Automatic Scheduling of Compute Kernels Across Heterogeneous Architectures

Robert F. Lyerly

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

Master of Science
in
Computer Engineering

May 7, 2014
Blacksburg, Virginia

Keywords: Scheduling, Compilers, Heterogeneous Architectures, Runtime Systems, High-Performance Computing

Scheduling of Compute Kernels Across Heterogeneous Architectures Using Machine Learning

Robert F. Lyerly

(ABSTRACT)

The world of high-performance computing has shifted from increasing single-core performance to extracting performance from heterogeneous multi- and many-core processors due to the power, memory and instruction-level parallelism walls. All trends point towards increased processor heterogeneity as a means for increasing application performance, from smartphones to servers. These various architectures are designed for different types of applications – traditional "big" CPUs (like the Intel Xeon or AMD Opteron) are optimized for low latency while other architectures (such as the NVidia Tesla K20x or Intel Xeon Phi) are optimized for high-throughput. These architectures have different tradeoffs and different performance profiles, meaning fantastic performance gains for the right types of applications. However applications that are ill-suited for a given architecture may experience significant slowdown; therefore, it is imperative that applications are scheduled onto the correct processor.

In order to perform this scheduling, applications must be analyzed to determine their execution characteristics (e.g. an application that contains a lot of branching may be better suited to a traditional CPU). Traditionally this application-to-hardware mapping was determined statically by the programmer. However, this requires intimate knowledge of the application and underlying architecture, and precludes load-balancing by the system. We demonstrate and empirically evaluate a system for automatically scheduling compute kernels by extracting program characteristics and applying machine learning techniques. We develop a machine learning process that is system-agnostic, and works for a variety of contexts (e.g. embedded, desktop/workstation, server). Finally, we perform scheduling in a workload-aware and workload-adaptive manner for these compute kernels.

This work is supported in part by US NSWC under contract N00178-09-D-3017/0022.

Acknowledgments

There are many people who helped me with this work, and for whom I am forever grateful. I would like to thank the following people specifically for helping me obtain my degree:

Dr. Binoy Ravindran, for setting me down a research path I would never dream of pursuing, and providing guidance in all of my research.

My committee members, Dr. Cameron Patterson and Dr. Paul Plassmann, for graciously taking time out of their busy schedules to help me with this work.

Dr. Alastair Murray, for the many brainstorming sessions, in-depth technical conversations about compilers, Linux mastery and being a steadfast friend through rough patches.

Dr. Antonio Barbalace, for his unbelievable knowledge of Linux internals and neverending humor, despite any situation.

Kevin Burns, for his system administration skills and great friendship through our SSRG careers.

The members of the System Software Research Group for their support, skill and friendship, including Curt Albert, Brandon Amos, Saif Ansary, Michael Drescher and Shawn Furrow.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Evolving Landscape of Processor Architectures

In recent years, processor architectures have shfited away from improving sequential performance, or reducing instruction latency, in favor of increasing overall processor throughput by enabling different levels of parallelism. This is due to the "brick wall", a light-hearted name for a collection of highly problematic processor architecture design issues. In his article "The Free Lunch Is Over", Herb Sutter describes three issues – the power wall, the instruction-level parallelism wall and the memory wall [54]. While Moore's law continues to hold true, the power dissipated by these ever more complex processors grows with every generation – increased power dissipation has resulted in stalled processor clock rates. Additionally, processors are unable to extract more instruction-level parallelism from general applications, as tricks such as speculative execution, out-of-order buffers and superscalar execution yield smaller and smaller speedups. Finally, main memory has scaled in size but not in speed, relative to the clock rates of processors.

Chip manufacturers have thrown in the metaphorical towel in regards to these problems, and have instead decided to rethink chip design in favor of parallelism. Newer chips (from smartphone processors like the Snapdragon 810 [4] to server CPUs such as the AMD Opteron 6376 [6]) have eschewed highly complex, latency-above-all designs in favor of simpler symmetric multiprocessor (SMP) that integrate multiple fully-functional processor cores on-die. This enables task- and data-level parallelism for applications that have parallel work that is able to be split among multiple processors. While scientific-computing applications are generally parallelizable, other applications such as Mozilla's recently announced collaboration with Samsung on their rendering engine Servo [2] can take advantage of multiple processing units. Even embedded systems, which eschew runtime performance in favor of power efficiency, have adopted this multicore approach. The trend for the foreseeable future is parallel.

