RICE UNIVERSITY

Productive Programming Systems for Heterogeneous Supercomputers

by

Max Grossman

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Vivek Sarkar, Chair
Professor of Computer Science
E.D. Butcher Chair in Engineering

John Mellor-Crummey
Professor of Computer Science and of
Electrical and Computer Engineering

Ray Simar
Professor in the Practice of Electrical and
Computer Engineering

Houston, Texas
December, 2016

ABSTRACT

Productive Programming Systems for Heterogeneous Supercomputers

by

Max Grossman

The majority of today's scientific and data analytics workloads are still run on relatively energy inefficient, heavyweight, general-purpose processing cores, often referred to in the literature as latency-oriented architectures. The flexibility of these architectures and the programmer aids included (e.g. large and deep cache hierarchies, branch prediction logic, pre-fetch logic) makes them flexible enough to run a wide range of applications fast. However, we have started to see growth in the use of lightweight, simpler, energy-efficient, and functionally constrained cores. These architectures are commonly referred to as throughput-oriented.

Within each shared memory node, the computational backbone of future throughput-oriented HPC machines will consist of large pools of lightweight cores. The first wave of throughput-oriented computing came in the mid 2000's with the use of GPUs for general-purpose and scientific computing. Today we are entering the second wave of throughput-oriented computing, with the introduction of NVIDIA Pascal GPUs, Intel Knights Landing Xeon Phi processors, the Epiphany Co-Processor, the Sunway MPP, and other throughput-oriented architectures that enable pre-exascale computing. However, while the majority of the FLOPS in designs for future HPC systems come from throughput-oriented architectures, they are still commonly paired with latency-oriented cores which handle management functions and lightweight/unparallelizable computational kernels. Hence, most future HPC machines will be heterogeneous in their processing cores.

However, the heterogeneity of future machines will not be limited to the processing elements. Indeed, heterogeneity will also exist in the storage, networking, memory, and software stacks of future supercomputers. As a result, it will be necessary to combine many different programming models and libraries in a single application. How to do so in a programmable and well-performing manner is an open research question. This thesis addresses this question using two approaches.

First, we explore using managed runtimes on HPC platforms. As a result of their high-level programming models, these managed runtimes have a long history of supporting data analytics workloads on commodity hardware, but often come with overheads which make them less common in the HPC domain. Managed runtimes are also not supported natively on throughput-oriented architectures.

Second, we explore the use of a modular programming model and work-stealing runtime to compose the programming and scheduling of multiple third-party HPC libraries. This approach leverages existing investment in HPC libraries, unifies the scheduling of work on a platform, and is designed to quickly support new programming model and runtime extensions.

In support of these two approaches, this thesis also makes novel contributions in tooling for future supercomputers. We demonstrate the value of checkpoints as a software development tool on current and future HPC machines, and present novel techniques in performance prediction across heterogeneous cores.

Acknowledgments

First, emphatic thanks have to go to my advisor, Professor Vivek Sarkar. His generosity and mentorship have had a huge impact on me for nearly the past decade. When I decided to return to school to earn a graduate degree as a member of the Habanero group, his immediate and unfaltering support of that was all the confirmation I needed that it was the right decision.

Second, I have to thank my mother, father, and extended family. The emphasis on education, happiness, and discipline that was instilled in me from an early age has undoubtedly been invaluable in pursuing this degree.

Third, I would like to thank Professor John Mellor-Crummey and Professor Ray Simar for their questions, feedback, and criticisms throughout my thesis proposal and defense. Their contributions have measurably improved the clarity and content of this dissertation.

Fourth, thank you to the members of the Habanero group, to collaborators outside of it, and to my friends. This accomplishment would not have been possible without the technical support, emotional support, and margaritas you have supplied.

There's a difference in life between things that are scary and things that are dangerous... There are plenty of things that are scary but not dangerous... Staying [in industry] was dangerous but not scary. The danger there, the risk of it was continuing to do something that didn't make me happy, and getting to 65 and looking back and going "Oh My God, I wasted my life". That is risk, that is danger.

- Jim Koch, Founder of Samuel Adams

Contents

Illustrations

Chapter 1

Introduction

According to the June 2016 Top 500 list [1], the top three supercomputers in the world are based on heterogeneous processing cores. Their combined sustained TFlop/s exceeds that of the next thirty machines on the list. Even more surprising, each uses a different throughput-oriented processor architecture: the Sunway TaihuLight uses the Sunway MPP [2], the Tianhe-2 uses Intel Knights Corner Xeon Phi (KNC) [3], and the Titan supercomputer uses NVIDIA Graphics Processing Units (GPU) [4]. If we look at the Top 500 list from a decade earlier, no machine in the top ten made use of accelerator cores or was heterogeneous in its processing elements. Looking forward, we also know that two of the three upcoming supercomputers funded under the United States Department of Energy's CORAL program (Summit at ORNL and Sierra at LLNL) are also heterogeneous.

From these observations we can conclude that over the last decade accelerated and heterogeneous supercomputing has demonstrated significant value for the high-performance computing (HPC) and scientific communities. The improved computational performance achievable through heterogeneous computing has enabled more scientific discovery through faster simulation times [5]. This has led to a sustained investment in the design and development of these large, heterogeneous machines by government organizations, academia, and industry. Rather than diminishing over time, that investment instead seems to be accelerating.

1.1 The Cost of Hardware Heterogeneity

The benefits of heterogeneous hardware are not without cost. In particular, programming heterogeneous HPC systems is much harder than programming homogeneous systems. This increased effort comes from five characteristics of software development on these systems:

1. **Composability**: Programming a machine with accelerators or with heterogeneous cores generally requires combining multiple programming models in a single application. For example, when programming a GPU-based supercomputer it is often necessary to combine CUDA [6] for the GPUs, OpenMP [7] for the CPU cores, and a communication library such as MPI [8] or OpenSHMEM [9] for communicating between nodes. This adds to the complexity of the code base, degrades application maintainability/portability, and adds an entirely new programming model for HPC programmers to become experts in.

2. **Tunability**: While HPC programming has always required more knowledge of hardware characteristics than many other domains of Computer Science, the energy efficiency improvements of accelerators and lightweight cores often come at the cost of removing "hardware programmer aids", e.g. deep and large cache hierarchies, branch prediction logic, pre-fetch logic, etc. With the loss of these programmer aids, an understanding of hardware characteristics becomes more important and has a magnified effect on the performance of applications. Not only that, but the characteristics of novel architectures often differ significantly from those that HPC programmers are accustomed to targeting in the past.

3. **Coherency**: While most HPC applications are designed with distribution in mind, adding accelerators magnifies the coherency challenges that arise from many discrete and explicitly managed address spaces. For example, using discrete GPUs today requires manual management of each GPU's global memory, constant memory, texture memory, and scratchpad memory.

4. **Scheduling**: Scheduling irregular workloads across homogeneous processing cores is an open research problem today. Having a heterogeneous mix of cores adds a new dimension to the scheduling problem, as different kernels may be more or less suited to different architectures.

5. **Tooling**: The debugging and profiling toolsets for HPC machines have evolved over decades to fit the needs of HPC programmers, and have naturally been designed to target current homogeneous platforms. As with any paradigm shift, the advent of accelerators and heterogeneous cores requires a re-thinking of techniques for HPC debugging and performance profiling.

In practice on today's heterogeneous supercomputers, most scientific applications use a combination of hand-coded kernels and high-level libraries to take advantage of all hardware resources. For example, an application might make use of MPI for inter-process communication/parallelism, OpenMP for intra-process parallelism, and CUDA [6], OpenACC [10], or OpenCL [11] for accelerator offload. Each of these models may be partially supplemented or entirely replaced by a library. Rather than write their own linear algebra kernels, a programmer might make use of the CUBLAS [12], CUSPARSE [13], MKL [14], or Trilinos [15] libraries.

On the other hand, distributed data analytics applications running on HPC or cloud platforms commonly forgo these programming systems for higher-level programming models that execute on managed runtimes. Today, the most common frameworks in use are Hadoop MapReduce [16] and Spark [17]. Both expose high-level, functional, but limited parallel APIs in Java or Scala and execute on the Java Virtual Machine (JVM). Both Hadoop and Spark are also often accessed through high-level, domain-specific libraries such as GraphX [18], MLlib [19], or Mahout [20].

However, efficient native execution of managed runtimes on accelerators would lead to massive performance degradation. Instead, accelerator enablement of these data analytics applications commonly takes the form of either 1) a low-level, hand-coded, labor-intensive, and application-specific integration of accelerators into an existing data analytics framework, or 2) a high-level, domain-specific library which can be configured to perform accelerator offload under the covers [21]. The hand-coded approach lacks generality and takes away the programmability benefits that data analytics frameworks commonly offer, but does enable application-specific, low-level optimizations. The domain-specific library approach lacks flexibility and programmer tunability, but offers ease-of-use.

One commonly proposed solution to productively programming distributed, heterogeneous machines is to hide this heterogeneity under domain-specific libraries or languages. While this approach may be effective for many use cases, a block-box, fixed-function solution can also be 1) overly restrictive in terms of the user optimizations enabled, and 2) the functionality offered. It is also entirely insufficient to enable novel application or algorithm development. Hence, programming heterogeneous platforms for scientific simulation and data analytics will continue to require:

1. The ability to choose from and combine multiple software components such that the right tool can be used for a given task. For example, the UPC++ [22] implementation of the HPGMG benchmark uses the one-sided, PGAS UPC++ library for sending small messages between distributed processes but also relies on MPI collectives as a more mature and well-performing implementation of collective communication patterns.

2. The ability to express and optimize application logic at the statement level such that novel approaches to real-world problems can explored outside of the constraints of a limited, domain-specific interface.

This thesis focuses on the following three principles as a solution to the problems highlighted above in heterogeneous and multi-tenant computing:

1. **Compatibility and composability with existing APIs and models.** Focusing on compatibility lowers barriers to adoption, adds interesting constraints

to the research problem, and takes advantage of the decades of development that have gone into existing programming systems. Composability is important because any reasonably complex HPC application will be a multi-tenant system, likely using multiple third-party systems together.

2. **Taking advantage of compile-time insights to improve runtime performance.** Recently, much of the HPC research community has begun shifting away from compiler-based programming systems as a result of 1) the costs associated with maintaining them, and 2) the introduction of new features in standard programming languages (e.g. C++ and Java lambdas). The creation of the UPC++ library from the UPC language is an example of this trend. This shift towards library-based approaches has placed the burden of efficient scheduling solely on runtime systems. However, compile-time information and source-to-source transformations can often be useful for improving runtime performance. Therefore, the use of static tools to supplement library-based approaches offers a promising path for well-performing programming systems.

3. **Integrated scheduling on resource- and workload-aware runtimes.** While discrete, disjoint systems are desirable from a software engineering perspective, their lack of awareness of each other presents challenges for efficient scheduling. Often, overly strict synchronization is used to ensure correctness when algorithmic dependencies cross software component boundaries (e.g. when an MPI message depends on the completion of an OpenMP task). Being aware of the full workload at runtime may enable more efficient scheduling policies as well as improve system inspectability and performance diagnosability.

1.2 Our Envisioned User Workflow

Figure 1.1 shows our envisioned high-level development workflow for two classes of users: domain experts with a limited computational background doing novel algo-

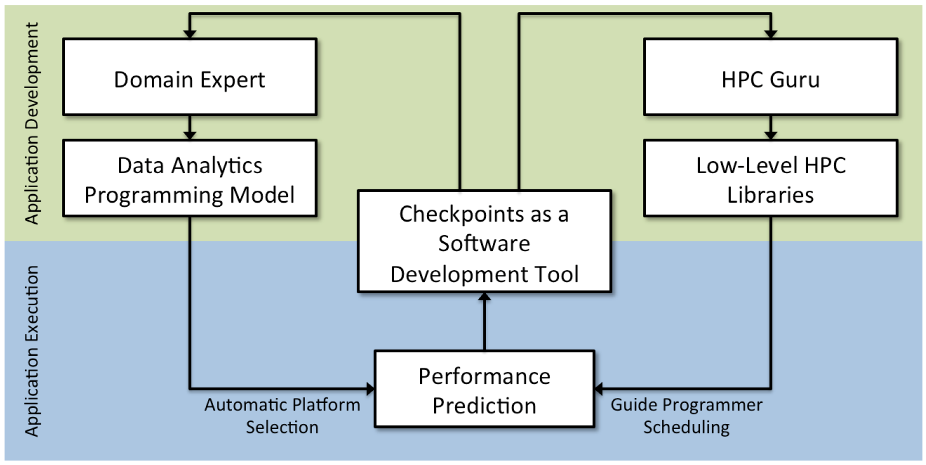
Application Development
Application Execution
Domain Expert
Data Analytics Programming Model
HPC Guru
Low-Level HPC Libraries
Checkpoints as a Software Development Tool
Performance Prediction
Automatic Platform Selection
Guide Programmer Scheduling

Figure 1.1 : User workflow

rithm development, and HPC Gurus who want the ability to tune and optimize applications at low levels.

Domain experts commonly look for high-level programming languages and models that allow them to focus on algorithm development and avoid thinking about low-level computing challenges, such as vectorization, scheduling, and data layout. In this thesis, we capitalize on work in distributed, data analytics programming systems such as Hadoop MapReduce and Apache Spark to offer these domain experts high-level programming models which enable this separation of concerns. Under the covers, we use automatic techniques to efficiently and natively execute these high-level, user-written kernels on heterogeneous HPC hardware.

On the other hand, HPC Gurus are interested in having the ability to tune and optimize their kernels, scheduling, resource management, and other low-level concerns. Programmer aids that enable quicker development of well-performing applications are important to this set of HPC users, so long as those aids are still incrementally tunable, do not hinder analysis of the application, and do not limit the expressiveness of the overall programming model. HPC Gurus also commonly have significant past experience developing applications using a certain set of HPC libraries or tools. While they are willing to learn new models, anyway that we can capitalize on that existing knowledge is beneficial. Hence, in this thesis we focus on enabling the composition of popular HPC libraries and languages such as CUDA, MPI, UPC++, and OpenSHMEM. As a result, we reap both performance and programmability improvements over hand-tuned HPC implementations.

However, writing application code is only half the HPC software development process. In general, the initial functionally correct implementation of an HPC application is not also a well-scaling and well-performing implementation. HPC developers require tools and frameworks for analyzing their applications for correctness and performance bugs. In this thesis, we explore the use of checkpoints and performance prediction for software development.

Checkpoints capture the state of an application at an instantaneous point in execution. Historically, they have primarily been used for building resilient applications. However, the ability to capture and replay a point-in-time that may be days into the execution of a long-running simulation can be an invaluable tool for rapid, iterative optimization or debugging of a region of code in an application.

Performance prediction, on the other hand, has obvious benefits for both domain experts and HPC Gurus for scheduling workloads across heterogeneous processors. However, each set of HPC developers would want to use this capability in different ways. Domain experts need not be exposed to scheduling decisions, and so it would be the responsibility of an underlying runtime to take advantage of performance predictions in scheduling user kernels. On the other hand, HPC Gurus are more likely to want to use performance prediction to inform the scheduling algorithms they write into their applications.

As part of this thesis, we explore all of the topics enumerated in this section: programming models, runtimes, and tools for both domain experts and HPC Gurus. We design these frameworks to fit into the workflow outlined in Figure 1.1. Later in this thesis, we illustrate how each set of HPC users would take advantage of the relevant frameworks.

1.3 Thesis Statement

Productive heterogeneous programming can be enabled by building programming systems that focus on 1) compatibility and composability with other tools and programming systems, 2) using compile-time insights to improve runtime performance, and 3) integrated resource- and workload-aware runtimes. Programming systems for heterogeneous HPC systems must balance high-level programmability with low-level tunability.

1.4 Contributions

This dissertation makes the following contributions that support this thesis statement:

1. HCL2, HJ-OpenCL, and SWAT (Spark With Accelerated Tasks) [23][24][25]: Each of these projects tackles heterogeneous programming through managed languages and runtime systems. Managed systems improve the inspectability of an application at runtime. These projects take advantage of that inspectability to automatically and transparently offload programmer-written parallel kernels to accelerators (in particular, GPUs). HJ-OpenCL offloads shared memory parallel loops from the HJlib [26] parallel programming library. HCL2 is a framework for offloading distributed Hadoop MapReduce applications. SWAT offloads distributed Apache Spark transformations. These projects share the same high level approach of using runtime bytecode-to-OpenCL code generation, automated memory management, and automated data serialization to enable transparent offload of JVM computation to accelerators. However, each project differs in the constraints placed on it by the source programming model (MapReduce, parallel loops, and Spark), in the extent to which scheduling and kernels are automatically optimized, in the amount of data transfer optimization possible, and in the flexibility of the code generator and automatic serializer.

2. HiPER (Highly Pluggable, Extensible, and Re-configurable Framework for

HPC) [27][28][29]: HiPER is a C++ runtime system and programming model that focuses on extensibility, pluggability, composability, and compatibility with existing programming models. The core component of HiPER is a task-parallel programming model sitting on top of a shared-memory lightweight work-stealing runtime. HiPER is explicitly designed to be extended with third-party modules. These modules add APIs to the HiPER programming model, and may use the HiPER runtime to achieve unified scheduling of all work in a system. Existing modules include a CUDA module, an MPI module, a UPC++ module, and an OpenSHMEM module. As such, while the core of HiPER is not itself a distributed or heterogeneous programming system, it is built with the flexibility to support future and current distributed and heterogeneous HPC platforms.

3. HYDOSO (HYbrid, Decomposition-based, Offline Sequence aligner for Online performance prediction): Performance prediction is a fundamentally important and challenging problem, particularly for scheduling of kernels and communication on heterogeneous and distributed systems. However, to date, this "Holy Grail" of HPC remains unsolved due the complexity of both hardware architectures and kernels. This work explores using techniques from genome alignment to discover similarities between kernels and predict future performance based on past observations. HYDOSO combines compile-time and run-time techniques to enable accurate and adaptive performance predictions on different architectures. HYDOSO is an example of a hybrid compile-time and run-time programmer aid that could support improved runtime performance in HiPER, HCL2, HJ-OpenCL, or SWAT.

4. CHIMES (CHeckpointing In-MEmory State) [30]: CHIMES is a checkpoint-restart framework motivated by the observation that checkpoints are a useful software engineering and programming system tool, and not simply useful for resiliency. Automated checkpointing enables the recall of nearly arbitrary

points in application execution. This recall can be useful for iteratively optimizing parallel regions, for reproducing faults, or for numerical regression testing. CHIMES uses a combined compile-time and run-time approach to take advantage of high-level semantic information in the application source code so as to minimize runtime overheads. A variant of CHIMES called CHIMES-lite is also used in HYDOSO, proving its usefulness in novel runtime and tool development. Like HYDOSO, CHIMES is also an example of an aid that could be integrated with HiPER, HCL2, HJ-OpenCL, or SWAT to improve programmability.

1.5 Outline

The remainder of this thesis will be structured as follows. Chapter 2 briefly summarizes emerging heterogeneous architectures and software components in HPC. Chapter 3 describes our work on the HJ-OpenCL, HCL2, and SWAT frameworks to enable managed, data analytics frameworks to exploit distributed, heterogeneous platforms. This work explores how new, high-level programming models might be used on future heterogeneous supercomputers. Chapter 4 describes our work on the HiPER framework that introduces an extensible programming model and a generalized work-stealing runtime to improve the composability of third-party HPC libraries. This work capitalizes on existing investments into HPC libraries and programmer familiarity with them while enabling unified, integrated, and more scalable scheduling on future platforms. Chapter 5 describes the CHIMES and HYDOSO frameworks for tooling to support either of the two approaches to programming future heterogeneous supercomputers introduced earlier, as well as other existing HPC programming systems. In this chapter, we present checkpoints as a powerful software development tool and make new contributions in heterogeneous performance prediction. Chapter 6 summarizes our contributions and discusses future research directions.

Chapter 2

Background

In this chapter, we outline the state of the heterogeneous computing world today in terms of both the hardware and software used.

2.1 Emerging Heterogeneous Architectures

Arguably the first instance of widespread heterogeneous computing in HPC came with the Tesla generation of NVIDIA GPUs released in 2006. Offering programmable shaders for the first time, Tesla cards allowed programmers to start to experiment with GPU computational parallelism and bandwidth in areas other than graphics. Since Tesla, the Fermi, Kepler, and Maxwell generations have continued to improve on the computational performance, memory bandwidth, energy efficiency, and flexibility of those early programmable graphics cards.

However, GPUs are no longer the only throughput-oriented architecture being used to build massively parallel HPC systems. In this section, we provide an overview of upcoming processing architectures that future HPC systems are likely to use to provide the majority of their computational performance.

2.1.1 NVIDIA Pascal and Volta

NVIDIA GPUs are organized around streaming multi-processors (SMs). Each SM can be thought of as a separate compute core on the GPU with its own program counter, vector arithmetic unit, scratchpad memory, registers, and other core-local resources. In the first GPU in the Pascal generation of NVIDIA GPUs (the GP100, released in 2016), 60 SMs reside on each GPU [31] each with the ability to issue two independent vector instructions of 32 single-precision operations. Each Pascal SM also has 32 double-precision units and 8 special-function units. In total, a single Pascal GP100 will be able to deliver up to 5.3 TFLOPS of double-precision arithmetic or 10.6 TFLOPS of single-precision arithmetic.

The GP100 includes 256 KB of registers per SM (4 KB of registers per vector element), a dramatic increase over past GPUs. Each SM also includes 24 KB of L1 cache, 64 KB of programmer-managed scratchpad memory, and 4 MB of L2 cache shared across all SMs. Finally, the GP100 supports 16 GB of high-bandwidth, stacked memory on the GPU with an aggregate memory bandwidth of 720 GB/s.

The Pascal generation of GPUs includes several hardware improvements intended to improve programmability as well. NVLink [32] improves inter-GPU bandwidth relative to the PCIe bus used today, from ∼30 GB/s to ∼40 GB/s. In certain hardware configurations, NVLink can also connect GPUs to the host CPU. As a result, data transfer patterns that would have previously resulted in high overheads may be feasible on Pascal, enabling new applications to take advantage of GPUs.

The Pascal generation is also the first generation to support page faulting on the GPU and a 512 TB virtual address space. As a result, truly unified memory across CPU and GPU is possible, with all transfers implicitly managed by the NVIDIA runtime and driver.

Of course, any Pascal-based system must also include a management processor, usually an x86 CPU. GPUs do not run a full operating system and so are unable to support essential capabilities like disk I/O, network communication, and access to other devices. Pairing a GPU with either x86 or ARM processors makes GPU-based HPC systems heterogeneous.

The generation of GPUs following Pascal is Volta, and will be used in the CORAL Summit and Sierra supercomputers. While a release date has yet to be announced, Volta is expected to be released sometime between mid-2017 and mid-2018. Details of the Volta architecture are still limited but we can expect 1) continued and improved use of high-bandwidth memory and NVLink to improve memory access performance, and 2) a re-designed SM architecture. More importantly, because Summit and Sierra will be POWER9-based systems, NVLink will be used as both an inter-GPU and GPU-to-CPU interconnect. This should drastically reduce CPU-GPU transfer overheads and create a more tightly linked host-device system.

2.1.2 Intel Knights Landing Xeon Phi

Intel's Knights Landing (KNL) generation of the Xeon Phi product line [33] has recently been deployed in several supercomputer installations, including the Cori supercomputer at NERSC [34]. Computationally, KNL is based around "tiles" each of which contains 2 low-power x86 cores and 4 AVX512 vector units. Each vector unit is capable of simultaneously issuing 32 single-precision operations. A single KNL chip contains 36 tiles. As a result, KNL supports up to ∼3 TFLOPS for double-precision operations and up to ∼6 TFLOPS for single-precision.

Each tile includes 1MB of L2 cache shared by all x86 and vector units on that tile. Tiles on the same chip are connected by a 2D mesh, across which cache coherence is configured using KNL "clustering modes". KNL supports up to 16 GB of on-chip high-bandwidth memory (MCDRAM) and up to 384 GB of DRAM. At boot, the user can select how to configure these different memories: 1) in cache mode, MCDRAM serves as a high-bandwidth and automatically managed cache for DRAM, 2) in flat mode, MCDRAM is explicitly exposed to the programmer through special allocation APIs, allowing them to specialize the allocations placed in MCDRAM, and 3) in hybrid mode, some amount of MCDRAM is used as a cache for DRAM and the rest is exposed to the programmer. A single KNL chip is able to achieve ∼400 GB/s to MCDRAM and ∼90 GB/s to DRAM.

Intel also supports including an Omni Path chip on-package, enabling the use of a high-performance interconnect for communicating between KNLs.

Compared to NVIDIA Pascal GPUs, we can draw a number of conclusions about

KNL:

1. The peak FLOPS supported by future GPUs will continue to exceed those on future Xeon Phi processors, though we currently have no understanding of how easy it will be to achieve a significant fraction of that peak on either chip.

2. The KNL will continue to offer more hardware performance programmer aids than GPUs. For example, consider that Pascal offers 1 MB of L2 cache for an entire GPU, but that KNL offers 1 MB per tile.

3. We can extrapolate that Pascal will likely also yield higher efficiency in terms of peak FLOPS per Watt, as fewer transistors will be dedicated to auxiliary functions.

4. The two will be evenly matched in terms of the amount of high-bandwidth memory each has access to, though it seems Pascal will likely be able to achieve higher throughput.

5. The pairing of Atom x86 cores with AVX512 vector units suggests that KNL will lean more heavily on automatic compiler vectorization to achieve peak computational performance, whereas GPUs will rely more on programmers to express their GPU kernels in a way that maps well to the architecture. This conclusion is supported by recent publications from Intel [35].

It is important to note that while the Cori supercomputer will be KNL-based, the larger upcoming Aurora machine at Argonne National Lab will be based on the following generation in the Xeon Phi product line, code named Knights Hill. Unfortunately, little or no information is available on changes in the Knights Hill generation.

KNL represents an odd hybrid of homogeneous and heterogeneous. At the tile granularity a KNL-based system will be homogeneous: each tile will contain the same processing elements. However, within each tile there is significant heterogeneity in the pairing of Atom x86 processors with vector units. However, if the Intel compilers are able to efficiently and automatically vectorize significant fractions of KNL applications this heterogeneity will never be exposed to the programmer.

2.1.3 Sunway MPP

The Sunway MPP [2] is the building block for the supercomputer currently at the top of the Top 500 list. The MPP is organized into four core groups on a single chip. Each core group consists of a single Management Processing Element (MPE) paired with a single Computer Processing Element (CPE). A CPE consists of an 8x8 mesh of lightweight vector cores. Each MPE supports 256-bit vector instructions but, as indicated by its name, is intended to be used primarily for management operations. Each core in the CPE mesh also supports 256-bit vector instructions, and it is expected that the CPE will handle the bulk of the computational load in an application. The primary difference between MPE and CPE cores noted in [2] is that an MPE core has two floating-point pipelines while CPE cores only have one. With a 1.45 GHz core clock, a single chip with four core groups can achieve a theoretical peak of 3.06 FLOPS of double-precision operations.

Each MPE includes 32 KB of L1 cache and 256 KB of L2 cache. Each CPE tile includes 64 KB of explicitly managed scratchpad memory. Each core group on a chip supports 8 GB of DRAM, leading to a total of 32 GB per chip.

We note a few interesting items in the design of the MPP:

1. The MPP has similar peak double-precision FLOPS to a KNL chip, despite very different designs.

2. Unlike Pascal, Volta, and KNL, the MPP has no high-bandwidth memory. It also has a relatively constrained amount of memory per chip, with only 32 GB supported compared to KNL's 384 GB.

3. Like KNL, the MPP relies heavily on auto-vectorizing compilers and vector

notations in source code to achieve high utilization of the vector units in its CPEs.

Therefore, while the MPP is the first chip discussed in this section which is currently deployed in a large-scale, production cluster, it also seems to be lacking in some areas (e.g. peak FLOPS, HBM, etc.) relative to Pascal and KNL.

2.1.4 Epiphany-V

The Epiphany-V [36] is the most experimental of the architectures discussed in this section, with no public planned HPC systems based on it. The Epiphany-V consists of an array of 32×32 grid of RISC cores, each with 64 KB of scratchpad memory and connected by a network-on-chip.

No hardware Epiphany-V chips are available as yet so all performance numbers are theoretical, but we can try to make some cautious comparisons. For example, Table 12 in [36] cites 8.55 GFLOPS/mm^2 for Epiphany-V, 7.7 for Pascal, and 5.27 for KNL assuming a 500 MHz frequency for the Epiphany. If we keep that assumption, we can estimate ∼1 TFLOPS double-precision peak for a single Epiphany-V. However, given that it is unclear what the core frequency will be for the final release of the Epiphany-V that is only an estimate.

If we focus on energy efficiency, the Epiphany-V becomes more impressive. The Epiphany-V white paper [36] reports a maximum chip power consumption of 2 Watts. Given ∼1 TFLOPS of compute, that would yield 500 GFLOPS/Watt. This is an outrageously high number relative to the other architectures discussed in this section (18.8 GFLOPS/Watt for Pascal, 14.69 GFLOPS/Watt for KNL) so it must be taken with a grain of salt. However, it suggests that while the Epiphany-V's per-chip floating-point performance may not equal its contemporaries, its energy efficiency may far exceed them.

2.1.5 FPGAs

Field Programmable Gate Arrays (FPGAs) are different from the systems described in this section in that they have no architecture, and are essentially re-programmable hardware. An FPGA program consists of a logic gate design which is flashed to the FPGA.

It is difficult to compare FPGAs to more conventional architectures as there are no cores, caches, or other architectural structures to compare to on an FPGA. However, the top-of-the-line FPGA from Altera [37] lists a peak performance of 10 TFLOPS of peak single-precision performance and 1 TB/s of memory bandwidth to 16 GB of high-bandwidth memory.

While these peak numbers are impressive, in reality, FPGAs often fall short for scientific, floating-point intensive workloads. Because the logic gate layouts necessary for floating-point units consume more space on the FPGA, the computational bandwidth for floating-point operations on FPGAs can often be limited relative to the peak. Additionally, FPGA compilation times run on the order of hours, making FPGA software development a more painstaking process. However, for integer arithmetic or fixed-point workloads, FPGAs can demonstrate significant efficiency improvements relative to fixed-architecture alternatives.

2.2 Production Programming Models for Emerging Supercomputers

In Chapter 1 we briefly summarized the state-of-the-art in heterogeneous programming: namely, that it requires the combination of several disjoint programming models to execute an application across the cores and nodes of a heterogeneous supercomputer.

This pain point has been broadly recognized by the HPC community, and different organizations and committees are taking steps to address it. We discuss related work before going into more detail on the contributions of this thesis. While many of these techniques are not widely used today, they are likely to become the status quo for heterogeneous programming in the near future. As such, this work cannot be treated as contemporary research (which will be discussed in the related work section of each chapter of this dissertation), but provide a yardstick with which to measure our work.

2.2.1 OpenMP

One of the largest and most promising efforts to improve the programmability of production-level heterogeneous programming has been undertaken by the OpenMP Architecture Review Board, primarily in the form of the introduction of OpenMP tasks in OpenMP 3.0 [38] and OpenMP accelerator support in OpenMP 4.0 [39]. With tasks, the flexibility and asynchrony of the OpenMP APIs for irregular applications was significantly improved, making them more amenable to efficient execution on pre-exascale and exascale platforms. Tasks have also been suggested as a possible avenue towards improving OpenMP's composability with other models and libraries, i.e. by "taskifying" calls to external libraries. With the addition of accelerator support, OpenMP became one of the first high-level programming models which offered parallel APIs for multiple architectures. This improves the maintainability of heterogeneous codes that use OpenMP, and improves programmability by offering consistent abstractions for programming multiple architectures.

Currently, OpenMP's accelerator support is a mixed bag of positives and negatives. Benchmarking of bleeding edge versions of OpenMP 4.0-compliant implementations show that real application codes running on GPUs using OpenMP are able to achieve a significant fraction of the performance that hand-coded CUDA versions can [40]. Additionally, early experiences show that performance portability is achievable to an extent on multiple architectures using OpenMP without modifications to the programmer-written kernel logic [41]. However, at the same time, support for accelerators has been slow to appear across OpenMP implementations, and most are still in-development. Flexibility built in to the specification of accelerator directives which was originally intended to enable the support of many types of accelerators also means that OpenMP accelerator directives are not usually performance portable across architectures. Additionally, at times, the accelerator directives do not offer enough information to the OpenMP compiler to fully optimize kernels in a way that a programmer might be able to. For example, OpenMP support on KNL platforms has met difficulties successfully vectorizing loops using only standard OpenMP directives. There is also no support for explicitly using scratchpad memory or other special-purpose memories, an important optimization for many accelerators. While OpenMP is arguably the most promising heterogeneous programming model for near-future heterogeneous computing, and shows promise in addressing programmability, composability, scheduling, and tooling challenges on future machines, it lacks some key features which limit its tunability and continues to saddle developers with coherency concerns.

The OMPT Tools API appeared in OpenMP 5.0 preview 1, released in November 2016 [42]. The Tools API will improve the inspectability of OpenMP runtimes by adding APIs to register notifications on specific events and collect traces from target devices. This would allow future performance profiling tools to offer deeper insights into the workload scheduled on OpenMP runtimes. In combination with OpenMP's accelerator APIs, this would conceivably enable much richer toolsets for heterogeneous systems. However, it is important to note that the scope of this API is naturally limited to work scheduled on the OpenMP runtime and that the inspectability of any third-party libraries would not improve.

2.2.2 Kokkos

Closely related to OpenMP, the Kokkos [43] project is an effort by Sandia National Laboratory to use C++ templates and parallel runtimes to produce a highly programmable and efficient parallel programming system that performs well across architectures. By offering abstractions for mapping application data to different parts of a heterogeneous platform's memory hierarchy (something that OpenMP lacks) and abstracting away the underlying execution engine, Kokkos is able to target programming models from Intel, AMD, and NVIDIA through a single high-level abstraction. While its programming model is restricted relative to OpenMP and focuses on synchronous loop parallelism, Kokkos has proved useful for many scientific applications [44]. Its integration with the Trilinos [15] scientific and mathematical packages has also made it easy for domain experts to use Kokkos without having to express their algorithm as a parallel computation. On the other hand, limitations in Kokkos's programming model restrict the domain of applications it is useful for. While Kokkos is used by the Trilinos package, it lacks any significant integration or composability with other programming systems. However, its ability to portably toggle the same parallel loop between architectures is powerful and is not yet supported by most existing parallel programming models.

2.2.3 Raja

The Raja portability layer [45] is similar in abstractions to Kokkos, and focuses on hiding non-portable programming model features from the programmer while enabling experimentation with loop nest ordering and data access patterns. Raja only supports the expression of loop parallelism in C++ and focuses on three abstractions to hide portability issues from programmers:

1. Data type encapsulation: By wrapping application data structures and pointers in Raja types, Raja is able to manage architecture-specific optimizations such as alignment and structure padding for the user.

2. Execution policy: Users may choose different execution policies to automatically toggle between different types of platforms on which to execute parallel loops. For example, Raja supports targeting OpenMP, OpenMP accelerators, and CUDA using different execution policies.

3. Index sets: Storing the iteration space of a parallel loop as a Raja object helps Raja enable programmer experimentation and optimization through automatic, compile-time loop transformations based on C++ template specialization.

2.2.4 CUDA

While the CUDA programming model itself has done little to improve its composability with other systems, there have been several new technologies introduced by NVIDIA which should improve the composition of host and device computation.

One of the largest programmability improvements in the accelerator domain is the upcoming introduction of hardware-supported Unified Memory [46] in NVIDIA GPUs. Put simply, Unified Memory on Pascal-generation or later GPUs will reduce the coherence problem for heterogeneous computing by automatically paging virtual pages to and from NVIDIA GPUs. It will still be the programmer's responsibility to avoid data races on pages shared between the host and device. Additionally, Unified Memory will do nothing to simplify management of the GPU's on-chip memory hierarchy, and it remains to be seen how Unified Memory will affect tooling, performance, and scheduling problems on GPU-based platforms.

2.2.5 GPU-Aware MPI

A recent development in GPU computing that has found its way into production software packages is GPU-Aware MPI [47]. GPU-Aware MPI enables the direct communication of data from a GPU in one node of a supercomputer to a GPU in another node. This simplifies the API by only requiring a single transfer call to be made, may improve performance if the networking hardware is able to support DMA from one GPU to the other without host intervention, and overall improves the composability of the two systems. However, this effort is too localized. While it improves MPI-CUDA applications, true composability is an all-to-all relationship and requires a more comprehensive approach to the problem.

2.3 Summary

While there are several efforts in the heterogeneous computing community to improve the state-of-the-art in production heterogeneous programming, many of these projects address the problem piecemeal by focusing on specific, current programming models and libraries. Most of these approaches do not consider future extensibility to support heterogeneity in other components of future HPC systems.

From this chapter, we can conclude two things. First, that the majority of future HPC systems are likely to include heterogeneous processing units in order to achieve high computational bandwidth without massive energy costs, and that as we enter this second wave of throughput-oriented architectures many chip manufacturers are in tight competition to produce the "best" HPC-oriented architecture. Second, that while some software projects (notably OpenMP) are working to improve the composability of HPC software modules, the next generation of software tools installed on these heterogeneous machines will tackle the problem piece-meal (e.g. by composing accelerator and host parallelism, or MPI communication and accelerator kernels). To support future heterogeneous machines, a more comprehensive solution is required.

Chapter 3

High-Level Programming Systems for Data Analytics Workloads on Heterogeneous HPC Systems

3.1 Motivation

Classically, the scientific workloads that HPC systems have been used for were programmed in low-level, prescriptive programming models. Programming models like OpenMP, MPI, and FORTRAN allowed HPC programmers to tune applications for the target platform while remaining sufficiently high-level to enable the development of reasonably portable large-scale scientific codes. These programming models avoided introducing significant overheads or bottlenecks and meshed well with optimizing and vectorizing compilers. This led to both efficient parallelization and well-performing straight-line code, producing well-scaling applications.

However, more recently, the users of large-scale HPC machines have been exploring the use of alternative programming models, primarily from the data analytics field. Data analytics and HPC share the problem of programming and scaling across large distributed machines in order to process massive datasets. However, while HPC scientific workloads tend to be compute-bound (hence the emphasis on computational performance), data analytics workloads tend to be communication-bound, I/O-bound, or memory-bound. For bandwidth-bound workloads, there is less of a need for a highly optimized execution platform. As a result, the data analytics community has gravitated towards higher level programming models running on managed runtime systems, and reaped productivity benefits. While managed runtimes and the high-level programming languages that accompany them have already been widely adopted by the scientific community at the desktop scale [48][49], only recently have the HPC and scientific communities begun experimenting with these systems at the cluster scale [50]. This experimentation has partially been driven by new workloads on HPC systems, and partially by the downward trend in memory per processing core causing previously compute-bound workloads to become bandwidth-bound.

On the other hand, as novel algorithms are developed in the data analytics and machine learning communities and as the frameworks used to implement them become more efficient, those communities are likewise seeing a diversification in workloads and an increase in the number of compute-bound problems. One example of this would be the rapid rise in popularity of neural nets for different image processing problems. As a result of these trends, the data analytics community is increasingly adopting hardware and software techniques from the HPC community. For example, accelerators are now commonly used to accelerate the training of neural nets. Hence, data analytics frameworks are more commonly being run on HPC-esque platforms for compute-bound data analytics problems.

The convergence of the scientific and data analytics communities leads to an open research problem: how can we efficiently execute managed runtimes on accelerators, without sacrificing the productivity and programmability that makes these data analytics frameworks popular to begin with? In this chapter, we present three pieces of work on this problem: HJ-OpenCL, HCL2, and SWAT. HJ-OpenCL is a shared-memory parallel programming model which we use to introduce foundational techniques in offloading data-parallel kernels from managed runtimes to accelerators. HCL2 and SWAT are extensions of the Hadoop and Spark distributed data analytics programming frameworks, and support offloading computational kernels to accelerators.

3.2 The Challenge of Managed Runtimes on Accelerators

A classical example of a managed system is the Java Virtual Machine (JVM). It offers a portable, abstract, stack machine model on which programming languages and programming tools can be constructed. Using a custom bytecode-based instruction set, the JVM natively supports high-level programming constructs such as exceptions, virtual methods, classes, inheritance, garbage collection, and multi-threaded synchronization. While it is a machine model, the JVM and its instruction set do not map natively down to any commodity hardware platforms. JVM implementers must efficiently support the stack machine model on top of conventional processors through runtime optimization tricks, such as just-in-time compilation. Fortunately, the hardware and software stacks which the JVM executes on can support all of its high-level features relatively efficiently.

While one of the strengths of the JVM is its portability, that portability is primarily limited to CPU architectures. Supporting the full instruction set and capabilities of the JVM on accelerators such as GPUs and FPGAs would either not be possible or incur large overheads, offsetting the performance benefit of the accelerator. Hence, as most data analytics frameworks are built on the JVM but native implementation of a full JVM is not feasible on accelerator cores, we must find another approach to running data analytics frameworks on accelerators.

We highlight five main challenges that any such approach must meet:

1. **Incompatibilities in instruction set between managed and native systems**: The JVM and other managed systems commonly use some intermediate representation for application instructions. Interpreted execution is used to enable portability across hardware platforms. Because these intermediate representations are not natively executable, we use runtime code generation to convert it to an executable format that is.

2. **Incompatibilities in data format between managed and native sys-**

tems: Managed systems may use custom data structures to store data internally. The format of those data structures may either be incompatible with accelerators, may not be knowable, or may be wasteful on memory-constrained platforms. In this work, we develop techniques that use reflection to convert JVM objects into a native format supported by accelerators. Our primary contributions are in extending the range of data types supported relative to previous work.

3. **Resource management in a multi-tenant system**: In general, data analytics platforms are highly parallel and will use several threads or processes within each shared memory node. It is necessary to efficiently share accelerators and other resources among them without causing synchronization overheads, over-subscription, or under-subscription. Adding to the complexity, you generally have resources in three different address spaces: the automatically managed JVM heap, the explicitly managed host native address space, and the accelerator.

4. **Scheduling in a multi-tenant system**: Scalably scheduling the computation and communication from multiple processes or threads requires an awareness of both the workload and hardware resources available. In this work, we present runtime designs that enable efficient scheduling. Additionally, we present techniques in automatic, machine learning-based workload characterization for automatic processor selection.

5. **Inspectability**: In any multi-tenant system, being able to diagnose performance and correctness bugs is critical to the usability of the framework. However, gaining insight into a system that combines JVM execution, native execution, and accelerator execution is a difficult problem for the domain experts that are the primary users of these frameworks.

```
forall (0, niters - 1, (iter) -> {
    // User-written parallel loop body
    ...
});
```

Figure 3.1 : An example HJlib parallel loop.

```
forall_acc (0, niters - 1, (iter) -> {
    // User-written parallel loop body
    ...
});
```

Figure 3.2 : An example HJ-OpenCL accelerated parallel loop.

3.3 Offloading Shared-Memory Parallel Java Programs Using HJ-OpenCL

HJlib [51] is a shared-memory, parallel programming library that supports the unification of a variety of parallel programming patterns, including task-parallelism, loop-parallelism, futures, and actors. HJlib includes the loop-parallel `forall` API shown in Figure 3.1. The `forall` loop takes a loop range and user-written loop body as input and executes each loop iteration in parallel across multiple JVM threads. This loop body can contain arbitrary user-written code, and is passed as a Java lambda. There is an implicit synchronization at the return of a `forall` call that waits for all parallel loop iterations to complete before returning to the calling code.

In this section, we develop techniques for offloading HJlib's parallel `forall` loop to OpenCL devices, in particular native threads executing on CPUs and GPUs. In this work, called HJ-OpenCL, we introduce a new identical API named `forall_acc` (illustrated in Figure 3.2. The semantics of `forall_acc` are the same as `forall`, but it supports transparent offload of the user-written parallel loop body to OpenCL devices.

The main functionality difference between `forall` and `forall_acc` are the limitations that `forall_acc` places on what can be written inside the parallel loop body.

In particular, the data types that can be accessed from the kernel are limited to primitives, arrays of primitives, objects containing only primitive fields, and arrays of those objects. HJ-OpenCL does not currently support exceptions, dynamic method dispatch, or network/file I/O inside of a `forall_acc` loop body.

Efficiently implementing `forall_acc` requires solving several research problems, which mirror the challenges presented in Section 3.2:

1. Code generation of OpenCL kernels from the JVM bytecode storing the logic for user-written kernels. In this work, we primarily rely on re-using the bytecode-to-OpenCL code generator developed as part of the APARAPI framework. We also extend APARAPI to support Java composite objects of primitive fields. (e.g. `java.util.ArrayList`).

2. Efficient serialization and deserialization of JVM objects at runtime to a format amenable to storage on an OpenCL accelerator. Like code generation, this support is only for Java objects that contain only primitive fields.

3. Automatic management of memory allocations and transfers, using Java reflection to determine the data in the JVM that must be transported to the accelerator and using inspection of the calling context of a `forall_acc` call to automatically remove redundant, inter-loop transfers.

The following sections elaborate on the implementation and limitations of the solutions to these challenges.

3.3.1 HJ-OpenCL Code Generation and Data Serialization

APARAPI [52] is an open-source tool for offloading Java applications to OpenCL accelerators. It includes a code generation module for converting JVM bytecode to OpenCL kernels at runtime as well as a runtime for automatically transferring captured values to the accelerator, launching the kernel, and transferring output back.

