Wang, Y. Pan, C. Yang, and J. D. Owens. Performance Characterization of High-Level Programming Models for GPU Graph Analytics. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2015.

[288] S. Yan, C. Li, Y. Zhang, and H. Zhou. yaSpMV: Yet Another SpMV Framework on GPUs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2014.

[289] Y. Yan, P.-H. Lin, C. Liao, B. R. de Supinski, and D. J. Quinlan. Supporting Multiple Accelerators in High-level Programming Models. In *International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*, 2015.

[290] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In *International Conference on Languages and Compilers for Parallel Computing (LCPC)*, 2010.

[291] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU Compiler for Memory Optimization and Parallelism Management. *SIGPLAN Not.*, 45(6):86–97, June 2010.

[292] Y. Zhang and F. Mueller. Auto-generation and Auto-tuning of 3D Stencil Codes on GPU Clusters. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2012.

[293] Y. Zhang, M. Sinclair, and A. A. Chien. *International Supercomputing Conference (ISC)*, chapter Improving Performance Portability in OpenCL Programs. 2013.