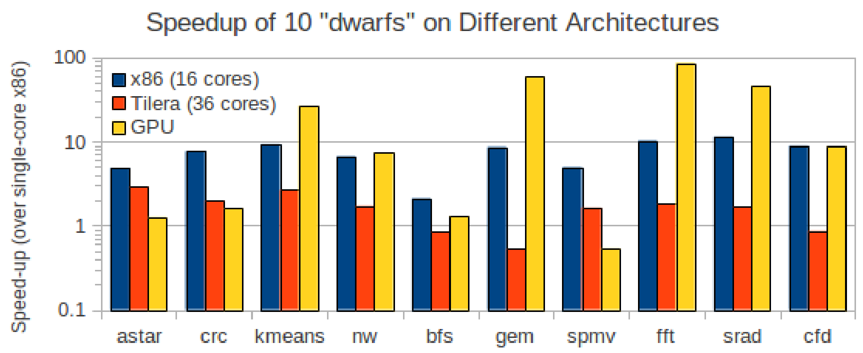


Figure 1.1: Speedup of benchmarks from the OpenDwarfs benchmark suite over execution on a single x86 core

More recently, systems have incorporated many types of processors with wildly varying design goals in order to provide high-performance execution for a variety of applications. Smartphones have a high amount of heterogeneity on-die with designs such as the Snapdragon 810. This SoC includes up to 4 traditional CPUs, several GPU cores, several DSPs and a microcontroller. Most workstations include traditionally “big” CPU cores and a discrete GPU card (using chips from NVidia or AMD) over PCIe. Intel has been marketing their Xeon Phi accelerator card [13] with plans to integrate it into motherboards with a dedicated socket [1]. In addition to parallelism, future computer systems will almost certainly be heterogeneous.

As figure 1.1 indicates, heterogeneity allows high performance for a variety of applications. This graph shows the speedup of compute-intensive portions of benchmarks over execution on a single x86 core when run on several different architectures[1]. All applications benefit from parallelism, but each execute differently based on the architecture. Some of the benchmarks have better performance on the 16 x86 host cores, while some have better performance on the GPU. Interestingly, some applications exhibit slowdown if executed on a certain architecture. However, the general trend indicates that parallelism and heterogeneity allow systems to execute high-performance applications more efficiently than homogeneous systems. Thus, we seek to utilize heterogeneity and parallelism to increase performance.

1.2 Current Limitations

Taking advantage of parallelism and heterogeneity can bring enormous performance benefits, but requires significant developer effort and esoteric knowledge of the underlying architecture to achieve maximum performance. Additionally, almost every heterogeneous-ISA device has

[1]These benchmarks were run on an AMD Opteron 6376 CPU, a Tilera TILEncore-Gx36 coprocessor and an NVidia Tesla C2075 GPU

its own device-specific parallel programming model, meaning that developers must spend time porting between various models to utilize different architectures[2]. These device-specific models require the developer to manually manage memory consistency between the host and device using device-specific APIs. This boilerplate code is tedious and error-prone, shifting focus away from the core application logic where developer effort should be spent. Ideally the programmer would write the compute-intensive portions of their application once and have the compiler manage translation to device-specific models while extracting maximum performance from that particular architecture. The compiler would ideally also handle data transfers between host and device transparently.

Additionally, different high-performance compute-bound applications execute differently on heterogeneous architectures based on the types of operations and the patterns of those operations. Additionally, depending on the problem size a given application may execute more or less efficiently by adding parallelism and heterogeneity. There are many ways to address this problem of deciding on which architecture a given application should execute its compute-bound sections (hereafter referred to as *scheduling*). There are two general approaches that previous work has taken:

1. *Utilize past history* – time the execution of the specified portion of the application and utilize this knowledge to make future scheduling decisions.
2. *Build a predictive model* – build some sort of predictive model to make scheduling decisions.

The first approach is reactive in nature and thus non-ideal, as it requires exhaustively evaluating the compute-bound region on a variety of architectures to determine the ideal scheduling. In addition, it is non-portable, does not adapt well to a varying workload and requires maintaining a monotonically-increasing database of past execution history in order to make scheduling decisions.