In HJ-OpenCL, we extend APARAPI's existing support for OpenCL code generation from JVM bytecode. While APARAPI's code generation module supports a reasonable subset of the JVM bytecode specification, many of the JVM's most commonly used features (e.g., object references and the `NEW` opcode) are not supported in APARAPI kernels. This work removes some of these limitations, while reusing much of the core code generation framework.

Additionally, we use The RetroLambda[53] tool at compile time to convert Java 8 JVM bytecode that uses the new `INVOKEDYNAMIC` opcode to instead use an inner class to store the closure for a lambda. This is not visible to the programmer, but is necessary for compatibility between APARAPI and HJ-OpenCL.

Storing JVM objects in OpenCL

This work focuses on supporting JVM object references in accelerated parallel regions, but limits the current scope of that work to only support references to primitive fields in those objects, or instance methods which only reference primitive fields. For example, it would be valid to reference objects of the type `Point` defined below so long as those references only used the methods `getX`, `getY`, `getZ`, or `distance`. Note that methods with object references as parameters and local variables are supported – the main constraint is that the method is not permitted to access a non-primitive field during GPU execution. Hence, use of the methods `getClosest` and `distanceToClosest` is not supported in accelerated parallel regions.

```
package edu.rice.hj.example.Point;

public class Point {
  private float x, y, z;
  Point closest;

  public Point(float x, float y, float z, Point closest) {
    this.x = x; this.y = y; this.z = z;
    this.closest = closest;
  }
  public float getX() { return x; }
  public float getY() { return y; }
  public float getZ() { return z; }
  public float distance(Point other) {
    return (float)Math.sqrt(Math.pow(other.x - x, 2) +  Math.pow(other.y - y,↩
         2) + Math.pow(other.z - z, 2));
  }

  public Point getClosest() { return closest; }
  public float distanceToClosest() {
    return distance(closest);
  }
}
```

A JVM class referenced from an accelerated parallel region are represented by a corresponding `struct` in the automatically generated OpenCL kernel definition. For each primitive field in the JVM object, a similarly typed field is created in the `struct` definition auto-generated by the modified APARAPI code generator. An example of the `struct` generated for the `Point` class above is shown below:

```
typedef struct __attribute__ ((packed)) edu_rice_hj_example_Point_s {
   float  x;
   float  y;
   float  z;
} edu_rice_hj_example_Point;
```

Note that this conversion is only carried out for object types which are used in an accelerated parallel region, not for all classes loaded by an HJlib application. If a referenced object has a superclass whose primitive fields are also referenced, those fields will be included inline in the generated `struct`. One side benefit of this approach is that only primitive fields are transferred to the accelerator, reducing data movement relative to approaches that copy the whole JVM object.

Once the referenced primitive fields of a class and its superclasses have been identified using Java reflection, it is straightforward to build a one-to-one mapping from Java primitive types to OpenCL primitive types. For example, reflection would tell us that the `Point` class above has a field named `x` whose type descriptor is “F”.

The type descriptor “F” can be converted to the OpenCL primitive type `float` using a lookup table.

Using the `packed` attribute for generated `structs` simplifies object serialization by guaranteeing that all `struct` fields are stored consecutively in memory.

Serializing and Deserializing JVM Data Structures

Runtime serialization and deserialization of dynamically loaded JVM objects is an expensive operation that generally takes O(`N`) time, where `N` is the size of the data to be serialized. Here we describe our strategy for serializing and deserializing JVM objects to the format defined by Section 3.3.1, which ensures that the resulting data is consumable by an OpenCL accelerator and matches the generated code. This work does not focus on optimizing this process. We rely on the techniques that will be presented in Section 3.3.2 to minimize the number of times expensive serialization and deserialization operations take place.

Data referenced from an accelerated parallel region can be classified as either: 1) local variables in the method scope enclosing the lambda creation site, or 2) fields in the class scope enclosing the lambda creation site. For example in Figure 3.3, `int b` is an example of a captured local variable and `int a` is an example of a captured field. Both of these variables would be captured by the lambda created for the accelerated parallel region and passed to `forall_acc`.

```
class Foo {
  private int a;

  void bar() {
    int b = ...;
    forall_acc (0, niters - 1, (iter) -> {
      ...
    });
  }
}
```

Figure 3.3 : Illustrative code snippet for lambda captures.

Local variables in the enclosing method are captured as fields in an anonymous inner class auto-generated by the RetroLambda pass, including `this`. Fields of the enclosing class are accessible through the `this` reference saved in the RetroLambda anonymous class. Both types of data are accessible from accelerated parallel regions, including fields of the enclosing `this` instance.

Prior to launching a parallel loop on an accelerator, we must iterate over each of these fields that is referenced from inside that parallel loop and decide how to store them on the accelerator. To do so, we start by classifying each into one of four categories: primitive, non-array object, array of primitives, or array of objects.

Supporting primitives on the accelerator is straightforward. Primitives are passed by value to the accelerator kernel using OpenCL's `clSetKernelArg` API. Their values are fetched from the JVM using the JNI APIs.

Serializing JVM Objects

Singleton objects which are shared across all iterations of a parallel loop may arise in a number of situations, such as the `center` object in Figure 3.4.

```
Point center = new Point(...);
forall(0, P - 1, (iter) -> {
  Point mine = points[iter];
  float distToCenter = mine.distance(center);
  ...
});
```

Figure 3.4 : Illustrative code snippet for singleton objects.

For this code snippet, the `center` object can be represented by a single `Point struct` allocated on the accelerator. The first step in creating that allocation is to serialize the JVM object to a byte buffer that matches the layout of the OpenCL `struct`. This can be done using the code template below, which takes as input the object to be transferred to the accelerator, a byte buffer to write the serialized object to, and a list of type descriptors and offsets for each field in the object being transferred. This field metadata is sorted in the same order as the fields appear in the OpenCL `struct` definition. The object to be transferred can be loaded by name using Java reflection.

```
void writeObjectToStream(Object ele, List<FieldDescriptor> structMemberInfo, ↩
    ByteBuffer bb) {
  for (FieldDescriptor fieldDesc : structMemberInfo) {
    TypeSpec typeDesc = fieldDesc.typ();
    long offsetInClass = fieldDesc.offset();

    switch (typeDesc) {
      case (TypeSpec.I):
        bb.putInt(Unsafe.getInt(ele, offsetInClass));
        break;
      case (TypeSpec.F):
        bb.putFloat(Unsafe.getFloat(ele, offsetInClass));
        break;
      ...
    }
  }
}
```

Figure 3.5 : Code snippet illustrating the serialization of an Object into a ByteBuffer.

The code snippet in Figure 3.5 relies on the `sun.misc.Unsafe` package to fetch both the offset of a field inside an object as well as the value of that field. The Unsafe package is used rather than Java reflection to enable future work on transferring whole JVM objects to accelerators and accessing fields by offsets rather than packing them into a `struct`.

The contents of the generated `ByteBuffer` can then be transferred to the accelerator and used to represent this JVM object. Given that the order of the fields in the `struct` and the order of the fields passed to `writeObjectToStream` are the same, the `packed` attribute used in Section 3.3.1 ensures that the `ByteBuffer` and OpenCL representations are compatible. The `byte[]` backing the `ByteBuffer` can then be passed through JNI to OpenCL and transferred to the accelerator. Once on the accelerator, this object can be referenced using an appropriately typed pointer, such as:

```
edu_rice_hj_example_Point *ptr;
```

Public fields in the `Point` object can be loaded using the `->` operator:

```
float tmp = ptr->x;
```

Methods of the `Point` class are mangled and include `this` as an explicit parameter. Thus, a call to `mine.distance(center)` would be transformed to:

```
static float edu_rice_hj_example_Point__distance(__global Point *this, ↩
    __global Point *other) {
  ...
}

float d = edu_rice_hj_example_Point__distance(mine, center);
```

For consistency, all object references on the accelerator are represented as `__global` pointers, pointing to memory in the accelerator's global address space. This design choice will be an important constraint when we discuss serializing arrays of objects.

Serializing JVM Arrays

Transferring arrays of primitives to an accelerator starts with passing the array directly through JNI to a native function. The native code then uses the appropriate JNI API to extract the buffer backing the Java array. For example, for an `int[]` in the JVM the function `GetIntArrayElements` would be used to extract an `int*` pointer to the contents of the JVM array. The values pointed to by the retrieved pointer can then be directly transferred to the OpenCL accelerator.

Transferring arrays of objects to an accelerator re-uses `writeObjectToStream` by iterating over the elements of the array and applying `writeObjectToStream` to each object with the same output `ByteBuffer`. This produces a single byte buffer that stores the contents of each object in the object array, consecutively. However, to keep the representation of object references as `__global` pointers consistent across object singletons and object arrays, it is necessary to allocate an additional array of `__global` pointers on the accelerator which represent the array of object references being serialized on the JVM. The `ByteBuffer` emitted by the successive calls to `writeObjectToStream` stores the data backing those object references. This additional array of pointers has the same length as the object array, and element `i` in the pointer array is pre-populated with the address of element `i` in the data buffer on the accelerator. This preprocessing is done in parallel on the accelerator itself after both buffers have been allocated and the contents of the data buffer have been transferred to the accelerator, but before the main computational kernel is launched.

Null pointers in arrays of objects require special treatment for consistency between the JVM and accelerator. When serializing an array of objects to the accelerator, an extra array of `byte`s is allocated whose length is the same as the array of objects. If element `i` in the array of objects is `null`, element `i` in this `byte` array is set to 1. Otherwise, it is set to 0. This `isNull` array is transferred to the accelerator along with the array of objects and used in a pre-processing kernel to initialize the `__global` pointer array described in the preceding paragraph. If an object reference in the JVM was `null`, the corresponding entry in the pointers array is set to `NULL`. The extra `isNull` array is necessary because the object references themselves are never transferred to or from the accelerator; only the contents of the pointed-to objects. The `isNull` array is necessary to determine whether each element of the corresponding array-of-pointers on the accelerator should be set to `NULL` or to point to an element in the array-of-structs that stores the object contents.

Deserializing Data Structures Back to the JVM

Transferring singleton objects, primitive arrays, and arrays of objects back from the OpenCL accelerator requires performing the same operations as described above, in reverse.

For objects, the serialized object is transferred back from the accelerator and the method `readObjectFromStream` (illustrated in Figure 3.6 is used to populate the associated JVM object with any changes. Note that HJ-OpenCL does not currently support atomic operations, reduction operations, or `synchronized` regions on shared objects inside accelerated parallel regions. If a field of a singleton object is written from multiple threads, no guarantees are made as to its state when it is restored in the JVM.

Arrays of primitives are restored in the JVM by transferring the accelerator buffer

```
void readObjectFromStream(Object ele, List<FieldDescriptor> structMemberInfo,↩
     ByteBuffer bb) {
  for (FieldDescriptor fieldDesc : structMemberInfo) {
    TypeSpec typeDesc = fieldDesc.typ();
    long offsetInClass = fieldDesc.offset();

    switch (typeDesc) {
      case (TypeSpec.I):
        Unsafe.putInt(ele, offsetInClass, bb.getInt)
        break;
      case (TypeSpec.F):
        Unsafe.putFloat(ele, offsetInClass, bb.getFloat)
        break;
    }
  }
}
```

Figure 3.6 : Code snippet illustrating the deserialization of an Object from a ByteBuffer.

directly into the JVM through a primitive pointer retrieved from JNI.

For arrays of objects, we first launch a post-processing accelerator kernel that iterates across the array of pointers storing the current state of object references on the accelerator. If pointer `i` no longer points to object `i` in the associated data buffer, it must either have been set to `null` or to point at a different object on the accelerator. If the former is true, we mark that object reference as having been nullified in the `isNull` array. Otherwise, the contents of object `i` are updated in the backing array to be the contents of the object instance pointed to by pointer `i` on the accelerator. The object referenced by pointer `i` may change if it now points to an object that was constructed on the accelerator, or to another object of the same type that was transferred from the JVM. More details on how dynamic object allocation is supported are available in Section 3.3.2.

Once the contents of the array of objects have been updated based on changes in the array of pointers, the contents of the array of objects are transferred out of the accelerator and used to update the contents of JVM objects by iterating over the JVM array and updating each object using `readObjectFromStream`. If an object reference is marked as null in `isNull`, the corresponding object reference in the JVM is also set to `null`.

This work assumes that no two elements in an array of object references transferred to the accelerator point to the same object.

Clearly, the serialization and deserialization of JVM objects and particularly arrays of JVM objects is an expensive operation. If `M` is the number of fields in a given type, serializing or deserializing an object of that type is `O(M)`. For an object array of length `N` that cost increases to `O(NM)`. Past work[54] has looked at using GPU parallelism and memory bandwidth to accelerate re-formatting the data layout of data structures. Our work could be extended to use similar techniques to accelerate data serialization.

3.3.2 HJ-OpenCL Runtime

This section describes the bytecode analysis techniques used to eliminate redundant data movement between successive accelerated parallel regions. This optimization reduces the serialization and deserialization overheads incurred from supporting object references in accelerated parallel regions. We also discuss our approach to dynamic memory management on the accelerator, which is necessary to support translation of the `NEW` bytecode.

Removing Redundant Data Movement Through Context Inspection

APARAPI supports basic PCIe transfer optimizations by analyzing the bodies of parallel regions for read-only and write-only buffers. If a buffer is detected as read-only, it is only transferred from the host JVM to the accelerator. If it is write-only, it is only transferred from the accelerator to the JVM.

In this work, we extend this capability by loading and inspecting the bytecode of the method in which this parallel region is launched and finding local variables or class fields that are passed to multiple, successive, accelerated parallel regions without being referenced from the JVM between those parallel regions. At a high level, a buffer (object, primitive array, or object array) will only be transferred back to the JVM after an accelerated parallel region if it may be read before the JVM launches another accelerated parallel region that is passed the same buffer. A buffer is only transferred to the accelerator prior to an accelerated parallel region if we find that this buffer may have been written from the JVM since the last accelerated parallel region that referenced it.

This analysis is currently intra-procedural and context-insensitive. As a result, when looking for reads following a parallel region the analysis will transfer a buffer back if there exists any control flow path from that parallel region to a method return that does not pass through a parallel for region that uses this buffer. Likewise, when looking for writes preceding a parallel region the analysis will transfer a buffer to the accelerator if there exists any control flow path from the start of the enclosing method to this parallel region that does not pass through any other parallel regions that use this buffer.

This analysis is particularly useful for applications that exhibit a pipeline of parallel regions. For example, the KMeans machine learning algorithm consists of a pipeline of two kernels: one kernel that classifies each data point into a cluster, and a second kernel that recalculates each cluster's centroid based on its member points. In this pipeline, we can safely skip both retrieving the point classifications after the first kernel and copying them back to the accelerator for the second kernel. Our analysis captures this information as long as 1) both parallel regions are launched in the same function, and 2) the analysis described below indicates the point classifications are not read or written from the JVM between the two parallel regions.

For each buffer used by any accelerated parallel region, the location in the source code of the last accelerated parallel region that referenced that buffer is stored by the HJ-OpenCL runtime. This last referenced location is updated following the completion of each parallel region for all buffers referenced. To determine if a given buffer must be transferred to the accelerator prior to launching a parallel region, we first check that the last referenced location is in the same method as the current accelerated parallel region. If it is not, then the transfer must be performed. Otherwise, we start from the last referenced location and traverse forward over bytecode instructions to verify that the preceding parallel region is post-dominated by the current parallel region and that the state of the buffer could not have been modified in the JVM. If any of the following conditions are met, our analysis indicates that the current buffer may have been modified and, therefore, must be updated on the accelerator before the parallel region can be launched:

1. If the current buffer is stored in a local variable slot and we find a local variable store opcode (e.g. `ASTORE_3`) to that slot.
2. If we encounter a return statement. Note that even though the last referenced location for a buffer is within the same method it may have been during a different call to this method. Checking for a return is necessary to check for this case.
3. If we encounter an `AASTORE` opcode.
4. If we encounter a `PUTFIELD` opcode.
5. If we encounter any type of method invocation.
6. If we encounter any opcode that may throw an exception (e.g. a divide-by-zero exception thrown by an `IDIV` instruction)

When deciding whether a buffer must be transferred back from the accelerator following an accelerated parallel region, we perform a similar traversal starting at the current parallel region and verify that the current parallel region is post-dominated by parallel regions that use the same buffer and that no reads of this buffer may be performed from the JVM between parallel regions. If any of the following conditions is met by any of the bytecode instructions following this parallel region and before encountering another parallel region that uses this buffer, the current buffer is transferred back to the JVM:

1. If the current buffer is stored in a local variable slot and we find a local variable load opcode (e.g. `ALOAD_3`) for that slot.
2. If we encounter a return statement.
3. If we encounter an `AALOAD` opcode.
4. If we encounter a `GETFIELD` opcode.
5. If we encounter any type of method invocation.
6. If we encounter any opcode that may throw an exception (e.g. a divide-by-zero exception thrown by an `IDIV` instruction)

These transfer decisions are cached for each parallel loop. This allows us to only perform this analysis once for each buffer in each parallel region regardless of how many times that region is entered, reducing overhead.

One special case is that of an accelerated parallel region that does not have any buffers that need to be transferred back to the JVM. That is, all buffers used by this kernel are also referenced by other accelerated parallel regions that post-dominate the current parallel region, and none of those buffers is referenced from the JVM. If this is the case, we can safely allow the non-producing accelerated parallel region to execute asynchronously on the accelerator while the host system continues JVM execution. All accelerated parallel regions wait for any preceding kernels before starting their computation.

Buffers are allocated the first time they need to be transferred to the accelerator and are freed after they are copied back to the host following the last accelerated parallel region to reference them. Hence, a buffer may be allocated on the accelerator even though a currently active parallel region is not working on it if the active parallel region is between two regions that do. This may lead to out-of-memory errors even when the working set for a single parallel region is within the limits of accelerator memory. Supporting out-of-core data and building a more resource-adaptive runtime is beyond the scope of this work.

Future work could extend this analysis to be inter-procedural and control flow sensitive, which would improve the accuracy of the redundant data movement elimination.

Dynamic Memory Allocation in Auto-Generated OpenCL Kernels

While supporting JVM object references on accelerators helps to expand the domain of applications that can be transparently offloaded, object references have limited usefulness without the ability to dynamically allocate new objects. However, the OpenCL standard does not include dynamic memory allocation as a supported operation and most accelerators have little or no native support for it. In this section, we describe a technique for supporting dynamic memory allocation on accelerators by transparently replacing the `NEW` opcode and object constructors during code generation.

On the accelerator, there are four global data structures used to support dynamic memory allocation in our approach:

- `heap`: A large, shared, byte-array pre-allocated in the OpenCL `__global` address space, where it is accessible to all threads.
- `top`: A single, shared atomic integer initialized to zero. When a thread performs a dynamic memory allocation, it atomically increments this integer by the number of bytes it is allocating. If the new value of the integer is less than or equal to the number of bytes in the `__global` heap, the allocation has succeeded. If the old value of the atomic integer were stored in `oldValue`, the current thread's allocation is now available at `(char *)heap + oldValue` and is followed by a contiguous chunk of the requested number of bytes.

- `complete`: An array of integers whose length is equal to the number of iterations in the current parallel loop. Each element in this array is initialized to zero at the start of processing a parallel region. Element `i` of this array is set to one when iteration `i` of the parallel for loop completes successfully on the accelerator. An iteration can only fail if a dynamic memory allocation cannot be satisfied by the heap.

- `anyFailed`: A single integer on the accelerator that is initialized to zero and set to one by any thread which fails a dynamic memory allocation.

Using these data structures (`heap`, `top`, `complete`, `anyFailed`), the dynamic memory allocation technique works as follows from the host application:

1. Buffers for the accelerator data structures described above are allocated. `complete` is zeroed.

2. The host application zeroes `top` and `anyFailed` and launches the generated kernel.

3. When the kernel completes, `anyFailed` is transferred back to the host. If it is non-zero, we return to step 2 and re-execute with a reset heap. However, iterations of the parallel loop that have marked themselves as completed in the `complete` buffer skip re-execution. Otherwise, if `anyFailed` is zero we continue.

This technique supports execution of parallel regions on accelerators whose dynamic memory allocation requirements exceed the size of the allocated heap by repeatedly retrying subsets of iterations in the same accelerated parallel region until all iterations succeed. This requires an extra allocation for any output object-typed arrays on the accelerator to persist the final object stored by a completed thread before resetting the heap. To support dynamic memory allocation, some extensions to APARAPI's code generation are also required.

First, all methods that perform dynamic memory allocation (i.e., use the `NEW` opcode) are identified, along with all callers of these methods. A field named `allocFailed` is added to the `This struct` that each thread has a thread-private copy of. This field is initialized to zero before performing the work for each iteration of the original parallel for loop. If an allocation fails, `allocFailed` is set to one and the memory allocation function returns. All calls to the memory allocation function as well as any calls that may lead to the memory allocation function being called are followed by a check that `allocFailed` is still zero. If `allocFailed` is found to be non-zero, the current function immediately returns as will all others on the current call stack. For functions with a non-void return value, a default value is returned based on its return type (e.g. `0` for `int`). Figure 3.7 illustrates the structure of the generated functions:

This generated code simulates an exception-like mechanism on the accelerator, albeit only for out-of-memory situations. At the top-level, if `allocFailed` is found to be non-zero, the element in `complete` corresponding to this iteration is not set and `anyFailed` is set to one. When the host discovers a non-zero value in `anyFailed`, this kernel will be rerun but any iterations with a non-zero value in `complete` will be skipped.

This technique does not make any guarantees on completion. With a very small heap or very large allocations, it is theoretically possible for this approach never to converge to completion. Future work could address this by reducing the level of parallelism at each re-execution if no progress was made. Halving the number of threads executing the parallel region would increase the chance of individual parallel iterations succeeding as contention for the heap decreases. In the worst case, this would devolve to a single-threaded parallel region and guarantee completion (but not efficiency) as long as the allocated heap was sufficiently large to support the dynamic allocations of each individual thread. If that were not the case, this condition would be easily detectable and handled by reverting to JVM execution.

This technique also assumes that the body of a single iteration of the enclosing

```
__global void *alloc(..., int *allocFailed) {
  __global void *allocation = ...;
  if (allocation == NULL) {
    *allocFailed = 1;
  } else {
    *allocFailed = 0;
  }
  ...
}

void foo(This *this, ...) {
  __global void *ptr = alloc(..., &this->allocFailed);
  if (this->allocFailed) return;
  ...
}

void bar(This *this, ...) {
  foo(this, ...);
  if (this->allocFailed) return;
  ...
}

__kernel void run(...) {
  This this;
  ...
  for (int i = tid; i < nthreads; i++) {
    ...
    bar(&this, ...);
    if (this.allocFailed) {
      complete[i] = 0;
      *anyFailed = 1;
    } else {
      complete[i] = 1;
    }
  }
  ...
}
```

Figure 3.7 : Code structure used to enable kernel abort on allocation failure.

`forall_acc` is idempotent: it does not modify its own input state. If a thread modified some input state, failed an allocation, and then was retried in a later kernel invocation, it would read partially updated state on the accelerator. This requirement is not currently enforced during code generation, but could be in future work. While this is a limitation, this assumption is implicitly true in the functional style programming that is common in parallel programming frameworks (e.g., Scala parallel collections) and in current JVM acceleration research[55].

3.3.3 HJ-OpenCL Performance Evaluation

In this section, we will start by summarizing the experimental setup and applications used to benchmark the HJ-OpenCL system. We will then begin the performance evaluation by comparing HJlib and HJ-OpenCL performance at the granularity of parallel regions, without any redundant transfer elimination. HJ-OpenCL will use a GPU accelerator. After enabling redundant transfer elimination we rerun all parallel regions and then note and explain any change in performance. Using this information, we statically select whether to run each parallel region as a `forall` (for JVM execution) or a `forall_acc` (for native CPU or GPU execution) and measure overall speedup of HJ-OpenCL using native threads on the CPU or GPU, comparing against parallel Java Streams. Finally, we measure how performance of one benchmark degrades as the size of the accelerator heap is artificially reduced.

Experimental Setup

All benchmarks in this section are run on the same hardware platform, containing a 12-core 2.80GHz Intel X5660 CPU, 48GB of system RAM, and 2 discrete NVIDIA M2050 GPUs each with 2.5GB of global memory. All tests are run using all 12 CPU cores but only 1 GPU, and with the maximum heap size of the JVM set to 48GB. All tests are repeated 30 times inside the same JVM to minimize the impact of random variations and to allow JIT compilation to improve JVM performance. The median execution time is reported. All tests are also run across a range of inputs to quantify where performance is lost or gained using accelerated parallel regions. These experiments use the Hotspot JVM v1.8.0_45 and the NVIDIA OpenCL 1.1 implementation included with CUDA 6.0.1.

Three benchmarks were used to evaluate HJ-OpenCL: KMeans, PageRank, and NBody.

KMeans iteratively finds `K` clusters in an input dataset of `P` points. Each iteration of KMeans is a two-stage pipeline: the first stage classifies each point into a cluster based on a Euclidean distance measure from each cluster's centroid to that point's location. The second stage computes new cluster centroids based on the point memberships calculated in the first stage. This pipeline is executed for `I` iterations or until the cluster centroids converge to a steady state. In our experiments we keep `I` as a constant, 10, but vary the number of clusters and data points. The first stage is parallelized across data points and the second stage is parallelized across clusters. The second stage allocates a new `Point` object for each cluster to store the re-calculated cluster centroid.

NBody simulates particle-particle force interactions and the resulting changes in particle positions. This implementation of NBody is precise, and does not use grid approximations to reduce the ratio of computation to communication. Each time step of an NBody simulation includes a two-stage pipeline: the first stage updates each particle's acceleration and velocity based on the position and mass of all other particles. The second stage updates each particle's position based on its re-computed velocity. In this evaluation we execute 20 timesteps and vary the number of points. Both stages of NBody are parallelized across particles.

PageRank is also an iterative application with a two-stage pipeline which assigns a rank to each node in a directed graph based on the ranks of its neighbors with inbound edges. The first stage in PageRank's pipeline assigns a weight to each edge in the graph based on the source node's rank and its number of outbound edges. The second stage uses edge weights to re-compute each nodes' rank based on the weights assigned to its inbound edges. We keep the number of iterations as a constant, 10, but vary the number of nodes and edges. The first stage of PageRank is parallelized across edges in the graph and the second stage is parallelized across nodes.

Table 5.1 details the characteristics of each benchmark as they relate to evaluating the contributions of this work. The first column indicates if any of the parallel regions in the benchmark contain object references. The second column indicates if this benchmark contains transfers to or from the accelerator which our transfer

Benchmark	**Obj Refs**	**Copy Elim**	**Dyn Alloc**
KMeans	Y	Y	Y
NBody	Y	Y	N
PageRank	Y	Y	N

Table 3.1 : Characteristics of each benchmark as they relate to the contributions of this work.

elimination algorithm identifies as redundant. The third column indicates if any dynamic memory allocations are performed in parallel regions of this benchmark.

Kernel Performance With Redundant Transfers

Initially, we compare performance of every parallel region running as a `forall` loop and as an accelerated `forall_acc` loop on a GPU with no redundant transfer elimination. All input and output singleton objects, primitive arrays, and object arrays are transferred to and from the accelerator in these experiments. The results for each benchmark are listed in Figures 3.8, 3.9, and 3.10. For each dataset and kernel, either the `forall` or `forall_acc` results are shaded gray to visually indicate the higher performing execution mode. In many cases, smaller datasets perform better when running on the JVM with the HJlib runtime, and larger datasets perform better when an accelerator is used (e.g. `updateClusters` in KMeans and `updateRanks` in PageRank).

These results show that even with redundant transfers, sufficiently large datasets which produce enough parallelism benefit from the accelerated parallel regions implemented in this work. Note that for improved performance to be achieved, the computational acceleration must be sufficient to offset both increased costs from transfers over the PCIe bus as well as costs from data serialization and deserialization.

Figure 3.8 shows that KMeans acceleration is primarily a function of the number of clusters (`K`) being calculated. `K` serves as a multiplier of the amount of work performed for each data point. Because communication scales by `O(P + K)`, computation scales by `O(PK)`, and `P` is generally much larger than `K`, increasing `K` increases the chances that the accelerator will have a measurably positive impact on overall parallel region execution time.

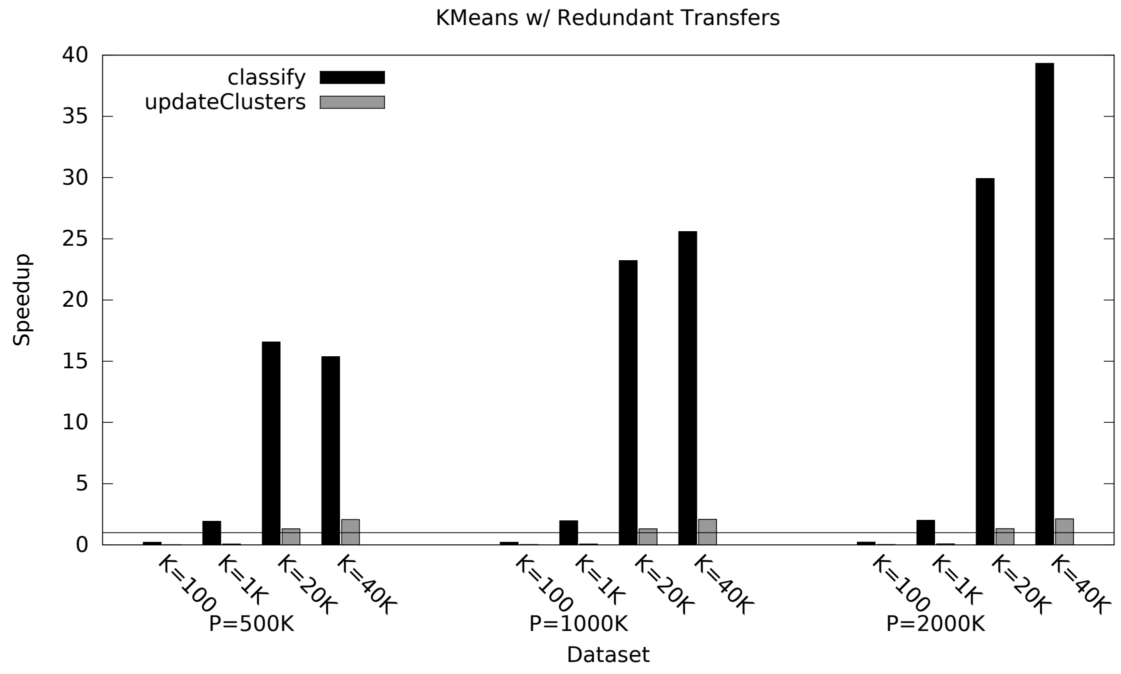

Figure 3.8 : GPU kernel speedup with redundant copies in the KMeans benchmark, relative to multi-threaded JVM execution.

Figure 3.9 shows that while accelerated parallel regions in NBody have an impact at larger datasets for `updateVel`, `updatePos` always runs faster on the JVM with the HJlib parallel runtime. The `updatePos` kernel includes a trivial amount of work but requires transferring and serializing JVM objects to and from the accelerator. Hence, overheads dominate accelerated execution of `updatePos` on the accelerator.

As we would expect, Figure 3.10 shows that for small node counts the `updateRanks` kernel in PageRank (which is parallelized across nodes) does not perform well on the accelerator. Like the kernels of KMeans and `updatePos` in PageRank, `updateRanks` is dominated by transfer and serialization overheads at smaller node counts with little computational work to accelerate. However, the `calcWeights` kernel always performs

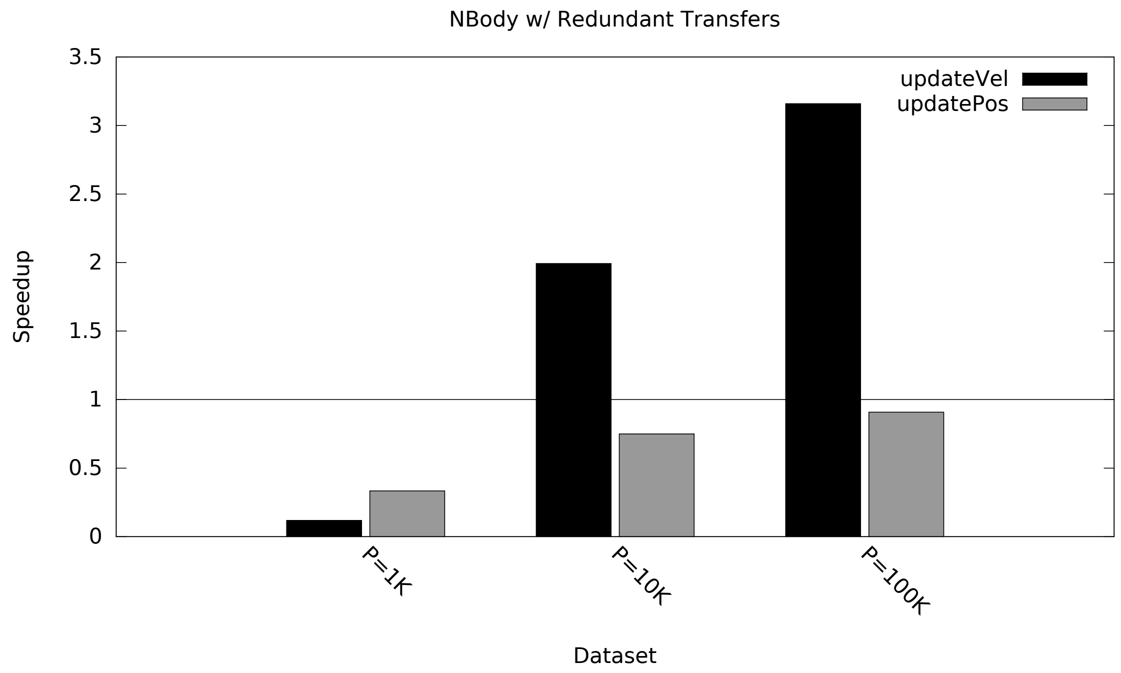

Figure 3.9 : Kernel speedup with redundant copies in the NBody benchmark, relative to multi-threaded JVM execution.

better on the accelerator than the JVM (for the data sizes that we studied).

Kernel Performance Without Redundant Transfers

Building on the results in Section 3.3.3, we enable the redundant transfer elimination described in Section 3.3.2 and rerun all experiments. For these experiments, we explicitly force all accelerated parallel regions to block on accelerator computation before returning to the host program. This simplifies the performance comparison in this section for kernels which have no buffers that must be transferred back to the JVM on completion and would therefore execute asynchronously.

Eliminating redundant transfers in KMeans does add some performance benefit. A summary of the results before enabling this optimization (copy-all) and after (elim) is provided in Figure 3.11. Redundant transfer elimination saves transferring 50% of data by eliminating all transfers from the accelerator following `classify` and to the

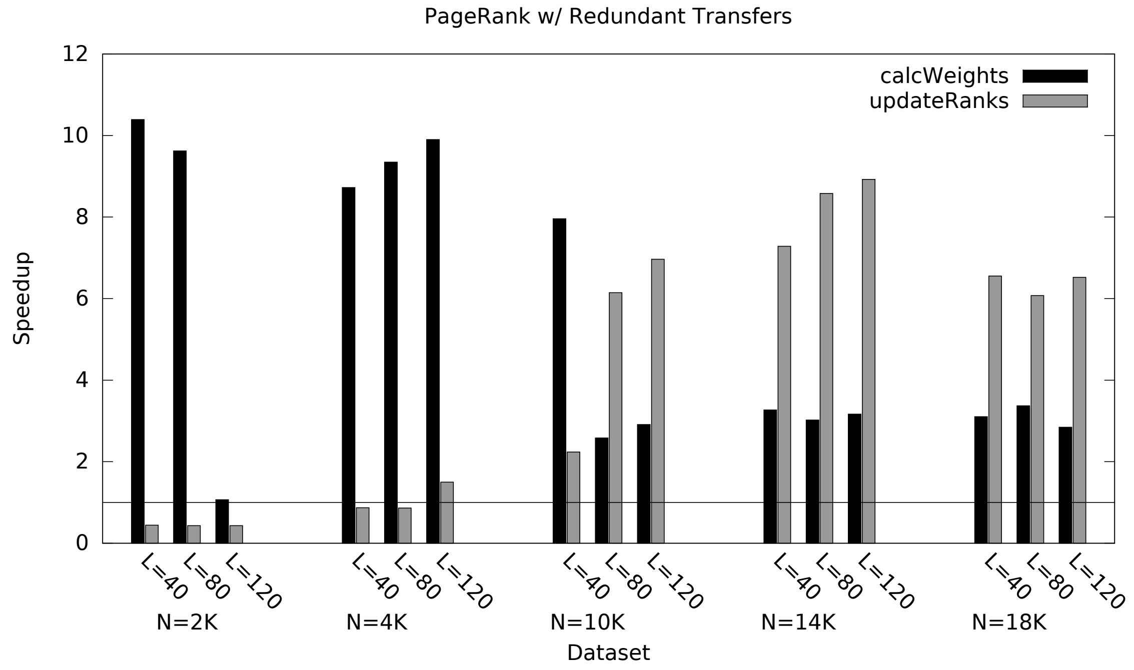

Figure 3.10 : Kernel speedup with redundant copies in the PageRank benchmark, relative to multi-threaded JVM execution.

accelerator before `updateClusters`. We can see that for every dataset this improves execution time. However, for kernels and datasets where JVM execution was faster than accelerated execution with redundant transfers (Figure 3.8), the performance improvement from redundant transfer elimination is insufficient to make those execution configurations now faster on the accelerator.

We find that for the NBody `updateVel` and `updatePos` kernels there is no improvement in performance. While our redundant transfer elimination algorithm keeps the velocity and position buffers on the accelerator between these two kernels, our NBody implementation is heavily computation-bound and so eliminating these transfers does not significantly improve the performance of the overall parallel region. For simulations of 100,000 particles this elimination reduces bytes transferred to and from the device by 58.8%.

The PageRank results with redundant transfers eliminated are summarized in Fig-

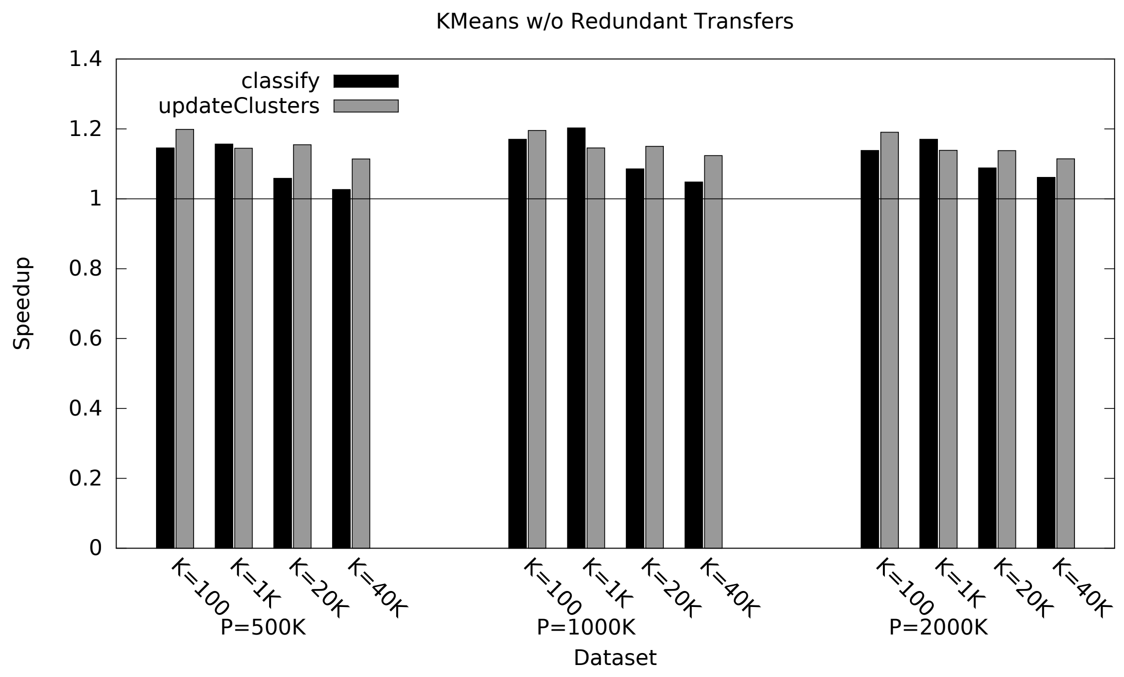

Figure 3.11 : Kernel speedup with redundant copy elimination in the KMeans benchmark, relative to multi-threaded JVM execution.

ure 3.12. Here, eliminating redundant transfers provides a clear performance benefit across all datasets. In every case, the optimized version performs better than the naive, copy-everything version. Redundant transfer elimination reduces the number of bytes transferred to and from the accelerator by 50%.

While the results for PageRank are similar to KMeans in that no kernel and dataset execution configuration which performed better on the JVM in Figure 3.10 now performs better on the accelerator, it is clear the inflection point at which the accelerator is the better choice has moved down as a function of `L`. For example, consider the kernel `updateRanks` for tests with `N=4K`. Let us fit a quadratic function `f(L) = E` to this data where `E` is the expected execution time for `updateRanks`. Without redundant transfer elimination, the fitted equations for `forall` and `forall_acc` predict an inflection point at `L=91`. With redundant transfer elimination, that inflection point reduces to `L=84`, increasing the number of datasets that execute faster on the accelerator.

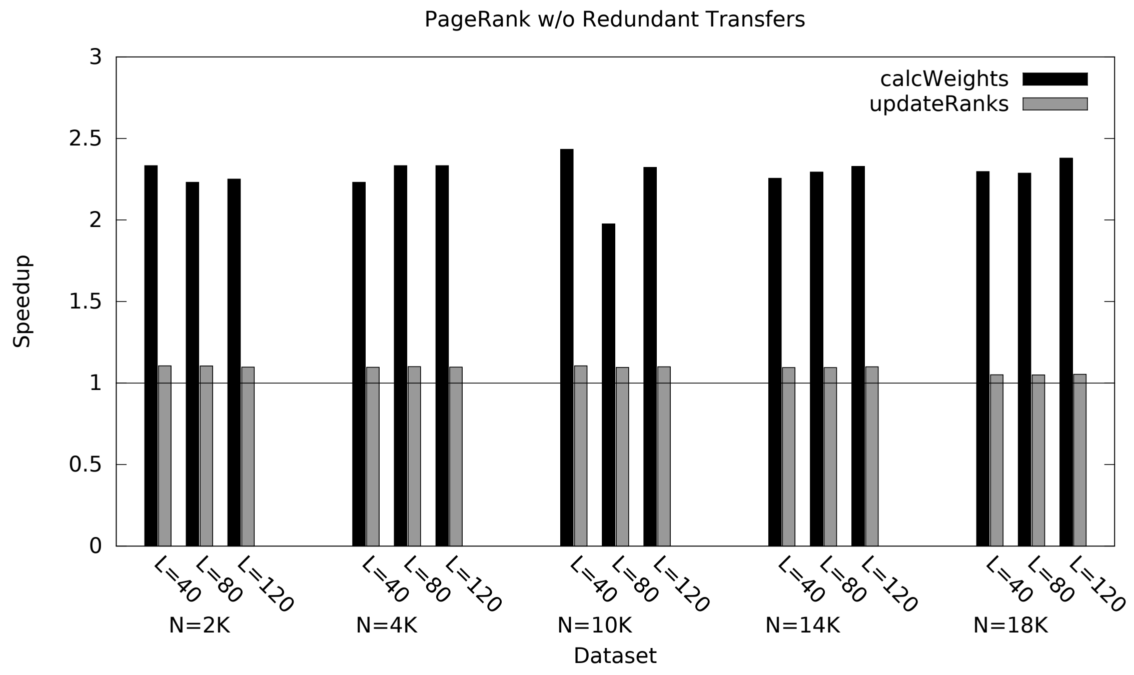

Figure 3.12 : Kernel speedup with redundant copy elimination in the PageRank benchmark, relative to multi-threaded JVM execution.

Overall Speedup

Using the insights gained in Sections 3.3.3 and 3.3.3, this section measures the overall speedup of whole benchmarks relative to a parallel implementation using Java Streams. We choose to compare performance against Java Streams because it is an industry standard with similar parallel functionality to HJlib's `forall` construct. Table 3.2 lists the kernels we choose to offload to the accelerator; `updatePos` in NBody is the only kernel that was not offloaded. Table 3.3 lists the overall speedup achieved on each benchmark and each dataset. All speedups are normalized to the execution time of an implementation that uses parallel Java Streams. We compare performance of HJlib's `forall` running in the JVM, `forall_acc` using OpenCL to run on the GPU, and `forall_acc` using OpenCL to run on the CPU.

Table 3.3 shows that the GPU-accelerated version of each application generally performs better than Java Streams, `forall`, and the CPU-accelerated version as the data size and parallelism of the input dataset increases.

The `forall` and Java Streams versions of each benchmark generally outperforms the GPU-accelerated version on smaller datasets where there is insufficient work for the GPU's parallelism to offset the overheads of data serialization and transfer.

The CPU-accelerated version (i.e., the version in which OpenCL code is executed on the CPU) of each benchmark offers an interesting tradeoff compared to the other execution platforms. Like `forall` and Java Streams, it executes on the CPU and therefore handles irregular computation and memory accesses better than the GPU. The CPU also handles non-coalesced memory accesses better than the GPU, a common access pattern when referencing arrays-of-structs rather than structs-of-arrays. Due to the serialization techniques described in Sections 3.3.1 and 3.3.1, all of these benchmarks operate on arrays-of-structs. However, when using the CPU as an accelerator we incur the same data serialization overhead that we do on the GPU, but the data transfer overhead is lower because it does not go over the PCIe bus. These results show that for some benchmarks there is a middle ground where the size of the dataset is sufficiently large for native CPU execution to demonstrate a performance benefit over JVM execution (`forall` or Java Streams) despite serialization overheads, but still small enough that the acceleration from GPU execution is insufficient to offset the transfer overheads. In particular, note the results for KMeans when `K=1K` or the PageRank dataset `N=4K, L=120`.

Automatic runtime identification of the best performing configuration (JVM, native OpenCL code on CPU, native OpenCL code on GPU) for a given kernel is a subject for future work.

Benchmark	Kernel	Accelerator?
KMeans	classify	Y
	updateClusters	Y
NBody	updateVel	Y
	updatePos	N
PageRank	calcWeights	Y
	updateRanks	Y

Table 3.2 : Kernels selected for acceleration.

Performance Degradation as Heap Contention Increases

One of the contributions of this paper is support for dynamic memory allocation on OpenCL accelerators. The techniques described in Section 3.3.2 enable acceleration of JVM applications where the dynamic memory allocations exceed the size of the heap that is allocatable on the accelerator. Because these techniques are based on retrying threads that fail to complete successfully, launching multiple kernels per parallel region becomes necessary, but naturally introduces overhead. In this section, we study how overhead and overall execution time of the KMeans benchmark increases as we artificially constrain the heap size to force allocation failures.

In KMeans, a new object is allocated on each iteration for each cluster that stores the new coordinates of that cluster. In these experiments, we test against the dataset with the largest `K`. For `K=40,000`, KMeans dynamically allocates 12 bytes per cluster (480KB in total).

Tables 3.4 and 3.5 show the number of kernel retries and total execution time for KMeans running on the GPU and CPU, as a function of the heap size. As we halve the heap size, the number of retries necessary approximately doubles. Overall execution time increases sub-linearly because each successive kernel retry in the same parallel region contains less work than the preceding one as more parallel iterations complete successfully. The execution time of Java Streams on the same KMeans dataset was 70,091 ms. Even with an artificially small heap size, HJ-OpenCL running on the GPU and CPU is able to maintain a performance advantage over Java Streams.

	Dataset	HJlib	CPU	GPU
KMeans	P=500K, K=100	0.97×	0.19×	0.09×
	P=500K, K=1K	1.01×	1.31×	0.61×
	P=500K, K=20K	1.23×	6.92×	10.26×
	P=500K, K=40K	1.21×	5.81×	11.94×
	P=1000K, K=100	1.05×	0.22×	0.10×
	P=1000K, K=1K	1.12×	1.63×	0.71×
	P=1000K, K=20K	1.06×	6.93×	10.36×
	P=1000K, K=40K	1.23×	7.51×	15.78×
	P=2000K, K=100	1.05×	0.21×	0.01×
	P=2000K, K=1K	1.22×	1.63×	0.71×
	P=2000K, K=20K	1.10×	7.42×	11.14×
	P=2000K, K=40K	1.23×	8.74×	18.33×
NBody	P=1K	0.50×	0.07×	0.08×
	P=10K	1.02×	0.61×	0.89×
	P=100K	0.89×	0.72×	1.23×
PageRank	N=2K, L=40	1.00×	0.74×	0.45×
	N=2K, L=80	1.04×	0.81×	0.45×
	N=2K, L=120	1.03×	0.80×	0.45×
	N=4K, L=40	1.03×	1.05×	0.89×
	N=4K, L=80	1.05×	1.05×	0.90×
	N=4K, L=120	0.93×	1.83×	1.53×
	N=10K, L=40	1.03×	1.41×	2.43×
	N=10K, L=80	0.98×	2.61×	5.95×
	N=10K, L=120	1.02×	1.85×	6.45×
	N=14K, L=40	0.96×	3.11×	7.05×
	N=14K, L=80	0.98×	1.69×	8.60×
	N=14K, L=120	0.98×	1.78×	9.03×
	N=18K, L=40	0.97×	3.06×	5.88×
	N=18K, L=80	0.95×	1.76×	6.57×
	N=18K, L=120	1.04×	1.66×	6.68×

Table 3.3 : Speedup of overall execution time for all benchmarks, relative to parallel Java Streams. This table compares HJlib's `forall` to HJ-OpenCL's `forall_acc` using the GPU or CPU as accelerators. The fastest performing platform for each test is highlighted.

Heap Size	Retries	GPU	
		Time	Slowdown
800KB	1	6,202 ms	
400KB	2	6,965 ms	1.12×
200KB	3	8,434 ms	1.36×
100KB	5	13,079 ms	2.11×
50KB	10	22,682 ms	3.66×

Table 3.4 : Performance degradation of the KMeans benchmark as the HJ-OpenCL heap size is reduced on the GPU, tested with 500,000 data points and 40,000 clusters.

Heap Size	Retries	CPU	
		Time	Slowdown
800KB	1	12,323 ms	
400KB	2	13,210 ms	1.07×
200KB	3	16,492 ms	1.34×
100KB	5	23,317 ms	1.89×
50KB	10	37,557 ms	3.05×

Table 3.5 : Performance degradation of the KMeans benchmark as the HJ-OpenCL heap size is reduced on the CPU, tested with 500,000 data points and 40,000 clusters.

The performance of CPU-accelerated HJ-OpenCL degrades at a slightly slower rate because the latency to transfer `anyFailed` from the accelerator to the JVM to check for failed allocations after every kernel launch is lower.

3.4 Accelerating Distributed Data Analytics Platforms Using HCL2 and SWAT

Section 3.3 focused on automatic offload of shared-memory parallel JVM programs and described the foundational techniques for transparent accelerator offload of parallel JVM regions. This section will build on and improve those techniques, with a focus on using them to accelerate distributed managed systems.

Our first experiences accelerating distributed data analytics platforms came in the form of HadoopCL [56]. HadoopCL accelerated Hadoop mappers and reducers using the same APARAPI code generation framework as HJ-OpenCL. While HadoopCL was an effective prototype and learning experience, it was far more restrictive than the HCL2 work that will be presented in this section as part of this dissertation. It relied more heavily on APARAPI for runtime memory management, code generation, and kernel offload, and as a result used many blocking operations. HadoopCL was limited to primitive data types only, and hence the application scope was also limited.

HCL2 shares the same goals as HadoopCL did: transparent acceleration of user-written Hadoop Mappers and Reducers. However, it differentiates itself by its more fully featured API and mature runtime, enabling evaluation on more complex benchmarks. Among other things, HCL2 also adds:

1. Auto-scheduling of tasks on HCL2 devices based on learned performance profiles (something HadoopCL completely lacked).

2. Use of the JVM device to allow for computation to run outside of OpenCL.

3. Automatic compiler optimizations/transformations for auto-generated OpenCL kernels.

4. An integrated profiling and debugging framework for studying and debugging HCL2 application behavior.

SWAT (Spark With Accelerated Tasks), on the other hand, was a follow-on to HCL2 that accelerates the computational regions of Spark programs. Spark is a distributed, functional, data-parallel programming model that focuses on iterative machine learning workloads. Spark programs consist of chained functional transformations (e.g. `map`, `reduce`) on distributed vectors using user-provided kernels. SWAT automatically offloads these user-provided kernels to accelerators.

The design of both the HCL2 and SWAT code generators and runtimes will be described and compared in this section.

3.4.1 Background: Hadoop MapReduce and Apache Spark

Hadoop MapReduce

Hadoop is a distributed MapReduce[57] programming system. It improves on other distributed frameworks in many areas, including programmability and flexibility.

Hadoop's programmability is derived from its high-level MapReduce programming model and its simple, object-oriented API. From a programmer's perspective, the MapReduce programming model (depicted in Figure 3.13) divides computation into two stages: map and reduce. The map stage applies a function to each of many input key-value pairs (kv-pairs), and outputs zero or more kv-pairs per input. Then, the reduce stage's kernel is applied to all map output values paired with the same key, reducing that collection of inputs to zero or more output kv-pairs per unique key. In addition to map and reduce, Hadoop also supports a combine stage that acts as an intermediate reduce and executes spatially near each map instance, reducing data movement and memory utilization as a result. The application-specific logic for each of these stages is implemented as single-threaded logic in Java classes. The workload for a Hadoop job can then be transparently mapped to multi-processor, shared-memory machines by taking advantage of the parallelism inherent in the MapReduce model.

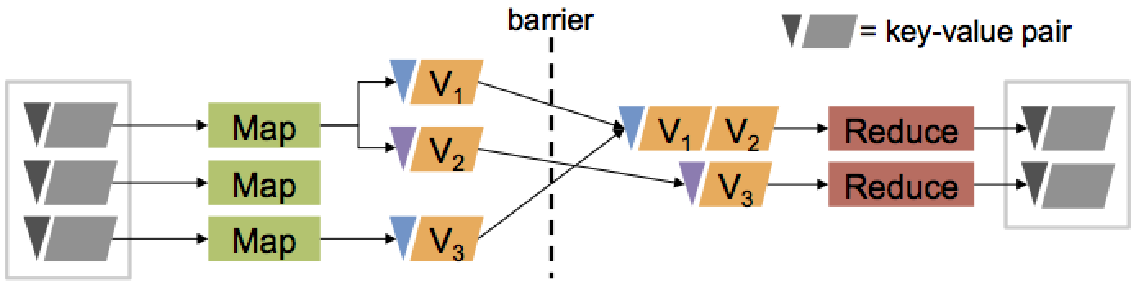

Figure 3.13 : The Map-Reduce execution flow.

A Hadoop job is an instance of a Hadoop application containing a map, a reduce, and an optional combine stage that is executed on user-specified input. The map, reduce, and combine stages of a Hadoop job are split into many parallel tasks. Each of these Hadoop tasks iterates over the input data assigned to it and applies the user-provided map or reduce function to each input kv-pair.

Hadoop TaskTrackers in each node pull tasks from a centralized Hadoop JobManager. The JobManager manages the tasks that make up each job, tracking which tasks are eligible for execution. Each TaskTracker manages a constant number of task slots in a node. A single slot generally maps to a single CPU core. The TaskTracker greedily pulls work from the JobManager as slots become available. Each task is executed in a Child JVM running as a separate process. In this way, Hadoop jobs that have been split into tasks can be scheduled across a distributed, homogeneous system and use all available CPU cores.

Hadoop is flexible in terms of the applications that can execute on it and the data types it supports. This flexibility can primarily be attributed to its use of the object-oriented JVM. The ability to represent, serialize, and strongly type-check user objects is useful when building complex applications.

Spark

Apache Spark, on the other hand, is a distributed, multi-threaded, in-memory programming system. The core abstraction of Apache Spark is that of a resilient distributed dataset (RDD). An RDD represents a distributed vector of elements. Elements in an RDD can be of any serializable type. RDDs are created and transformed through massively parallel, functional transformations. For example, a new RDD might be created by applying a parallel `filter` operation to an existing RDD which filters out all values below a given threshold.

RDD creation is lazy: creating an RDD object in a Spark program does not necessarily evaluate and populate its contents. Only certain operations in Spark programs force evaluation, resulting in long chains of lazily evaluated RDDs as one RDD is transformed into another. RDD resiliency derives from this ancestry tracking. By maintaining information on how RDDs are created rather than their actual contents, Spark guarantees that lost data can be recovered through re-computation without storing large amounts of intermediate data on disk.

One of Spark's strengths is its API, i.e. the transformations that it supports on RDDs. Spark transformations run in parallel across the machines that an RDD is stored on. Transformations are functional: they are applied to one RDD and produce another. This leads to long chains of lazily evaluated RDDs, linked by functional transformations. The transformations that Spark supports include `map`, `reduce`, `filter`, `reduceByKey`, `groupByKey`, `join`, `distinct`, and more. This variety of transformations greatly expands the flexibility of Spark relative to its predecessor, Hadoop MapReduce.

Transformations generally take some Scala lambda `f`, apply it to the input RDD using the semantics of the transformation, and produce some output RDD. For example, the `map` transformation applies `f` to each element of the input RDD, producing the corresponding element in the output RDD.

We refer to the processing of a single RDD partition by a single transformation as a Spark "task". A single RDD is split into multiple partitions. All elements in the same partition are stored on the same machine, but different partitions may be stored on different machines. Hence, partitions are the granularity of distribution in Spark.

Hence a Spark task, Hadoop task, and a single parallel loop in HJ-OpenCL are all similar forms of shared-memory parallelism.

Spark also supports broadcast variables. Spark broadcast variables are read-only data structures accessible on every node of a Spark cluster. Broadcast variables are an efficient way to share read-only data among all tasks in a Spark job.

Accelerating Spark and MapReduce

Both Spark and MapReduce jobs are partitioned into tasks which in turn apply some user-defined kernel across a chunk of inputs. These tasks are each analogous to a shared-memory parallel loop, and as a result we can re-use many of the code generation, data serialization, and dynamic memory management techniques developed as part of HJ-OpenCL (see Section 3.3). As a result, we also inherit the same constraints on supported user kernels (e.g. no exceptions, dynamic method dispatch, nested object references, etc.).

Instead, the novel contributions made by HCL2 and SWAT focus on resource management and workload scheduling in multi-tenant systems. Whereas in HJ-OpenCL we focused on efficiently executing one shared-memory parallel loop after another, in HCL2 and SWAT the computational workload consists of many concurrently launched parallel loops coming from different parallel tasks in the same node. The HCL2 and SWAT runtimes are responsible for managing the scheduling of these parallel loops on accelerators, in addition to any necessary serialization or communication.

Additionally, the work on HCL2 and SWAT emphasizes maintaining compatibility with the existing APIs that MapReduce and Spark users will already be familiar with. Recall that this work on acceleration of managed runtimes focuses on domain experts. Hence, we want to avoid changes to and constraints on the high-level MapReduce and Spark APIs.

One important difference between the HCL2 and SWAT works comes from the ways in which MapReduce and Spark managed shared-memory parallelism. In MapReduce, multiple processes are spawned in parallel in each node with each processing a different task. Spark instead uses a single process with multiple threads, distributing the workload across threads rather than processes. This differentiation places different constraints on the solution for acceleration of MapReduce and Spark.

Additionally, HCL2 is implemented as a modification to the core Hadoop code. SWAT is built as a third-party JAR that does not require changes to the core Spark code. The latter approach simplifies installation and deployment, but we will see in Section 3.4.8 that modifying the core runtime leads to higher speedups by enabling optimizations that are not possible as a third-party JAR.

In the following sections, we will cover in detail the HCL2 and SWAT APIs, code generation extensions, memory management, runtime scheduling, and built-in tools. We focus on areas in which HCL2 and SWAT go further than the work in HJ-OpenCL.

3.4.2 APIs for Accelerated Data Analytics

The HCL2 and SWAT APIs are different from each other in two regards. First, because each was designed to emulate the APIs of their host systems, the HCL2 and SWAT APIs mirror many of the differences between the Hadoop and Spark APIs. Second, and more importantly, because SWAT was a later work that built on lessons learned with HCL2, its APIs are also simpler and require fewer changes to existing Spark programs.

HCL2 API

As Section 3.4.1 described, Hadoop programmers write Java classes to implement custom Mapper and Reducer logic. The guiding principle of the HCL2 API was to retain as much similarity to Hadoop MapReduce as possible. As a result, HCL2 applications are developed entirely in the Java programming language and compiled into JARs, like Hadoop applications. Similar to Hadoop, the map, combine, and reduce stages are each defined by Java classes which extend type-specific Mapper, Combiner, and Reducer superclasses. Figure 3.14 depicts example HCL2 mapper and reducer implementations.

The types in each mapper and reducer superclass name (e.g., `IntPairBooleanLongMapper`) indicate the input and output key and value types for that computation. These superclasses are auto-generated for a range of primitive, composite, and sparse vector data types. For example, `PiMapper` takes

```
public class PiMapper extends
    IntPairBooleanLongMapper {
  void map(int pid, double valx, double valy) {
    ...
  }
}

public class PiReducer extends
    BooleanLongBooleanLongReducer {
  void reduce(boolean inside,
      HadoopCLLongValueIterator values) {
    ...
  }
}
```

Figure 3.14 : Example HCL2 Mapper and Reducer implementations extending type-specific Mapper and Reducer superclasses.

a kv-pair of (`int`, `Pair`) as input and outputs a kv-pair of (`boolean`, `long`). The `Pair` type is an HCL2-supported composite type containing two double-precision floating-point values.

HCL2 supports globally shared read-write sparse vectors within a Hadoop job. Many Hadoop applications exhibit a pattern of 1) initialize global data on task setup, 2) read and modify global data at each kv-pair, and 3) if global data was modified then write those modifications to the Hadoop Distributed Filesystem (HDFS) on task cleanup. Therefore, to support this pattern, HCL2 exposes a simple API for interacting with global sparse vectors.

The API for initializing global sparse vectors is straightforward. During job initialization, a sparse vector Java object is passed to HCL2 with flags indicating if it is writable and a unique ID to identify that global vector.

These globals are made accessible to the application code through the API below. Each global sparse vector is keyed by its unique integer ID. The dimensions, values, and length for that sparse vector can be fetched using that ID.

```
int[] getGlobalIndices(int GID);
```

```
double[] getGlobalVals(int GID);
int getGlobalLength(int GID);
```

Utility functions are also provided for quick lookup, and for manipulation of elements in the global vectors using supported mathematical operations (e.g., increment).

These user-initialized global sparse vectors are stored in HDFS files so that they can be accessed from inside the Hadoop job. Section 3.4.5 provides more details on how these global data structures are managed at runtime and made accessible to user computation inside a job.

SWAT API

In a vanilla Spark program, RDDs are created by applying transformations or operations to other RDDs. In the example code snippet below, an RDD of integers is created from a file stored in HDFS, and a new RDD is created where element `i` contains the value of element `i` in the first RDD, multiplied by two:

```
val input = sc.objectFile[Int](hdfsPath)
val doubled = input.map(i => 2 * i)
```

To run this kernel on an accelerator, SWAT simply requires that the `input` RDD be wrapped by a custom SWAT RDD object using a `cl` API call (shown below). No other code change is required, and this is the only method exposed by SWAT. By wrapping the RDD object, SWAT can intercept transformations performed on it and replace the JVM implementations of those transformations with semantically equivalent but accelerated versions.

```
val input = cl(sc.objectFile[Int](hdfsPath))
val doubled = input.map(i => 2 * i)
```

SWAT currently supports intercepting and accelerating calls to Spark `map` and `mapValues` transformations. Other transformations could be supported, but we have not found motivating application kernels that use other transformations and would benefit from acceleration. For example, `filter` kernels tend to be short-lived and the GPU offload time would be dominated by overheads. Future work could investigate the use of kernel fusion across chained transformations to produce larger GPU kernels. These fused kernels might offset the offload overheads, making offload of lightweight transformations like `filter` profitable.

Note that Spark's built-in broadcast variables are similar to the global sparse vectors introduced by HCL2, hence no new APIs had to be introduced in SWAT to support that functionality.

3.4.3 Runtime Code Generation

While HCL2 uses the code generation work from APARAPI and HJ-OpenCL unchanged, SWAT makes framework-specific extensions to it.

APARAPI-SWAT

SWAT extends the code generation work done as part of HJ-OpenCL to support references to Spark-specific data structures in kernels. In particular, we support the `SparseVector` and `DenseVector` classes from Spark's MLlib, and the Scala `Tuple2` class used to store key-value pairs in Spark. Figure 3.15 illustrates an example of the OpenCL kernel code generated to store and manipulate a `DenseVector` object.

One important item to note in the definition of the `DenseVector` struct is the addition of a `stride` field. During serialization of `DenseVector` JVM objects to native structs that can be accessed on the GPU, we tile and stride `DenseVector` objects to improve memory access coalescing on the GPU. This transformation places the `i`th element of neighboring `DenseVector` objects adjacent to each other. In our implementation, we tile 32 `DenseVector` objects together before striding them because NVIDIA GPUs schedule threads in "warps" of 32 threads. However, the implementation is structured so that this can be easily tuned when porting to new architectures. These same optimizations are also performed on `SparseVector` objects.

```
typedef struct __attribute__ ((packed)) dv {
  __global double*  values;
  int  size;
  int  stride;
} DenseVector;

static int DenseVector__size(
    __global DenseVector *this) {
  return (this->size);
}

static double DenseVector__apply(
    __global DenseVector *this, int index) {
  return (this->values)[this->stride * index];
}
```

Figure 3.15 : Generated OpenCL kernel code for storing and manipulating a Spark `DenseVector` object.

The automatic optimization of the user-written kernels during code generation is beyond the scope of this work, but ideas from related work [58][59][60] could be integrated into this code generator in the future.

3.4.4 Runtime Accelerator Memory Management

At runtime, HCL2 and SWAT must dynamically allocate and release accelerator memory. Because memory allocation on GPUs generally requires synchronizing all running kernels on the device, both HCL2 and SWAT pre-allocate all device memory during initialization of a task. HCL2 and SWAT must also both contend with two entities competing for host heap allocations within a single process: the JVM heap and the OpenCL runtime.

One major difference between the constraints placed on the HCL2 and SWAT memory managers is that while SWAT must support multi-GPU memory management from within a single process, HCL2 only needs to support single-CPU memory management. This is a result of the design of Spark and Hadoop. Hadoop spawns a process per task, while Spark generally runs one process per shared-memory node and has multiple threads within that process handling different tasks. As a result, all of the GPUs within a single node are managed from a single process in SWAT, while in HCL2 processes are spread across GPUs.

HCL2 Memory Management

The HCL2 Runtime explicitly pre-allocates and manages both OpenCL and JVM memory buffers. This memory management was implemented to limit dynamic allocations on the JVM's heap and on OpenCL devices so as to prevent operating system and JVM out-of-memory errors, as well as limit overhead from excessive JVM garbage collection or device synchronization.

There are 3 types of buffers used in the HCL2 runtime, each used for a different stage of processing. Each buffer type has a fixed number of buffer instances that can be instantiated at any time. Each buffer instance is either 1) owned by the component of the HCL2 runtime which is currently operating on it, 2) stored temporarily in a queue of pending work, or 3) stored in a pool of free, pre-allocated buffer instances which are not in active use. The three buffer types are described in detail below:

1. *Input Buffer*: An Input Buffer is used to store input data in the JVM. These buffers are filled with input data from the current task's input stream before having their contents transferred to the accelerator. Input Buffers encapsulate primitive Java arrays which store Java objects in a format that OpenCL can process.

2. *Output Buffer*: An Output Buffer is used in the JVM to store data output by an accelerator kernel. Like Input Buffers, Output Buffers store Java objects as primitive arrays. Output Buffers are transferred to directly from the OpenCL device. Once the OpenCL outputs have been pulled from the OpenCL address space into an Output Buffer, the contents of these buffers are written to the next stage in the MapReduce pipeline.

3. *Kernel Buffer*: A Kernel Buffer is a JVM object that consumes little JVM memory but serves as a handle to a set of OpenCL buffers in the OpenCL

address space. To access these buffers, each HCL2 runtime component must first acquire the Kernel Buffer handle associated with them. Kernel Buffers are allocated from a pre-allocated pool before having the contents of an Input Buffer transferred to the corresponding OpenCL buffers. Kernel Buffers are active for as long as the OpenCL kernel using them. Outputs from the kernel are transferred back from the Kernel Buffer to an Output Buffer.

HCL2's memory management system consists of the 1) pre-allocation of these three types of buffers, 2) the acquisition and release of these buffers to and from a pre-allocated pool, and 3) the passing of these buffers between HCL2 runtime components as they make their way from input aggregation to final output.

SWAT Memory Management

SWAT takes a more general-purpose approach to multi-device memory management.

clAlloc is a thread-safe, single-accelerator memory management library built on top of the OpenCL APIs. It exposes two data structures: 1) an `allocator` object for each OpenCL device in a platform that serves as a context/handle for clAlloc operations on that device, and 2) `region` objects which represent a contiguous block of allocated memory on a single OpenCL device. Its API is as follows:

1. `clalloc_init(device)`: Initialize an `allocator` instance for the selected device.

2. `cl_allocate(nbytes, allocator)`: Allocate `nbytes` bytes on the device associated with `allocator`, returning a `cl_region` handle for the allocated memory or `NULL` if the allocation failed.

3. `cl_free(region, try_to_keep)`: Release the device memory represented by `region` for future allocations. If `try_to_keep` is true, clAlloc will make a best-effort to not use that memory to satisfy future allocations as it may be re-used soon.

4. `cl_reallocate(region)`: Using the provided `region`, attempt to re-allocate the same device memory. Successfully re-allocating memory guarantees that it has not been used to satisfy another allocation since `region` was originally allocated, and so its state is consistent with previous operations performed on the same `region`. If the `region` is not already free this simply increments a reference counter, allowing multiple kernels to share the same `region`.

5. `get_pinned(region)/release_pinned(buf)`: Fetch or release a page-locked buffer `buf` in host memory that matches the size of `region`. Page-locked buffers are necessary for performing asynchronous communication to or from accelerators.

6. `set_region(region, buf, nbytes)/get_region(buf, region, nbytes)`: Fill or fetch the contents of a region on an OpenCL device using a corresponding host buffer.

clAlloc adds a higher-level API on top of the standard OpenCL APIs, including features that enable higher layers of the software stack to perform data sharing across kernels and efficient data communication. clAlloc pre-allocates all device memory when it is initialized and partitions memory up for allocation requests on-demand using the OpenCL `clCreateSubBuffer` API.

Free device memory is represented by a free list, sorted by offset into the device memory address space. When a clAlloc `region` is freed with `cl_free` and with `try_to_keep` set to false, it is merged into any neighboring regions to reduce fragmentation. If `try_to_keep` is set to true, it is not merged.

clAlloc also stores free regions in buckets for efficient allocation. For a single device, `B` buckets are created. Each bucket `b` from 0 to `B`-1 stores all free regions on that device with a size between 2^b (inclusive) and 2^{b+1} (exclusive). A special-purpose bucket is used to store any regions larger than $2^B - 1$. The free regions in each bucket are kept sorted by size, from smallest to largest. Regions freed with `try_to_keep` set to true are not kept in these buckets, only in the global free list for each device.

When allocating `nbytes`, clAlloc starts with the smallest bucket that may have a region of size `nbytes`, searching larger buckets until a free region that is large enough to satisfy this allocation is found. The free region is then trimmed to `nbytes`, any leftover space is re-inserted in the free list and free buckets, and the allocated region is returned.

If no free region is found in the buckets list, the allocator reverts to a linear search of the device-global free list for adjacent free regions that can be merged to produce a sufficiently large free region to satisfy this allocation. This step will only succeed where the previous one failed if some regions freed with `try_to_keep` set to true can be merged with neighboring free regions to de-fragment device memory. If this step fails, a `NULL` region is returned.

To support multi-GPU memory management in SWAT, an `allocator` object is created for each device at initialization. Because `allocator` objects are already thread-safe, multiple threads sharing GPUs within a single Spark process can safely allocate and release device memory at runtime.

3.4.5 Runtime Coordination

Like memory management, the runtime coordination for HCL2 and SWAT is significantly different because of the use of processes for parallelism in Hadoop versus the use of threads in Spark.

HCL2 Runtime Coordination

As described in Section 3.4.1, the Hadoop TaskTracker launches a separate Hadoop Child process for each task it processes. In HCL2, the TaskTracker is modified to assign each Child a single device to work with (Section 3.4.6 will describe in more detail how devices are chosen for tasks). Within that Child process, the HCL2 runtime is responsible for scheduling execution of user-defined computation on the device assigned to the task, as well as handling any necessary communication or management work.

HCL2 supports three types of HCL2 “devices”: native OpenCL threads on GPUs, native OpenCL threads on CPUs, and execution in the JVM. HCL2 is sufficiently flexible to support additional OpenCL architectures as they become available.

During initialization of the Child process, the HCL2 Runtime loads the global sparse vectors described in Section 3.4.2 from HDFS. If this Child is assigned the JVM device, then no further action is necessary as the globals are now in the JVM’s address space. If this child is assigned an OpenCL device, OpenCL buffers are pre-allocated and initialized with these global values before processing begins.

When using the JVM device, the HCL2 Runtime mirrors the workflow of a normal Hadoop Child process. It iterates single-threaded over the input kv-pairs, calls the user-defined map or reduce function on each, and outputs kv-pairs one at a time. The work described in this paper does not significantly change this process.

When running on an OpenCL device, the HCL2 runtime chunks input and output data points into data buffers. The HCL2 runtime follows the following steps to process these data buffers on an OpenCL device:

1. The bytecode loaded for this task’s `map()` or `reduce()` function is translated to an OpenCL kernel using the techniques described in Section 3.3.1.

2. A dedicated input I/O thread, called the Input Aggregator, buffers many kv-pairs from the input stream for this task in to an Input Buffer `D`.

3. Once it is full, `D` is passed to the Buffer Executor, a separate thread which acquires a Kernel Buffer on the OpenCL device assigned to this task, transfers the inputs contained in `D` to the OpenCL memory buffers associated with that Kernel Buffer, and launches the OpenCL kernel created in step 1.

4. The Buffer Executor detects the completion of processing for `D`, transfers its outputs back to an Output Buffer in the JVM, and passes that Output Buffer

to a dedicated I/O thread, the Output Writer, to be written out.

5. If there are inputs left to process, control loops back to step 1. Otherwise, this task terminates.

Note that while these steps are described sequentially, most of the actual processing of a data buffer `D` is asynchronous and does not require a component (e.g., Input Aggregator) to block on `D` completing unless the storage or compute resources required for forward progress by a component have been exhausted. Examples of resources that may cause blocking are pre-allocated OpenCL memory buffers, pre-allocated JVM buffers, or the OpenCL device.

SWAT Runtime Coordination

SWAT's runtime design is similar to HCL2 in that it uses a pipeline of JVM threads to aggregate inputs, execute kernels, and produce outputs.

Sitting on top of the clAlloc memory management layer described in Section 3.4.4 is the **SWAT Bridge**, a bridge between the components of SWAT running in the JVM and those sitting on top of OpenCL. SWAT Bridge's upward exposed APIs are expressed in terms of JVM or Spark objects, which it then translates into commands to the OpenCL-centric layers below.

The Bridge's primary responsibilities are the caching of data on OpenCL devices across kernels and tasks, the creation and management of native SWAT contexts, the setting of arguments to OpenCL kernels, and the management of asynchronously executing OpenCL operations (including kernel executions and data communication). It exposes APIs to the JVM that enqueue work for accelerators and block or poll on their completion. At a high level, the Bridge accepts input buffers from the JVM, places them on the accelerator, launches computation on those buffers at the JVM's request, retrieves outputs, and signals the JVM on completion of various stages and as resources are released.

The Bridge stores two mappings for caching data on OpenCL devices: one mapping from unique RDD partition IDs to their clAlloc regions, and another from unique broadcast variable IDs to their clAlloc regions. When layers higher in the software stack indicate that a partition or broadcast variable should be allocated and populated on a device, the Bridge first checks if an entry already exists for it in one of the cached mappings. If it does and a call to `cl_reallocate` succeeds on it, the Bridge can skip creating a separate allocation. This saves space on the device through deduplication, and reduces data communication to the device. When freeing regions associated with cache-able data, the `try_to_keep` flag is set to true.

In practice, we found that caching partitions of RDDs on the device was not useful. The limited size of device memory and the scale of Spark datasets meant that RDD partitions were never re-used before being evicted from device memory: the memory used to store them had to be allocated to store other data before the application came back around to a re-use of that partition. Broadcast variables, on the other hand, are frequently used, may be shared across multiple stages of a Spark application, and are usually smaller than an RDD partition. We generally see benefits from caching them.

Besides caching, the bridge's other main responsibility is exposing an API that allows higher layers to enqueue asynchronous OpenCL operations and check for their completion. In support of this, the bridge implements an asynchronous runtime that coordinates asynchronous OpenCL operations with the host application using pthreads condition variables, OpenCL events, and OpenCL event callbacks. This runtime is illustrated in Figure 3.16. Below the dotted line in Figure 3.16 are OpenCL operations managed by the OpenCL runtime including data communication, kernel execution, and SWAT-specific callbacks. These callbacks are identified by the light gray boxes. All of these operations are asynchronous with respect to the host JVM, with OpenCL events being used to maintain inter-operation dependencies.

The functions that the SWAT Bridge exposes to higher layers in SWAT are listed

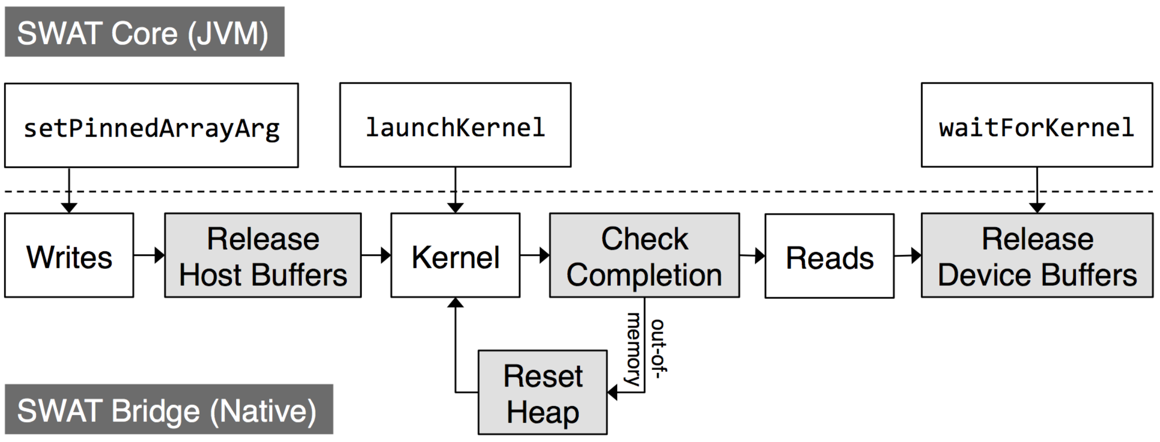

Figure 3.16 : The flow of event-driven actions in the SWAT Bridge at runtime.

below:

1. `setPinnedArrayArg`: This function initiates the transfer of the contents of a host page-locked buffer to a buffer on an OpenCL device. It also sets the appropriate arguments of an OpenCL kernel to point to the same OpenCL buffers. This call is non-blocking, and creates OpenCL events for later operations to depend on.

2. `launchKernel`: This function launches a new kernel processing data initialized using `setPinnedArrayArg`. This kernel is made dependent on the writes started by `setPinnedArrayArg` using OpenCL events. Each kernel launch is uniquely identified by a 64-bit sequence number, which is incremented by one on each kernel launch.

3. `waitForKernel`: This function forces the current JVM thread to wait for a specific kernel launch to complete, identified by its sequence number. This involves a wait on a condition variable which is set by the box labelled "Release Device Buffers" in Figure 3.16.

4. `waitForInputBuffersRelease`: This function forces the current JVM thread to wait for a set of transfers to the device to complete, signaling that the host

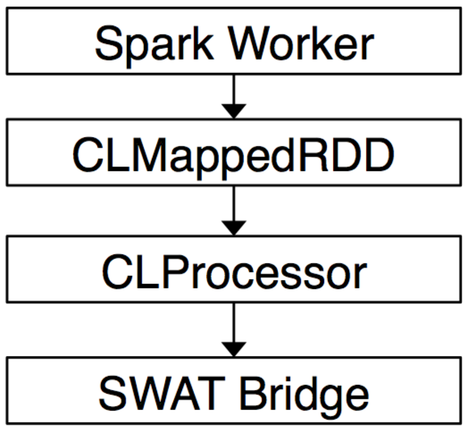

Figure 3.17 : Example stack trace of entry point to SWAT Core.

buffers they originate from are now available for re-use by the host.

All of these APIs are thread-safe as they may be concurrently called by multiple JVM threads.

On the other side of the Java Native Interface from the SWAT Bridge is SWAT Core. SWAT Core refers to the components of SWAT that sit inside the JVM. SWAT Core runs inside a Spark Worker JVM. The interface between Spark and SWAT is a custom SWAT RDD class (illustrated in Figure 3.17). When Spark has a partition to process it calls a `compute` method on the custom RDD, passing it an iterator over an input partition. The `compute` method returns an iterator over output items. SWAT currently adds two custom RDD classes: one each for the Spark `map` and `mapValues` transformations. Both of these classes hand off processing of the input partitions to a shared code base in `CLProcessor`.

`CLProcessor` has four main responsibilities: 1) setup and configuration of the SWAT environment, 2) input buffering and serialization, 3) launching a GPU batched kernel, and 4) output deserialization and writing. At a high level, `CLProcessor` accumulates many input items from the input iterator, launches a batched OpenCL kernel on the accumulated inputs, and returns the accumulated outputs to Spark through the iterator which was returned by the RDD `compute` method.

There are five categories of objects that the `CLProcessor` is responsible for initializing:

1. OpenCL objects: This includes SWAT-specific items such as clAlloc allocators, as well as OpenCL-specific items like compiled kernels and OpenCL contexts.

2. Native Input Buffers: These are JVM handles on native, page-locked buffers allocated from clAlloc. Multiple page-locked buffers may be grouped into a single Native Input Buffer handle if they are required to serialize a given input type. For example, accumulating vectors from a `DenseVector` input iterator requires three native input buffers: one buffer to store the values of each vector, one buffer to store the length of each vector, and one buffer to store the offset of each vector in the values buffer.

3. JVM Input Buffers: These are small JVM objects that contain the logic to serialize items from an input iterator into backing Native Input Buffers. Sometimes it is necessary to store small temporary buffers in JVM Input Buffers. An actively buffering JVM Input Buffer is always backed by a Native Input Buffer in which it stores the accumulated and serialized input items.

4. Native Output Buffers: Page-locked host buffers that SWAT Bridge transfers the outputs of an OpenCL kernel into, asynchronously.

5. JVM Output Buffers: JVM objects backed by Native Output Buffers that expose an iterator interface which `CLProcessor` can use to deserialize and fetch output elements from Native Output Buffers.

Only one JVM input buffer and JVM output buffer are created by `CLProcessor`. Multiple native input and output buffers are created, but are limited to a fixed number to prevent out-of-memory errors caused by excessive native buffer allocation. Only a single JVM input or output buffer is necessary as different native buffers can be swapped in and out as the storage backing the JVM buffers' interfaces.

The CLProcessor uses two JVM threads: a dedicated reader thread and the main Spark thread. A reader thread is spawned by the main Spark thread when the SWAT RDD `compute` method is called. The reader thread retrieves an iterator for a given input partition. The reader thread is responsible for accumulating items from the input iterator, through the JVM Input Buffer, and into a Native Input Buffer. It then initiates the asynchronous input copies to the accelerator from the Native Input Buffer and launches an asynchronous kernel to process them, using the SWAT Bridge. Illustrative pseudocode for the reader thread is listed in Algorithm 1.

```
currentInputSeqNo = 0;
lastSequenceNo = -1;
done = false;
jvmInputBuf = createJVMInputBuffer();
jvmInputBuf.setNativeInputBuf(fetchNativeInputBuffer());
;
while not done do
    jvmInputBuf.accumulate(inputIter);
    ;
    nextNativeInputBuf = fetchNativeInputBuffer();
    jvmInputBuf.transferOverflowTo(nextNativeInputBuf);
    jvmInputBuf.nativeInputBuf.copyToAccelerator();
    ;
    currNativeOutputBuf = fetchNativeOutputBuffer();
    bridge.setupOutputBuffers(currNativeOutputBuf);
    ;
    kernelSequenceNo = currentInputSeqNo++;
    done = bridge.run(kernelSequenceNo);
    if done then
        lastSequenceNo = kernelSequenceNo;
    end
end
```

Algorithm 1: Pseudocode for SWAT Core reader thread.

The main Spark thread for the current partition first retrieves an output iterator from the SWAT RDD object's `compute` method and then repeatedly calls that iterator's `hasNext` and `next` methods to retrieve output items for the current partition, until `hasNext` returns false. `hasNext` checks that there are no remaining output items by verifying that the input iterator is finished, there are no pending OpenCL kernels, and that there are no pending output items left in any Native Output Buffers. `next`, on the other hand, either immediately returns an output item from a currently active Native Output Buffer or loads a new Native Output Buffer by waiting for the appropriate kernel launch to finish, based on a kernel sequence number. Illustrative pseudocode for the output iterator logic is listed in Algorithm 2.

```
currentOutputSeqNo = 0;
jvmOutputBuffer = ∅;
;
Procedure next
    if jvmOutputBuffer == ∅ then
        currNativeOutputBuffer =
            bridge.waitForFinishedKernel(
                currentOutputSeqNo);
        currentOutputSeqNo++;
        jvmOutputBuffer.fillFrom(currNativeOutputBuffer);
    end
    return jvmOutputBuffer.next;
;
Procedure hasNext
    return lastSequenceNo == -1 or
        currentOutputSeqNo ¡= lastSequenceNo or
        jvmOutputBuffer.hasNext;
```

Algorithm 2: Pseudocode for the SWAT Output Iterator.

3.4.6 Performance Prediction in HCL2

One of the novel contributions of the HCL2 work was the integration of a heterogeneous device performance prediction framework. This performance prediction framework was used by an auto-scheduler to automatically select which device a given Hadoop kernel should run on. HCL2 supports three types of HCL2 ”devices”: native OpenCL threads on GPUs, native OpenCL threads on CPUs, and execution in the

JVM. SWAT, on the other hand, has a simplified device selection problem because it does not support the OpenCL CPU device. SWAT relies on the programmer to select JVM or GPU execution.

While each Hadoop task only uses a single device, multiple tasks being processed in parallel by different processes may be assigned to the same device. Running multiple tasks simultaneously on the same device keeps device utilization high even when some tasks are blocked on I/O, at the cost of potentially increased overhead from context switching and resource contention.

Deciding which device to assign to a given task is a complex problem. While the performance tradeoffs between OpenCL CPU and GPU devices are well understood[61][62], the tradeoffs between OpenCL and JVM execution are more interesting. Using the JVM eliminates the need to perform transfers into a separate address space (as is necessary in OpenCL). Depending on the characteristics of the data in an application, using the JVM may also offer memory and I/O benefits. OpenCL's batched execution model requires buffering of many input data points. This increases the working set size of HCL2 tasks and may produce bursty I/O. However, batching reads in HCL2 also enables optimizations in the runtime that can hide I/O overhead.

HCL2 uses an internal auto-scheduling framework which constructs a relationship between the computational load on a HCL2 device and a task's expected execution time. By learning from past executions of tasks of the same type, persisting this information across jobs, and using low overhead techniques to construct this relationship, the auto-scheduling framework can match or beat the performance of manual, programmer-defined scheduling. The following sections will discuss the techniques used by the auto-scheduler in more detail.

Measuring and Storing Past Performance

HCL2 stores per-device historical performance information for every task type. This historical information is used to characterize task performance on each device in a platform. A task type corresponds to a single mapper, combiner, or reducer class in Hadoop.

For every possible `(task-type, device-type)` tuple, the HCL2 auto-scheduler stores a list of past performance data points. Each element in this list includes two things: the computational bandwidth achieved by an instance of this task type in kv-pairs/ms, and an estimate of the average load on all devices in the system during the execution of that task.

Computational bandwidth is measured differently for OpenCL and JVM devices. For both, it is straightforward to measure the number of kv-pairs processed in each task by incrementing a counter for each kv-pair read from the input of a task. To calculate the time taken to process those kv-pairs on OpenCL CPU and GPU devices, a millisecond-granularity timestamp is taken at the start and end of every kernel launch.

Because JVM execution is not batched and each kv-pair is processed individually, placing timing statements around every call to a map or reduce function would significantly add to the overhead of the HCL2 runtime when auto-scheduling on JVM devices. Instead, we can only time the overall task. While this technique induces less overhead than the technique for OpenCL devices due to fewer timing statements, it also strictly underestimates the computational bandwidth of the JVM as other operations, such as I/O, are included in the elapsed time measured.

The estimated average load during a task's execution is measured in units of tasks running per device, and is calculated as the mean of the device load at the start of the task and at the end of the task.

The computational bandwidth of a task is calculated by the task itself and communicated to the TaskTracker when the task completes successfully. The TaskTracker writes these metrics to a local file immediately so that they can be reloaded on startup if the TaskTracker is shut down. The TaskTracker also passes these metrics to the auto-scheduling framework so that task characterizations can be constructed or revised.

Task Characterization

In the HCL2 auto-scheduler, task characterizations are created for each task type. Task characterizations use the historical data described in Section 3.4.6 to predict the performance of instances of that task type on each device in a platform, and provide a confidence measure for that performance prediction. Each HCL2 task characterization constructs an internal function:

$$f(D, L) \rightarrow R$$

from device type D and current device load L to expected execution rate R where R is measured in kv-pairs per millisecond.

f has a different shape (e.g., linear, exponential, etc.) for different device types. The function for each device type was chosen experimentally using performance data from manually scheduled runs. We plotted the performance of different devices running KMeans against device load and looked for trends in the data. Based on the trends we observed (discussed below), we chose a function shape to fit to the data for each device.