The second approach, if performed correctly, allows a more flexible and portable scheduler to make decisions on a wide variety of systems with limited or no prior knowledge. Recent approaches have used machine learning for the implementation of the scheduling logic. Given a representative set of benchmarks, a properly designed and trained machine learning model should be able to accurately predict on which architecture certain classes of applications should execute. However, the state-of-the-art in using machine learning to perform scheduling of applications onto architectures is also limited:

1. The state-of-the-art models are trained in systems where the heterogeneous architectures are vastly different, such as systems coupling a CPU with a discrete GPU accelerator. Many emerging systems (especially in the embedded space) couple together

[2]OpenCL is an attempt to bridge this gap, but the quality of implementations vary from vendor to vendor

several types of architectures, such as traditional multicore CPUs with integrated GPUs and DSPs, making the scheduling decision much more difficult. The Snapdragon 810 system-on-a-chip (SoC) processors combine multicore CPUs, a GPU and several DSPs onto a single die, all programmable using Qualcomm's development kits [4].

2. The models do not consider scheduling with respect to external workload. These models all use a static/offline prediction model that always maps the application to a single architecture. However, varying levels of external workload may affect the performance of an application on an architecture which ultimately changes the scheduling decision.

3. The generated machine learning models are not portable. Models are generated on a per system basis, meaning that the models generate predictions based on the relative efficiency of the architectures in a system. This is problematic, given the proliferation of heterogeneous architectures and the near infinite combinations of those architectures within systems. Any new combination of architectures in a system requires incurring heavy model generation costs.

We thus seek to address both the translation and scheduling problems in this work.

1.3 Research Contributions

In this thesis, we present the following contributions:

1. We implemented a partitioning tool that takes an application written using C and parallelized using the OpenMP parallel programming standard; it automatically analyzes & refactors the application so the compute-intensive portions (including data transfers and execution) can be executed on a variety of devices, including a multicore x86 host, a manycore "smart network card" and a server-class GPU.

2. We implemented two feature extractor tools that characterize compute-intensive applications by generating a set of application features. These features are used as inputs into the scheduler to make a scheduling decision.

3. We implemented a scheduling daemon that makes scheduling decisions for applications based on their extracted features. The daemon uses IPC to communicate with clients; it also maintains workload information about the devices in the system.

4. We generated machine learning models to make scheduling deicisions for applications. We evaluated several different models, including models that incorporate external workload information and unified models that can be used for any combination of architectures in a system.

1.4 Thesis Organization

The thesis is organized as follows:

- Chapter 2 presents background on the programming models & architectures used for evaluation in this thesis.
- Chapter 3 presents related work in the area of heterogeneous scheduling.
- Chapter 4 presents the partitioning tool used to automatically refactor applications to be able to execute on heterogeneous architectures. In addition, it discusses the implementation of the feature extractor, the scheduling daemon, and generation of machine learning models that incorporate external workload. We subsequently analyze the performance generated models
- Chapter 5 presents a second approach to the scheduling problem using OpenCL applications that require no refactoring. We generate and evaluate a unified machine learning model, i.e. a model that can be used for any combination of architectures in a system.
- Chapter 6 recapitulates the lessons learned from the work, and chapter 7 discusses possible future research directions.

Chapter 2

Background

2.1 Architectures

In this work we chose to investigate scheduling for a wide variety of architectures, intermixing several types of architectures within several systems. There were three main classes of architectures used in scheduling:

1. *Central Processing Units (CPU)* – classic general-purpose processors that are designed to handle a variety of different types of applications.
2. *Graphics Processing Units (GPU)* – general-purpose highly parallel processors initially designed for graphics computation with a recent shift towards general-purpose parallel computation.
3. *Manycore Coprocessors* – accelerator add-on cards with dozens of simplified CPU cores. There is no clear line between multicore and manycore, although multicore chips tend to have 16 or fewer cores.

Below we discuss the three classes of architectures, and the specific models that were used in this work.