For JVM and OpenCL CPU devices, we found there was a clearly linear relationship between device load and task processing rate. Therefore, we use linear regression to construct a linear function from device load to task processing rate. Related works[63][64] have also used linear relationships to predict CPU performance. Generating f has a computational complexity of $O(C^2N)$, where C is the number of features and N is the number of data points. For OpenCL CPU devices, we only consider the load on that device so C is equal to one. For JVM devices, we consider the load on all devices in the system so C may be greater than one.

As an example, the function constructed for OpenCL CPU devices running the Pairwise mapper was $f(OpenCLCPU, L) = 26.67 - 1.17L$. This relationship indicates that adding more load to an OpenCL CPU device causes the expected execution rate for all tasks on that device to drop.

For OpenCL GPU devices, there was no clear relationship between device load and task performance. Rather, we observed two clusters of performance: a small cluster of slow executions caused by initialization overheads, and a larger cluster of higher-performing executions. We chose to use K-nearest neighbors to estimate task performance on GPUs. This approach predicts task performance using the mean of the K performance measurements most similar to the current one in terms of device load. K-nearest neighbors naturally disregards outliers.

The computational complexity of predicting the performance of a given task on a given GPU using K-nearest neighbors is $O(N)$ where N is the number of past performance data points. For GPUs, only the load on the GPU in question is considered as an input when predicting performance.

Before predicting task performance, a task characterization must first report if it has sufficient historical performance data on which to base a prediction. In our implementation, a task characterization is "confident" it can make an accurate performance prediction if there are any similar past executions in its historical performance data. A past execution is similar if it is for the same device and executed at a similar device load. We use an n-dimensional Euclidean distance measure to determine similarity between device loads, where n is the number of devices in a platform. Any device loads within a constant, experimentally-chosen radius of the current device load are considered similar.

Types of Scheduling Decisions

There are two types of scheduling decisions in HCL2: speculative and performant.

Speculative scheduling decisions are made to fill in gaps in a task characterization's knowledge of a particular device's performance. A speculative scheduling decision is made when a task characterization indicates it has no confidence in its performance predictions for a device at the current device load. By scheduling the current task on the no-confidence device, a performance data point is added to that task characterization, allowing it to better predict performance for future task executions. Having a well-defined model of each task's performance on each device is important in making accurate and well-performing scheduling decisions.

While speculative scheduling may lead to suboptimal task placement, any short-term performance gains that are lost are outweighed by the long-term benefits of well-characterized performance. As a result, there is generally a period of suboptimal scheduling and performance during the early executions of a new task type. Section 3.4.8 will characterize this further.

Performant scheduling decisions are made to achieve maximum performance for a task, given the available HCL2 devices in a platform and the task characterization constructed from past executions on those devices.

Auto-Scheduler Core

The core of the HCL2 auto-scheduler resides in the TaskTracker and is responsible for:

1. Keeping track of the current device load in a node, measured in tasks executing per device.

2. Selecting a device for each task based on the current load in the node and the known task characterizations (described in Section 3.4.6).

3. Communicating the selected device to the task.

The current load for all devices is stored in an integer array, with one entry for each device. The auto-scheduler increments the load for a device when a new task is assigned to it. The auto-scheduler also maintains a mapping from executing tasks to the device each is running on. When a task signals the TaskTracker that it completed successfully, the TaskTracker signals the auto-scheduler to remove that task from its accounting. The auto-scheduler uses the task-to-device mapping to decrement the appropriate device load.

In our implementation, the device for a given task is selected by first querying the ”confidence” level of its task characterization for each device in the current platform. If the task characterization has no confidence for one or more devices at the current device loads, this task is speculatively scheduled on a randomly selected no-confidence device. Otherwise, a performance prediction for each device is made. The task is assigned the device with the highest predicted performance.

Once a task has been assigned a device, a device ID is passed to the Child process running that task as a Java environment variable. This environment variable is read by the HCL2 Runtime and work is only scheduled on the selected device.

Static Scheduler

In addition to the manual programmer-controlled scheduler and the auto-scheduler, HCL2 supports a third Device Scheduler: the static scheduler. The static scheduler uses the task performance profiles generated by the auto-scheduler to make scheduling decisions, but does not update those performance profiles. The static scheduler avoids the computational overheads incurred when performance profiles are updated with new performance data.

3.4.7 Framework-Specific Tooling

Both the HCL2 and SWAT frameworks used similar techniques to supply programmer aids to help with debugging and optimizing applications built on top of them.

Debugging Auto-Generated Kernels

One of the strengths of SWAT and HCL2 is that it is possible to disable accelerated execution and run programmer-written kernels in the JVM. As a result, users can test and debug their applications from entirely within a managed runtime, taking advantage of its out-of-bounds checks, arithmetic exceptions, and other exceptions to verify the correctness of their kernel.

However, it sometimes becomes necessary to analyze the generated OpenCL kernels themselves. Both HCL2 and SWAT use runtime checkpointing of kernel inputs to enable offline debugging of both correctness and performance of auto-generated OpenCL kernels.

On each kernel launch, a snapshot is taken of OpenCL buffer contents and sizes, kernel source code, and any other state necessary to fully recreate the same kernel launch. This snapshot is written to a dump file on disk, ensuring that the saved state persists even if the process writing it crashes.

The buffers which must be saved are determined based on directional information (`IN`, `OUT`, `INOUT`) passed down from higher software layers. Only the contents of `IN` and `INOUT` buffers are written to the dump file. All buffers have their dimensionality saved.

Upon successful completion of the associated kernel launch, the dump file is deleted from disk to prevent out-of-space errors in storage-constrained systems.

Once HCL2 or SWAT job execution completes or fails, any dump files remaining on disk must be associated with kernel launches that either failed or were in-progress at job termination. A utility was implemented to parse the generated dump files and perform an identical re-execution based on their contents. This offline execution can be inspected using other debugging tools, such as print statements, `gdb`, `gprof`, `cuda-gdb`, or `nvprof`.

Analyzing Runtime Performance

Additionally, the HCL2 and SWAT runtimes both include instrumentation to help with visualizing time the runtime spends in I/O, computation, serialization, and other work.

At the core of this profiling infrastructure is a runtime logging component that logs timestamps of important events in both JVM and native execution. For example, timestamps are written at the start of input aggregation, at each OpenCL kernel launch, and for each event in the OpenCL profiling API. By correlating JVM and OpenCL timestamps within a node, fine-grain profiling is achieved. However, neither HCL2 nor SWAT currently support correlating these events across multiple nodes.

Following job completion, the timestamp logs can be fetched, post-processed, and visualized using a standalone tool developed as part of HCL2 and SWAT. This produces a visual timeline of input aggregation, output dumping, kernel processing, thread blocking, and other important HCL2-specific states. Examples are shown in Figures 3.20 and 3.21.

3.4.8 HCL2 and SWAT Performance Evaluation

In this section we evaluate the performance gains and losses made using the HCL2 and SWAT frameworks.

Experimental Setup

All benchmarks and metrics are evaluated on a hardware platform containing a 12-core 2.80GHz Intel X5660 CPU with 48GB of system RAM and two NVIDIA M2050 GPUs each with 2.5GB of device memory in each node. Nodes in this platform are connected by QDR Infiniband. Our experiments are limited to a maximum of nine nodes (one master, eight workers) by hardware availability. All experiments were run with 12 Spark executor threads or 12 Hadoop Child processes in each node. The softwate platform consists of JDK 1.7.0_80, Scala 2.11.5, Spark 1.2.0, HDFS 2.5.2,

Dataset	**Size**	**# Items**	**Scala Type**
Hyperlink	16 GB	1,289,970K	(Int, Int)
Census	14 GB	49,166K	DenseVector
ImageNet	1.3 GB	40,646K	(Int, DenseVector)

Table 3.6 : Characteristics of each dataset.

and ICC 15.0.2. For the overall and task-level performance results, each benchmark was tested ten times at each node count. For the more detailed performance analysis, median runs were selected for study.

We use six benchmarks to evaluate SWAT:

1. PageRank: A graph algorithm that ranks nodes in the graph based on the nodes that link to them.
2. Connected: Connected components graph algorithm.
3. NN: A simple neural net implementation.
4. Fuzzy: A probabilistic clustering algorithm.
5. KMeans: An iterative clustering algorithm.
6. Genetic: A genetic, evolutionary algorithm. In this case, we use a genetic algorithm to find cluster centroids.

For these six benchmarks, we evaluate on three datasets. For PageRank and Connected we use the Hyperlink Graph available from the Web Data Commons [65]. For Fuzzy, KMeans, and Genetic we evaluate on the Census dataset available from the UCI Machine Learning Repository [66]. For NN we evaluate on a subset of the images in the ImageNet dataset [67]. The size of each dataset is listed in Table 3.6.

We use five kernels from the Mahout [20] machine learning framework to evaluate HCL2: KMeans, Pairwise Similarity, Fuzzy KMeans, Dirichlet Clustering, and the

Naive Bayes Trainer. All HCL2 experimental runs are done using a subset of the Wikipedia dataset [68].

All Spark applications were implemented using Spark's Scala API, and all Hadoop applications were implemented using Hadoop's Java API.

Overall Speedup and Scalability

Figure 3.18 shows the overall speedup HCL2 and SWAT achieved relative to Hadoop and Spark on 1 worker node. We note a number of trends.

For both HCL2 and SWAT, there are two clear categories of benchmarks: those which see little or no improvement from acceleration, and those with 2× or greater end-to-end speedup. For SWAT, the Genetic, KMeans, Fuzzy, and NN benchmarks all show speedups between 2× and 3×, while PageRank and Connected either show no change, or slight slowdowns. For HCL2, Dirichlet, Fuzzy, and KMeans see up to an order of magnitude performance improvement, while Bayes and Pairwise see little speedup. We explain this through the characteristics of the applications: speedups are achieved when non-trivial computation is present in the application logic being accelerated by GPUs. For applications that are I/O bound on disk or network bandwidth and have small computational kernels, HCL2 and SWAT demonstrate little or no improvement.

Figure 3.18 also suggests that HCL2 is able to achieve much greater improvements relative to its baseline than SWAT. While this is true, it is primarily a result of the simplified programming constructs that HCL2 supports relative to SWAT, simpler benchmarks, and HCL2's more intrusive programming model which enables more compile-time optimizations and less runtime work.

Figure 3.19 shows the scalability of each benchmark running on Hadoop, HCL2, Spark, and SWAT when moving from two to eight worker nodes. Linear scalability would be denoted by a 4× speedup on the y-axis. At the scale of only eight worker nodes, it is difficult to make conclusions about the scalability of any framework. In

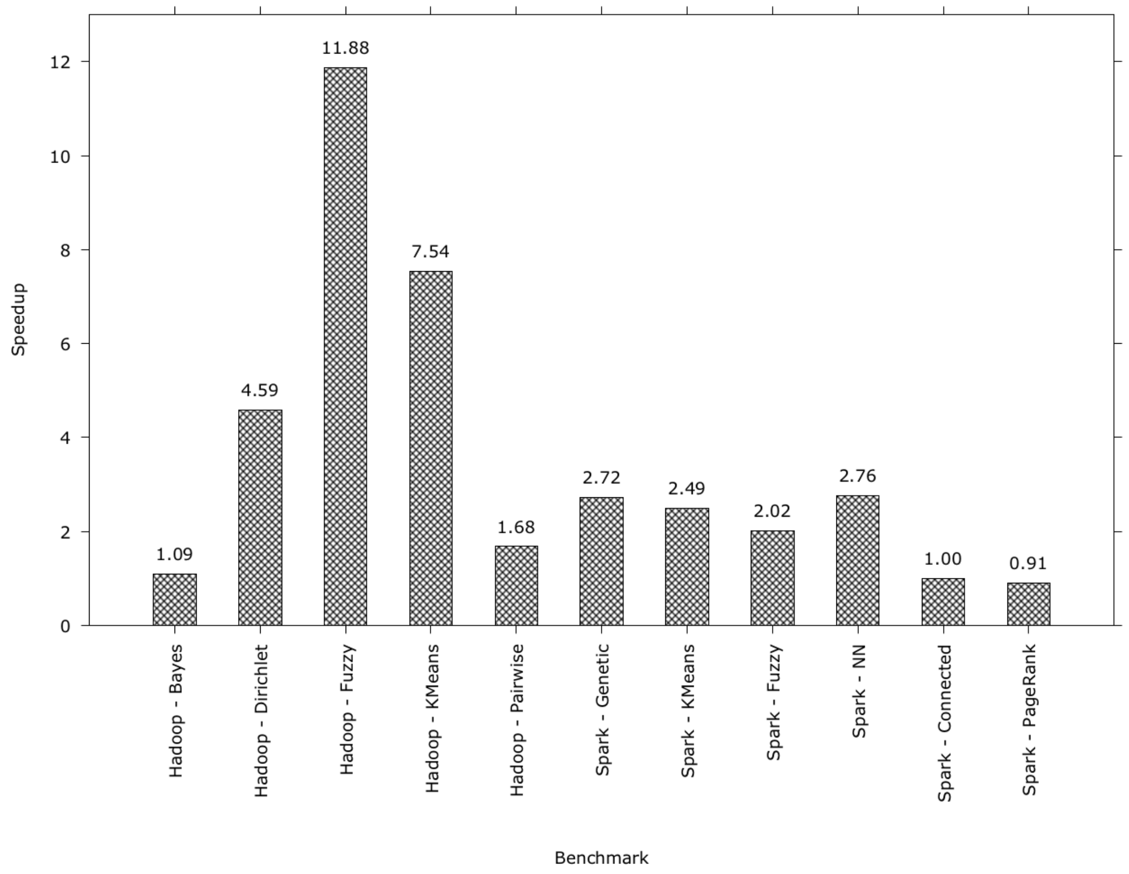

Figure 3.18 : Overall speedup of each HCL2 and SWAT benchmark using 1 master node and 1 worker node.

general, none consistently achieves linear scalability. However, these applications are not perfectly parallel and have collect or reduction stages which would make perfect scalability unlikely.

We do note that HCL2 consistently scales worse than Hadoop. HCL2 (unlike SWAT) modified core Hadoop code, including the Hadoop TaskTracker that manages each worker node. The modifications made to the TaskTracker for tracking GPU usage added synchronization, and affected the scalability of the overall HCL2 framework as a result of higher overheads in dispatching tasks.

We also note that in Figure 3.19, the two SWAT applications with the worst scalability (PageRank and Connected) also demonstrated the lowest speedups in Figure 3.18. This poor scalability is caused by the same network and I/O bottlenecks that caused poor speedups when comparing SWAT to Spark.

Execution Timelines

Focusing on performance within a single Spark or Hadoop worker node, we can use the custom HCL2 and SWAT profiling tools described in Section 3.4.7 to better understand the behavior of these frameworks and explain the performance of different applications.

In this study, we focus on the benchmarks for HCL2 and SWAT which demonstrated the least and most speedup relative to their respective baselines. For HCL2, that is Bayes and Fuzzy KMeans. for SWAT, that is PageRank and Genetic.

Parsing HCL2's profiling logs produced the statistics in Table 3.7 for Fuzzy KMeans and Bayes. Because HCL2 focuses on accelerating computation, it makes sense that the performance improvement from using HCL2 would be larger for compute-bound applications like Fuzzy KMeans than for more I/O-bound applications like Bayes.

On the other hand, in SWAT we categorize the work performed into three categories:

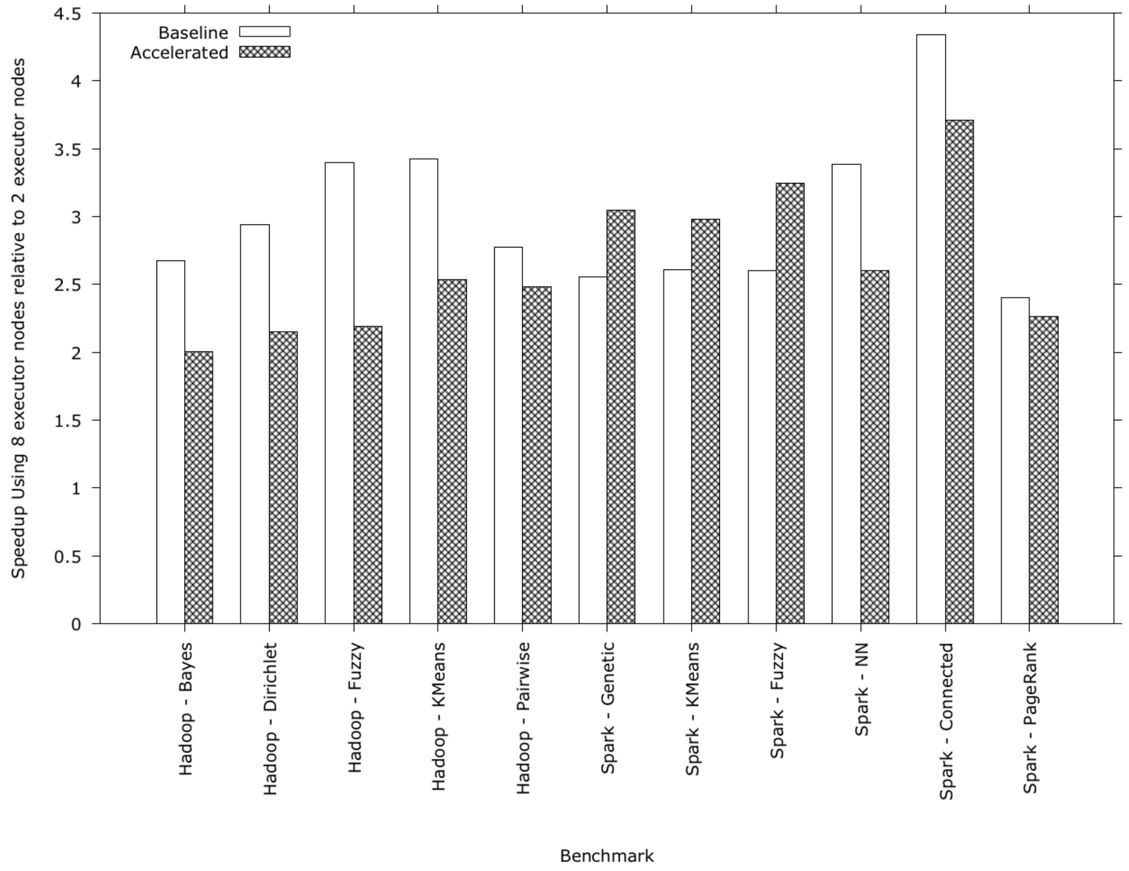

Figure 3.19 : Speedup of Hadoop, HCL2, Spark, and SWAT relative to themselves when going from 2 to 8 worker nodes.

	Map Stage			Reduce Stage		
	Read	Exec	Write	Read	Exec	Write
Fuzzy	5%	94%	1%	23%	65%	12%
Bayes	97%	2%	1%	22%	73%	5%

Table 3.7 : Percent execution time spent by HCL2 in read I/O, kernel execution, and write I/O while executing Fuzzy KMeans and Bayes.

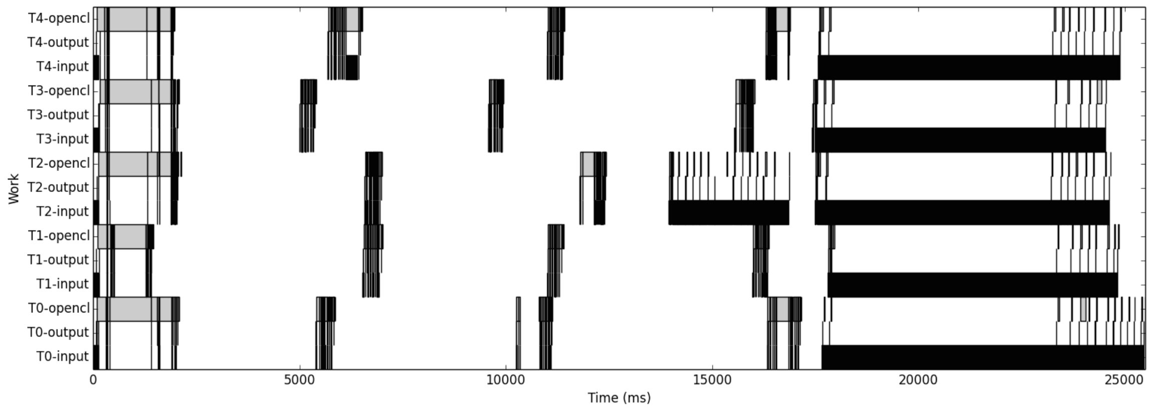

Figure 3.20 : PageRank execution timeline. Light gray indicates input I/O, dark gray indicates OpenCL operations, and black indicates output I/O. No dark gray is visible at this time scale as little computation is performed in PageRank.

1. Input I/O, which includes deserialization and disk I/O.
2. OpenCL operations, which includes both data communication with and execution on the OpenCL device.
3. Output I/O, which includes serialization and disk I/O.

Figure 3.20 shows an execution timeline for the PageRank benchmark, and Figure 3.21 shows it for the Genetic benchmark. Clearly, PageRank is dominated by input and output I/O while Genetic is dominated by computation. Combining these observations with Amdahl's Law explains the higher overall speedups achieved for the Genetic benchmark compared to the PageRank benchmark.

SWAT Hardware Utilization

In the SWAT project, we also spent time performing a deeper investigation of hardware resource utilization. In particular, we look at CPU utilization and system memory utilization.

Figure 3.22 shows the change in CPU and memory utilization for the PageRank benchmark. Figure 3.23 shows the same information for the Genetic benchmark.

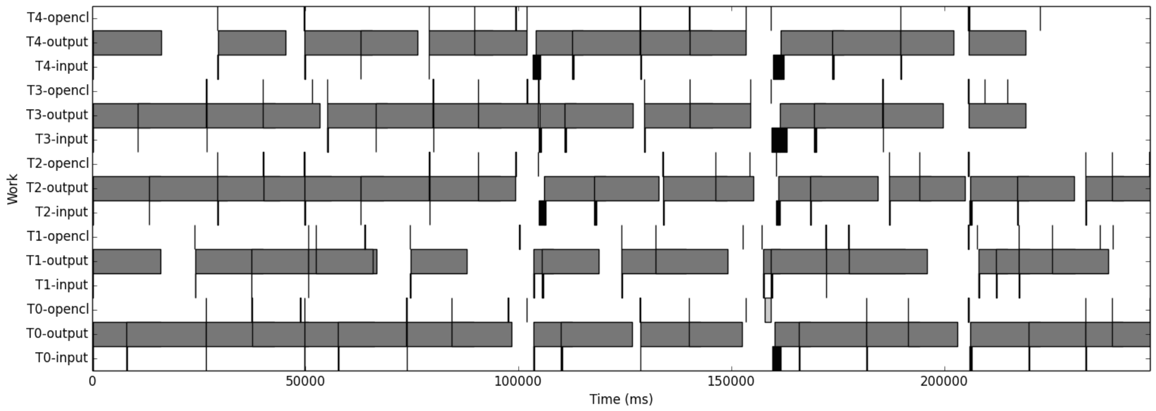

Figure 3.21 : Genetic execution timeline. Light gray indicates input I/O, dark gray indicates OpenCL operations, and black indicates output I/O. Note that this figure is dominated by dark gray, indicating a large amount of time in OpenCL operations.

Table 3.8 lists peak utilization information for all benchmarks.

We observe that the results in Figure 3.22 and Table 3.8 support the conclusion that PageRank is not a compute-bound benchmark, only achieving a peak CPU utilization of 55% when running on Spark. Similarly, Figure 3.23 and Table 3.8 show that Genetic is compute-bound, achieving a peak CPU utilization of 90% when running on Spark.

Performing a comparative study between Spark and SWAT, we note that CPU utilization drops by an average of 31% across all benchmarks when using SWAT, but system memory utilization increases by an average of 25%. Both of these results are expected. It is natural for the host utilization to drop if the main compute workload is now offloaded to an accelerator. We also expect system memory utilization to increase as SWAT allocates extra management data structures and host buffers for input and output accumulation.

Note that while there is an increase in system memory utilization with SWAT, the memory controls implemented as part of the SWAT Core are effective in keeping system memory utilization stable throughout the job: it does not oscillate or monotonically increase. In fact, it closely resembles the behavior of Spark's memory

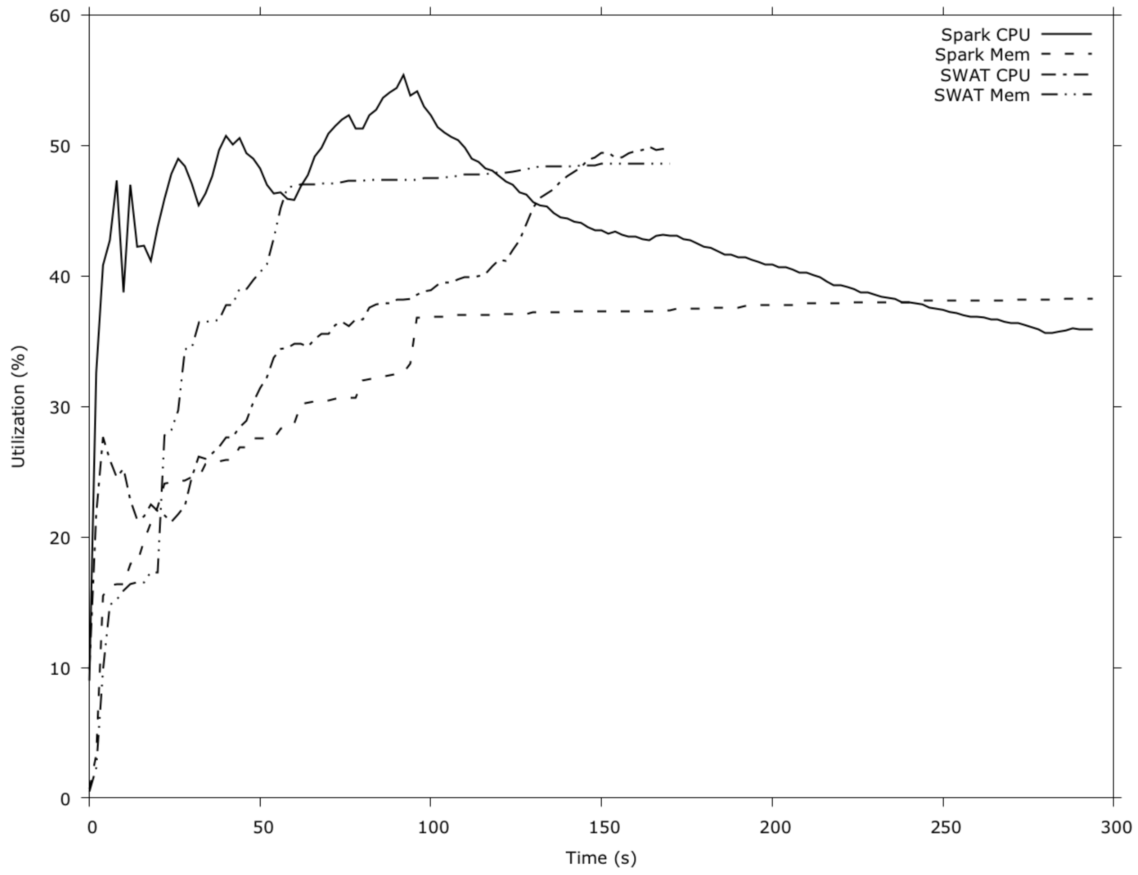

Figure 3.22 : Host processor and memory utilization of the PageRank benchmark running on Spark and SWAT.

utilization, albeit with a constant factor added on top.

HCL2 Auto-Scheduler Evaluation

To evaluate the HCL2 auto-scheduler, 20 jobs of each benchmark were run consecutively: 10 with auto-scheduling applied only to the map stage followed by 10 with auto-scheduling applied only to the reduce stage. During mapper auto-scheduling, reduce tasks were assigned to the same device as was chosen for manual scheduling. During reducer auto-scheduling, the HCL2 static scheduler was used for mapper tasks.

Figure 3.24 shows the progression of performance over all auto-scheduled runs of

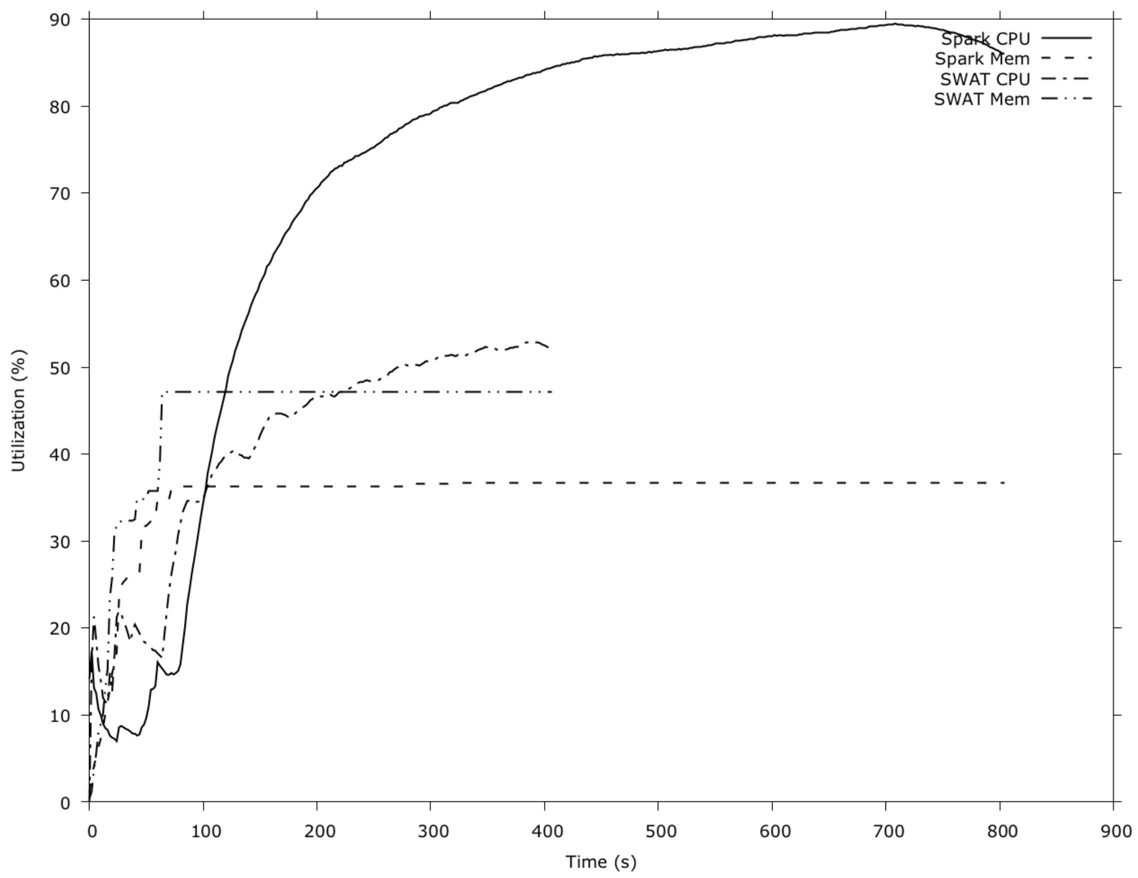

Figure 3.23 : Host processor and memory utilization of the Genetic benchmark running on Spark and SWAT.

Benchmark	**Peak CPU Utilization**			**Peak Sysmem Utilization**		
	Spark	**SWAT**	**% Change**	**Spark**	**SWAT**	**% Change**
PageRank	55%	50%	-9%	38%	49%	+27%
Connected	28%	36%	-22%	81%	91%	+12%
NN	89%	66%	-26%	77%	85%	+10%
Fuzzy	92%	52%	-43%	30%	42%	+43%
KMeans	85%	45%	-47%	37%	47%	+30%
Genetic	90%	53%	-41%	37%	47%	+29%
Average			**-31%**			**+25%**

Table 3.8 : Resource Utilization summary across all benchmarks

Fuzzy KMeans and Bayes on Platform A. These benchmarks were chosen as representations of good and poor auto-scheduling. For Fuzzy KMeans, the HCL2 auto-scheduler task placement quickly converges. There is little loss in performance relative to manually scheduled jobs for most auto-scheduled runs. Bayes requires more exploration of the task performance characteristics before the HCL2 scheduler converges. This is due to poor performance when executing the map stage of Bayes on GPUs, leading to drastic performance variation for any Bayes job which speculatively schedules map tasks on the GPU.

We observe a downwards spike in Fuzzy KMeans execution time on run eleven, caused by the switch from mapper auto-scheduling to reducer auto-scheduling. At this point, no historical performance information is available on Fuzzy KMeans reducer performance, so speculative scheduling decisions are made on suboptimal devices.

Note that these graphs show the raw execution time for individual runs as a way of illustrating the real-world performance progression one could expect from using this auto-scheduling framework. Random variations in performance can be attributed to environmental factors as other jobs in the same compute cluster use a shared resource (e.g. the network).

The only benchmark on either platform that fails to achieve parity with manual scheduling is Pairwise on Platform B, where only approximately 83% of peak manually-scheduled performance is achieved. This is a result of scheduling the mapper stage in OpenCL CPU threads instead of on the JVM. This mis-scheduling is caused by inaccuracies in the technique used to calculate task performance on the JVM, described in Section 3.4.6. This is only a factor for kernels where an OpenCL device performs similarly to the JVM. The Pairwise Mapper is an example where the resulting scheduling actually resulted in a significant performance loss.

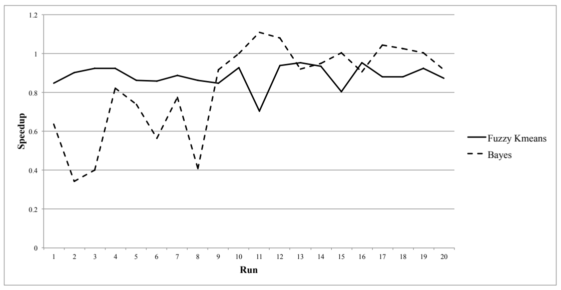

Figure 3.24 : Progression of execution time for auto-scheduled HCL2 Fuzzy KMeans and Bayes jobs relative to the mean execution time of manually scheduled HCL2 jobs on Platform A.

Table 3.9 shows how many auto-scheduled jobs completed within 10% of the execution time of the manually scheduled jobs, and how well the fastest auto-scheduled job performed relative to the fastest manually scheduled job. The main outlier (explained above) is the Pairwise benchmark on Platform B.

One item of note in Table 3.9 is the relative ability of the auto-scheduler on each platform to reach performance similar to manual scheduling. Platform B is measurably better at achieving performance parity than Platform A on four out of the five benchmarks (Pairwise's poor performance was explained earlier). The logs of the auto-scheduling system explain that this is a result of Platform A having twice as many GPUs as Platform B. Because the confidence of a task's performance prediction uses a distance measure based on system load, this increased dimensionality in the device load vector leads to an increased amount of time spent performing speculative execution on Platform A than Platform B, as a larger search space must be covered.

Benchmark	Platform A		Platform B	
	# Runs	Relative Perf.	# Runs	Relative Perf.
KMeans	17/20	0.99x	19/20	1.02x
Fuzzy KMeans	8/20	0.98x	18/20	0.97x
Dirichlet	7/10	0.95x	9/10	0.98x
Pairwise	17/20	0.96x	0/20	0.83x
Bayes	11/20	1.06x	19/20	1.04x

Table 3.9 : Relative performance of auto-scheduled and manual runs based on the number of auto-scheduled runs whose execution time was within 10% of the mean of the manually scheduled runs, and the relative speedup of the fastest auto-scheduled run relative to the mean of the manually scheduled runs.

Hence, more time and more jobs are spent with tasks being speculatively scheduled on suboptimal devices on Platform A.

It is also important to understand the overhead added when performing auto-scheduling. This overhead comes from three places: the timing statements used to measure the elapsed time of device computation, the computation necessary to revise a task's performance characterization based on new performance data, and the computation needed to make a scheduling decision based on task performance profiles. We can measure this overhead by directly recording the time spent in each job on revising performance characterizations and making scheduling decisions using millisecond-granularity timing statements. Note that the HCL2 Static Scheduler from Section 3.4.6 was added as a way of reducing auto-scheduling overhead by using stored task characterizations but not updating them.

Table 3.10 shows the average percentage of total execution time spent revising performance characterizations, making performant scheduling decisions, and making speculative scheduling decisions for all auto-scheduled jobs. Note that performant scheduling consumes an order of magnitude more execution time than speculative scheduling. This is a result of much more performant scheduling being done than speculative scheduling, as full task performance characterization is often achieved after at most 5 jobs.

Benchmark	Speculative	Performant	Revising
KMeans	0.004 %	0.07 %	0.02 %
Fuzzy KMeans	0.0026 %	0.07 %	0.04 %
Dirichlet	0.0021 %	0.07 %	0.02 %
Pairwise	0.0017 %	0.03 %	0.04 %
Bayes	0.0027 %	0.07 %	0.5 %

Table 3.10 : Overhead added by the auto-scheduler. These percentages are means across all auto-scheduled runs on all platforms.

Benchmark	Platform A		Platform B	
	Map	Reduce	Map	Reduce
KMeans	100 GPU	2 CPU	100 GPU	2 CPU
Fuzzy KMeans	50 GPU	2 CPU	20 GPU	2 CPU
Dirichlet	100 GPU	N/A	20 GPU	N/A
Pairwise	12 CPU	2 CPU	12 CPU	2 CPU
Bayes	100 JVM	2 GPU	50 JVM	2 JVM

Table 3.11 : Number of tasks placed on each type of device by the HCL2 auto-scheduler in Platforms A and B.

Table 3.11 shows how many tasks the auto-scheduler placed on each device type for the fastest run of each benchmark on Platforms A and B. In most cases, the scheduling decisions made by the auto-scheduler are identical to those made manually by expert tuning. The most common difference was execution by the auto-scheduler of tasks on the OpenCL CPU device instead of the JVM. This is likely a result of the measurement error discussed previously. For most tasks, this has little impact on performance as they both execute on the same physical architecture. The auto-scheduler used the GPU for the Bayes reduce stage on Platform A, and was able to achieve a 6% performance improvement relative to manual scheduling on the CPU as a result of lowered contention for CPU cycles.

3.5 Related Work

3.5.1 Past Work in Shared-Memory Programming Frameworks

While little practical related work exists in this area, several past research projects have explored accelerated JVM systems.

The Rootbeer [69] and APARAPI [52] projects explored offloading shared-memory parallel Java programs to GPUs using APIs similar to Java's `Runnable` interface. Rootbeer generally supported more advanced JVM features inside of offloaded kernels, such as exceptions and object references, while APARAPI constrained the supported subset of the JVM bytecode specification. Both offer fully automatic memory management, using Java Reflection to inspect data structures referenced from offloaded kernels and transfer them to the accelerator. Both Rootbeer and APARAPI also use code generation from JVM bytecode to create accelerator-compatible kernels. They differ in that Rootbeer generates these kernels statically at compile-time, while APARAPI does so lazily at runtime immediately prior to kernel launch.

Past work in [70] and [71] built more advanced offload capabilities by using the APARAPI framework to offload parallel loops in the Habanero-Java parallel programming language (HJlang). These works extended APARAPI to add support for multi-dimensional array references, global barriers, and exceptions within an offloaded kernel.

Building on lessons learned from [70] and [71], the same authors are currently also exploring offloading parallel Java streams from within the IBM JVM itself [72][73]. This work differentiates itself by performing runtime code generation from within the JVM's Just-In-Time compiler, generating PTX kernels targeting NVIDIA GPUs. Sitting within the JVM, this work is also able to use JVM internal information to optimize redundant data transfers and data serialization. In addition, this work is one of the first examples of an accelerated JVM programming system which performs automatic device selection. In [73], the authors explore how Support Vector Machines trained on kernel feature vectors can accurately select the GPU or JVM for a given kernel.

Similar to [72] and [73], the work in the JaBEE project [74] also modified JVM internals to enable JVM accelerator offload, but using the J3 JVM. JaBEE arguably supports the most general object model out of all related work, supporting nearly arbitrary object references within offloaded kernels and dynamic memory allocation of new objects. However, the performance evaluation of JaBEE shows that this support for complex JVM operations causes significant overheads, to the point where the performance benefit of GPU execution is lost. This underscores the importance of selectively offloading JVM kernels based on the JVM features they use.

3.5.2 MapReduce-based Frameworks

To a lesser extent, past research efforts have also explored accelerating user-written MapReduce kernels from within or on top of the Hadoop MapReduce framework.

The work in [75] and [76] take similar approaches to accelerating MapReduce applications. Both ignore the problem of kernel compatibility by having the programmer manually write accelerator-compatible kernels, with [75] calling those kernels through JNI and [76] using Hadoop Pipes to pipe input data to external GPU processes. Both works use automatic load-balancing runtimes in each worker node of a Hadoop installation to distribute work among both GPU accelerators and CPU cores, deciding the work distribution using speculative execution of a small number of tasks on both architectures to determine the relative processing rate of each for a given Hadoop job.

HeteroDoop [77], on the other hand, generates accelerated Hadoop MapReduce programs from annotated, high-level, sequential applications, including native bridge code and accelerator kernels for supporting GPU offload.

3.5.3 Functional, Spark-Based Frameworks

The most recent explorations into managed runtime acceleration has focused on the relatively new Apache Spark framework. Apache Spark is a functional, distributed, and multi-threaded programming system that uses distributed vectors, in-memory caching of sub-vectors, and one-way transformations applied to those vectors to support programmable distributed execution.

SparkCL [78] began the exploration of accelerating Spark applications by using a lightly-modified version of APARAPI to automatically generate accelerator-compatible versions of user-written Spark transformations and launch them on any detected OpenCL device. While this straightforward integration was a useful proof-of-concept, in practice it would suffer from limitations in the APARAPI execution model, including blocking communication, redundant transfers, and limited support for complex data types. Additionally, the evaluation of SparkCL presented in the paper is limited.

HeteroSpark [79], on the other hand, closely resembles the design of HeteroDoop [76]. It pipes input data to external processes running compiled GPU applications through Java RMI, which then process the received data on an accelerator using the specified kernel and return the results. This model allows GPU programmers to implement and optimize their own GPU kernels, but as a result requires GPU expertise from the user.

3.6 Discussion of Selectively Supporting JVM Features

While the works presented in this chapter support some high-level JVM features (e.g. object references) inside of accelerator kernels, limitations remain on the type of JVM bytecode and JVM objects that can be supported on accelerators. For example, in HJ-OpenCL and SWAT only objects which had exclusively primitive fields accessed inside accelerator kernels were supported. This is not an absolute limitation: JaBEE[74] is an example of existing work that supports nested object references. However, JaBEE's results demonstrate that adding this support cancelled out the performance advantage of accelerator execution. Not only does indirection complicate the serialization logic, but multiple layers of indirection is not generally a pattern that fits well with simpler accelerator architectures. The works presented in this chapter chose to focus on kernels that were most likely to be amenable to acceleration, kernels that operate on primitive or simple object types.

This brings up a tension between how much of the JVM bytecode specification can be supported on accelerators, and how much should be supported. The choice to move to accelerators is usually made primarily for performance, and secondly for energy efficiency. We believe the goal of research like ours should be to investigate how certain portions of the JVM specification are supportable on the accelerator, what compromises must be made on performance and/or energy efficiency, and how those compromises change with application or dataset characteristics. JaBEE [74] did this for indirect object references. Previous work in HJ-OpenCL did this for exceptions [71] and global synchronization [70]. Works presented in this thesis did the same for dynamic memory allocation, framework-specific data types, and simple composite objects. Through experimentation the community can converge on a reasonable subset of the JVM bytecode specification that can be supported on accelerators without sacrificing performance/energy efficiency.

3.7 Summary

At the start of this chapter, we highlighted five fundamental challenges to accelerating managed data analytics frameworks. These works address these challenges in the following ways:

1. **Incompatibilities in instruction set between managed and native systems**. HJ-OpenCL, HCL2, and SWAT all extend on APARAPI's code generation module to support dynamic generation of OpenCL kernels from JVM bytecode. While HJ-OpenCL provides foundational extensions (e.g. object

support), HCL2 and SWAT both provide framework-specific code generation to facilitate common usage patterns. HJ-OpenCL extended APARAPI to support the `NEW` bytecode and object references inside of accelerated kernels. HCL2 and SWAT add framework-specific knowledge to APARAPI to enable code generation for Hadoop- and Spark-specific data types.

2. **Incompatibilities in data format between managed and native systems**. HJ-OpenCL extended on the state-of-the-art in JVM-to-accelerator data serialization by supporting runtime serialization of user-defined, composite objects to a format an OpenCL device can manipulate. SWAT extended on that work to also support useful Spark-specific data structures, such as `scala.Tuple2`, `DenseVector`, and `SparseVector`.

3. **Resource management in a multi-tenant system**. HCL2 and SWAT take different approaches and make novel contributions in managing accelerator memory, JVM memory, and native memory together in multi-tenant applications.

4. **Scheduling in a multi-tenant system**. HCL2 uses a historical performance prediction framework to automatically schedule user-written kernels on to well-performing platforms based on device load. Both HCL2 and SWAT use asynchronous runtimes to schedule management operations (e.g. serialization) and application operations (e.g. accelerator kernels) on a shared system.

5. **Inspectability**: The HCL2 and SWAT provide intrinsic support for debugging and optimizing both the runtime itself and application kernels being scheduled by the runtime.

This chapter presented novel work offloading computational kernels from managed runtimes and data analytics frameworks to accelerators. We address challenges in code generation, data serialization, resource management, and automatic scheduling. These techniques produce significant speedups relative to JVM baselines while retaining the programmability benefits of the offloaded frameworks for domain experts. These techniques prepare these frameworks to support future heterogeneous HPC platforms.

Chapter 4

Improving the Scalability, Programmability, and Composability of HPC Libraries on Heterogeneous Systems

4.1 Motivation

While Chapter 3 focused on using high-level, managed data analytics programming models to program HPC systems, HPC systems have classically been programmed using compiled, native libraries and languages. This approach generally improves performance and transparency, enabling HPC Gurus to optimize their code however they see fit. This section will discuss contributions of this thesis in making native HPC systems more extensible, flexible, and tunable for future platforms.

These works are motivated by the changes we see in HPC today. The architectural evolution of the field over the last decade eclipses the decades leading up to it. With NVRAM, high-bandwidth memory, more revolutionary architectures, the slowing of Dennard scaling, and a number of other changes affecting HPC, we can expect the next decade to bring even more heterogeneity and change. With these changes in HPC hardware, an increased heterogeneity can also be expected at the software layer.

The composability of heterogeneous software components with each other is an all-to-all relation. That is, we do not just need GPU-Aware MPI. We also need GPU-Aware OpenSHMEM, and GPU-Aware UPC++, and FPGA-Aware MPI, and more. However, forcing every software component to be aware of all others is not scalable from a development perspective. Instead, in this work we focus on globally coordinating work across the system from a single runtime system and through a single, consistent API. Thus, only one programming and runtime system needs be aware of the different hardware and software components available.

This full-system awareness comes with three primary benefits in three different research areas:

1. **Programming Model**: By including multiple types of software components in a single programming system we have the opportunity to develop novel API extensions at the interfaces between software components. These new APIs can improve programmability by allowing the programmer to express dependencies, computation, or other logic that spans multiple components. For example, it becomes possible to make the execution of a GPU kernel predicated on an incoming MPI message.

2. **Runtime Scheduler**: With more information exposed to a single scheduler, we can use that information to make better scheduling decisions in order to improve the scalability of the overall system.

3. **Tool Enablement**: With more of the work in a system scheduled on a single runtime, that runtime can offer more comprehensive and informational hooks to tools, much like OMPT [42] does for OpenMP runtimes.

In this thesis, we introduce HiPER (a Highly Pluggable, Extensible, and Reconfigurable Framework for HPC) [29]: a programming system for HPC workloads that focuses on extensibility, composability, and by extension programmability. It is designed with heterogeneous HPC systems in mind. While HiPER is built on a lightweight, efficient, work-stealing runtime, its main contributions are in its scheduling and API flexibility.

4.2 HiPER Design and Implementation

At a high level, the HiPER system consists of three components: 1) a platform model, 2) a generalized work-stealing, multi-threaded runtime, and 3) pluggable, third-party software modules.

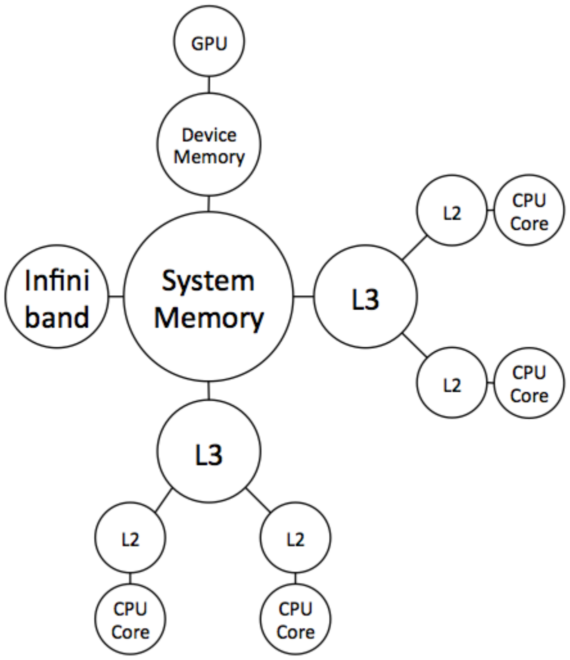

Figure 4.1 : An example of the HiPER Platform Model.

The HiPER Platform Model offers an abstraction of the heterogeneous hardware resources across which the workload of an application will be distributed. The Generalized Work-Stealing Runtime manages load-balancing and execution of user-created tasks placed at different locations in the Platform Model. The Pluggable Software Modules sit on top of the runtime and expose familiar APIs to the user (e.g. MPI, OpenSHMEM) while placing tasks on the HiPER Platform to be executed by the work-stealing runtime.

4.2.1 HiPER Platform Model

The HiPER Platform Model consists of an undirected, unweighted graph. Nodes within the graph logically represent hardware components that software libraries may utilize, and are referred to as "places" [80]. Figure 4.1 depicts an example HiPER Platform Model.

Edges between places in the platform graph logically represent direct accessibility between hardware components. For example, a direct edge between system memory and GPU device memory indicates that data in system memory is directly transferrable to that GPU's device memory. There is no strict requirement that there be a one-to-one mapping of places or edges in the platform model to physical hardware or connections. However, some similarities are likely desirable for improved performance fidelity.

The HiPER Platform Model is implemented as an in-memory graph structure. It is loaded from a JSON-formatted file at HiPER runtime initialization. HiPER comes with utilities for automatically generating JSON platform configuration files using the HWloc library [81], but users are also free to edit these configurations.

4.2.2 Generalized Work-Stealing Runtime

The HiPER "Generalized Work-Stealing" runtime depends upon the Platform Model and consists of four components: a set of persistent worker threads, task deques of eligible tasks at each place in the platform model graph, a pop and steal path for each thread which traverses some subset of the places in the platform model, and an API for enqueueing tasks to these deques in the platform model.

Persistent Thread Pool

Like most work-stealing runtimes, a generalized work-stealing runtime contains a persistent set of worker threads on which all tasks are executed. The number of worker threads to create is defined in the JSON file used to initialize the platform model.

Tasks are defined as suspendable single-threaded streams of execution, and may synchronize on other tasks or create new tasks.

As in Realm, HPX, and QThreads [82][83][84], the HiPER runtime threads use runtime-managed stacks to enable task suspension. When a HiPER task blocks on a synchronization operation, HiPER will suspend that task without blocking a CPU core on it by swapping its call stack off of the current thread, wrapping its continuation in a task, and making the execution of that task predicated on the satisfaction of the appropriate synchronization event.

Per-Place Task Deques

Each place in the platform model includes `N` task deques, where `N` is the number of runtime threads configured in the platform configuration file. The ith deque in a place contains only eligible tasks that are ready to begin executing and which were spawned by the ith worker thread. Hence, given a place and a thread looking for work to do it is a straightforward and efficient process to differentiate between tasks created by that same thread and tasks created by other threads. Executing a task created by the same thread likely encourages locality, while executing a task created by other threads encourages load balance.

Per-Thread Pop and Steal Paths

Each worker thread has one "pop path" and one "steal path". Each of these paths is an ordered list of places in the platform model. A path defines the sequence of places a runtime thread will traverse when searching for a task to execute. When traversing a pop path, a runtime thread will only check for work that it created. A steal path is similar, but runtime threads traversing a steal path will only look for work created by other runtime threads. Figure 4.2 depicts an example path through the platform model from Figure 4.1. Pop and steal paths are also loaded from the platform configuration JSON file.

Hence, each runtime thread's logic simply consists of:

1. Search along its pop path for any work created by the same thread at any place, where work created by thread `T` is always placed in the `T`th task deque in a place.

2. If no work has been found yet, search along its steal path, only looking for work

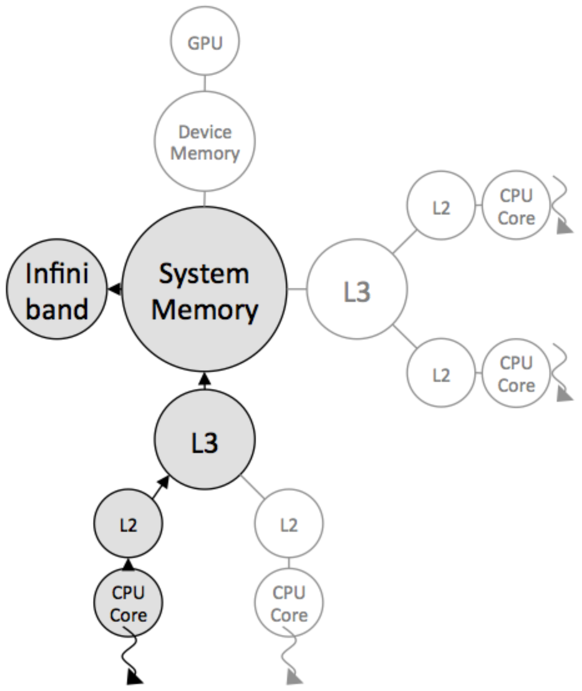

Figure 4.2 : An example of a pop or steal path through the HiPER Platform Model.

created by other threads.

3. Repeat #2 until either work is found or a runtime shutdown signal is received by this thread.

When a runtime thread discovers a task along either its pop or steal path, that task is immediately executed.

Task Creation APIs

The generalized work-stealing runtime must also expose APIs for placing and removing tasks at deques in the platform model. These APIs may be used by a programmer, but are also key to implementing the pluggable HiPER modules described in Section 4.2.3. HiPER currently only supports C++ APIs.

The `async` API creates a task executing `body` at the place closest to the current runtime thread:

```
async([] { body; });
```

The `async_at` API creates a task executing `body` at a specific place:

```
async_at([] { body; }, place);
```

HiPER's API and runtime also support the use of promises and futures for inter-task synchronization. A promise in HiPER is a single-assignment, thread-safe container for some value. A future is a read-only handle on that value. Promises and futures can serve as a flexible synchronization channel from one source task to many sink tasks. Sink tasks may block on the future, only being released when another task performs a put on the promise. In HiPER, a programmer can manually satisfy a promise or block on a future.

Promise and future objects can be created in HiPER using standard C++ constructors and getters:

```
promise_t *p = new promise_t();
future_t f = p->get_future();
```

Satisfying a promise, blocking on a future, and fetching the put value from a future are simple member function calls:

```
p->put(NULL);
f->wait();
val = f->get();
```

Additionally, the `async_future` API creates a task and returns a future which will be automatically satisfied when that task completes, while the `async_await` API creates a task whose execution is predicated on the satisfaction of a future object:

```
future = async_future([] { body; });
async_await([] { body; }, future);
```

Bulk task synchronization is possible using the `finish` API. `finish` waits for all tasks created in `body` before returning, including transitively spawned tasks.

```
finish([] { body; });
```

Many combined variants of the task creation APIs exist as well. For example, `async_future_await` creates a task whose execution is predicated on the satisfaction of a future, and returns a future that is satisfied when that task completes.

HiPER also comes with an `async_copy` API which asynchronously transfers data from a location in one place to a location in another place:

```
async_copy(dst_loc, dst_place, src_loc,
    src_place, nbytes);
```

4.2.3 Pluggable Software Modules

The final component of the HiPER system is its pluggable modules. A single pluggable module adds user-visible APIs that can be called to schedule tasks doing module-specific work on the HiPER work-stealing runtime. These tasks may perform arbitrary logic. For example, an MPI module would extend the HiPER user-visible APIs with functions from the MPI standard and would schedule inter-rank communication on the HiPER runtime. A complete HiPER module includes:

1. An initialization function registered with the HiPER runtime which is called once during the life of a process.

2. A finalization function registered with the HiPER runtime which is called once during the life of a process.

3. A set of optional, special-purpose functions registered with the HiPER runtime. For example, a module may register itself as responsible for handling data transfers between places of certain types in the platform model.

4. A set of functions added to the global HiPER namespace and accessible to programmers. These functions extend the capabilities of HiPER to make use of a new hardware or software component (e.g. GPUs, MPI, hard disks). These user-facing functions are commonly implemented as the placement of tasks at special-purpose nodes in the platform model. As a result, all work created by HiPER modules is scheduled together on a single unified runtime. Examples of

modules supported today include modules for CUDA, MPI, OpenSHMEM, and UPC++.

One of the key characteristics of HiPER modules is that they do not require that the software or hardware component they support be aware of HiPER or of the other HiPER modules.

To illustrate these points, we will perform several case studies on existing HiPER modules below. Note that in general, these HiPER modules may only implement a useful subset of the APIs they are implementing (e.g. the MPI module implements a subset of the MPI standard) and are not necessarily full, specifications-compliant implementations of the corresponding HPC libraries.

MPI Module

The MPI module implements a subset of the APIs in the MPI standard, relying on a full MPI library to handle the actual messaging.

Regarding the platform model and thread configuration, the MPI module relies on a single "Interconnect" place existing in the platform model and that place being on a single thread's pop and steal paths. This allows the MPI module to configure the underlying MPI implementation in `MPI_THREAD_FUNNELED` mode, keeping MPI runtime overheads low. It is up to individual modules to make these assertions about the current platform model during module initialization.

Many MPI APIs are implemented using the following flow:

1. A C++ lambda is created which captures the inputs to the MPI API being implemented, and which calls the underlying MPI library's implementation of that API.

2. This lambda is passed to the `async_at` API in Section 4.2.2, targeting the Interconnect place in the platform place graph.

3. A `finish` scope is used to block the calling task on the completion of the spawned task. Under the covers, this deschedules the calling task until the spawned MPI task completes.

For example, the HiPER Module implementation of `MPI_Send` is shown below:

```
finish([&] {
    async_at([&] {
        ::MPI_Send(buf, count, datatype,
            dest_rank, tag, comm);
    }, interconnect);
});
```

On the other hand, asynchronous MPI APIs (e.g. `MPI_Isend`) are implemented by issuing the non-blocking calls from a wrapping task and periodically polling on the result. When the communication has completed, a promise is satisfied by the HiPER runtime. The future associated with that promise is returned to the user. It can be used to register other HiPER work on the completion of the asynchronous MPI communication.

Polling for the completion of an asynchronous MPI operation is implemented as follows. A look-up table is kept in the MPI module which stores the MPI result object associated with an asynchronous operation and the promise to satisfy when that operation completes. When a new asynchronous MPI operation is launched, it adds itself to this table. If it is the first entry in the table, it also spawns a task at the Interconnect place which scans the lookup table for any completed operations. If that task finds completed operations, it removes them from the look-up table and satisfies the appropriate promise. If the look-up table is non-empty after the scan finishes, it will then recursively spawn another instance of itself to again check the table. This way, the worker thread visiting the Interconnect place does other useful work while only periodically checking for completed asynchronous operations.

Using future-producing APIs like these enables programmers to compose MPI messages with other work in the system, such as task parallel computation. For example, the code snippet below would trigger a task on the receipt of an asynchronous MPI message.

```
fut = MPI_Irecv(...);
async_await([=] { body; }, fut);
```

UPC++ Module

UPC++ differs from MPI in that it was designed as a C++, object-oriented communication library. As such, intercepting function calls is not sufficient for the implementation of the HiPER UPC++ module. Whole classes must be wrapped. Like MPI, UPC++ can be configured to be thread-safe but has fewer internal performance bottlenecks when configured for single-threaded access. We therefore use the same platform constraints for UPC++ as were used for MPI.

The UPC++ module primarily consists of a set of C++ classes extending UPC++ classes, overriding super class functions when necessary to offload the work of the parent function to the Interconnect place. In addition, UPC++ also has several top-level APIs which are wrapped in a similar manner to the MPI module.

The following code snippet is an illustration of UPC++'s asynchronous copy and tasking APIs, taken from the UPC++ implementation of HPGMG-FV [85]:

```
upcxx_finish {
  for (...) {
    upcxx::async_copy(..., copy_e);
    upcxx::async_after(..., copy_e, data_e)(
        [=] { body; });
  }
}
```

In this snippet, several copies are created, each with a remote asynchronous task triggered once the copy completes. After all tasks and copies have been spawned, the `upcxx_finish` scope blocks the current thread on completion of all outstanding copies and tasks. We argue there are a number of programmability challenges with these APIs, illustrated by this example:

1. Because events are passed as arguments, the source-sink relationship between `async_copy` and `async_after` in the above snippet is unclear.

2. The purpose of the `data_e` event as an output event of `async_after` is similarly unclear.

3. The use of thread-blocking APIs like `upcxx_finish` wastes CPU resources and limits scalability.

To address these programmability concerns and improve the composability of UPC++'s APIs, we made the following extensions in the implementation of the UPC++ module:

1. Like the asynchronous MPI APIs, `async_copy` was refactored to return a future object. This clarifies the input and output relations of its dependencies.

2. A new function, `remote_finish`, was added to the UPC++ APIs which combines HiPER's `finish` API and UPC++'s `upcxx_finish` API to wait on remote work without blocking the current thread.

With these extensions, the previous HPGMG-FV code snippet becomes:

```
upcxx::remote_finish([&] {
  for (...) {
    copy_e = upcxx::async_copy(...);
    upcxx::async_after(..., copy_e, [=] { body; });
  }
});
```

OpenSHMEM Module

The OpenSHMEM specification currently does not make any guarantees about thread safety; therefore, the OpenSHMEM Module has the same platform constraints as the MPI and UPC++ libraries: a single Interconnect place must exist which is only visited by the master thread. Like MPI, the OpenSHMEM specification consists entirely of functions and is not object-oriented like UPC++. As a result, many of the supported OpenSHMEM APIs are implemented using the `async_at-finish` pattern described along with the MPI module in Section 4.2.3.

The OpenSHMEM specification includes distributed locking routines. `shmem_set_lock` and `shmem_clear_lock` enable critical sections across OpenSHMEM PEs. A naive implementation of these APIs using the `async_at-finish` pattern would lead to a deadlock in the following situation:

1. A HiPER runtime thread calls `shmem_set_lock`, suspending the calling task and shipping the `shmem_set_lock` to the Interconnect place.
2. A second HiPER runtime thread currently executing a task inside of an OpenSHMEM critical section calls `shmem_clear_lock`, also shipping that task to the Interconnect but behind the previous `shmem_set_lock` call in the task queue.
3. The master thread picks up the first task and calls `shmem_set_lock`, deadlocking because it will never reach the `shmem_clear_lock` task.

Instead, the HiPER implementations of `shmem_set_lock` and `shmem_clear_lock` use promises and futures to chain locking calls together within a node. The task for the next locking call in the chain is not made eligible for execution until the preceding call to unlock is complete.

The integration of OpenSHMEM into HiPER also enabled the development of novel APIs. For example, the OpenSHMEM specification includes `wait` APIs which allow an OpenSHMEM process to block on a remote put into its address space. While these are useful APIs for point-to-point synchronization, their blocking nature may waste CPU cycles and reduce application scalability. One extension to the OpenSHMEM APIs enabled as part of the HiPER module implementation was an asynchronous variant which makes a task's execution predicated on a put by a remote process, called `shmem_async_when`:

```
shmem_async_when(mem_addr, wait_for_val, [=] {
    body;
  });
```

CUDA Module

The CUDA Module supports basic CUDA operations, such as blocking data transfers, asynchronous data transfers, and asynchronous CUDA kernels.

The CUDA Module is the only module discussed here which registers special-purpose functions with the HiPER runtime. In particular, it registers itself as handling any copies to or from GPU places. Anytime a call to HiPER's `async_copy` API reads or writes a GPU place, it is automatically handed off to the CUDA Module.

The CUDA Module uses the same polling technique as the MPI Module (described in Section 4.2.3) to support asynchronous CUDA operations satisfying HiPER promises.

4.2.4 Example HiPER Usage

Consider a three-dimensional stencil application, in which the cells of a three-dimensional, regular grid are distributed in only the z-direction among MPI ranks. Let us assume that in this simplified application, a single data-parallel kernel is run across the z values a given rank is responsible for before a halo exchange occurs with neighboring ranks. This process repeats on each of several time iterations.

In an MPI+OpenMP implementation, this application could be implemented as something like the following:

```
for (t = 0; t < nt; t++) {
  // Process ghost regions on this rank in parallel
#pragma omp parallel for
  for (...) { }

  // Transmit ghost regions to neighbors, and post receives
  MPI_Isend(..., &reqs[0]);
  MPI_Isend(..., &reqs[1]);
  MPI_Irecv(..., &reqs[2]);
  MPI_Irecv(..., &reqs[3]);

  // Process remainder of z values on this rank
#pragma omp parallel for
  for (...) { }

  // Wait for all sends/recvs to complete
  MPI_Waitall(4, reqs);
}
```

Adding MPI+CUDA instead produces a slightly longer code snippet. More importantly, doing so introduces more blocking operations which may waste host CPU cycles. Additionally, the inter-statement dependencies in the straight-line sequence of API calls is unclear as a result of a lack of composability between the CUDA and MPI APIs:

```
for (t = 0; t < nt; t++) {
  // Process ghost regions on this rank in CUDA
  stencil<<<...>>>(...);

  // Copy ghost region from CUDA device
  cudaMemcpy(..., cudaMemcpyDeviceToHost);

  // Transmit ghost regions to neighbors, and post receives
  MPI_Isend(..., &reqs[0]);
  MPI_Isend(..., &reqs[1]);
  MPI_Irecv(..., &reqs[2]);
  MPI_Irecv(..., &reqs[3]);

  // Process remainder of z values on this rank
  stencil<<<...>>>(...);

  // Wait for all transmissions to complete
  MPI_Waitall(4, reqs);

  // Copy received ghost region to CUDA device
  cudaMemcpy(..., cudaMemcpyHostToDevice);
}
```

However, it may also be possible to improve performance by combining MPI, OpenMP, and CUDA by processing the smaller ghost region with OpenMP to avoid a `cudaMemcpy` while still offloading the main computational region to CUDA. The code snippet below not only requires that the programmer have expertise in OpenMP, CUDA, and MPI, but also understand how to manage their interaction safely.

```
for (t = 0; t < nt; t++) {
  // Process ghost regions on this rank in parallel
#pragma omp parallel for
  for (...) { }

  // Transmit ghost regions to neighbors, and post receives
  MPI_Isend(..., &reqs[0]);
  MPI_Isend(..., &reqs[1]);
  MPI_Irecv(..., &reqs[2]);
  MPI_Irecv(..., &reqs[3]);

  // Process remainder of z values on this rank
  stencil<<<...>>>(...);

  // Wait for all transmissions to complete
  MPI_Waitall(4, reqs);

  // Copy received ghost region to CUDA device
  cudaMemcpy(..., cudaMemcpyHostToDevice);
}
```

In contrast, expressing the same computational pattern in HiPER's future-based, composable programming model would look like the following (assuming the user already has the CUDA and MPI modules installed):

```
for (t = 0; t < nt; t++) {
  // Place an outer finish scope to ensure all work completes before
  // continuing to the next time step
  finish([&] {

    // Asynchronously process ghost regions on this rank in parallel
    ghost_fut = forasync_future([] (z) { ... });

    // Asynchronously exchange ghost regions with neighbors
    reqs[0] = MPI_Isend_await(..., ghost_fut);
    reqs[1] = MPI_Isend_await(..., ghost_fut);
    reqs[2] = MPI_Irecv(...);
    reqs[3] = MPI_Irecv(...);

    // Asynchronously process remainder of z values on this rank
    forasync_cuda(..., [] (z) { ... });

    // Copy received ghost region to CUDA device
    async_copy_await(..., reqs[2], reqs[3]);
  });
}
```

Note that in the code listing above, dependencies are expressed more naturally and between different software components. Each asynchronous operation waits on precisely the futures it needs to in order to ensure its dependencies are maintained, and input/output relations are visible as return values and API parameters. At the same time, the future-based APIs used to express CUDA parallelism and MPI communication remain syntactically similar to their standard variants in order to take advantage of existing expertise.

4.3 HiPER Evaluation

4.3.1 Experimental Setup

The experiments in this section were run on one of two platforms: the Edison supercomputer at NERSC [86] or the Titan supercomputer at ORNL [87]. Edison is a Cray XC30 with 2×12-core Intel Ivy Bridge CPUs and 64 GB DDR3 in each node. Titan is a Cray XK7 with a 16-core AMD CPU, an NVIDIA K20X, and 32GB of DRAM in each node. For the experiments listed below, Cray SHMEM v7.4.0 and GCC v4.9.3 were used on Titan. GCC 5.2.0 was used on Edison. All flat UPC++, MPI, or OpenSHMEM experiments are run with 1 process pinned to each core. All hybrid experiments on Edison are run with 2 processes and 12 threads per process, and on Titan are run with 1 process and 16 threads per process.

Our benchmark suite consists of:

1. HPGMG-FV [85]: "Implements full multigrid algorithms using finite-volume...

methods". Uses the UPC++ and MPI modules. This is a weak scaling benchmark, and was run with `log2_box_dim`=7 and `target_boxes_per_rank`=8 based on the advice of the HPGMG-FV developers.

2. ISx [88]: Integer sort benchmark. Uses the OpenSHMEM module. This is a weak scaling benchmark, and was run with 2^{29} keys to sort per process.

3. UTS [89]: Unbalanced tree search. Uses the OpenSHMEM module. This is a strong scaling benchmark, and it was run with the T1XXL dataset.

4. Graph500 [90]: Parallel, distributed breadth first search of a graph. Uses the MPI module. This is a strong scaling benchmark, and was run using 2^{31} vertices with edge factor set to 16.

4.3.2 Regular Workloads

We start by running performance experiments focused on regular applications (HPGMG-FV and ISx) to demonstrate that the HiPER framework is low overhead, and that improvements to composability and programmability do not come at the cost of performance. For regular applications, we do not expect HiPER to improve overall performance.

Figure 4.3 depicts the total execution time spent in the solving stages of HPGMG-FV. Figure 4.4 shows the weak scaling of ISx up to 1024 nodes on Titan. We note that for each benchmark the HiPER and reference hybrid implementations are comparable in performance. While the Flat OpenSHMEM implementation of ISx outperforms the two hybrid versions at smaller node counts, it scales poorly to 512 and 1024 nodes (i.e. 8,192 PEs and 16,384 PEs) due to a global all-to-all.

4.3.3 Irregular Workloads

We now turn to more irregular benchmarks, UTS and Graph500. Due to the asynchrony of HiPER's APIs and the ability to more naturally express the algorithmic

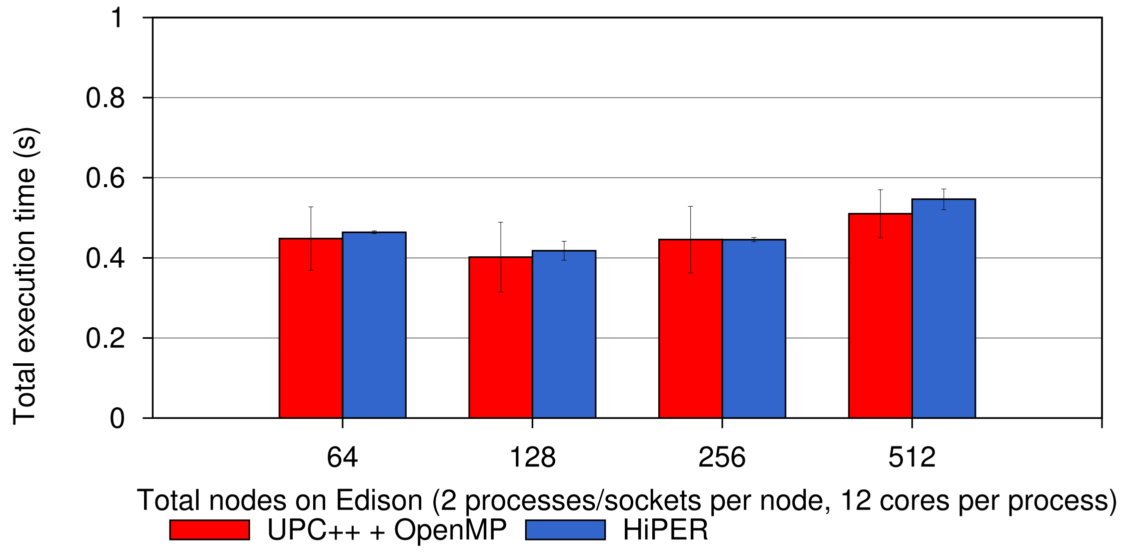

Figure 4.3 : Total HPGMG solve time on up to 512 Edison nodes. Error bars indicate a 95% confidence interval.

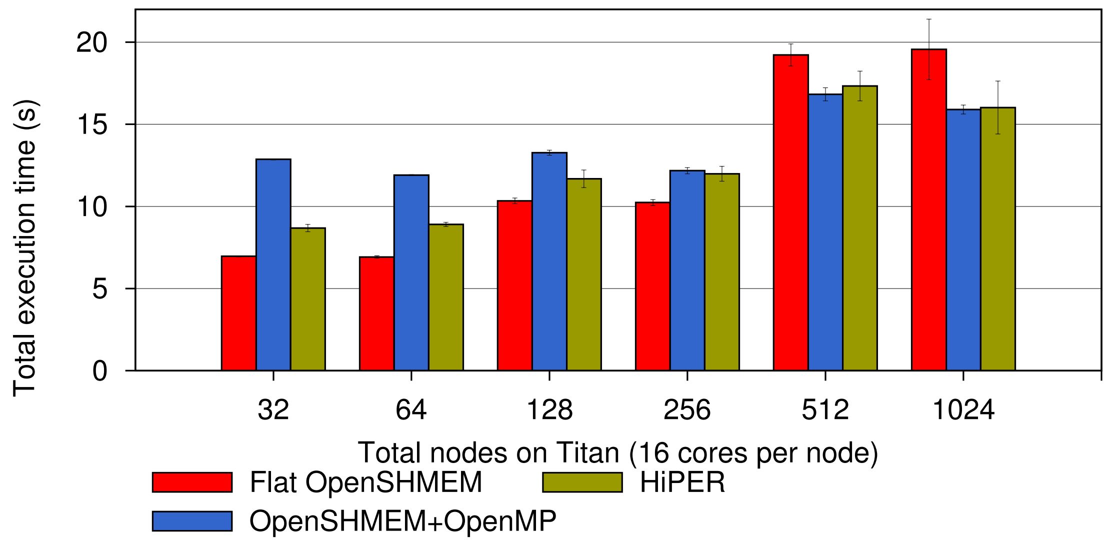

Figure 4.4 : Total ISx execution time. Weak scaling up to 1024 nodes on Titan.

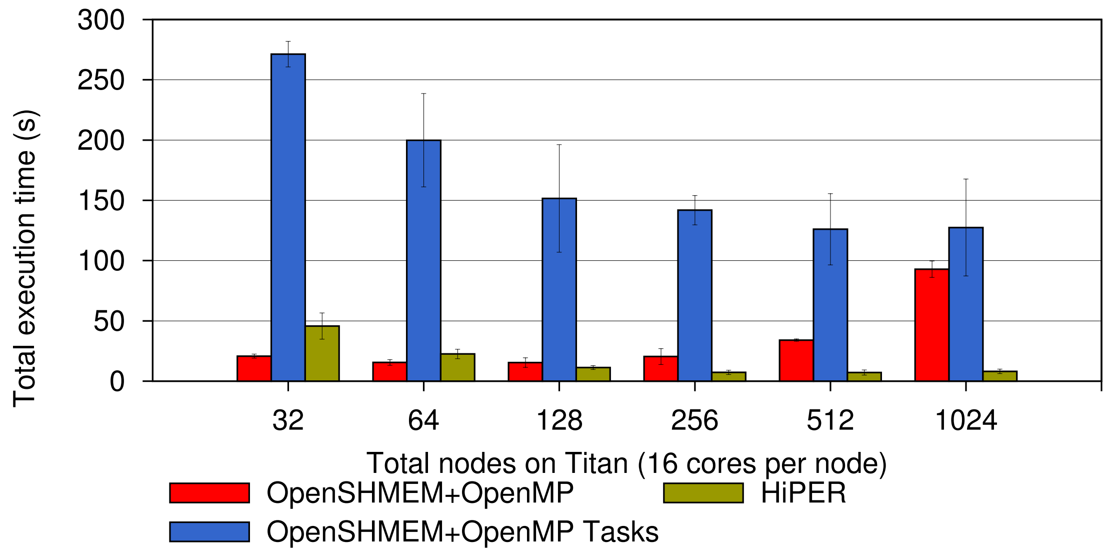

Figure 4.5 : Total UTS execution time.

dependencies of a parallel algorithm, we expect these types of workloads to perform better on HiPER and be more naturally expressible using its APIs.

UTS

Figure 4.5 shows the overall execution time of UTS using OpenSHMEM+OpenMP, OpenSHMEM+OpenMP Tasks, and AsyncSHMEM.

The hand-coded OpenSHMEM+OpenMP version of UTS scales similarly to HiPER up to 128 nodes, but starts to degrade as contention from distributed load balancing increases.

Because of the lack of integration between OpenSHMEM and OpenMP, the OpenSHMEM+OpenMP version that uses OpenMP's tasking APIs performs slowly as coarse-grain synchronization is required to join all tasks before performing distributed load balancing using OpenSHMEM.

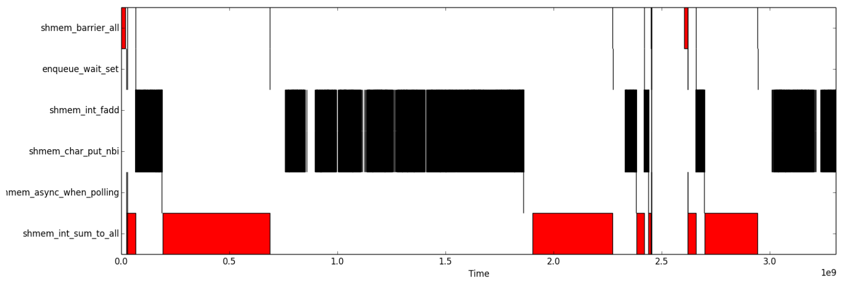

Figure 4.6 : A trace of OpenSHMEM calls and their elapsed time inside a single PE from an execution of Concurrent-CRC on 128 nodes.

Graph500

We implement HiPER versions of the two OpenSHMEM-based implementations of Graph500 presented in [91], named Concurrent and Concurrent-CRC. In contrast to UTS, Graph500 is an entirely network- and memory-bound benchmark. For the Concurrent and Concurrent-CRC implementations of Graph500, the addition of multi-threaded parallelism does not benefit parallelism.

Instead, HiPER is used for concurrency, programmability, and unified scheduling for Graph500. We also use it as a basic illustration of how HiPER enables tooling.

The Concurrent and Concurrent-CRC reference implementations of Graph500 use polling to detect incoming RDMA from remote OpenSHMEM PEs. We use the novel `shmem_async_when` API introduced in this work and enabled by HiPER to replace this polling with asynchronously spawned tasks which are triggered when an incoming RDMA is detected. This change reduces clutter in the code, and hands over the problem of polling intervals and other scheduling issues to the global HiPER scheduler. It also improves the ability of the HiPER system to visualize the application workload by exposing more semantic information to the runtime. For example, Figure 4.6 demonstrates how the HiPER runtime can be used to visualize traces of execution from a single PE with high-level semantic information attached by HiPER modules.

Figure 4.7 reports overheads from using the HiPER runtime to schedule triggered

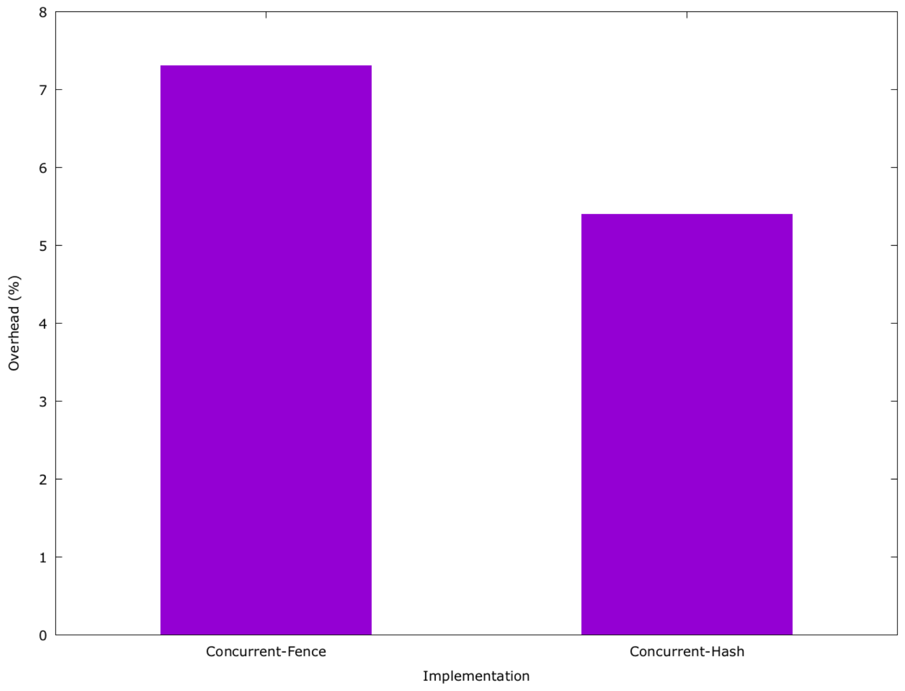

Figure 4.7 : HiPER Overheads for Graph500.

asynchronous tasks rather than using manual polling. In general, we see approximately 5% overhead. While this is larger than desired, it is an upper limit. There has not yet been any exploration of more complex, cooperative, non-greedy scheduling algorithms for computation and communication on the HiPER runtime. Most of this overhead likely comes from memory allocation system calls needed for task creation in the runtime, which are unnecessary in a manual polling approach.

4.4 HiPER Related Works

We identify two categories of related works for HiPER: frameworks designed to improve the composability of multiple software components, and frameworks designed to improve the scalability or programmability of heterogeneous platforms.

4.4.1 Composable Frameworks

To frame the discussion of related work, we start by defining six types of software components that are likely to be in use on future HPC systems and which should therefore be composable with each other:

1. **Hand-coded, accelerator kernels**: These are kernels executed on the accelerator in an HPC system which can be modified by a programmer. The fact that these kernels are modifiable means that composability is defined by the programming system used to write them.

2. **Third-party accelerator libraries**: These libraries offer accelerator kernels which a programmer cannot modify. As a result, software components of this type are more constrained in their composability unless the library developer has deliberately exposed sufficient functionality.

3. **Hand-coded, host kernels**: These are kernels that are executed on the host, latency-oriented cores of a system and as a result may be single-threaded. Like the accelerator variant, the composability of these kernels with other software components depends on the programming system used.

4. **Third-party host libraries**: Like accelerator libraries, but intended for execution on the host, latency-oriented cores of a heterogeneous system.

5. **Inter-node communication libraries**: These libraries enable the communication of data between shared-memory nodes. Their composability with other components is dependent on the functionality they expose and support internally.

6. **Intra-node communication libraries**: These libraries enable explicit data movement within a node (e.g. CPU to GPU). Their composability with other components is dependent on the functionality they expose and support internally.

We focus on evaluating related works in terms of the extent to which they enable composability between different types of software components.

Some past works have already explored the use of thread offload for inter-node communication using techniques similar to this work. In [92], [93], and [94], the authors demonstrated the performance and scalability benefits of a dedicated communication thread to which all communication is funneled. However, these works were all hard-coded to a single communication library (MPI or UPC++), dedicated an entire OS thread to communication (likely hurting the performance of more compute-bound applications), and in the case of [94] depended on changes to the MPI runtime itself. Hence, none of these works are applicable to other software components.

GPU-Aware MPI [47] enabled the direct communication of data from a GPU sitting in one node of a cluster to another GPU sitting in a different node. This work demonstrated both performance benefits from a more direct data path and programmability benefits as a single asynchronous MPI call is required. While this work is restricted to composing NVIDIA GPUs and MPI, the techniques are more generally applicable. A future direction of the HiPER project would allow registered modules to query for other modules which they can offer tighter integration with. With this capability, a GPU-Aware communication module could use techniques from [47]. However, today this capability does not exist in HiPER.

The recent introduction of `target` and `task` directives, as well as the `depend` clause in OpenMP [39] have made composing accelerators and host parallelism using OpenMP possible. A programmer may create dependencies between tasks running on the host and accelerator kernels. While the abstractions offered by OpenMP mean that this enables the composability of any accelerator with host parallelism, in reality this support has only been added for GPUs to date. However, like HiPER, the higher level abstractions of OpenMP mean that the composability it enables has a much broader scope than most related works.

There have also been several research projects into composing GPUs with host parallelism [95][96][83]. In general, the approach taken by these works is similar to that taken by work on communication offload: a dedicated GPU management thread is used to schedule work across all GPUs in the system. These works have similar challenges, in that they are usually hard-coded to a single accelerator type and lose a whole OS thread to GPU management.

XKaapi [97] contributes a work-stealing, locality-aware runtime for scheduling tasks with internal parallelism across CPUs and GPUs. This runtime offers automatic data coherency across CPUs and GPUs and automatic load balancing across devices. While the data coherency contributions are less relevant today with the upcoming release of hardware-supported GPU Unified Memory [46], the load balancing contributions of XKaapi would ease programmer burden when combining CPUs and GPUs. Supporting this kind of native capability would require significant extensions to the HiPER runtime.

Another approach to the composability of multiple software components is to hide them from the programmer entirely. In projects such as Legion [98][99], the aim is to use high level abstractions to completely hide the underlying components being used. Legion, for example, allows the programmer to express accelerator parallelism, host parallelism, and distribution without using any low-level frameworks. While there are clear programmability benefits to this approach, it also can suffer from a lack of fine-grained tunability and extensability. With HiPER, new modules can quickly add new capabilities.

Lithe [100] focuses on composing libraries that use one or more processing units on a shared multi-processor. It proposes APIs which allow these libraries to request and yield cores, relative to a parent in the Lithe scheduler. While this is a scalable and elegant solution, it does require modifications to libraries to make them composable and its scope is limited to composing systems that share the same computational resource (e.g. an OpenMP host runtime and Intel MKL).

4.4.2 Heterogeneous Programming Frameworks

The work presented in EXOCHI [101] is divided into two parts. The Exoskeleton Sequencer (EXO) includes hardware support for a number of useful features in a shared-memory heterogeneous platform, including a MISP exoskeleton which allows interaction between a IA32 processor and non-IA32 accelerator using user-level interrupts, an Address Translation Remapping mechanism which has the IA32 processor in a heterogeneous processor handle page faults by proxy for an accelerator to support a shared virtual address space, and Collaborative Exception Handling which again uses an IA32 processor as a proxy for handling hardware exceptions on a non-IA32 accelerator. C for Heterogeneous Integration (CHI) provides a programming environment for EXO-like architectures by adding inline accelerator-specific computation, fork-join or producer-consumer style parallelism for sections of inline accelerator code, and a way to specify input/output/resident memory regions for accelerator code segments. CHI extends OpenMP to provide a familiar programming environment for heterogeneous, shared-memory processors. In the paper, performance improvements of up to 10.97× were demonstrated for highly-optimized media applications. This work is similar to OpenACC in its OpenMP-like interface, and supports multiple ISAs in a single binary. Due to the hardware support added by EXO, CHI is a high-performance and productive programming environment for shared-memory heterogeneous accelerators. It is unclear how applicable this type of programming environment would be for processors with discrete memory. However, with the recent addition of hardware-supported shared memory for PCIe NVIDIA GPUs, the EXOCHI work is relevant to future HPC platforms.

Merge [102] describes a programming model, compiler, and runtime built on EXOCHI that uses programmer annotations of functions to understand the target architecture as well as necessary conditions for correct execution. These architecture-specific implementations of common functions are often at different granularities, reflecting the architecture-specific optimal granularity for different processors. By filling a work queue with tasks that include metadata about supported architectures Merge can schedule available work on all processors in a heterogeneous platform and do so efficiently thanks to EXO's hardware support. Merge decomposes a MapReduce application into side-effect free C++ tasks for scheduling on the Merge runtime. Merge was evaluated on the same heterogeneous processors as EXOCHI as well as a 32-way Unisys SMP system and demonstrated up to 8.5× and 22× speedup on those platforms, respectively. Merge's main contributions are a higher-level MapReduce framework on top of EXOCHI and the dynamic selection from multiple architecture-specific implementations of the same function to achieve high utilization and well-performing mappings of tasks to architectures.

Phalanx [103] describes a library-based, PGAS, task-based programming model and runtime system for distributed and heterogeneous CPU+GPU machines. Like HiPER, Phalanx includes a platform model composed of places representing processors and attached memories. Like other works, Phalanx uses a tree-based rather than graph-based platform model (as in HiPER). Phalanx exposes a task-based programming model where tasks may themselves include internal parallelism, similar to thread blocks within a CUDA grid. Each task is described as either "streaming" in which case each sub-task is entirely independent, or "parallel" in which case sub-tasks within the same grouping task may synchronize with each other. Task objects can be layered on top of each other, allowing programmers to specify different scheduling constraints at different work granularities. Phalanx allows asynchronous tasks to be launched without execution predicates, or with a future-based API that declares one task as dependent on others. Phalanx also defines a memory model that accounts for discrete address spaces in a single program, supporting both distributed execution and discrete accelerators. Phalanx includes pointer objects which encode both the memory address and place of the referenced object. Phalanx offers explicit memory management APIs for allocation, de-allocation, and transfer. The authors of [103] implement Phalanx on top of GASNet, OpenMP, and CUDA and demonstrate scalable performance on shared- and distributed-memory implementations of dense matrix-matrix multiplication, sparse matrix-vector multiplication, and 2D FFT across a wide variety of platforms.