2.1.1 Central Processing Units

The CPUs used in this work are modern "big" processors, with many of the latest advancements in process technology and architectural innovation. CPUs are considered latency-oriented processors in that they strive to execute a single thread of instructions as quickly as possible (although this has changed in recent years with multicore and SIMD, which introduce task-level and data-level parallelism, respectively). CPUs utilize instruction-level parallelism to get as many instructions as possible "in-flight". All of the processors we used are multi-core, and some contain support for simultaneous multithreading. These processors include out-of-order execution, are superscalar and superpipelined, and contain a SIMD-processing unit per core. These chips also feature large caches and hardware prefetching capabilities to hide memory access latencies. Finally, these chips contain variable, but high-frequency clock rates. The three CPUs used in this work are the AMD Opteron 6376, the Intel Core i7-4770 and the Intel Xeon E5-2695.

AMD Opteron 6376

The AMD Opteron 6376 is one of AMD's highest-end server offerings, containing 16 CPU cores with the 32 nm AMD Pileriver microarchitecture [6]. Each core contains a 4-way instruction decoder that can dispatch instructions to two integer clusters or a floating-point unit. Each integer cluster contains two arithmetic logic units (ALU) and two address generator functional units, while the floating-point unit contains two SIMD-lanes and two fused multiply-and-add (FMAD) units[1]. In terms of SIMD capabilities, each core supports SSE 4.2 (128-bit integer & floating-point operations) and AVX (256-bit floating-point operations). This processor does not contain SMT support, meaning that the number of physical cores on die is the number of cores reported by the operating system. The processor contains a 48 kB L1 and a 1 MB L2 cache per core, with a 16 MB L3 cache shared by all cores. Finally, the processor is clocked at 2.3 GHz.

Intel Core i7-4770

The Core i7-4770 is one of Intel's higher-end desktop CPUs with 4 cores (8 with SMT) and uses the 22nm Intel Haswell microarchitecture [29]. This core also contains a 4-way instruction decoder, but utilizes 8 execution ports to launch operations from the reorder buffer onto the functional units. Each core contains two integer, two floating-point and a memory functional unit, all of which can execute SIMD instructions. The Haswell microarchitecture supports SSE 4.2 as well as AVX2 (256-bit integer & floating-point operations). The processor contains a 32 kB L1 cache per core; a 1 MB L2 and an 8 MB L3 cache are shared between all cores. The processor also contains an integrated Intel HD 4600 GPU, which was not evaluated. Finally, the processor is clocked at 3.4 GHz and can be boosted to 3.9 GHz for compute-intensive applications.

Intel Xeon E5-2695

The Xeon E5-2695 brings the Haswell microarchitecture into the server setting. The microarchitecture of this processor is very similar to the Core i7 but the package contains 12

[1]The AMD moniker for fused multiply-and-add is fused multiply-and-accumulate (FMAC)

cores (24 with SMT) instead of 4. Additionally, the processor contains multi-socket and ECC support for servers, but removes the integrated Intel HD 4600 GPU. Each core contains the same 32 kB L1 cache, but the shared L2 and L3 caches have been expanded to 3 MB and 30 MB, respectively. Finally, the procesor is clocked at 2.4 GHz, and can be boosted to 3.2 GHz for compute-intensive applications.

2.1.2 Graphics Processing Units

The GPUs used to evaluate heterogeneous scheduling range from older-generation desktop- and server-grade GPUs to newer GPUs that blur class distinctions by simultaneously targetting gaming graphics and high-performance computation. GPUs are often advocated as a counter-balance to CPUs in that they are throughput-oriented. They aim to get as much independent parallel computation executing as possible; this design point is inspired by their graphics background, which requires rendering every pixel on screen as quickly as possible. The processor cores in these architectures contain wide SIMD (or SIMT for NVidia) lanes, each of which corresponds to a single thread in the parallel computation[2]. These cores execute multiple threads in lock-step and suffer severe performance penalties for control-flow divergence between individual threads. Additionally, these architectures feature programmer-managed caches (although recent models include CPU-like hardware caches) and hide memory access latencies by using hardware to swap between groups of threads (*warps* in NVidia terminology, or *wavefronts* in AMD terminology) that are not stalled on memory accesses. These architectures contain lower clock frequencies than their CPU counterparts, but sustain high instruction throughput through massive parallelism. The four GPUs used in this work are the NVidia GeForce GTX 560 Ti, the NVidia Tesla C2075, the NVidia GeForce GTX Titan and the AMD Radeon R9 290X.

NVidia GeForce GTX 560 Ti

The GeForce GTX 560 Ti (hereafter referred to as the GTX 560 Ti) is a desktop-grade GPU that uses NVidia's Fermi microarchitecture [45] at a 40 nm feature-size. The GTX 560 Ti is composed of 14 Fermi *streaming multiprocessors* (SM), each of which contains 32 *CUDA cores* for a total of 448 CUDA cores. Each CUDA core contains a 32-bit integer and a 32-bit floating-point unit, the latter of which can execute fused multiply-and-add instructions. Each SM additionally contains 16 load/store units and four special function units (for transcendental operations). Warps, or groups of 32 threads, are fetched & decoded with two warp schedulers. The instructions for all threads within the warp are then issued using two corresponding warp dispatch units. Each SM has a 64 kB configurable L1/shared memory cache, and all SMs are connected to 1.2 GB of global memory. The GPU is clocked

[2]GPU threads are a hardware entity, meaning they are much more lightweight than traditional operating system threads

at 1.464 GHz.

NVidia Tesla C2075

The Tesla C2075 (hereafter referred to as the Tesla) is the server-grade equivalent of the GTX 560 Ti aimed at HPC rather than gaming. This GPU also contains the Fermi microarchitecture with the same number of CUDA cores. However, because the Tesla is geared towards HPC, it contains full double-precision floating point hardware support, whereas the graphics-oriented GeForce cards use some form of double-precision emulation. This enables the Tesla to have 4x the double-precision performance of the GTX 560 Ti. Other than the enhanced double-precision compute performance, the differences between the two GPUs are small – the Tesla has a slightly slower clock frequency of 1.15 GHz, but a wider memory bus (384-bit vs. 320-bit) and a larger global memory (6 GB) with ECC support.

NVidia Geforce GTX Titan

The GeForce GTX Titan (hereafter referred to as the Titan) contains NVidia's newest Kepler GK110 microarchitecture which uses a 28 nm process technology [46]. Although the Titan is gaming-oriented, it is also billed as suitable for HPC (although server-grade Tesla Kepler GPUs exist). Besides adding programmability features, the Titan contains a redesigned streaming multiprocessor (renamed SMX) architecture. Each of the 14 SMXs in the Titan contain 192 CUDA cores, 64 separate double-precision units, 32 special function units and 32 load/store units for a total of 2,688 cores. The number of warp schedulers and warp dispatch units was increased to 4 and 8, respectively (warps still consist of groups of 32 threads). The Titan still contains 64 kB of configurable L1 cache/shared memory, but features an expanded register file and 6 GB of global memory. Finally, the Titan is clocked at a lower 837 MHz, but the increased number of cores enables significantly higher performance than the Fermi architecture[3].

AMD Radeon R9 290X

The Radeon R9 290X processor contains AMD's newest Graphics Core Next compute architecture on a 28 nm process [5]. The Graphics Core Next Architecture changes the underlying implementation of the *compute units* (CU) within the GPU. Instead of utilizing a set of 16 4-way VLIW-based SIMD lanes (which comprised a single 64 work-item wavefront), the CUs consist of a set of 4 16-wide SIMD lanes, each of which can execute instructions from separate wavefronts (similarly to NVidia's terminology, AMD refers to each processing unit in each SIMD lane as a *stream processor*) and fetch memory. This change allows limited out-of-order

[3]NVidia moved from separate base and graphics clocks in the Fermi architecture to a unified clock in the Kepler architecture

execution on the GPU, as different lanes can be filled from a re-order buffer. In addition to the SIMD lanes, each CU contains a scalar functional unit used to handle branching, synchronization and an ALU for use in address generation. The GPU contains 44 CUs for a total of 2816 stream processors. However, this GPU is tailored to graphics applications and therefore has neutered double-precision floating-point performance. Each CU contains a 16 kB read/write cache and a 16 kB read-only cache; a 1 MB L2 cache is shared by all CUs, and sits in front of 4 GB of global memory. The Radeon R9 290X is clocked at 1 GHz.