The PaRSEC system [104] is a dataflow programming and runtime system for heterogeneous platforms. PaRSEC uses a dataflow model of an application to automatically manage the scheduling of both coarse-grain user-written tasks and the implicit data movement required by those tasks across heterogeneous computational units. PaRSEC focuses on a re-thinking of how parallel programs are expressed, allowing programmers to explicitly express and tune their applications in the dataflow model. All composability and mapping to heterogeneous hardware is handled automatically behind the scenes.

4.5 HiPER Discussion and Conclusions

While HiPER's use of modules, generalized work-stealing, and an abstract platform model makes it a general framework, it is important to consider where it might struggle to enable composable components. In particular, we believe HiPER's main challenge is in supporting components that share the CPU with the HiPER runtime itself. For example, supporting composable MKL would require logic in the HiPER runtime for 1) forfeiting CPU cores for the use of MKL, and 2) scheduling MKL on those cores specifically. Indeed, this is the exact challenge that Lithe [100] solves, demonstrating that this type of composition will require modifications to the software components themselves, an undesirable property and something HiPER deliberately avoids.

An important item to note is the tooling that HiPER enables. Like any unified scheduler, the HiPER runtime is aware of all of the work executing on a system. Hooks have been added to the HiPER runtime which enable programmers to gather statistics on time spent in different calls to different modules. While any standard performance profiler could provide this information, HiPER can add high-level, module-specific semantic information about performance bottlenecks. This information was useful in optimizing applications and the HiPER runtime itself. Similar ideas are being explored in the OpenMP Tools APIs [42].

In conclusion, HiPER is a framework for enabling the composition of a variety of software components, including accelerator libraries, communication libraries, storage libraries, and host parallelism. Using a foundational work-stealing runtime, an abstract platform model, and pluggable software modules HiPER enables unified scheduling of near-arbitrary software components, as well as the expression of dependencies between them. This paper has explained the high-level system design, described in detail its implementation, illustrated the programmability and performance benefits of such a system, and pointed out opportunities for future optimizations in the HiPER framework.

Chapter 5

Supporting HPC Programmers with Novel Tooling

Recall in Section 1.2 we described an envisioned HPC workflow in which both domain experts and HPC gurus are able to productively program heterogeneous platforms using different programming models. Chapter 3 illustrated how the HJ-OpenCL, HCL2, and SWAT works enable productive and well-performing heterogeneous execution of applications written by domain experts and executed on managed runtimes. Chapter 4 discussed the design and implementation of the novel HiPER runtime system for supporting native execution on current and future heterogeneous platforms. HiPER focuses on being usable by HPC gurus, improving the composability and tunability of low-level frameworks without removing optimization opportunities.

The workflow in Section 1.2 also argued that novel HPC tooling would be necessary to augment and support these novel programming models. In particular, this thesis makes concrete contributions in the use of software checkpointing as a software development tool and in the area of performance prediction. This thesis will also discuss how these or similar techniques integrate with and complement the runtime and programming model contributions described in Chapters 4 and 3.

5.1 Background: Checkpointing

5.1.1 Motivation

Application checkpointing is a well-studied problem with a variety of use cases. An application checkpoint is a snapshot of program state that includes sufficient information to resume execution of an instance of that application from an intermediate point in time. Application checkpointing is most applicable to the following use cases in scientific computing:

1. Resiliency against software or hardware errors.

2. Debugging of application failures or numerical errors.

3. Performance profiling and tuning of application hotspots.

Resiliency is the classic motivating example for checkpointing. Creating periodic checkpoints allows a crashed process to be immediately resumed from its most recent checkpoint.

Checkpointing also enables debugging and performance tuning. Creating periodic checkpoints simplifies the process of reproducing a program error or analyzing a performance hotspot by allowing programmers to resume from a checkpoint immediately prior to the relevant code region. For long-running scientific applications, this can save days or weeks of time and enable more rapid iteration on an application. Checkpoints can be integrated into automated testing environments to protect against future regressions, or used as representative application inputs for a performance auto-tuning framework.

Modern debugging and performance profiling tools generally induce a significant amount of overhead. This can make program behaviors difficult to reproduce. Checkpointing techniques are uniquely suited to resolve these issues by allowing the application developer to compile and run their application with the highest optimization settings by default, but re-compile with different compiler flags or added instrumentation before resuming from a checkpoint. To make it feasible to run with checkpointing permanently enabled, checkpointing frameworks must keep overheads low and retain original application behavior.

The existing research in application checkpointing has primarily focused on resiliency, with some attention given to debugging using record-replay techniques. However, we argue that research in checkpointing should place just as much emphasis on their use as a software development tool or as an underlying infrastructure for other tools, as it does on resiliency.

For example, in the application acceleration work in [105] and [106], work focused on offloading the computational kernels of a geophysical simulations to GPUs. These geophysical simulations are long-running and their behavior and data varies over time during a single simulation. This means that a given simulation run may only encounter a state which triggers a bug after multiple hours or days of execution. Manually created, application-specific checkpoints enabled rapid debugging and resolution of difficult-to-reproduce bugs in these projects.

Additionally, in the HCL2 and SWAT projects presented in Section 3.4 checkpoints were invaluable in enabling performance profiling and debugging of auto-generated OpenCL kernels. In large-scale, multi-tenant systems such as HCL2 and SWAT, it can be difficult to iteratively inspect and modify the behavior of components which are logically far away from the application code, sitting in the lower levels of the software stack. Checkpoints allow for the capture of application and runtime state at these lower levels of the software stack and the repeated replay, inspection, and optimization of accelerator kernels. Then, the techniques which were found to be effective in optimizing the kernels captured by checkpoints can be plugged back into the overall framework from which the checkpoint was captured.

In this work, we use a compiler- and library-based approach to checkpointing to support all three use cases of checkpointing: resiliency, correctness debugging, and performance profiling. This hybrid compile-time and run-time approach has a number of merits relative to more low-level techniques proposed in past work including fewer platform dependencies, reduced overhead, and a more user-tunable checkpointing process.

CHIMES (CHeckpointing of In-MEmory State) is a checkpoint-restart framework that uses compile-time insights to guide and optimize runtime checkpointing. As a result, CHIMES demonstrates and average of $\sim$5% checkpointing overhead across a wide range of benchmarks and multiple HPC hardware platforms.

5.1.2 CHIMES Design and Implementation

CHIMES design focuses on satisfying the following three constraints:

1. Checkpoints should include all user-visible heap and stack data structures.

2. Checkpoint creation should require minimal assumptions about the execution platform.

3. CHIMES itself should offer sufficient pluggability to be extended to handle third-party data types without negatively impacting performance.

While some past work supports checkpointing kernel-level state (e.g. file descriptors) to some extent [107][108], CHIMES's emphasis on software development rather than full-application resiliency makes this support a lower priority. The most common use case of CHIMES would be to capture in-memory application state prior to a parallel, computational bottleneck such that the correctness or performance of that bottleneck could be analyzed. Parallel kernels are unlikely to include file I/O, network communication, or other system calls that would require checkpointing of kernel-level state. Therefore, CHIMES's emphasis on checkpointing user-visible state is driven by the motivation of this research.

Past work on checkpointing has often added certain platform dependencies, such as running inside of a certain virtualization environment or installing a custom kernel module [108]. These constraints may not be satisfiable in certain deployment environments for performance or security reasons. Therefore, we constrain the research problem to make minimal assumptions about the platform support available for efficient checkpoint creation.

As in Chapter 4, we recognize that HPC applications are multi-tenant systems and that CHIMES may be required to support checkpointing of third-party state. For example, user-managed, opaque data structures are quite common in HPC libraries such as pthreads, CUDA, OpenMP, and MPI. Initialization of these data structures can only be properly handled by that third-party library. As a result, we also ensure that CHIMES includes pluggability as a first-class citizen in its design such that initialization of certain third-party data types can be handled by the necessary third-party routines.

This section describes the main contributions of this work: a compiler- and library-based approach to checkpointing single-threaded and OpenMP programs. We start with an overview of what a CHIMES checkpoint contains and then cover step-by-step how the compile-time and run-time workflows 1) create a single checkpoint, and 2) resume from it.

Anatomy of a CHIMES Checkpoint

A CHIMES checkpoint includes the following application state:

1. Per-thread stack contents, including variable names, sizes, types, and values.
2. Global state, including variable names, sizes, types, and values.
3. Constant state, including variable names, sizes, types, and values.
4. Function addresses and function names.
5. Thread hierarchy information indicating which OpenMP threads spawned other OpenMP threads.
6. Heap state changed since the last checkpoint.
7. Metadata on aliased pointers in the host application.
8. Metadata on the pointer hierarchy in the host application (i.e. pointers that point to other pointers).

9. User-provided checkpoint data.

We assume that the heap of the host application accounts for the majority of its in-memory state. Checkpoints are stored on disk as binary files and all checkpoints are incremental in their storage of heap contents. To restore the full contents of an application's heap from a checkpoint, it may be necessary to also traverse backwards through a chain of predecessor checkpoints to find the current state of all bytes in the heap.

Checkpoints are created by programmer-inserted calls to `checkpoint()`, giving the programmer the ability to place checkpoints prior to important, buggy, or long-running code regions.

Note that the current checkpoint format does not include metadata on open files, signal handlers, or other system-managed state. Handling these types of system-specific state leads to less platform flexibility and higher overheads as more system interaction must be instrumented. These types of objects can be restored by custom user callbacks during checkpoint creation and resume.

Compile-Time Analysis and Transformations

Figure 5.1 shows the high-level workflow of the CHIMES compilation pass. CHIMES processes one .c file at a time. The input file first goes through a preprocessing pass that performs some lightweight transformations to simplify the main transformation pass later, including hoisting expressions with side effects out of return statements or function call parameters.

Following the preprocessing pass, LLVM bitcode is generated from the preprocessed file and passed through an LLVM analysis pass. The analysis pass's primary purpose is to produce information on 1) intra-procedural pointer aliasing, and 2) the memory locations modified within each procedure.

During intra-procedural alias analysis, pointer variables within a function that may alias are marked as part of a single alias group. A globally unique alias group

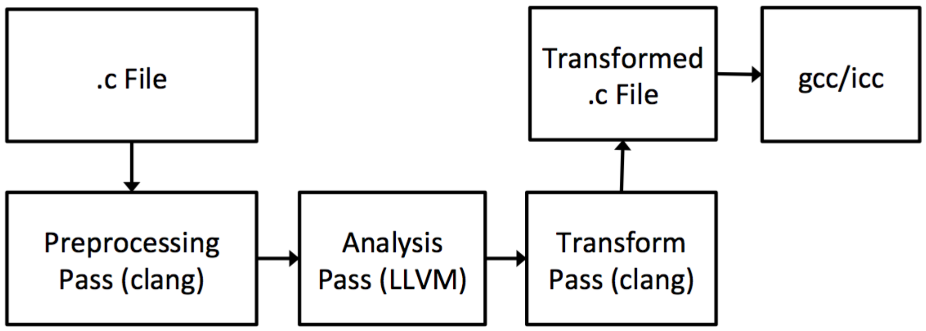

Figure 5.1 : The CHIMES compilation workflow.

ID is generated for each alias group in each function. The analysis pass also tracks which alias groups are pointed to by other alias groups.

The analysis pass identifies alias group change locations, i.e., source code locations where a `STORE` to a member of an alias group occurs. This information is used at runtime to calculate heap state that may have changed since the last checkpoint was taken.

Alias groups that may be modified between two checkpoints are collected by propagating alias group change information from the original change location down control flow paths until it encounters 1) a `checkpoint` call, or 2) a redirection of control flow that may lead to a checkpoint being created. This generally leads to the aggregation of alias group change locations at checkpoint calls, at function calls, at conditional branches, and at return statements. These aggregate alias group change locations record all of the alias group IDs that may have been modified by `STORE` operations since the last aggregate alias group change location.

In addition to storing alias groups that have definitely changed, it is necessary to store alias groups that may be changed by a call to an externally defined function or function pointer. Any arguments passed by reference and all global values are conservatively marked as “possibly changed” and added to the next alias group change location as such. If at runtime CHIMES finds that the external call was instrumented by CHIMES, these “possibly changed” alias groups are removed from the change location. Otherwise, these alias groups and any alias groups indirectly reachable from them are conservatively added to the change location as definitely changed. This ensures that if an external library modifies any state tracked by CHIMES, the changes are included in the next checkpoint.

The analysis pass also generates metadata on:

1. Global, constant, and stack variables
2. Alias groups passed as parameters to function calls or returned by functions
3. Heap management locations, such as calls to `malloc`, `calloc`, `realloc`, and `free`
4. The call tree for this compilation unit, including any externally defined functions that are called but are currently unresolvable.
5. The OpenMP pragmas and clauses in the source code.

Once the analysis pass completes, the metadata generated by it is passed to the transformation pass, which is implemented as a standalone clang tool using LibTooling [109]. The CHIMES transformation pass performs a source-to-source transformation of the preprocessed source code. This transformation primarily inserts calls to CHIMES library functions that track application state identified by the analysis pass (e.g. stack variables, globals, function addresses).

Every compilation unit (i.e. input file) has a static, one-time module initialization function inserted which passes module information to the CHIMES runtime prior to entering the application's `main`. This `module_init` function is depicted in Figure 5.2.

The transformation pass also instruments each function with a variety of CHIMES runtime callbacks that are used for tracking stack variables, heap allocations, inter-procedural alias creation, alias group change locations, entrance or exit from OpenMP parallel regions, or changes to the call stack.

```
static int module_init() {
  libchimes_init_module(...);
  return 0;
}
static const int __libchimes_module_init =
    module_init();
```

Figure 5.2 : The CHIMES `module_init` function is used to pass module-specific information to the CHIMES runtime before entering `main`.

As part of the transformation pass, jumps and labels must be inserted in any functions that may be on the stack when a checkpoint is created. The labels allow a resume of a checkpoint to skip to the original checkpoint location while reproducing the original call stack using only jump operations and function calls. A label is added to every callsite that may directly or transitively create a checkpoint, to each CHIMES callback that registers stack variables, and before each OpenMP parallel region that may have a checkpoint created inside.

By inserting jumps between these labels, we build a control flow tree within each function that allows the transformed application to jump from the entry point of a function, through stack variable registrations, and into any parallel regions or function calls necessary to reproduce the stack and thread state of the application when checkpointed. The root of the tree is the entrypoint of the function. The children at each layer of the tree are any parallel regions spawned from the current node of the tree or checkpoint-causing function calls made. This model supports resume from arbitrary call stacks and nested parallel regions, including recursive ones. The use of this control flow tree will be illustrated further in Section 5.1.2.

CHIMES Checkpointing Runtime

The transformations described in Section 5.1.2 add instrumentation to the host application. This instrumentation registers all checkpointable state with the CHIMES runtime. This section expands on the state stored by the CHIMES runtime and how that state is used to create checkpoints.

At a glance, the CHIMES runtime stores the following:

1. A mapping from the address of a heap allocation to its metadata.
2. A list of global and constant variables, along with associated metadata.
3. A mapping from function names to their addresses in the running application.
4. A mapping from each alias group to all other alias groups that have become aliased with it at runtime.
5. Points-to information for each alias group.
6. Per-thread stack trace information, stored as a stack of integer IDs.
7. A full call tree for the program, dynamically constructed from the per-compilation unit call tree information passed to `libchimes_init_module` as described in Section 5.1.2.

Precise and correct alias analysis is vital for CHIMES checkpointing: the mapping from aliases to heap allocations is used to determine what heap regions may have changed since the last checkpoint and need to be included in the next checkpoint. We perform inter-procedural alias analysis at runtime by passing the alias group information for function parameters and return values to CHIMES callbacks at the entry of each function, exit of each function, and before each callsite. For example, the alias groups of a formal parameter and an actual parameter will be merged following a function call. This analysis works across functions in different compilation units and through function pointer calls as long as both the source and target are transformed by CHIMES.

To illustrate the CHIMES runtime, we consider a simple example function in Listing 5.3 that creates a checkpoint. Pseudocode of the transformed code generated by the CHIMES transform pass for this function is shown in Figure 5.4.

```
int *sum_alloc(int *a, int b) {
  int sum = *a + b;
  *a = sum + b;
  int *alloc = (int *)malloc(sum *
      sizeof(int));
  checkpoint();
  return alloc;
}
```

Figure 5.3 : A simple code example calling `checkpoint`.

```
int *sum_alloc(int *a, int b) {
  libchimes_enter_func("sum_alloc", sum_alloc,
      ...);
  if (____libchimes_resuming) goto lbl_0;

  lbl_0: int sum;
  libchimes_register_stack_var(&sum, ...);
  if (____libchimes_resuming) goto lbl_1;
  sum = *a + b;
  *a = sum + b;

  int *alloc;
  lbl_1: libchimes_register_stack_var(
      &alloc, ...);
  if (____libchimes_resuming) {
    switch (libchimes_next_call()) {
      case (0): goto lbl_2;
      default: abort();
    }
  }
  alloc = (int *)malloc(sum * sizeof(int));
  libchimes_register_heap(alloc, ...);

  libchimes_alias_groups_changed(...);
  lbl_2: checkpoint();

  libchimes_leaving_func(...);
  return alloc;
}
```

Figure 5.4 : An example of the transformed code generated from Figure 5.3.

Upon entering `sum_alloc`, the CHIMES runtime is notified that a new entry should be pushed on the current thread's stack by the `libchimes_enter_func` callback. The information passed to `libchimes_enter_func` also assists with inter-procedural alias analysis for the parameters of `sum_alloc`.

Then, the transformed code checks to see if the current program execution is a resume from a checkpoint using `____libchimes_resuming`. We assume it is not for this example, Section 5.1.2 will provide more detail on how a checkpoint is resumed. Next, the `sum` and `alloc` stack variables are registered using `libchimes_register_stack_var`, and the heap memory allocated in `alloc` is registered with the runtime using `libchimes_register_heap`.

Immediately before creating a checkpoint, `libchimes_alias_groups_changed` is called to inform the runtime of which alias groups have been modified since the last checkpoint. This call would inform the CHIMES runtime that `sum` and `alloc` have both had their values set.

The `checkpoint` function has three main steps. First, it serializes all program state outside the heap into byte buffers, including stack variables, per-thread stack traces, global variables, and constants.

Second, the checkpoint function determines what parts of the heap need to be checkpointed based on 1) alias group change tracking, and 2) hashing of heap contents. The first stage is straightforward: CHIMES has been collecting a set of modified alias groups since the last checkpoint. Combined with a mapping from alias groups to heap allocations, CHIMES can construct a set of user heap allocations that may have changed since the last checkpoint.

In the second stage, CHIMES subdivides heap allocations into evenly sized chunks and computes a hash for the contents of each chunk. The chunk size is configurable, but defaults to 4MB. Hashing is done using the xxHash library [110]. Hashes are stored between checkpoints and only chunks whose hashes have changed since the last checkpoint are added to this checkpoint. Once `checkpoint` has determined exactly

```
> CHIMES_CHECKPOINT_FILE=chimes.2.ckpt \
        ./a.out ...
```

Figure 5.5 : An example resume of an application using CHIMES.

which regions of the heap need to be checkpointed, in-memory copies of each region are made.

Finally, the serialized byte buffers from the first step of `checkpoint` and the heap contents from the second step are passed to a dedicated checkpointing thread which writes them out to disk. The checkpointing thread uses asynchronous writes to keep the checkpointing thread off-core. If an out-of-memory error occurs while preparing data for checkpointing, the `checkpoint` function becomes blocking and writes heap state directly from the application buffers.

After the call to `checkpoint` returns in Figure 5.4 we call `libchimes_leaving_func` and return from `sum_alloc`. `libchimes_leaving_func` aids with inter-procedural alias analysis for return values and pops from the stack trace for this thread.

Resuming From a Checkpoint

The previous section covered checkpoint creation. In this section, we look at how a checkpoint can be used to resume program execution from the point-in-time that the checkpoint was created.

Specifying the checkpoint file to use when resuming is as simple as setting an environment variable and running the original executable with the same command-line arguments, as shown in Figure 5.5.

During initialization, the CHIMES runtime will detect check for a `CHIMES_CHECKPOINT_FILE` environment variable and, if found, load the serialized program state from it. Restoring program state from the serialized state is a three step process.

First, during runtime initialization at the start of program execution CHIMES reads the contents of the checkpoint file and stores the deserialized data. CHIMES uses the deserialized heap, constant, and globals data to construct a mapping from the addresses of objects in the address space of the original execution to their addresses in the current execution. This information is used to update pointers stored in the stack, heap, and globals. The pointer translation process uses a self-balancing binary tree to store the mapping from addresses in the checkpointed address space to their addresses in the current address space. Each node in this tree is an address in the checkpointed address space and the number of allocated bytes that follow it. A binary tree is used to keep lookups efficient.

The second step of the restore process is to restore the thread and stack state of the program using the labels and jumps discussed in Section 5.1.2. Figure 5.4 shows an example of the code generated to support this step. Upon entering a function with `____libchimes_resuming` set to true, control flow will jump to each stack variable registration, passing updated addresses for each of these variables to the CHIMES runtime. Once all stack variables have been traversed, `libchimes_next_call` is used to pop the next entry from the checkpointed stack for the current thread. The value popped determines which label to jump to next. This jump may target a function call or a nested parallel region. At the completion of this step, all threads will be inside a call to `checkpoint` with the same stack trace that the original program followed to create the checkpoint being restored, but with stale stack state. Note that this approach does not support restoring checkpoints taken from beneath function pointer calls, as there are no guarantees that the function pointer's value will be correct on resume. Future work could remove this restriction by special-casing the restore of function pointers.

This label-jump approach is the main reason for the CHIMES preprocessing stage. During the CHIMES preprocessing stage one of the transformations performed is to hoist any expressions with side effects out of function argument lists. If this step were not taken, jumping to a function call would cause its arguments to be evaluated with only partial program state restored.

The third step of the restore process happens from inside the final `checkpoint` call. First, the values of all stack variables are restored using the values deserialized from the checkpoint file. Then, all pointers in the stack, heap, and global variables are translated from the old address space to the new address space using the address information collected from the previous two steps. This step also finds all variables whose type is either a pointer-to-pointers or a pointer-to-structs and recursively performs the translation for all pointers reachable from each variable.

This pointer translation step is complicated by the flexibility of the C programming language. `void*` pointers may point to data structures that contain pointers which need to be translated. If these "hidden" data structures are unreachable from anywhere else, the obfuscation of a `void*` type prevents CHIMES from identifying all pointers in the program. We have implemented a feature (disabled by default) that brute force searches any heap allocations behind `void*` pointers for pointers that can be updated. While this feature has not caused unexpected behavior when enabled, it is possible it could mutate data which appears to be a pointer from the old address space but which is not.

After the address translation completes each thread returns from the `checkpoint` call and execution continues as usual with a fully restored program.

Pluggability

In CHIMES, we include a number of hooks to allow users to add custom checkpoint and restore functionality to their applications as needed.

Users can insert custom data in CHIMES checkpoints using `register_checkpoint_handler`, shown in Figure 5.6.

During checkpoint creation, `handler` is called and passed the pointer `data` as its first argument. If handler wishes to add state to the checkpoint, it must set its second

```
void register_checkpoint_handler(
    void (*handler)(void *, void **,
        size_t *),
    void (*restore)(void *, heap_tree *,
        chimes_stack *),
    void *data);
```

Figure 5.6 : Signature of `register_checkpoint_handler`, used to insert custom information into a CHIMES checkpoint.

```
void register_custom_init_handler(
    const char *type_name,
    void (*handler)(void *));
```

Figure 5.7 : Signature of `register_custom_init_handler`, used to register custom data handlers in CHIMES.

argument to be a valid buffer on the heap and set the third argument to be the length of this buffer.

On resume, `restore` is called and passed the address of the restored buffer, a data structure that can translate pointers in the old address space to the new address space, and a data structure that can look up stack variables by name and scope.

Users can also register custom handlers for restoring objects of a certain type using `register_custom_init_handler`, shown in Figure 5.7.

If CHIMES finds an object whose type matches `type_name`, it will pass the address of this object to `register_custom_init_handler`. This is useful for restoring objects specific to third-party libraries (e.g. CUDA, pthreads).

Optimizations

Section 5.1.1 pointed out that for a checkpointing system to be feasible it must be efficient, adding little overhead. A naive implementation of the techniques described in this section would lead to significant overheads for many applications: taking the address of stack variables and functions impedes compiler optimization, excessive function calls and runtime logic adds overhead, and frequent checkpointing would considerably add to execution time. In this section we describe techniques used to limit the overhead incurred by the CHIMES runtime, and evaluate their effectiveness in Section 5.1.3.

One of the most effective optimizations implemented is the CHIMES ShortCut Mode (SCM). For SCM, a duplicate version of each function is emitted with most of the CHIMES instrumentation removed. The only instrumentation kept is heap registration callbacks, which are necessary to associate each allocated buffer with an alias group.

Calling the SCM version of a function reduces overhead, but has some constraints. The called function and all of its callees must be known functions which definitely will not checkpoint. This can be determined using the global call tree constructed at runtime. If the SCM version of a function is called then all changes to and aliasing of alias groups must be evaluated ahead-of-time based on statically known information. This can reduce the accuracy of this information, but not the correctness.

The CHIMES runtime is also aware of the overhead it adds to the host application and limits checkpoint creation to keep overhead below a certain threshold, when possible. Expensive CHIMES runtime callbacks are instrumented to measure the time spent inside. The total time in the CHIMES runtime is tracked and compared to the overall wallclock time of the application. If this ratio exceeds a threshold, checkpoints are not created. This threshold was set to 5% in our experiments. This system includes a maximum allowable period between checkpoints and forces checkpoint creation if that period is exceeded, even if the estimated overhead is greater than the allowable threshold. In our experiments, we set this period to be 60 seconds.

During experimentation, we also found that taking the address of functions and stack variables can significantly degrade the ability of the compiler to optimize application code. We addressed this problem by using POSIX `dlsym` to fetch function addresses, and used liveness analysis to reduce the set of stack variables that had to be registered.

5.1.3 CHIMES Performance Evaluation

We evaluate CHIMES performance based on three metrics: overhead added, checkpoint size relative to the size of the application in memory, and number of checkpoints created. We use benchmarks from the Rodinia benchmark suite [62], benchmarks from the SPEC benchmark suite [111], the Lulesh [112] application, the CoMD [113] application, the UTS [89] application, and a custom 3D stencil benchmark called Iso3D that is representative of wavefront propagation simulations from the energy industry.

All benchmarks and metrics are evaluated on two hardware platforms. Platform A contains a 12-core 2.80GHz Intel X5660 CPU, 48GB of system RAM, and is connected to a GPFS storage system by QDR Infiniband. Platform B is an IBM Power 755 node containing 4 eight-core 3.86GHz POWER7 CPUs with 4-way simultaneous multithreading (128 hardware threads in total), 256GB of system RAM, and is also connected to a GPFS storage system through QDR Infiniband. The GNU C Compiler was used on both platforms, v4.8.5 on Platform A and v4.4.7 on Platform B.

We compare performance of both single-threaded and OpenMP multi-threaded programs. Single-threaded tests are denoted with the label “CPP”. Multi-threaded tests are denoted with the label “OMP”.

All tests are repeated 10 times and the median result is used to build the graphs below. Table 5.1 lists the execution time and memory consumed for each application running on Platform A without CHIMES.

Overheads

To evaluate the overhead of CHIMES, we start by running a transformed version of the application linked with an empty runtime library (referred to as Empty tests). This evaluates the overhead added by only the inserted function calls and other source

Benchmark	**CPP**		**OMP**	
	Time	**Space**	**Time**	**Space**
Iso3D	147.86s	3.21GB	36.38s	3.22GB
Lulesh	32.63s	2.76MB	167.60s	80.32MB
CoMD	101.23s	308.32MB	101.63	314.66MB
UTS	64.07s	15.26MB	5.84s	183.11MB
RodBackprop	103.00s	19.97GB	44.35s	19.97GB
RodBfs	124.11s	2.50GB	121.71s	2.50GB
RodB+tree	10.96s	73.97MB	2.00s	73.97MB
RodHeartwall	112.24s	28.73MB	11.70s	28.73MB
RodHotspot	54.52s	384.00MB	18.65s	384.00MB
RodKmeans	101.46s	132.11MB	16.92s	132.11MB
RodLavamd	122.70s	20.56MB	11.68s	20.56MB
RodLud	8030.28s	16.00MB	8207.23s	16.00MB
RodMyocyte	227.46s	286.91MB	18.81s	286.91MB
RodNn	184.50s	15.32MB	29.06s	15.32MB
RodNw	49.62s	19.20GB	27.99s	19.20GB
RodParticlefilter	9.62s	200.33MB	9.29s	3.07GB
RodSrad	91.47s	85.14MB	12.25s	85.14MB
SPECBotsAlgn	761.60s	1.35MB	63.82s	1.35MB
SPECBotsSpar	791.65s	757.83MB	737.69s	72.31MB
SPECSmithwa	97.51s	0.27MB	0.38s	11.33MB
SPECKDTree	0.22s	40.06MB	40.10s	6.96MB

Table 5.1 : Median execution time and peak memory consumption for the baseline version of each application on Platform A.

code instrumentation. Then, we use a CHIMES library that implements all of the functionality from Section 5.1.2 but does not actually create checkpoints (referred to as No-Checkpoint tests). This measures the overhead added by tracking the state of the application. Finally, we test with the full CHIMES runtime library and measure any increase in overhead caused by creating checkpoints on disk (referred to as Checkpoint tests). All overheads are measured relative to the original application, compiled with gcc -O3.

Note that in some cases the Empty tests may demonstrate higher overhead than the others because No-Checkpoint and Checkpoint tests are often able to enter SCM mode in cases where Empty tests do not.

Figures 5.8 and 5.9 show the results of running the single-threaded tests on both hardware platforms. In general, we see an expected trend of increasing overhead from the Empty tests to the No-Checkpoint tests to the full Checkpoint tests. The median overhead for the Checkpoint tests across all applications on Platform A is 4.3%, and on Platform B is 4.1%.

Figure 5.8 shows a significant slowdown for RodiniaBackprop caused by the CHIMES code transformations interfering with compiler optimizations when stack variable and function addresses are taken, as discussed in Section 5.1.2.

In Figure 5.8, the RodiniaNw results show significant overhead with checkpointing enabled, and in both Figures 5.8 and 5.9 we see similar behavior for SPECKDTree. We find that the added execution time comes from a wait at execution termination for the last and only checkpoint to complete being written to disk. These benchmarks are not characteristic of the long-running iterative scientific applications targeted by this and other checkpointing work. They are short-lived applications for which checkpointing offers little value.

In Figure 5.9, the RodiniaB+tree results show negative overheads when running the No-Checkpoint test. This result is caused by cache behavior. RodiniaB+tree performs many small heap allocations. In CHIMES, a small header (8 bytes) is added

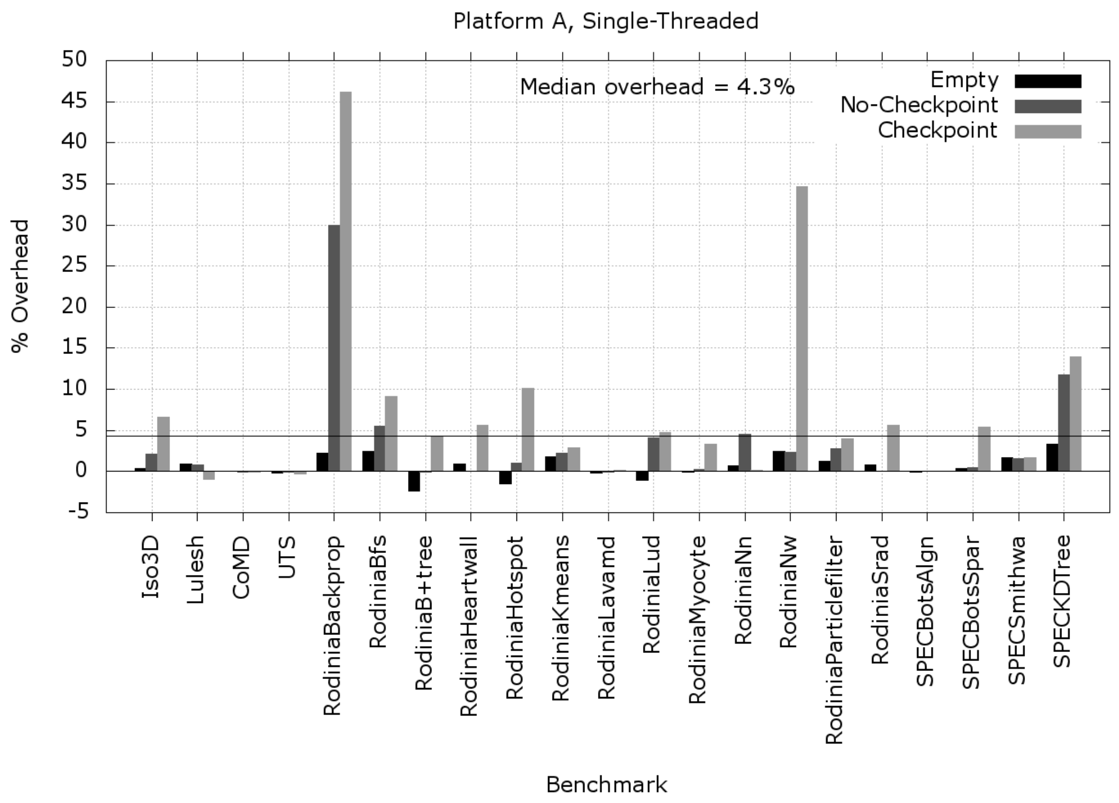

Figure 5.8 : Overheads on Platform A during single-threaded tests.

to each of these allocations, improving the cache characteristics of RodiniaB+tree by pushing more allocations onto separate cache lines. If these allocation headers are removed, No-Checkpoint overhead becomes 1.5%. Note that Platform A and B both have the same L1 cache line size (64 bytes), but that Platform B has a longer L2 cache line (128 bytes vs. 64 bytes for Platform A). This explains why Platform A does not demonstrate this behavior for RodiniaB+tree: its smaller L2 cache lines cause similar caching behavior with and without the added header.

In the OpenMP results, most of the outliers mimic the results from the single-threaded programs. The main difference is the CoMD Empty test on Platform B, where ~50% overhead is recorded. This is caused by the lack of SCM mode in the Empty tests. Without SCM mode, some tight parallel loops are run with instrumenta-

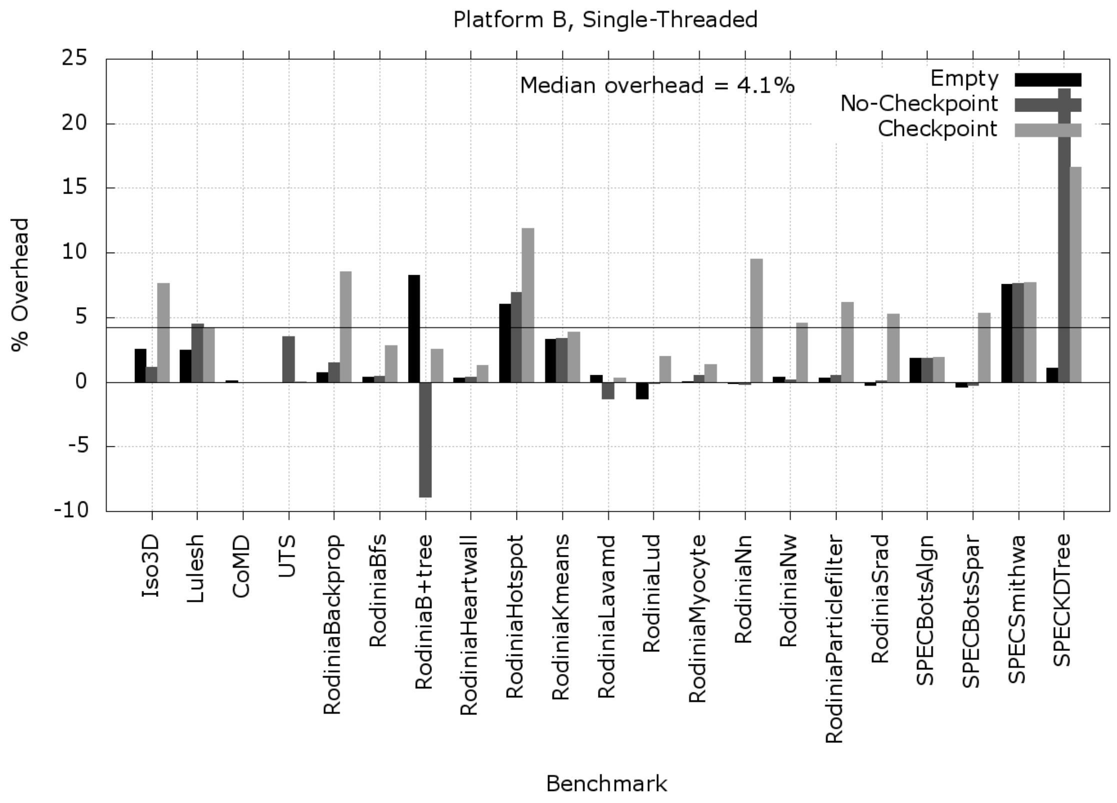

Figure 5.9 : Overheads on Platform B during single-threaded tests.

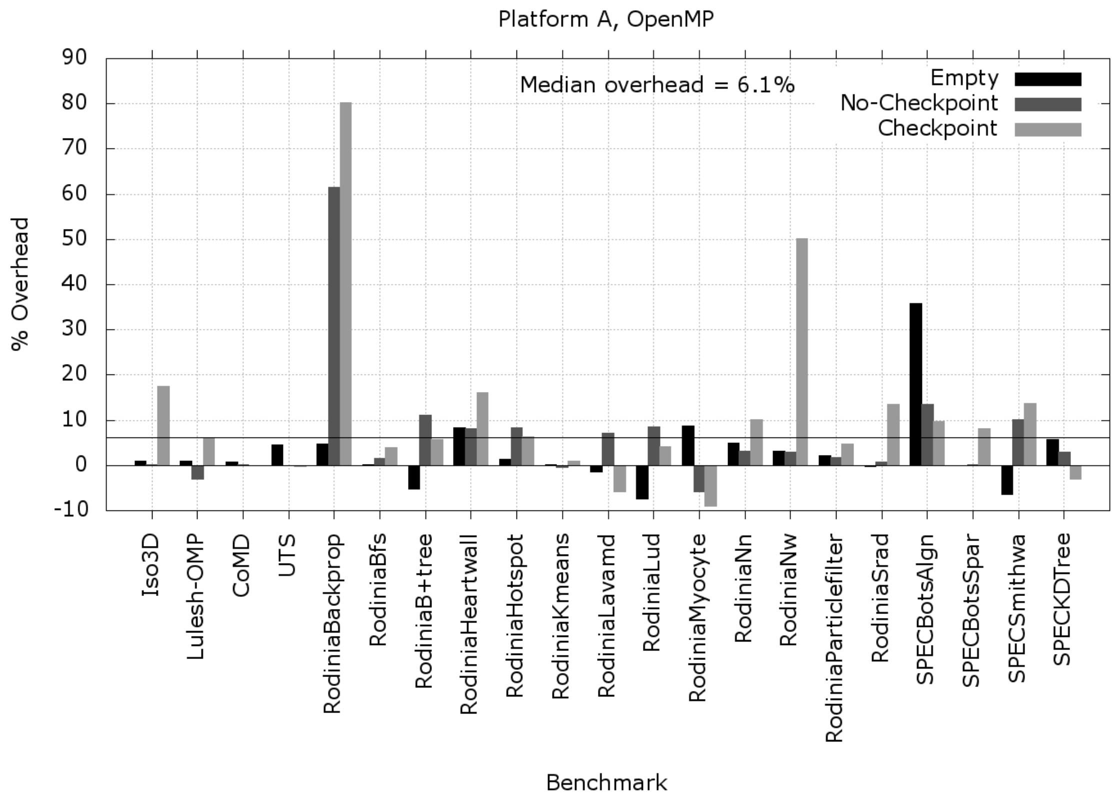

Figure 5.10 : Overheads on Platform A during multi-threaded OpenMP tests.

tion enabled, which includes one inserted synchronization point. This synchronization adds significant overhead on Platform B because it is more parallelism than Platform A (128 hardware threads on Platform B vs. 12 on Platform A).

Otherwise, the OpenMP tests perform similarly to the single-threaded tests, with an average overhead of 6.1% on Platform A and 5.3% on Platform B.

Number of Checkpoints

When evaluating the overhead of CHIMES, it is important to also consider how many checkpoints are being created. Figure 5.12 shows the number of checkpoints created by each application on Platforms A and B.

Some benchmarks execute for an insufficient amount of time to create more than

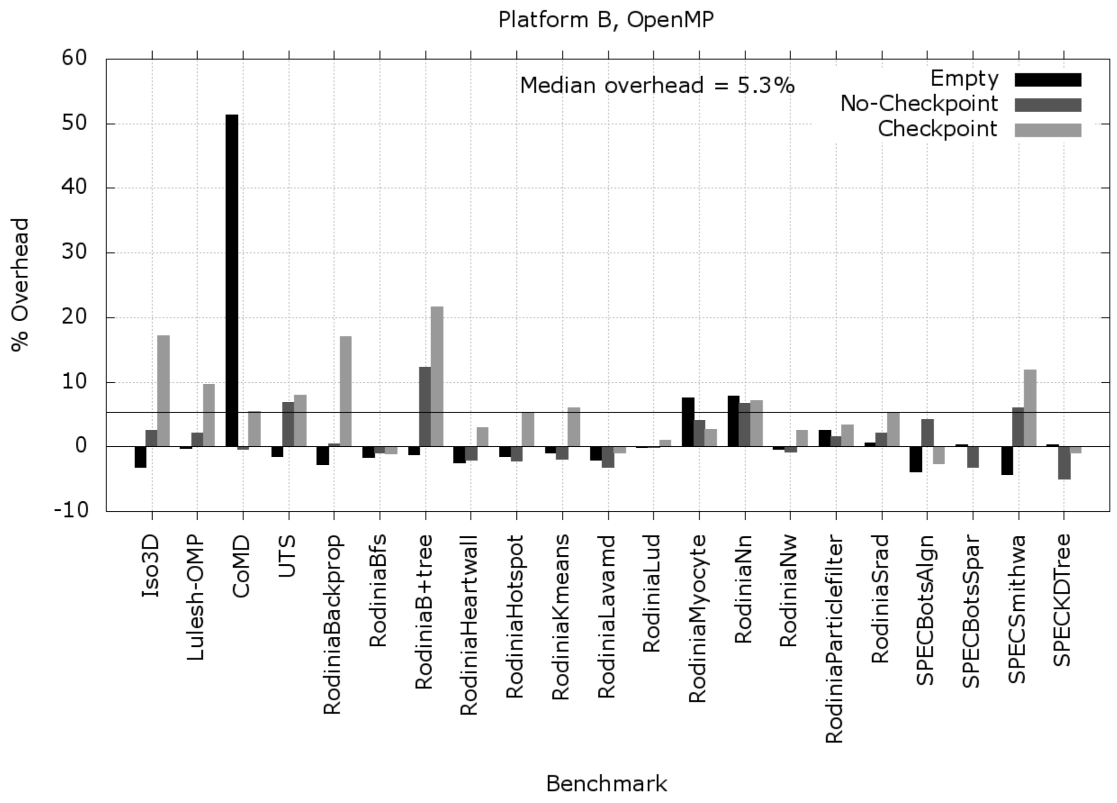

Figure 5.11 : Overheads on Platform B during multi-threaded OpenMP tests.

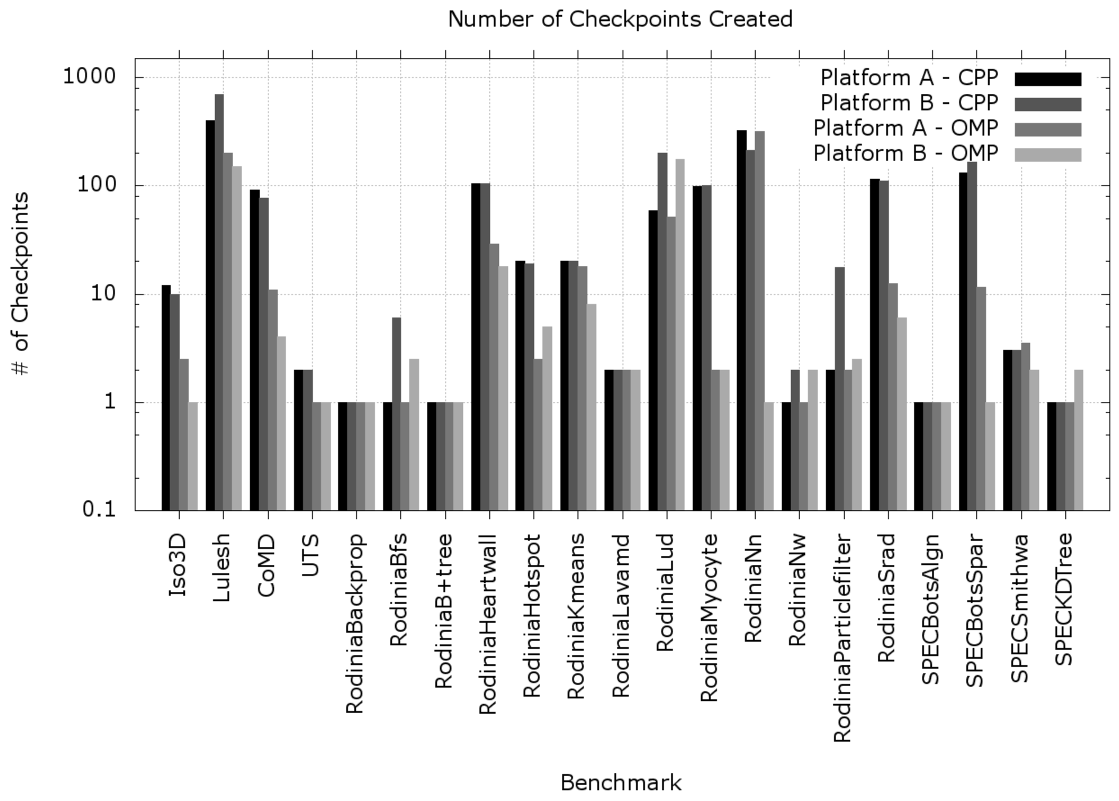

Figure 5.12 : Median number of checkpoints created on Platforms A and B for each benchmark.

one checkpoint, though many produce on the order of tens or hundreds of benchmarks. Note that OpenMP applications tend to produce fewer checkpoints than single-threaded applications as instrumentation is added to track thread state, thereby increasing overheads and leading to more checkpoint throttling.

Checkpoint Efficiency

Checkpoint efficiency is a measure of the size of the checkpoints created for an application, relative to its total size in memory. Figure 5.13 shows the checkpoint efficiencies for all benchmarks on Platforms A and B. 100% efficiency indicates that the size of the application's in-memory state and the size of the checkpoints are the same. For

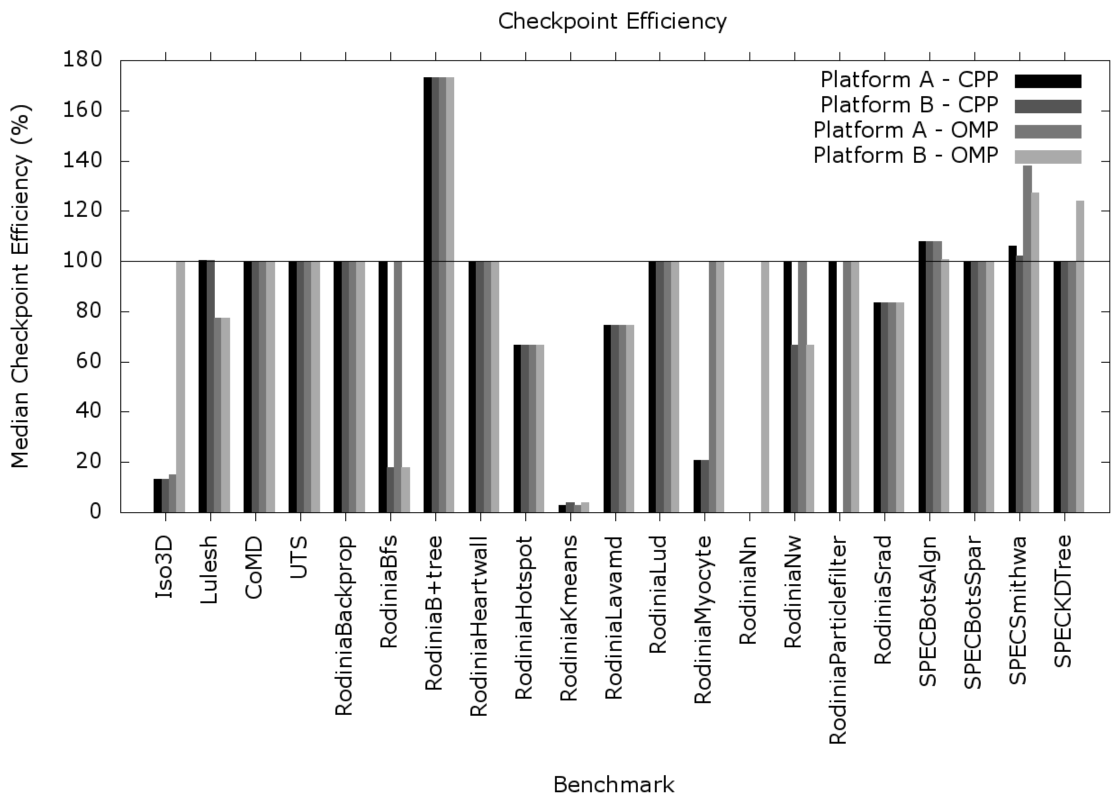

Figure 5.13 : Median checkpoint efficiency on Platforms A and B across all checkpoints created by test runs of all applications.

some applications, we see that the change set tracking and hashing described in Sections 5.1.2 and 5.1.2 successfully reduced the amount of application state that had to be checkpointed. However, it is quite common for applications to regularly touch all application state (e.g. on every time step), so in many cases a checkpoint is a full copy of the running application. In some cases where the application working set is small, the checkpoint is appreciably larger due to CHIMES-specific objects added to the checkpoint. For instance, CHIMES includes alias set information in the checkpoint, which is not considered a part of the running application's working set.

Conclusions

There are many tradeoffs in the design of a checkpointing framework: how comprehensive the checkpointable state is, how much attention is given to efficiency, how much control the user is given over checkpoint creation, the layer in the software stack to implement the framework, etc. In this work we present a novel checkpointing framework that has the following characteristics:

1. Automated checkpointing of stack, heap, and other user-level objects through source code inspection and transformation.
2. A highly efficient, overhead-aware runtime for multi-threaded programs that handles program state tracking and checkpoint creation.
3. Support for user specification of checkpoints.
4. Support for pluggable user functionality in the creation and restoration of checkpoints.
5. A combined compiler and library approach to checkpointing which uses insights gained from the source code to enhance efficiency.

The evaluation in Section 5.1.3 shows that this framework is not only flexible enough to handle checkpointing of real-world scientific applications, but that it does so efficiently and transparently to the user. CHIMES supports real-world, long-running, scientific applications that have large memory footprints, use function pointers, use complex types, and use complex build systems. Not only does CHIMES support them, but CHIMES makes it easier to build and improve them by easing debugging, performance hotspot analysis, and resilient application development.

5.2 Decomposition-Based Performance Prediction

5.2.1 Background

Categorizing Performance Prediction Techniques

The ability to predict the performance of a given kernel on a given architecture has many applications. These predictions can be used to improve device selection on heterogeneous systems by providing the scheduler with completion time estimates across a variety of computational resources. They can also be used to improve the partitioning of a workload across multiple devices, by predicting the relative computational bandwidth of each device. Performance predictions can be used to make frequency and voltage throttling decisions for improved energy efficiency. Performance predictions can even be used in the procurement of new systems by predicting the behavior of existing computational workloads on new architectures.

In this work we focus on predicting the performance of loop-parallel kernels implemented in both OpenMP and CUDA. We assume that these kernels are run in isolation, i.e. they have access to all processing elements in a given processor. We also assume that we have access to all of the source code for the kernel, besides certain math intrinsics (e.g. `sin`, `sqrt`, etc.). The actual use case for these performance predictions is left open so as to not overly constrain the impact of this work, but at a minimum we require that the predictions made be useful for runtime scheduling decisions.

We make no assumptions about the type of data structures accessed from the kernel, nor do we perform any automatic data layout transformations based on the results of performance prediction. That work is considered beyond the scope of this work, though the framework developed in this work could be used to guide automatic data layout optimizations. This work also does not account for special-purpose memories (e.g. GPU constant memory, scratchpad, etc.), but support for that would be a straightforward extension.

Performance predictions can be categorized into three levels of accuracy:

1. **Binary**: Given two execution platforms, a binary performance predictor indicates which is faster. For example, it might predict that a given kernel will run faster on an x86 processor than on a GPU.

2. **Relative**: Given a collection of execution platforms, a relative performance predictor estimates the relative computational bandwidth of each platform relative to the others. For example, it might predict that a given kernel will run $2\times$ faster on a GPU and $4\times$ faster on an FPGA, relative to x86.

3. **Absolute**: Given a single kernel, an absolute performance predictor produces a performance prediction in terms of absolute elapsed time or processor cycles.

On the other hand, performance predictors can be split into four categories: programmer-provided, analytical, simulator-based, and learning-based.

Programmer-provided performance prediction uses hints or estimates from the programmer to determine the time a particular kernel will take on a given architecture. This manually-specified information is then generally used by the runtime system to aid with scheduling decisions. For example, in [114] the programmer is responsible for differentiating between "low" priority tasks which should be chunked together for accelerator execution, and "high" priority tasks which may be executed immediately on the host. In [95], the authors instead phrase performance prediction in terms of a task's "affinity" to a particular architecture. Programmers must provide abstract, relative affinities for each task on each available architecture (e.g. GPU, CPU, FPGA). The scheduler then uses runtime resource availability and task affinities to select the architecture to schedule a task on. Similarly, the OSCAR compiler [115] uses compile-time annotations of cycle estimates for each kernel to select architectures at runtime.

Of course, programmer-provided predictions are prone to human error, not future-proof, generally lack awareness of runtime state (e.g. loop counts), lack a confidence measure, and are not generalizable. It is often difficult to predict the behavior of a kernel across a wide variety of inputs, particularly since past works commonly constrain programmer-provided estimates to a constant. However, programmer-provided predictions are easily tunable and inspectable (i.e. they can be simple to reason about for programmers).

Analytical models, on the other hand, try to model some behavior of the target system or architecture in a closed and human-derived form based on understanding of the architecture. For example, in [116] the authors construct performance and energy predictions functions by inputting sampled performance counters from a kernel into a human-defined performance prediction function. This prediction function is constructed using the developer's understanding of architectural features. Then, the generated predictions are used to select the number of cores to run a parallel region on, and the number of SIMT threads to run on each core.

While analytical models can benefit from human intuition and understanding of hardware characteristics, they share many of the same challenges as programmer-provided prediction. These analytical models are generally hard-coded to a single architecture's features and rely on the creator of the analytical model having a comprehensive understanding of that architecture's behavior. Analytical models are also often too simple to capture the full behavior of a complex multi-tenant system.

On the other hand, **simulator-based performance prediction** uses cycle-accurate or near cycle-accurate hardware simulators to estimate performance [117]. The accuracy of simulator-based techniques can of course far exceed the previous techniques discussed, as the actual behavior of the hardware is being modeled rather than some abstraction of it. However, constructing a cycle-accurate simulator is a major engineering effort, making their creation costly in terms of developer hours. Additionally, because simulators model architectures at a finer granularity, they take longer to produce their predictions.

Finally, **learning-based performance prediction** uses a statistical model applied to a curated set of features to model hardware behavior. Creating a learning-based performance model generally requires the selection of the statistical model to use (e.g. linear regression, logistic regression, support vector machine) and a set of curated features that are relevant to kernel performance. The curated features can be features of the architecture being targeted, of the kernel being executed, of runtime state, or of any other relevant inputs the model developer selects.

Once the features and statistical model are selected, a performance model is then trained using that statistical model and a set of sample, curated input datasets for sample, curated kernels. When it comes time to create a performance prediction, the trained performance model is applied to the same set of curated features.

The work described here builds directly on the work described in [73], a learning-based approach to performance prediction. In [73], the authors train a Support Vector Machine (SVM) to perform a binary performance prediction for a given kernel. This binary performance prediction decides whether to run the given kernel on a POWER CPU or an NVIDIA GPU. The SVM is trained on a feature vector consisting of high-level kernel features, including items like:

1. The loop range of the parallel loop to offload.
2. The number of instructions in each kernel.
3. The percentage of kernel instructions that are memory accesses, arithmetic operations, method calls to math functions, or branch instructions.
4. The number of coalesced, strided, and offset memory accesses.

The SVM is trained offline and evaluated online to decide which device to run a given kernel on. The results presented show that for the selected set of kernels, the accuracy of the binary predictions made can reach 99%. Hence, the work described in [73] is an initial exploration into accurate learning-based techniques for binary performance prediction.

Guiding Principles

Here, we present a learning-based performance prediction model called HYDOSO (HYbrid, Decomposition-based, Offline Sequence aligner for Online performance prediction) that builds on lessons learned in [73]. The development of HYDOSO was guided by the following observations and principles:

1. The set of all kernels is massively diverse, but including kernel features in any execution performance model is crucial for accurately predicting performance of previously unseen kernels. Hence, accurately training a performance model across the space of all kernels is an important but intractable problem. Instead, it may be beneficial to decompose the performance prediction problem to work on smaller blocks of code, and then combine the per-block insights to generate full-kernel predictions.

2. Thanks to instruction pipelining and other latency-hiding mechanisms, the performance of a block of code depends on both its composition as well as the ordering of the instructions it contains.

3. In performance prediction, confidence measures for a given prediction are crucial to deciding whether a prediction should be relied on at runtime. These confidence measures can also guide further offline performance model training.

4. Runtime values which are indeterminable at compile time can drastically affect the performance of kernels (e.g. runtime-defined loop iteration counts).

5. Re-training accurate performance models online without significantly disrupting application performance is a difficult problem. However, spending cycles offline to continually improve performance models is cheap.

Items 4 and 5 above are supported by the approach to performance prediction in [73]. However, items 1, 2, and 3 are insights which require a drastically different approach.

Based on the above guiding principles, we design and implement the HYDOSO performance prediction framework to operate at the basic block level. HYDOSO trains a block-level performance model offline for straight-line sequences of code. This performance model is then combined with run-time state to make accurate online performance predictions. HYDOSO focuses on making accurate binary and relative performance predictions, but Section 5.2.3 also evaluates the accuracy of its absolute performance predictions.

5.2.2 HYDOSO Design and Implementation

The HYDOSO framework consists of two software pipelines: a model training pipeline, and a model-use pipeline. Offline, the model-training pipeline collects performance data for all available architectures across a set of sample kernels and datasets. The model-use pipeline uses the collected performance data to predict kernel performance at runtime from within a "predictive application". A predictive application is one that uses the HYDOSO APIs to query for performance predictions.

For this investigation, our evaluation will focus on performing binary and relative performance predictions on x86 and NVIDIA GPU platforms. However, our discussion includes investigation into how this work could apply to performance prediction for JVM kernels as well.

A HYDOSO Performance Model

In HYDOSO, a "performance model" consists of a large set of short, straightline sequences of instructions with metadata attached to each. Each member of this set is referred to as a "sample" and includes:

1. The sequence of instructions that this sample captures performance information on (e.g. `LOAD,STORE,ADD,...`).

2. The average observed latency of this sequence of instructions, in milliseconds.

3. The observed standard deviation of this sequence of instructions, in milliseconds.
4. The minimum observed latency of this sequence of instructions.
5. The maximum observed latency of this sequence of instructions.
6. The architecture this performance sample is for.
7. The architecture configuration this performance sample is for.
8. The number of observations this performance sample is based on.

This approach differs from past approaches in that it does not build a closed-form representation of the performance of a kernel. Instead, the performance model is a collection of past performance measurements. For the remainder of this discussion, we will refer to this construct as the "sample database". Later sections will describe how this performance model is created, and how it is used to predict performance.

Creating a HYDOSO Sample Database

Creating a HYDOSO Sample Database requires block-level, low overhead sampling of representative kernels on representative datasets. Recall from Section 5.2.1 that we focus this work on composing block-level performance predictions together to perform full-kernel performance prediction, based on the observation that the space of all kernels is much more diverse and more difficult to characterize than the space of all basic blocks.

In this work, we have explored three different ways to sample basic blocks in target kernels: full application sampling, checkpoint-based sampling, and Java Agent-based sampling. Full application sampling and checkpoint-based sampling support performance prediction of native execution on x86 and NVIDIA GPU processors, while Java Agent-based sampling supports performance prediction for the JVM.

All three of these approaches rely on the same fundamental capability: lightweight, transparent insertion of timestamp sampling at instruction-level granularity. For native x86 execution, this capability is supported by reading the RDTSC (or Time Stamp Counter) register. The RDTSC register counts cycles, and offers efficient and low-latency measurement. Sampling of RDTSC is inserted using LLVM. For interpreted x86 execution on the JVM, a Java Agent is instead used to dynamically transform the bytecode of a JVM application at runtime and insert JNI calls to a library which in turn samples the RDTSC register. Finally, for native GPU execution we can insert calls to the CUDA `clock` API using LLVM, which reads a clock register on each GPU SM similar to the RDTSC register.

With the question of capturing low-overhead, high-accuracy timestamps answered, we next need to consider where to insert this sampling. Note that in order to make the performance information captured with these timestamps relevant, we need to minimize the amount of perturbation introduced to the instrumented application both in terms of runtime overheads as well as compile-time interference. As a result, only a single block is instrumented at a time and all instrumentation is inserted as external function calls to minimize interference with the compiler's intra-procedural register allocator. Instrumenting a block implies using the low-overhead methods described previously to take a timestamp at the beginning and end of a target block, and storing the elapsed number of cycles in memory for later output.

In both LLVM- and Java Agent-based sampling, this process begins with parsing the instruction-level representation of the kernel and its callees (either as LLVM bitcode or JVM bytecode) into a directed control flow graph of call-less basic blocks. This process starts by producing a control flow graph of basic blocks for each function from an instruction-level representation. Then, basic blocks in the produced control flow graphs are split at call sites, and edges added from the call site to the entry basic block of the callee and from any return points in the callee to the call site. The end result is an inter-procedural control flow graph where conditional branches, jumps, and function calls are all modeled as edges in the graph. Repeated edges in this graph (i.e. due to loops in the source code) are annotated with either 1) a constant loop count if it can be derived at compile-time, 2) a variable whose runtime value controls how many times this edge is repeated, or 3) a special annotation indicating that we do not have a way to estimate the number of times this edge is repeated.

Then, given a sample kernel within a sample application and a sample dataset to run it on, we selectively instrument a single block at a time in that kernel's inter-procedural control flow graph. The exact implementation of this depends on whether we are performing full application sampling, checkpoint-based sampling, or Java Agent-based sampling.

In full application sampling, the original application source code is passed to LLVM for insertion of RDTSC- or `clock`-based instrumentation. No other transformations are applied. To collect samples the full application must be repeatedly transformed and run on the sample dataset, once for each call-less basic block in each kernel we are training. This process can be time-intensive if the target kernel makes up a small fraction of overall application execution time.

In checkpoint-based sampling, collecting performance samples is a multi-step process:

1. The original application has checkpoint creation points inserted prior to the start of each parallel kernel. These checkpoint creation points capture the state of the program prior to that parallel kernel.

2. The checkpointed application is run once on a sample dataset, producing a set of application checkpoints.

3. A "resume program" is auto-generated for each checkpoint which reads a checkpoint and re-executes that parallel region on the application state stored in that checkpoint.

4. Then, the same LLVM-based transformation techniques used in full application sampling are used to instead instrument these resume programs, followed by

re-executing each checkpoint and collecting performance samples from those re-executions.

Semantically, full application sampling and checkpoint-based sampling produce the same output. However, checkpoint-based sampling does not require repeatedly rerunning the full application and so can speed up the model training process.

On the other hand, a Java Agent is a small applet that is pre-loaded by the JVM before starting execution of a JVM program. The bytecode of the loaded user program is passed to the Java Agent, which it can then transform prior to the JVM executing it. Using a Java Agent we are able to instrument a JVM kernel at instruction-level granularity to insert reads of the RDTSC register. However, this process has the same downside as full application sampling in that it requires full application execution and is not checkpoint-based.

Regardless of whether full application sampling, checkpoint-based sampling, or Java Agent-based sampling is used, the output of this stage is a list of triples, where each triple contains:

1. The instructions in an instrumented call-less basic block.
2. A list of samples of the latency of that call-less basic block.
3. The platform that those samples were collected on (e.g. x86, NVIDIA GPU, JVM).

The collected latency samples are adjusted to account for the expected overhead incurred by the timestamp collection. We do this by empirically determining the number of cycles measured for an empty block using our timestamping methods, and then subtract this number of cycles from each latency sample. For reference, on our evaluation platform we measured an overhead of 32 cycles on our x86 evaluation platform and 71 cycles on our GPU evaluation platform.

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
  #pragma hydoso kernel omp omp_kernel_lbl
  ...
}
```

Figure 5.14 : An example HYDOSO directive added to an OpenMP parallel region, indicating that the user may request performance predictions for this kernel.

```
__global__ void kernel(...) {
#pragma hydoso kernel cuda cuda_kernel_lbl
}
```

Figure 5.15 : An example HYDOSO directive added to a CUDA kernel, indicating that the user may request performance predictions for this kernel.

We aggregate these triples into a single HYDOSO Sample Database (described in Section 5.2.2) by merging information for identical blocks on the same architecture. This HYDOSO Sample Database then becomes our "performance model" for the tested platforms.

User Interface for Predictive Applications

As defined earlier, a predictive application is one that "uses the HYDOSO APIs to query for performance predictions". In Section 5.2.2, we covered what a HYDOSO performance model contains and how it is generated. The first step to using that model in a predictive application is to have the user specify the kernels whose performance they wish to predict.

This is done by applying compiler directives to the body of the kernel. For example, marking a target OpenMP kernel is shown in Figure 5.14.

On the other hand, marking a CUDA kernel is shown in Figure 5.15.

In each of the above code snippets, the programmer needs to supply the name of the platform they will be executing on (e.g. `omp`, `cuda`) as well as a unique label for

```c
// A data structure for capturing the predicted performance
// of a given kernel
typedef struct {
  // Iterations per ms
  double predicted_rate;
  // Confidence measure
  double score;
  // Information on platform configuration, e.g. how many
  // threads per CUDA block.
  device_thread_config config;
} hydoso_prediction;

// A data structure for storing runtime application state
// in, to be passed to the HYDOSO runtime
typedef struct {
  // List of names of each variable being passed down
  const char **var_names;
  // Values of each variable, for now limited to
  // integer-typed variables
  int *var_values;
  // Number of variables
  unsigned nvars;
} hydoso_app_vars;

hydoso_prediction hydoso_predict(const char *kernel_name,
        hydoso_app_vars *vars);
```

Figure 5.16 : The signature of `hydoso_predict`, which is used to query for new performance predictions in HYDOSO.

this kernel (e.g. `omp_kernel_lbl`, `cuda_kernel_lbl`).

With the target kernels marked the user is able to query the HYDOSO runtime for performance predictions at runtime. This is done using a function call, to which the user passes the name of the kernel they would like to predict performance for as well as runtime state that HYDOSO can use to refine the inter-procedural control flow graph for that kernel. This API is called `hydoso_predict` and is depicted in Figure 5.16.

`hydoso_predict` then returns an estimate of the rate at which the target loop-

```
hydoso_prediction pred_omp = hydoso_predict("omp_kernel", ↩
    NULL);
hydoso_prediction pred_cuda = hydoso_predict("cuda_kernel",↩
     NULL);
// A relative performance prediction
double relative_perf = pred_cuda.predicted_rate / pred_omp.↩
    predicted_rate;
```

Figure 5.17 : An example usage of `hydoso_predict` to produce a relative performance prediction between two kernels.

```
hydoso_prediction pred_omp = hydoso_predict("omp_kernel", ↩
    NULL);
hydoso_prediction pred_cuda = hydoso_predict("cuda_kernel",↩
     NULL);
// A binary performance prediction
bool is_omp_faster = pred_omp.predicted_rate > pred_cuda.↩
    predicted_rate;
```

Figure 5.18 : An example usage of `hydoso_predict` to produce a binary performance prediction.

parallel kernel will run in terms of iterations per millisecond, as well as a confidence score for that prediction. The HYDOSO framework does not assume anything about how these predictions are used by the programmer.

The value returned by `hydoso_predict` is an absolute performance prediction. It can be used to produced relative and binary performance predictions. Figure 5.17 depicts how a relative performance prediction is calculated by dividing one absolute prediction by another.

Likewise, Figure 5.18 shows how a binary performance prediction can be made by comparing one absolute prediction to another.

Compile-Time Application Transformations for Predictive Applications

After a user has created a predictive application using the APIs described in Section 5.2.2, this application is analyzed (but not transformed) by the HYDOSO framework.

First, the same techniques that were described in Section 5.2.2 are used to construct an interprocedural control flow graph for each of the annotated kernels.

Second, for each call-less basic block in the generated interprocedural CFGs we use algorithms from substring matching/genome alignment to align the call-less basic block to the sampled blocks in a Sample Database, looking for the match that produces the highest score based on the ordering and composition of instructions in each block. We record both the best match, and its confidence score. In particular, the current implementation uses the Needleman-Wunsch algorithm [118].

Third, each interprocedural CFG is converted into a "latency graph" by assigning latency estimates to each block. Currently, the latency estimate of a block is configurable and may be set to be the mean, minimum, or maximum of the latency measurements made for the highest scoring aligned block from the Sample Database. Hence, our original graph of straightline sequences of instructions becomes a graph with the same structure, but containing estimated latencies for each node.

The latency graphs for each annotated kernel are then stored on disk.

Application compilation takes place normally and without any other HYDOSO passes.

Generating Instruction Lists

As described above, HYDOSO relies on accurately aligning one instruction list to another for its performance predictions. Instruction lists are generated from LLVM bitcode or JVM bytecode through a straightforward one-to-one mapping of instructions to HYDOSO's instruction representation. For example, the LLVM bitcode sequence in Figure 5.19 would result in a HYDOSO instruction sequence of

```
%div = fdiv double 1.000000e+00, %conv
%1 = load i32, i32* %x.addr, align 4
%idxprom = sext i32 %1 to i64
%weights = getelementptr inbounds %class.openmp, %class.↩
    openmp* %this1, i32 0, i32 2
%2 = load double*, double** %weights, align 8
```

Figure 5.19 : An example instruction sequence.

`FDIV,LOAD,CAST,GEP,LOAD`.

However, this alignment process makes an implicit assumption that instructions with the same representation in HYDOSO's instruction set also have similar latencies. This is true for many instructions, but may not be the case for memory accesses. A given load or store operation might hit anywhere in the cache hierarchy on a given architecture, and the resulting latency can change by orders of magnitude as a result.

To address this, we extend the conversion of LLVM bitcode to HYDOSO's instruction set by analyzing the access patterns for any memory access and adding this information to the instruction stream. With this added analysis, the generated instruction sequence for the LLVM bitcode in Figure 5.19 instead becomes `FDIV,STACKLOAD,CAST,GEP,COALESCEDLOAD`. A given access pattern refers to the distribution of offsets from an identical base address across all threads for a single instance of that instruction. A single instance of an instruction is defined by a single instruction in the generated instructions for a kernel, as well as a unique point in the iteration space of any loop that contains that instruction. We currently support differentiating between five different types of access patterns:

1. **Broadcast**: A broadcast access pattern guarantees that for a given instance of a given instruction executed by any thread, that access will always be to the same memory address.

2. **Stack**: A stack access references a stack-allocated variable, which is likely to be in registers or some other thread-local memory.

```
%base = load i32*, i32** %ptr.addr
%address = getelementptr inbounds i32, i32* %base, %i
%val = load i32, i32* %address
```

Figure 5.20 : An example pointer offset calculation.

3. **Strided**: A strided access implies that neighboring threads are accessing memory at a constant stride relative to each other. For example. thread 0's access starts at byte 0, thread 1's access starts at byte 20, thread 2's access starts at byte 40, etc. This pattern can be caused by intra-thread looping over a chunk of an array.

4. **Coalesced**: A coalesced access pattern is a special case of strided where the accesses by all threads are to neighboring elements in a primitive array.

5. **Random**: A random access pattern indicates an access pattern that our framework was unable to analyze. For example, if the offset of an access is determined by a value loaded from memory it would be labeled as Random.

These access patterns are constructed recursively by applying mathematical operations on top of one another in the same sequence they are used in the original instruction sequence. For example, Figure 5.20 shows a common instruction pattern in LLVM bitcode for computing the address of the `i`th element in an array, where `i` is the index of the current iteration of a parallel loop.

The value loaded by the first `load` in Figure 5.20 would be labeled a Broadcast, assuming that the HYDOSO framework was able to analyze the value loaded from `%ptr.addr` and guarantee it is the same base address across all threads. Then, the `getelementptr` instruction is used to calculate an offset address from which is purely determined by the parallel iterator, `%i`. Hence, this is the addition of a Coalesced access pattern to a Broadcast pattern, which in turn produces a Coalesced access pattern. This analysis would result in the final `load` in Figure 5.20 being labeled a

Coalesced load.

HYDOSO at Runtime

At runtime, the user must set a `HYDOSO_GRAPHS` environment variable to be a colon-separated list of the latency graphs generated at compile-time. These graphs are loaded and parsed by the HYDOSO runtime on initialization of a predictive application and stored in a dictionary, mapping from user-provided kernel name to latency graph.

As described in Section 5.2.2, the only HYDOSO API is `hydoso_predict`, which accepts a kernel label and runtime variable state and returns a prediction for the performance of that loop-parallel kernel, in iterations per millisecond.

When a call to `hydoso_predict` is made at runtime, the HYDOSO runtime will first look up the latency graph for the target kernel. It will produce a single-threaded latency estimate for this kernel by simulating the latency graph and producing a latency estimate for the full graph.

Given a node in the latency graph, the latency estimate for the subgraph from that node to kernel termination is calculated by summing a latency estimate for the current node with a latency estimate computed across its successor nodes.

The latency estimate of a single node is tuneable at runtime, and can be either the mean, maximum, or minimum of the latency measurements for the block that node was aligned to in the Sample Database.

The latency estimate calculated across successor nodes in the latency graph is also tuneable, and depends on the type of successors the current block has. There are four different types of successor relations:

1. **Direct**: The current block has a single successor which it jumps to directly.

2. **Call**: The current block has a single successor which it jumps to directly, but which is in another function. This relation represents a function call in the original source code.

3. **Branch**: The current block has more than one successor which it may conditionally branch to.

4. **Repeated**: The current block is a loop-ending conditional block which is evaluated `L` times for the execution of a single instance of the loop. The first `L-1` times it takes one branch. On the last evaluation, the conditional block branches to the other branch.

Given a type of successor relation, the latency estimation for successors can be tuned in the following ways:

1. For **Direct** and **Call** successor relations, the latency estimate for successor blocks is simply the latency estimate for the single target block (which is recursively defined based on it and its own successors).

2. For a **Branch** successor relation, the latency estimate over all successors can be calculated as either the mean, maximum, minimum, or sum of the latency estimates across all successors.

3. For a **Repeated** successor relation, the latency estimate over all successors is the latency estimate for the body of the loop, multiplied by a loop repeat estimate, plus the latency estimate for the non-repeated branch.

The loop repeat estimate is also tuneable. If the loop repeat was a constant determinable at compile time during the inter-procedural CFG generation, the constant is used. If it was dependent on a runtime value, we assume that value was passed in the `vars` parameter to `hydoso_predict` and its runtime value is used. Otherwise, a tuneable “repeat guess” is used which defaults to ten.

Note that using a repeat guess implicitly sacrifices the ability to reliably estimate absolute performance, and means that the HYDOSO framework will only be able to estimate relative performance. Hence, an accurate combination of compile-time and run-time analysis is necessary to produce accurate full-kernel latency estimates, but that even without that combination accurate relative performance estimates are still possible.

For a single call to `hydoso_predict`, a performance estimate is produced for the target kernel on all architecture configurations the HYDOSO performance model was trained on. For OpenMP kernels the architecture configuration consists of the number of threads used and the number of SIMT threads assigned to each core. For CUDA programs, the architecture configuration consists of the number of threads to create per thread block. Only the architecture configuration with the lowest estimated latency/highest estimated execution rate is returned to the user for the target architecture.

Recall that for each call-less basic block in the latency graph, we also computed a confidence score using the Needleman-Wunsch algorithm which indicates how accurately the target block aligned to a block in our Sample Database. To produce a whole-kernel confidence score for the user, we return a weighted average of the confidence scores for each block, weighted by the number of instructions in each block.

5.2.3 HYDOSO Performance Evaluation

Evaluation of the accuracy of the HYDOSO framework was carried out on the Titan machine at ORNL [87]. Titan is a Cray XK7 with a 16-core AMD CPU, an NVIDIA K20X, and 32GB of DRAM in each node. We focus our use of HYDOSO on estimating the performance of a variety of kernels on a variety of datasets on the AMD CPU and NVIDIA GPU.

Our evaluation uses the benchmarks and datasets shown in Table 5.2.

We focus on training performance models for only the platform configurations shown in Table 5.3, using checkpoint-based sampling to train the Sample Database.

We evaluate the accuracy of the HYDOSO framework in terms of its binary, relative, and absolute performance prediction accuracy. All benchmarks are evaluated

Suite	**Benchmark**	**# Datasets**	**# Kernels**
Rodinia [62]	bfs	2	2
	b+tree	1	2
	cfd	1	5
	hotspot	2	1
	hotspot3D	2	1
	lud	2	2
	nw	2	2
	particlefilter	2	10
	pathfinder	2	1
	srad	2	2
Polybench [119]	doitgen	1	2
	gemm	1	1
	gesummv	1	1
	jacobi-1d-imper	1	2
Parsec [120]	blackscholes	1	1
Parboil [121]	mri-q	1	2
NAS [122]	BT	1	35
	CG	1	8

Table 5.2 : Benchmarks

Platform	**Configuration**
OpenMP	1 thread per core
CUDA	64 threads per block
	128 threads per block
	256 threads per block
	512 threads per block

Table 5.3 : Tested platform configurations

HYDOSO Tunable	**Values**
OpenMP Block Latency Estimation	MIN, MAX, MEAN
CUDA Block Latency Estimation	MIN, MAX, MEAN
OpenMP Divergence Estimation	MIN, MAX, MEAN, SUM
CUDA Divergence Estimation	MIN, MAX, MEAN, SUM
Repeat Guess	10, 100

Table 5.4 : HYDOSO Tunables

using all possible permutations of the HYDOSO tunables shown in Table 5.4. We refer to a single, unique instance of these tunable values as an estimation strategy in that it guides how the runtime performance predictor estimates unknown values or branches. There are 288 unique estimation strategies possible using the tunables in Table 5.4.

We also evaluate the accuracy of the HYDOSO framework using three strategies for generating the HYDOSO Sample Database:

1. **Full**: In Full training, we generate a Sample Database that contains performance information from all of the benchmarks and kernels in Table 5.2. Hence, when we use this Sample Database to make performance predictions on those same kernels, it includes performance data from the kernel being predicted as well as many other kernels.

2. **Singleton**: In Singleton training, a Sample Database is generated for each benchmark in Table 5.2 that contains only the performance data from that benchmark. We then use that Sample Database to only make predictions for that same benchmark. As a result, we expect Singleton Sample Database to be quick to train, to be highly specialized for its target benchmark, but to not generalize well to other benchmarks.

3. **Exclusive**: In Exclusive training, a Sample Database is generated for each benchmark that contains performance data from all benchmarks in Table 5.2

Average Block Length	97.29 instructions
Minimum Block Length	1 instruction
Maximum Block Length	3,364 instructions
Average Block Latency for OpenMP	161.77 cycles
Minimum Block Latency for OpenMP	2.29 cycles
Maximum Block Latency for OpenMP	11,372.53 cycles
Mean % STD of Block Latency for OpenMP	31.11%
Average Block Latency for CUDA	3,037.42 cycles
Minimum Block Latency for CUDA	59.69 cycles
Maximum Block Latency for CUDA	138,913.55 cycles
Mean % STD of Block Latency for CUDA	21.44%

Table 5.5 : Statistics on the blocks contained in the Sample Database used in this evaluation

except for that benchmark. Hence, Exclusive training should demonstrate that HYDOSO can effectively generate performance predictions even for unseen kernels.

Finally, we compare the binary accuracy of the HYDOSO framework to the SVM-based techniques from [73] that inspired this work.

Analysis of the Sample Database

We start by analyzing the block-level data generated from all kernels for our Full Sample Database. The Full database, including block-level performance information from all benchmarks on all datasets, consists of information on 261 unique blocks backed by 2,837,972,766 unique block latency samples. Table 5.5 lists various statistics on the blocks contained in this table.

Binary Performance Estimate Accuracy

Next we consider how different estimation strategies perform in predicting the correct platform configuration to target using a Full Sample Database. Out of the total of 288 possible estimation strategies, eleven achieve a maximum of 86.21% accuracy

Estimation Class	**Repeat Guess**	
	10	**100**
Best	63.64%	36.36%
Worst	50.00%	50.00%

Table 5.6 : Percentage of the best and worst estimation strategies which used repeat guesses of 10 or 100.

across all kernels and datasets in choosing the correct fastest device. 34 strategies achieve better than 75% accuracy. The ten worst estimation strategies all achieve an accuracy of 22.41%. This demonstrates that the selection of an accurate estimation strategy is important, and does impact the accuracy of the performance predictions made.

To understand in more detail how HYDOOSO's estimation strategies affect binary prediction accuracy, we look at trends in the eleven best estimation strategies (which each achieved 86.21% accuracy) and the ten worst estimation strategies (which each achieved 22.41%). Tables 5.6, 5.7, 5.8, and 5.9 shows these trends for the repeat guess, block latency estimation, OpenMP divergence estimation, and CUDA divergence estimation tunables.

It is important to point out that the correct estimation strategy is both architecture and application dependent. For example, an application demonstrating highly divergent threads on a GPU might benefit from using SUM as its divergence estimation strategy so as to model the execution of multiple branches by a single divergent warp. More regular applications might prefer MIN, MAX, or MEAN. Hence, we do not expect a single strategy to perform perfectly across all kernels. It is still interesting to study trends in the accuracy of each strategy.

Table 5.6 is relatively uninteresting, and suggests that the choice of repeat guess is unimportant in binary performance prediction. This is primarily because the HYDOSO framework is able to compute accurate runtime loop counts for most kernels using either compile-time constants or runtime values.

Estimation Class	**OpenMP Block Latency**			**CUDA Block Latency**		
	MIN	**MAX**	**MEAN**	**MIN**	**MAX**	**MEAN**
Best	0.00%	100.00%	0.00%	100.00%	0.00%	0.00%
Worst	100.00%	0.00%	0.00%	0.00%	100.00%	0.00%

Table 5.7 : Percentage of the best and worst estimation strategies which used block latency estimation strategies of MIN, MAX, and MEAN.

Estimation Class	**OpenMP Divergence**			
	MIN	**MAX**	**MEAN**	**SUM**
Best	18.19%	27.27%	27.27%	27.27%
Worst	60.00%	20.00%	20.00%	0.00%

Table 5.8 : Percentage of the best and worst estimation strategies which used divergence estimation strategies of MIN, MAX, MEAN, and SUM for OpenMP.

Table 5.7 shows that block latency estimation strategy is a major differentiator between accurate and inaccurate binary performance prediction strategies. For both the best and worst binary performance predictors, all predictors in the same class agree on the OpenMP Block Latency and CUDA Block Latency estimation strategy to use.

Relative Performance Estimate Accuracy

Next, we consider how accurately the HYDOSO framework is able to estimate the relative computational performance of multiple platforms for a given kernel and dataset. A HYDOSO relative performance prediction between two devices is determined by

Estimation Class	**CUDA Divergence**			
	MIN	**MAX**	**MEAN**	**SUM**
Best	72.73%	0.00%	27.27%	0.00%
Worst	0.00%	20.00%	20.00%	60.00%

Table 5.9 : Percentage of the best and worst estimation strategies which used divergence estimation strategies of MIN, MAX, MEAN, and SUM for CUDA.

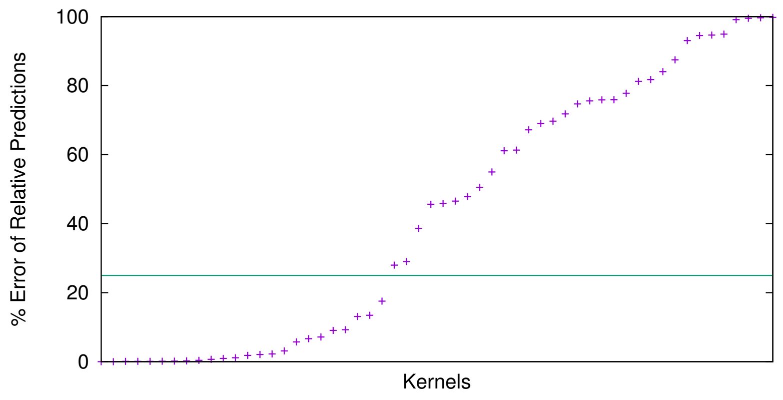

Figure 5.21 : The accuracy of relative performance predictions from HYDOSO, plotted on a linear scale and with a y-axis limited to 100.00%. The horizontal line marks 25%.

dividing the predicted execution rate of one by the predicted execution rate of the other. Like in Section 5.2.3, this section evaluates the accuracy of HYDOSO across all kernels.

Figures 5.21 and 5.22 plot the accuracy of HYDOSO's relative performance predictions across all kernels using the Full Sample Database, sorted by accuracy. Figures 5.21 and 5.22 plot the same data, but Figure 5.21 has a linear y-axis limited to 100.00% in order to highlight the accurate data points. Figure 5.22 has a log-scale y-axis and is not limited. For ∼42% of all tested kernels, HYDOSO achieves an accuracy of 10% or better in its prediction of relative performance for CPU and GPU. For ∼50% of all tested kernels, an accuracy of 25% or better is achieved.

However, Figure 5.22 also shows that HYDOSO's accuracy at relative predictions is poor on some kernels. To explain why, we studied the three kernels which achieved the lowest accuracy and how HYDOSO's alignment approach failed to properly account for their performance characteristics. For simplicity, we refer to these kernels as A, B, and C with A being the least accurate.

The majority of the computation performed in Kernel A is included in a call to `cosf` followed by a call to `sinf`, and so predicting the latency of these operations

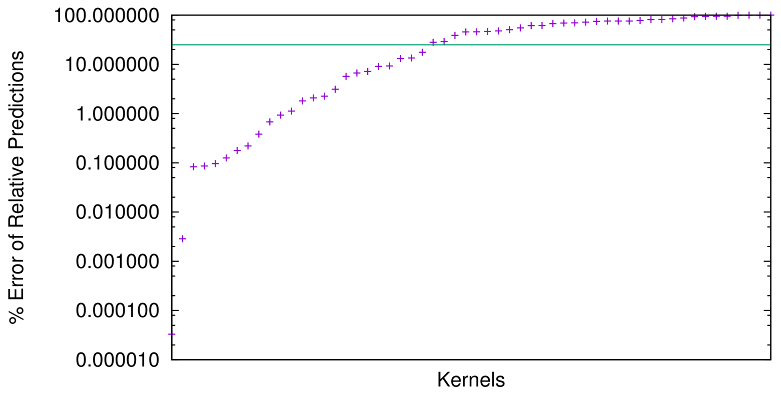

Figure 5.22 : The accuracy of relative performance predictions from HYDOSO, plotted on a log scale. The horizontal line marks 25%.

at runtime is key to accurate performance predictions. However, we believe that the LLVM representation of these trigonometric functions that we are predicting on is different from the version actually being compiled and executed. Recall that we use the LLVM infrastructure to analyze the kernels, but we must use NVIDIA's proprietary compiler toolchain (`nvcc`) to generate executable programs. It appears that the implementations of these trigonometric functions significantly differ between the two infrastructures. Indeed, as expected NVIDIA's implementation appears to be much more efficient as we are predicting a processing rate of 1.542152 iterations/ms but observe 23.749224 iterations/ms. This reveals a drawback to the current implementation, in that there are cases where the LLVM representation of a program and the actual instructions executed may differ significantly.

Kernel B, on the other hand, is a heavily divergent with nested conditions in a short-lived kernel. The control flow of a single parallel iteration is data-dependent, and so on GPUs we can expect unpredictable performance as warps diverge to differing degrees. Indeed, looking closer at the predicted performance for CUDA we observe that on a given kernel execution HYDOSO predicted a processing rate of 258,854.27 iterations/ms but observed 596,623,113.18 iterations/ms, a large error in GPU predicted performance. However, for OpenMP running on an architecture that is more suited to divergent threads and irregular control flow the predictions are much more accurate. HYDOSO predicts 434,567.90 iter/ms for OpenMP on Kernel B and observes 412,225.50 iterations/ms.

Finally, the bulk of computation in Kernel C is contained in a tight loop which repeats thousands of times for each parallel iteration. Today, HYDOSO does not explicitly account for the cost of branch instructions. For kernels where many branch instructions are executed and account for a significant amount of execution time, larger prediction errors can be observed.

Absolute Performance Estimate Accuracy

The most difficult form of performance prediction is absolute: cycle- or second-accurate estimation of the performance of kernels. To date, only programmer-provided and simulator-based performance predictors have been successfully used to produce accurate absolute performance predictions.

Figures 5.23 and 5.24 plot the percent error in absolute performance predictions for OpenMP and CUDA using a Full Sample Database, relative to the measured performance of each parallel region. As described in Section 5.2.2 HYDOSO absolute performance predictions are made in terms of iterations per millisecond.

Figure 5.23 clearly demonstrates that while HYDOSO is able to achieve accurate absolute predictions on a small number of kernels (~6% of all CUDA kernels and ~16% of all OpenMP kernels show less than 25% error), for the vast majority HYDOSO is unable to produce cycle-accurate performance predictions.

Accuracy Changes With a Subsetted Sample Database

The experiments described in Sections 5.2.3, 5.2.3, and 5.2.3 were run with the Full Sample Database, which included samples from the same kernels as were being evaluated. In this section, we explore how removing samples collected from the same application being evaluated affects the accuracy of performance predictions (Exclu-

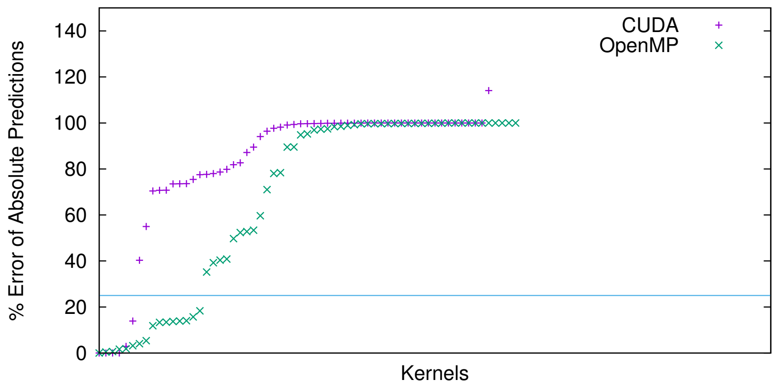

Figure 5.23 : The accuracy of absolute performance predictions from HYDOSO plotted on a linear y-axis. The horizontal line marks 25%.

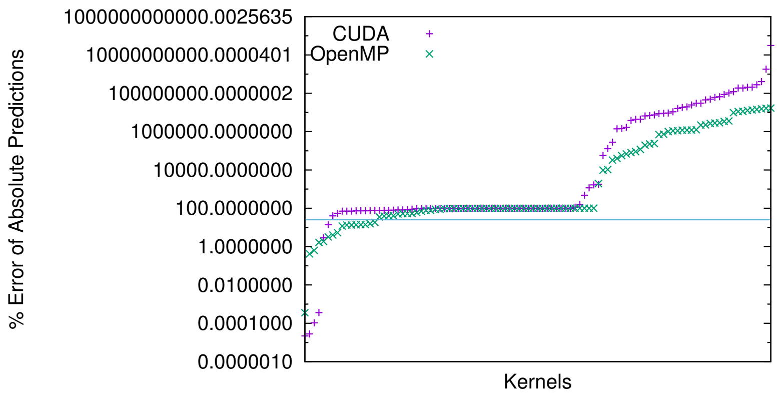

Figure 5.24 : The accuracy of absolute performance predictions from HYDOSO plotted on a logscale y-axis. The horizontal line marks 25%.

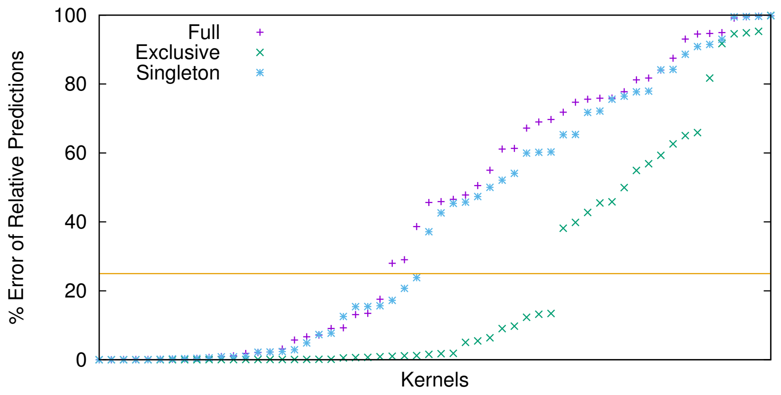

Figure 5.25 : The accuracy of relative performance predictions from HYDOSO when using a Full, Exclusive, or Singleton Database.

sive Sample Database). We also investigate how using a database trained only with samples from the same application performs (Singleton Sample Database). In both comparisons, we focus on any change in the relative performance predictions made, as first described in Section 5.2.3.

Figure 5.25 plots the percent error in relative performance predictions for tests run with the a Full, Exclusive, and Singleton Sample Database. For Exclusive tests, applications are run with databases that are not trained with any latency samples taken from that application. For Singleton tests, applications are run with databases that include only latency samples taken from that same application. Figure 5.25 demonstrates that the least accurate method for performing relative performance predictions is to use a Singleton Sample Database and that the most accurate is to use an Exclusive Database.

From Figure 5.25 we can conclude two things. First, that one of the benefits of a block-centric approach to performance characterization is a reduced search space in terms of the instruction sequences that must be trained. As a result, the relative performance of unseen applications is still predictable when using an Exclusive Sample Database. Indeed, it appears that it is more important to carefully curate your kernels than it is to train on the exact kernels you will be testing on.

Second, as we expect, training a performance model for a single application with only that application can also be accurate, but that the reduced number of samples make it less preferrable. With a smaller sample set (even when it is a more focused sample set), small errors in data collection yield large mispredictions during runtime. In general, a Singleton Sample Database performs a worse characterization of the behavior of hardware.

Comparison to Past Work

In [73], the authors trained a Support Vector Machine (SVM) to perform binary performance predictions based on the composition of a kernel. We reproduce these results on the kernels and datasets used in this work, and compare the accuracy of HYDOSO to the SVM-based approach.

In particular, we train an SVM on the following features of each kernel:

1. Number of parallel loop iterations
2. Total instructions
3. Percentage of memory instructions
4. Percentage of arithmetic instructions (e.g. divide, add, subtract, etc)
5. Percentage of special math operations (e.g. cos, sin, tan, sqrt, pow)
6. Percentage of branches performed
7. Percentage of other instructions
8. Percentage of memory accesses that are coalesced memory accesses
9. Percentage of memory accesses that are strided memory accesses
10. Percentage of memory accesses that are other accesses

We then re-apply that SVM to the experimental data to see if it accurately predicts the faster device based on the observed performance.

In our tests, using the above features to train an SVM yielded 91.5% accuracy in binary performance predictions when choosing between OpenMP and CUDA. These results mirror the results presented in [73] and the binary performance prediction accuracy achieved by HYDOSO in Section 5.2.3. However, note that the approach in [73] is limited to providing binary performance predictions while HYDOSO supports binary, relative, and absolute performance predictions.

5.2.4 HYDOSO Discussion and Future Work

Section 5.2.3 demonstrated that HYDOSO is able to achieve similar accuracy for binary performance predictions to previous work [73] while also offering accurate predictions of relative performance between multiple devices. However, Section 5.2.3 also demonstrates that the HYDOSO framework is not achieving accurate absolute performance predictions, but that doing so may be possible.

One approach to improving HYDOSO's accuracy on unseen blocks would be through improvements to the block alignment process. In particular, today the alignment process uses Needleman-Wunsch which decomposes to scoring alignments of individual instructions. In the current implementation alignments of identical instructions are given a score of 10. Otherwise, the alignment of two instructions is given a score of -5. Experimenting with changing these scores, or customizing scores based on the two mismatched instructions being aligned might produce more meaningful alignments for unseen blocks. For example, we might configure these alignment scores to encourage arithmetic instructions to be aligned with other arithmetic instructions (e.g. a subtraction aligned to an addition) rather than penalizing all mis-alignments equally. Intuitively, this makes sense as these instructions would use similar hardware resources.

HYDOSO also does not explicitly account for the cost of branches, as latency measurements only include the body of each basic block. Again, this could introduce errors to the absolute performance predictions. Similar to accounting for instrumentation overheads, adding a training stage that uses a micro-benchmark to measure the latency cost of various branch instructions and then including this information in the predictions made would address this inaccuracy.

In the description of this work we were careful to not constrain the use of HYDOSO to any specific application or use case, and only focus on the accuracy of the generated performance predictions. However, in future work it would be important to evaluate the efficacy of the HYDOSO framework by integrating it with a higher-level compilation framework. For example, we are currently considering two use cases. First, HYDOSO could be integrated with the OSCAR compiler to replace the programmer-provided performance predictions [115] it uses for scheduling with auto-generated performance predictions. Second, HYDOSO could be integrated with a polyhedral framework for automatic loop and data layout transformation, using HYDOSO performance predictions as a cost function to guide automatic code optimization.

5.3 Related Work

5.3.1 Related Work to Checkpointing

Arguably the most well-known tool for checkpointing is Berkeley Lab's Linux Checkpoint/Restart tool (BLCR) [123]. In [123], the authors of BLCR present three design choices for every checkpointing system: user-level vs. kernel-level, the amount of user- and kernel-level state that can be checkpointed, and the level of integration with other components in the platform (e.g. MPI). BLCR uses kernel-level checkpointing to support pausing and resuming of MPI applications. BLCR is implemented as a Linux kernel module and supports checkpointing a wide range of user-level and kernel application state. BLCR uses custom callbacks to enable integration with other tools or libraries. BLCR takes a stop-the-world approach to checkpointing, pausing all threads until a checkpoint is fully persisted to disk.

DMTCP[124] takes a similarly low-level approach to checkpointing. Instead of adding a kernel module, DMTCP uses the `fork` and `abort` system calls to create a core dump of application state that can be restored from. To support checkpointing of additional program state not captured in this core dump (e.g. file descriptors), DMTCP wraps system calls as well and tracks their usage. Like BLCR, DMTCP dumps all application state with each checkpoint, leading to multi-second checkpoint times for applications with working sets in the megabytes[124].

IGOR[125] uses dirty page tracking to reduce the size of checkpoints by only checkpointing heap regions that have been modified. An application image is constructed from the pages written to disk to enable resume of the application. On resume, the user can select a point to resume from and IGOR will restore from the preceding checkpoint before using interpreted execution to move program state to the desired point in the program.

The work presented in [126] and [127] is the most similar to our work. The authors use a combined compile- and run-time approach to checkpoint OpenMP applications. At compile-time, the application code is analyzed to determine which arrays have been "dirtied" since the last checkpoint and need to be checkpointed. This information is propagated inter-procedurally at compile time. This analysis allows checkpointing of arrays to be started early, immediately after the last write to an array prior to a checkpoint being taken.

Power-Check [128] focuses on limiting the impact of checkpointing on power consumption. Assuming a checkpoint model that requires global synchronization and quiescing of application threads, they note that checkpointing periods are I/O intensive but not compute-intensive, offering the chance for power throttling techniques to be effective. They design an energy-aware I/O subsystem that can either sit under other checkpointing libraries (e.g. BLCR, DMTCP, CHIMES) or be used directly for application-specific checkpointing. In our work, we do not consider the opportunities for energy saving during checkpointing as our CHIMES system tries to overlap I/O intensive checkpointing with compute-intensive application execution, limiting the opportunities to throttle power without significantly degrading application performance.

Each of these checkpointing implementations has contributed to the state-of-the-art. However, each is limited in how it satisfies the motivation in Section 5.1.1. All five previous works lack in transparency: checkpoints are mostly opaque containers whose contents are not easily mapped back to developer-visible constructs such as variables or functions.

BLCR, DMTCP, IGOR, and Power-Check pause all running threads in an application when creating a checkpoint and do not continue until the full checkpoint is flushed to disk, increasing overheads. The lack of source code analysis in these approaches exacerbates the efficiency problem even further. From BLCR [123], "large user applications often already do their own checkpointing for fault tolerance, and can often do it much more efficiently than an automated checkpoint system can, since they know exactly which parts of their application need to be saved and which can be discarded or regenerated". Only by analyzing application source code can we gain the insights necessary for efficient checkpointing.

The work in BLCR, DMTCP, IGOR, and Power-Check are also similar in that each is closely tied to the underlying platform. For example, BLCR is built as a Linux kernel module. While a kernel-level approach allows a checkpointing framework to manage state that user-level approaches do not have access to, this limits flexibility for future platforms and restricts the environments it can be deployed to. The compiler-based approach taken in [126] and in our work is more flexible and platform agnostic, tied only to the semantics of the programming model.

The work in [126] is also limited in several ways:

1. It cannot support multiple compilation units: the full call graph must be available at compile time.

2. It does not support resuming from the checkpoints it creates.
3. The target language is FORTRAN, which simplifies the problem of checkpointing by not considering pointer aliasing.

5.3.2 Related Work to Performance Prediction

Section 5.2.1 briefly summarized related work in performance prediction by splitting those works into four categories: programmer-provided, analytical, simulator-based, and learning-based. This section will go into more detail on how each of the works touched on there perform or use performance prediction.

Programmer-Provided

In [114], the authors look at using task priority to schedule workloads across heterogeneous processing units. By accumulating low-priority tasks for later, batched execution on a bandwidth-optimized accelerator and immediately executing high-priority tasks on latency-optimized cores, the authors demonstrate a natural fusion of message passing and accelerator programming. The authors rely on programmer-provided task priorities to differentiate between low and high priority tasks. To some extent, these "priorities" are actually performance prediction proxies for selecting which tasks will execute well on an accelerator and which will execute well on the host.

Likewise, in [95] the authors use task "affinities" to indicate the architectures on which a task is better executed. Rather than use a bi-modal low and high priority model, like in [114], this work allows affinities to multiple architectures to be specified on a single task as integers. This allows the programmer to make relative performance predictions to help the runtime scheduler select the execution platform for a device.

The work presented in [115] adds a `hint` directive to an existing heterogeneous compiler which allows the programmer to specify an estimate of the number of clock cycles required to run an annotated kernel on the accelerator. This programmer-provided, absolute performance prediction is then used to make runtime scheduling decisions as to when to launch tasks on the accelerator. Note that the accelerator architecture targeted in [115] is a simpler, embedded architecture. Hence, making cycle-accurate predictions is a more reasonable task.

Analytical

The work presented in [116] focuses on modeling different layers of parallelism in modern multi-core systems, e.g. multiple threads in a single SMT core, multiple cores on a single processor, multiple processors in a SMP node. The authors construct an analytical model that takes in hardware counters collected at runtime and uses that information to select either optimal performance or optimal energy-performance configurations for different layers of parallelism. They then use that information to tune the parallel structure of each parallel region in an application, based on the hardware counters from that parallel region. This process is performed entirely online during application execution, and so does not require an offline training phase as HYDOSO does.

ANATOMY [129] is a tool for modelling the performance of the main memory of a system, rather than the processing units. Using an analytical queueing model to simulate the behavior of DRAM, the authors aim to aid hardware designers in tuning their hardware configurations based on representative workloads. However, the high accuracy of this work is a result of its high-fidelity representation of DRAM hardware features, and so as hardware evolves the analytical model may have to be continually updated with it.

In [130], the authors use symbolic evaluation of “work flow graphs” (WFG) to analyze the memory access patterns and synchronization of a given GPU kernel in order to predict its performance on a target GPU platform. Using knowledge of GPU architectures and memory hierarchy, the authors focus on precise modeling of low-level hardware behavior such as bank conflicts, coalesced memory accesses, SIMD divergence, and warp/instruction-level parallelism using compile-time analysis of the kernel paired with a small performance model generated from micro-benchmarks. This framework is intended to be used as a guidance for compiler or programmer optimizations, and as such is able to estimate how certain elements of a kernel (e.g. thread divergence, synchronization, bank conflicts) contribute to its overall performance. This work demonstrates accurate absolute performance estimates across dense matrix-matrix multiplication, FFT, parallel prefix sum, and sparse matrix-vector multiplication kernels.

Simulator-Based

Graphite [117] is a parallel architecture simulator that looks to address the main limitations of simulator-based performance prediction (high overheads) by distributing the simulator workload across nodes and cores of a compute cluster. Graphite has a pluggable infrastructure which enables the addition of new performance models for novel architectural features, and demonstrates highly accurate (¡10%) predictions. While Graphite helps to overcome the limitation which it targets, high overheads from simulation-based performance modeling, it is still fundamentally an x86-based simulator. Results presented in [117] show that even using it for similar (but not x86) architectures introduces larger errors. Hence, Graphite may not be useful for future heterogeneous platforms.

Learning-Based

In [131], the authors present the DPAPP model: a learning-based performance and energy model that uses a multi-variate regression process based on hardware counters to continually learn architecture characteristics at runtime. Like [116], the authors use their DPAPP model to guide runtime concurrency control at multiple parallelism layers, with the goal of reducing energy consumption when possible without sacrificing performance. Multi-variate regression is selected as the performance model because of its low computational overhead for evaluation and prediction at runtime. DPAPP includes an offline learning phase in which representative, curated benchmarks and datasets are used to characterize each layer of parallelism on a target architecture. Then, hardware events are collected at runtime for different parallel phases of a program and plugged into the generated regression model in order to estimate well-performing parallelism configurations with possible energy improvements.

In [73], the authors focus explicitly on selecting a target architecture for a kernel, given multiple options. A Support Vector Machine-based statistical model is trained offline on kernel features (e.g. percentage of arithmetic operations, percentage of memory load operations, etc.) which enables efficient selection of a target architecture at runtime. This work focuses on selecting between JVM execution and GPU execution, and integrates with the IBM J9 JVM [132].

Several past works [24][133][134] describe similar approaches to using performance prediction to improve scheduling of tasks based on runtime profiling. HadoopCL[24] aims to auto-select the execution architecture for Hadoop MapReduce tasks. Kaleem et al.[133] aim to automatically partition a workload across CPUs and GPUs in an integrated, shared memory system using precise performance prediction for each device. Qilin[134] uses an adaptive mapping framework to automatically select execution platforms for data-parallel, stream kernels. These past works use speculative scheduling of tasks across multiple architectures to characterize the performance of tasks on each architecture. The end of that speculative scheduling stage is determined in these frameworks based on convergence of some framework-specific confidence measure that captures how well the framework believes it has characterized the behavior of each task in an application. Once speculative scheduling is complete, the framework uses the generated task performance models to guide future scheduling of tasks. These frameworks are entirely online and base their scheduling on historical performance of tasks.

5.4 Conclusions

This chapter has prevented novel work in HPC tools. In Section 5.1, we motivated, described, and evaluated the CHIMES framework, a checkpointing tool for use in debugging performance and correctness bugs in multi-threaded applications. In Section 5.2, we described the HYDOSO framework for performance prediction which combines compile-time, instruction-level analysis with runtime state to improve the accuracy of predictions for loop-parallel kernels.

Low-overhead and accurate tooling will be key to debugging and addressing correctness and performance issues on current and future HPC machines. While much of the past research on the programmability of heterogeneous supercomputers has focused on 1) the expressiveness and safety of programming models, and 2) the intelligence and efficiency of runtimes. However, building programming models and runtimes that will work efficiently and optimally in every (or even most) situations seems unlikely. Therefore, identifying inefficiencies in current and future HPC applications must rely heavily on 1) appropriate tools being available which offer sufficient insight without significantly impacting the behavior of an application, and 2) appropriate tuning knobs in programming models and runtimes that can be used to fix issues identified using those tools.

Chapter 6

Future Work & Conclusions

In the introduction, we identified the following five challenges as being key to solving heterogeneous programmability:

1. Composing many heterogeneous management libraries.
2. Enabling low-level tunability of accelerator kernels.
3. Managing coherency among heterogeneous and physically discrete address spaces.
4. Scheduling of workloads across processing units with heterogeneous architectures.
5. Providing tools that improve the ability of programmers to analyze system state in multi-tenant, heterogeneous applications.

The work presented in Chapters 3, 4, and 5 addresses these challenges in the following ways.

Composing many heterogeneous programming libraries is challenging, as in the past it has generally been done by making one library more "aware" of the other (e.g. GPU-Aware MPI), an unscalable solution. HiPER focuses on enabling the composition of multiple heterogeneous programming libraries on a multi-core system. Through a task-parallel programming model, platform abstractions, and a modular plug-in system for heterogeneous libraries, HiPER makes it simple for library developers to integrate with other programming systems and exposes a cleaner, more expressive API for expressing cross-module dependencies.

While optimizing compilers and other automatic techniques continue to improve, low-level and manual tunability of kernel code, workload scheduling, and other optimization knobs remains an important part of HPC and heterogeneous programming. While HiPER abstracts away many low-level programming concerns, such as multi-core load balancing, it still offers low-level tunability by enabling programmers to hand-tune their own kernels and scheduling. For example, HiPER's platform model enables fine-grain locality control across runtime threads, its library-based programming system integrates cleanly with existing optimizing/heterogeneous compilers, and its place paths allow programmers to tune the priority of different task types. HCL2 and SWAT, on the other hand, will auto-generate and auto-optimize low-level kernel code and scheduling decisions for the programmer, but also expose tunables and the ability to insert manual modifications into those kernels. This represents a hybrid human-machine model of tunability.

Coherency in a multi-processor environment can be a burden to programmers and, when mis-managed, a performance bottleneck. HCL2 and SWAT both automatically manage multiple discrete address spaces for the programmer (including JVM, native, and accelerator), while HJ-OpenCL uses runtime program inspection to automatically remove redundant accelerator transfers. HiPER, on the other hand, allows programmers to more naturally express the communication of data between separate address spaces using a future-based programming model, such that with a single abstraction an accelerator kernel can be made dependent on accelerator communication which is dependent on inter-node communication.

HiPER, HCL2, and SWAT all offer unified scheduling of communication and computation on a single runtime system. This provides the runtime system with more information on the state of the application, enabling more complex scheduling heuristics. Additionally, HYDOSO makes novel contributions in performance prediction across heterogeneous processing units, which can enable more accurate scheduling decisions on heterogeneous platforms.

Finally, each of these works have explored novel tooling capabilities for heterogeneous systems. CHIMES presents a novel exploration of checkpoints as a software development tool, demonstrating low overhead checkpoint creation and including a discussion of how checkpoints can be a powerful for application developers. HCL2, SWAT, and HiPER show how integration of tools and instrumentation into unified runtime systems can enable powerful HPC tooling. HYDOSO offers a framework for performance prediction, which can be a useful tool for application developers or runtime developers when tuning scheduling and energy consumption.

Several of these projects have exciting future directions and ongoing development.

While the HYDOSO framework has demonstrated preliminary effectiveness as a performance prediction tool, that value can only be practically realized as an underlying layer for a higher level software engineering system. Continuing work with HYDOSO looks to integrate it into polyhedral, heterogeneous code optimizers and the OSCAR compiler's heterogeneous programming support as a guide for both static code transformations and dynamic runtime scheduling decisions. HYDOSO's use with the Epiphany co-processor [36] will also be explored in future work.

Ongoing work with SWAT is exploring its use in larger scale applications, particularly in genome alignment. The main challenge in working with SWAT remains the restrictions on the kernels and data structures that can be automatically offloaded based on what the SWAT-APARAPI code generator can accept. Hence, a three-stage process of porting existing, complex Spark kernels to SWAT is being explored. In the first stage, a pre-processing Spark transformation is used to simply convert spark-formatted data into a format that SWAT can accept. Second, a post-process Spark transformation is developed which converts from a SWAT-compatible output format to the format expected by downstream transformations. Finally, additional source code-level transformations are made to the core computational kernel to use the new input and output formats as well as support automatic offload. While techniques like these require more programmer effort and are not entirely automatic, the overall effort to accelerate a Spark application will still be much lower than with hand-coded accelerator support.

With collaborators on the HiPER project, we are also exploring how new interfaces between task-parallel runtimes and communication libraries can enable tighter, cooperative, and more composable integration between the two. While this work is primarily motivated by OpenSHMEM, the hope is that the insights gained will be applicable to other communication libraries as well.

The thesis of this dissertation focused on the use of composable programming systems, combined runtime and compile-time analysis/optimization, and integrated resource-aware runtimes to solve all five of the above challenges. Each of the software components presented in this dissertation makes novel contributions in at least one of these three solution areas and advances the state-of-the-art in productive programming for heterogeneous supercomputers.

Bibliography

[1] Top500, “Top500.” http://www.top500.org.

[2] J. Dongarra, “Report on the Sunway TaihuLight System.” http://www.netlib.org/utk/people/JackDongarra/PAPERS/sunway-report-2016-old.pdf, June 2016.

[3] G. Chrysos, “Intel® Xeon Phi Coprocessor - The Architecture.” http://gec.di.uminho.pt/Discip/MInf/cpd1314/SCD/Intel_Xeon-PhiArch.pdf, 2014.

[4] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable Parallel Programming With CUDA,” *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[5] Wen-Mei Hwu, “Addressing the Accelerator Programming Challenges in Exascale Systems.” http://www.mcs.anl.gov/events/workshops/ashes/2016/slides/session1/Hwu_AsHES-Keynote-Hwu-2016.pdf, The Sixth International Workshop on Accelerators and Hybrid Exascale Systems (AsHES). 2016.

[6] NVIDIA, “Compute Unified Device Architecture Programming Guide.” https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2007.

[7] OpenMP Language Committee, “OpenMP API, Version 4.0.” http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, July 2013.

[8] Message Passing Interface Forum, “MPI: A Message-Passing Interface Standard, Version 3.1.” http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, 2015.

[9] OpenSHMEM Community, "OpenSHMEM Application Programming Interface Version 1.3." http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.3.pdf.

[10] OpenACC Standards Committee, "OpenACC: Directives for Accelerators." http://www.openacc.org/About_OpenACC, 2011.

[11] Munshi, Aaftab, "The OpenCL Specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*, pp. 1–314, IEEE, 2009.

[12] NVIDIA, "cuBLAS." https://developer.nvidia.com/cublas.

[13] NVIDIA, "cuSPARSE." https://developer.nvidia.com/cusparse.

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

[15] 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, *et al.*, "An Overview of the Trilinos Project," *ACM Transactions on Mathematical Software (TOMS)*, vol. 31, no. 3, pp. 397–423, 2005.

[16] The Apache Software Foundation, "Hadoop." http://hadoop.apache.org/.

[17] The Apache Software Foundation, "Spark: Lightning-Fast Cluster Computing." https://spark.apache.org/.

[18] The Apache Software Foundation, "GraphX." http://spark.apache.org/graphx/.

[19] The Apache Software Foundation, "Apache Spark MLlib." http://spark.apache.org/mllib/.

[20] C. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, "Map-Reduce for Machine Learning on Multicore," in *NIPS. Vol 6*, 2006.

[21] Y. Jia, "Caffe: An Open Source Convolutional Architecture for Fast Feature Embedding." http://caffe.berkeleyvision.org/, 2013.

[22] Y. Zheng, A. Kamil, M. Driscoll, H. Shan, and K. Yelick, "UPC++: A PGAS Extension for C++," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 1105–1114, May 2014.

[23] M. Grossman, S. Imam, and V. Sarkar, "HJ-OpenCL: Reducing the Gap Between the JVM and Accelerators," in *Proceedings of the Principles and Practices of Programming on The Java Platform*, pp. 2–15, ACM, 2015.

[24] M. Grossman, M. Breternitz, and V. Sarkar, "HadoopCL2: Motivating the Design of a Distributed, Heterogeneous Programming System With Machine-Learning Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 762–775, 2016.

[25] M. Grossman and V. Sarkar, "SWAT: A Programmable, In-Memory, Distributed, High-Performance Computing Platform," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pp. 81–92, ACM, 2016.

[26] S. Imam and V. Sarkar, "Habanero-Java Library: a Java 8 Framework for Multicore Programming," in *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pp. 75–86, ACM, 2014.

[27] Max Grossman, Vivek Kumar, Zoran Budimlic, Vivek Sarkar, "Integrating Asynchronous Task Parallelism with OpenSHMEM," in *OpenSHMEM Workshop*, 2016.

[28] M. Grossman, J. Shirako, and V. Sarkar, "OpenMP as a High-Level Specification Language for Parallelism," in *12th International Workshop on OpenMP*, 2016.

[29] M. Grossman, V. Kumar, N. Vrvilo, Z. Budimlic, and V. Sarkar, "A Pluggable Framework for Composable HPC Scheduling Libraries," in *Submitted for publication to AsHES 2017*, IEEE, 2017.

[30] M. Grossman and V. Sarkar, "Efficient Checkpointing of Multi-Threaded Applications as a Tool for Debugging, Performance Tuning, and Resiliency," in *IEEE International Parallel and Distributed Processing Symposium*, IEEE, 2016.

[31] NVIDIA, "NVIDIA Tesla P100." https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016.

[32] NVIDIA, "NVIDIA NVLink High Speed Interconnect." http://www.nvidia.com/object/nvlink.html.

[33] A. Sodani, "Knights Landing (KNL): 2nd Generation Intel® Xeon Phi Processor," in *Hot Chips 27 Symposium (HCS), 2015 IEEE*, pp. 1–24, IEEE, 2015.

[34] National Energy Research Scientific Computing Center, "CORI Overview." http://www.nersc.gov/users/computational-systems/cori/.

[35] Saito, Hideki and Preis, Serge and Panchenko, Nikolay and Tian, Xinmin, "Reducing the Functionality Gap Between Auto-Vectorization and Explicit Vectorization," in *International Workshop on OpenMP*, pp. 173–186, Springer, 2016.

[36] A. Olofsson, "Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip." https://www.parallella.org/wp-content/uploads/2016/10/e5_1024core_soc.pdf, 2016.

[37] Altera, "Stratix 10 FPGA and SOC." https://www.altera.com/products/fpga/stratix-series/stratix-10/overview.html.

[38] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 3.0." http://www.openmp.org/mp-documents/OpenMP3.0.0.pdf.

[39] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 4.0 - July 2013." `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[40] I. Karlin, T. Scogland, A. C. Jacob, S. F. Antao, G.-T. Bercea, C. Bertolli, B. R. de Supinski, E. W. Draeger, A. E. Eichenberger, J. Glosli, *et al.*, "Early Experiences Porting Three Applications to OpenMP 4.5," in *International Workshop on OpenMP*, pp. 281–292, Springer, 2016.

[41] M. Martineau, J. Price, S. McIntosh-Smith, and W. Gaudin, "Pragmatic Performance Portability with OpenMP 4. x," in *International Workshop on OpenMP*, pp. 253–267, Springer, 2016.

[42] A. Eichenberger, J. Mellor-Crummey, M. Schulz, N. Copty, J. DelSignore, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "OMPT and OMPD: OpenMP Tools Application Programming Interfaces for Performance Analysis and Debugging," in *International Workshop on OpenMP (IWOMP 2013)*, 2013.

[43] Computer Science Research Institute of the Sandia National Laboratories, "Kokkos Github." `https://github.com/kokkos/kokkos`.

[44] The Trilinos Project, "Trilinos User Group Meeting 2015." `https://trilinos.org/community/events/trilinos-user-group-meeting-2015/`.

[45] R. Hornung, J. Keasler, *et al.*, "The RAJA Portability Layer: Overview and Status," *Lawrence Livermore National Laboratory, Livermore, USA*, 2014.

[46] Mark Harris, "Inside Pascal: NVIDIAs Newest Computing Platform." `https://devblogs.nvidia.com/parallelforall/inside-pascal/`.

[47] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda, "GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evalua-

tion," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 10, pp. 2595–2605, 2014.

[48] NumPy Developers, "NumPy." http://www.numpy.org/.

[49] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A Fresh Approach to Numerical Computing," *CoRR*, vol. abs/1411.1607, 2014.

[50] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan, "Scaling Spark on HPC Systems," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pp. 97–110, ACM, 2016.

[51] S. Imam and V. Sarkar, "Habanero-Java Library: A Java 8 Framework for Multicore Programming," in *11th International Conference on the Principles and Practice of Programming on the Java Platform, PPPJ*, vol. 14, 2014.

[52] Gary Frost, "APARAPI in AMD Developer Website." http://developer.amd.com/tools/heterogeneous-computing/aparapi/.

[53] Luontola, Esko, "Retrolambda: Use Lambdas on Java 7." https://github.com/orfjackal/retrolambda, 2013.

[54] S. Che, J. W. Sheaffer, and K. Skadron, "Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems," in *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, p. 13, ACM, 2011.

[55] J. J. Fumero, M. Steuwer, and C. Dubach, "A Composable Array Function Interface for Heterogeneous Computing in Java," in *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, p. 44, ACM, 2014.

[56] M. Grossman, M. Breternitz, and V. Sarkar, "HadoopCL: MapReduce on Distributed Heterogeneous Platforms Through Seamless Integration of Hadoop and OpenCL," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, (Washington, DC, USA), pp. 1918–1927, IEEE, 2013.

[57] Jeffrey Dean, Sanjay Ghemawt, "MapReduce: Simplified Data Processing on Large Clusters," Communications of the ACM 51.1 (2008): 107-113.

[58] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral Parallel Code Generation for CUDA," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, p. 54, 2013.

[59] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimization," *ACM SIGPLAN Notices*, vol. 44, no. 4, pp. 101–110, 2009.

[60] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA Code Generation for Affine Programs," in *Compiler Construction*, pp. 244–263, Springer, 2010.

[61] Owens, John D and Houston, Mike and Luebke, David and Green, Simon and Stone, John E and Phillips, James C, "GPU Computing," Proceedings of the IEEE 96.5 (2008): 879-899.

[62] 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 *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44–54, IEEE, 2009.

[63] Matthew Curtis-Maury, Christos D. Antonopoulos, Dimitrios S. Nikolopoulos, "Predication-based Power-Performance Adaptation of Multithreaded Scientific

Codes," IEEE Transactions on Parallel And Distributed Systems, Volume 19, No. 10, Pages 1396-1410.

[64] Matthew Curtis-Maury, Ankur Shah Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, Martin Schulz, "Prediction Models for Multi-Dimensional Power-Performance Optimization on Many Cores," Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. ACM, 2008.

[65] H. Mühleisen and C. Bizer, "Web Data Commons-Extracting Structured Data from Two Large Web Corpora.," *LDOW*, vol. 937, pp. 133–145, 2012.

[66] A. Asuncion and D. Newman, "UCI Machine Learning Repository." http://archive.ics.uci.edu/ml/, 2007.

[67] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A Large-Scale Hierarchical Image Database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pp. 248–255, IEEE, 2009.

[68] Wikimedia Foundation, "Wikipedia Data Dumps." https://dumps.wikimedia.org/enwiki/.

[69] P. C. Pratt-Szeliga, J. W. Fawcett, and R. D. Welch, "Rootbeer: Seamlessly Using GPUs from Java," in *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pp. 375–380, IEEE, 2012.

[70] A. Hayashi, M. Grossman, J. Zhao, J. Shirako, and V. Sarkar, "Accelerating Habanero-Java Programs with OpenCL Generation," in *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pp. 124–134, ACM, 2013.

[71] A. Hayashi, M. Grossman, J. Zhao, J. Shirako, and V. Sarkar, "Speculative Execution of Parallel Programs with Precise Exception Semantics on GPUs," in *Languages and Compilers for Parallel Computing*, pp. 342–356, Springer, 2014.

[72] K. Ishizaki, A. Hayashi, G. Koblents, and V. Sarkar, "Compiling and Optimizing Java 8 Programs for GPU Execution," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 419–431, IEEE, 2015.

[73] A. Hayashi, K. Ishizaki, G. Koblents, and V. Sarkar, "Machine-Learning-Based Performance Heuristics for Runtime CPU/GPU Selection," in *Proceedings of the Principles and Practices of Programming on the Java Platform*, pp. 27–36, ACM, 2015.

[74] W. Zaremba, Y. Lin, and V. Grover, "JaBEE: Framework for Object-Oriented Java Bytecode Compilation and Execution on Graphics Processor Units," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pp. 74–83, ACM, 2012.

[75] A. Abbasi, F. Khunjush, and R. Azimi, "A Preliminary Study of Incorporating GPUs in the Hadoop Framework," in *The 16th CSI International Symposium on Computer Architecture and Digital Systems (CADS 2012)*, pp. 178–185, IEEE, 2012.

[76] K. Shirahata, H. Sato, and S. Matsuoka, "Hybrid Map Task Scheduling for GPU-Based Heterogeneous Clusters," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pp. 733–740, IEEE, 2010.

[77] A. Sabne, P. Sakdhnagool, and R. Eigenmann, "HeteroDoop: A Mapreduce Programming System for Accelerator Clusters," in *Proceedings of the 24th In-*

ternational Symposium on High-Performance Parallel and Distributed Computing, pp. 235–246, ACM, 2015.

[78] O. Segal, P. Colangelo, N. Nasiri, Z. Qian, and M. Margala, "SparkCL: A Unified Programming Framework for Accelerators on Heterogeneous Clusters," *arXiv preprint arXiv:1505.01120*, 2015.

[79] P. Li, Y. Luo, N. Zhang, and Y. Cao, "HeteroSpark: A Heterogeneous CPU/GPU Spark Platform for Machine Learning Algorithms," in *Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on*, pp. 347–348, IEEE, 2015.

[80] 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 Notices*, vol. 40, pp. 519–538, ACM, 2005.

[81] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications," in *PDP 2010-The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, 2010.

[82] S. Treichler, M. Bauer, and A. Aiken, "Realm: An Event-Based Low-Level Runtime for Distributed Memory Architectures," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pp. 263–276, ACM, 2014.

[83] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A Task Based Programming Model in a Global Address Space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, p. 6, ACM, 2014.

[84] K. B. Wheeler, R. C. Murphy, and D. Thain, “Qthreads: An API for Programming With Millions of Lightweight Threads,” in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–8, IEEE, 2008.

[85] M. Adams, “HPGMG 1.0: a Benchmark for Ranking High Performance Computing Systems.” https://bitbucket.org/hpgmg/hpgmg, 2014.

[86] National Energy Research Scientific Computing Center, “Edison.” http://www.nersc.gov/users/computational-systems/edison/.

[87] Oak Ridge Leadership Computing Facility, “Titan.” https://www.olcf.ornl.gov/titan/.

[88] U. Hanebutte and J. Hemstad, “ISx: A Scalable Integer Sort for Co-design in the Exascale Era,” in *Partitioned Global Address Space Programming Models (PGAS), 2015 9th International Conference on*, pp. 102–104, Sept 2015.

[89] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, “UTS: An Unbalanced Tree Search Benchmark,” in *Languages and Compilers for Parallel Computing*, pp. 235–250, Springer, 2007.

[90] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, “Introducing the Graph 500,” *Cray Users Group (CUG)*, 2010.

[91] M. Grossman, V. Kumar, Z. Budimlic, and V. Sarkar, “Experiences Developing Regular and Irregular Applications on OpenSHMEM,” in *Supercomputing 2016 PGAS Booth Poster*, 2016.

[92] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, “Integrating Asynchronous Task Parallelism With MPI,” in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 712–725, IEEE, 2013.

[93] V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić, and V. Sarkar, "HabaneroUPC++: A Compiler-Free PGAS Library," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, p. 5, ACM, 2014.

[94] K. Vaidyanathan, D. D. Kalamkar, K. Pamnany, J. R. Hammond, P. Balaji, D. Das, J. Park, and B. Joó, "Improving Concurrency and Asynchrony in Multithreaded MPI Applications Using Software Offloading," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 30, ACM, 2015.

[95] A. Sbîrlea, Y. Zou, Z. Budimlíc, J. Cong, and V. Sarkar, "Mapping a Data-Flow Programming Model onto Heterogeneous Platforms," in *ACM SIGPLAN Notices*, vol. 47, pp. 61–70, ACM, 2012.

[96] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta, "Productive Programming of GPU Clusters with OmpSs," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 557–568, IEEE, 2012.

[97] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin, "Xkaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 1299–1308, IEEE, 2013.

[98] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing Locality and Independence with Logical Regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 66, IEEE Computer Society Press, 2012.

[99] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: A High-Productivity Programming Language for HPC with Logical Regions," in *Pro-*

ceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 81, ACM, 2015.

[100] H. Pan, B. Hindman, and K. Asanovic, "Lithe: Enabling Efficient Composition of Parallel Libraries," *Proc. of HotPar*, vol. 9, 2009.

[101] Wang, Perry H and Collins, Jamison D and Chinya, Gautham N and Jiang, Hong and Tian, Xinmin and Girkar, Milind and Yang, Nick Y and Lueh, Guei-Yuan and Wang, Hong, "EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-Core Multithreaded System," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 156–166, 2007.

[102] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: A Programming Model for Heterogeneous Multi-Core Systems," in *ACM SIGOPS Operating Systems Review*, vol. 42, pp. 287–296, ACM, 2008.

[103] Garland, Michael and Kudlur, Manjunath and Zheng, Yili, "Designing A Unified Programming Model for Heterogeneous Machines," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pp. 1–11, IEEE, 2012.

[104] Bosilca, George and Bouteiller, Aurelien and Danalis, Anthony and Faverge, Mathieu and Hérault, Thomas and Dongarra, Jack J, "PaRSEC: Exploiting Heterogeneity to Enhance Scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[105] Max Grossman, Mauricio Araya-Polo, "Efficient Static and Dynamic Memory Management Techniques for Multi-GPU Systems," in *Workshop on Runtime Systems for Extreme Scale Programming Models and Architectures*, November 2015.

[106] Max Grossman, Mauricio Araya-Polo, "Distributed, Heterogeneous Scheduling Techniques Motivated by Production Geophysical Applications," in *Workshop*

on Many-Task Computing on Clouds, Grids, and Supercomputers, November 2015.

[107] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–12, IEEE, 2009.

[108] J. Duell, "The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart," *Lawrence Berkeley National Laboratory*, 2005.

[109] The Clang Team, "Clang LibTooling." http://clang.llvm.org/docs/LibTooling.html.

[110] Yann Collet, "xxHash: Extremely Fast Hash Algorithm." https://github.com/Cyan4973/xxHash.

[111] K. M. Dixit, "The SPEC Benchmarks," *Parallel computing*, vol. 17, no. 10, pp. 1195–1209, 1991.

[112] I. K. et al., "Lulesh Programming Model and Performance Ports Overview," Tech. Rep. LLNL-TR-608824, Lawrence Livermore National Laboratory, December 2012.

[113] ExMatEx team at Los Alamos National Laboratory and Lawrence Livermore National Laboratory, "CoMD Proxy Application." http://www.exmatex.org/comd.html.

[114] J. Lifflander, G. C. Evans, A. Arya, and L. Kale, "Dynamic Scheduling for Work Agglomeration on Heterogeneous Clusters," in *Proceedings of (PLC'12) Multicore and GPU Programming Models, Languages and Compilers Workshop at IPDPS 2012*, May 2012.

[115] A. Hayashi, “Studies on Automatic Parallelization for Heterogeneous and Homogeneous Multicore Processors,” in *Waseda University PhD Dissertation*, 2012.

[116] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos, “Online Power-Performance Adaptation of Multithreaded Programs Using Hardware Event-Based Prediction,” in *Proceedings of the 20th Annual International Conference on Supercomputing*, pp. 157–166, ACM, 2006.

[117] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, “Graphite: A Distributed Parallel Simulator for Multicores,” in *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pp. 1–12, IEEE, 2010.

[118] S. B. Needleman and C. D. Wunsch, “A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins,” *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[119] L.-N. Pouchet, “Polybench: The Polyhedral Benchmark Suite.” `http://www.cs.ucla.edu/pouchet/software/polybench`, 2012.

[120] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pp. 72–81, ACM, 2008.

[121] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, “Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing,” *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[122] Bailey, David H and Barszcz, Eric and Barton, John T and Browning, David

S and Carter, Robert L and Dagum, Leonardo and Fatoohi, Rod A and Frederickson, Paul O and Lasinski, Thomas A and Schreiber, Rob S and others, "The NAS Parallel Benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[123] J. Duell, "The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart," *Lawrence Berkeley National Laboratory*, 2005.

[124] J. Ansel, K. Aryay, and G. Coopermany, "DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–12, IEEE, 2009.

[125] S. I. Feldman and C. B. Brown, "Igor: A System for Program Debugging via Reversible Execution," in *ACM SIGPLAN Notices*, vol. 24, pp. 112–123, ACM, 1988.

[126] G. Bronevetsky, D. Marques, K. Pingali, S. McKee, and R. Rugina, "Compiler-Enhanced Incremental Checkpointing for OpenMP Applications," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–12, IEEE, 2009.

[127] G. Bronevetsky, K. Pingali, and P. Stodghill, "Experimental Evaluation of Application-Level Checkpointing for OpenMP Programs," in *Proceedings of the 20th Annual International Conference on Supercomputing*, pp. 2–13, ACM, 2006.

[128] R. R. Chandrasekar, A. Venkatesh, K. Hamidouche, and D. K. Panda, "Power-Check: An Energy-Efficient Checkpointing Framework for HPC Clusters," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pp. 261–270, IEEE, 2015.

[129] N. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan, "ANATOMY: An Analytical Model of Memory System Performance," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, pp. 505–517, ACM, 2014.

[130] Baghsorkhi, Sara S and Delahaye, Matthieu and Patel, Sanjay J and Gropp, William D and Hwu, Wen-mei W, "An Adaptive Performance Modeling Tool for GPU Architectures," in *ACM SIGPLAN Notices*, vol. 45, pp. 105–114, ACM, 2010.

[131] M. Curtis-Maury, F. Blagojevic, C. D. Antonopoulos, and D. S. Nikolopoulos, "Prediction-Based Power-Performance Adaptation of Multithreaded Scientific Codes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 10, pp. 1396–1410, 2008.

[132] IBM Corporation, "IBM J9 JVM." https://www.ibm.com/support/knowledgecenter/SSYKE2_7.0.0/com.ibm.java.lnx.70.doc/user/java_jvm.html.

[133] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali, "Adaptive Heterogeneous Scheduling for Integrated GPUs," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pp. 151–162, ACM, 2014.

[134] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 45–55, IEEE, 2009.