2.1.3 Manycore Coprocessors

These processors represent a future direction in CPU-like processor architectures. They attempt to gain the performance of throughput-oriented architectures through simpler CPU-like cores to leverage existing CPU programming models instead of the more complex GPU/single-work-item programming models. Because they seek to be compatible with existing processors and programming models, these processors contain mechanisms that make them OS-capable (e.g. interrupts and virtual memory). They feature cores that shed the power-hungry requirements of out-of-order execution in favor of in-order superscalar or VLIW execution. However, they do include branch prediction capabilities, a feature most modern GPUs lack. These architectures are gaining traction in various markets – Facebook uses Tilera's processors for their datacenters [9] and there are several Top500 supercomputers (including the fastest supercomputer in the world, the MilkyWay-2) that accelerate computation using the Xeon Phi coprocessors [3].

Tilera TILEncore-Gx36

The TILEncore-Gx36 coprocessor is a new custom architecture from Tilera fabricated on a 40 nm process and targeted towards multimedia servers [56]. The architecture consists of 36 3-way VLIW/in-order cores connected to a grid or mesh interconnect. Each core is programmed using a custom RISC ISA designed by Tilera. Each operation in a VLIW instruction word corresponds to one of three execution pipelines. The first pipeline is used for arithmetic and logic operations, including multi-cycle instructions such as arithmetic multiply. The second pipeline is also used for arithmetic and logic operations in addition to control flow (branches & jumps). The final pipeline is used solely for memory accesses. None of the pipelines support floating-point arithmetic, meaning it must be emulated in software. Additionally, there is no specialized SIMD hardware, although the arithmetic and logic units can perform SIMD instructions on 64-bit operands. Each processor is a part of an on-grid *tile* that also contains a 32 kB L1 and a 256 kB L2 cache. Access to the 8 GB of global memory (distributed between 4 memory channels) is serviced either through direct access if the tile is directly connected to a memory controller (and is therefore responsible for servicing all memory requests to that memory bank), or is routed through the grid to the appropriate memory bank. The TILEncore-Gx36 runs a custom version of Red Hat Enterprise Linux and is programmable using C/C++ through a custom GCC compiler. Standard Linux parallel programming models (i.e. PThreads & OpenMP) are available, meaning existing applications can be recompiled for the TILEncore-Gx36. The processors are clocked at 1.2 GHz; however, the entire PCIe board consumes at most 50W of power.

Intel Xeon Phi 3120A

The Xeon Phi architecture (codenamed "Knight's Corner") is a meet-in-the-middle design, allowing the programming flexibility of CPUs with the peak parallel performance of GPUs [13][50]. The Xeon Phi 3120A contains 57 physical cores, each with 4-way SMT, for a total of 228 logical cores manufactured with a 22 nm process. These cores operate using a subset of the x86 ISA, with added vector extensions for the custom 16-wide vector processing unit (VPU)[4]. Each core maintains 4 thread contexts, and issues from these contexts in an in-order fashion to two execution pipelines in a round-robin fashion. The processor cores cannot issue from the same context in consecutive cycles, meaning that there must be at least two threads per core to achieve maximum throughput. The first pipeline can execute instructions on the VPU, the scalar floating-point unit or the two scalar integer units, while the second pipeline can only execute instructions on the two scalar integer units. Each core contains a 32 kB L1 cache and a 512 kB slice of a distributed L2 cache. Each core/L2 cache slice is attached to a ring interconnect which allows access to 6 GB of memory distributed between 12 memory channels. As previously mentioned, the Xeon Phi runs its own stripped-down version of Linux and supports of a variety of tradtional and emerging programming models. The processor is clocked at 1.1 GHz.

2.2 Parallel Programming Models

The architectures mentioned in section 2.1 can be programmed using a variety of models. We focused on two widespread and industry-accepted programming models in our scheduling work.

Throughout this work, we denote a *compute kernel* as a section of parallel work to be performed on some parallel processor[5]. A compute kernel may execute at varying levels of granularity (e.g. data-level or task-level) and can be specified using a variety of programming models, but in general the compute kernel contains several pieces of independent work that are executed by some parallel processor, either concurrently (through context switching) or in parallel (multiple execution units operating at the same time). In general, compute kernels contain a large amount of arithmetic or logic computation and should benefit from parallelism in processing resources (although in practice they may not).

[4]The VPU is 16-wide for 32-bit arithmetic

[5]A compute kernel has no connection to an operating system kernel

Additionally, *coprocessor* and *device* are used interchangeably throughout. Although there are slight semantic differences (coprocessor implies non-host CPUs, while device refers to any compute resource), we generally use coprocessor and device to mean a processor resource on which to execute a compute kernel.

2.2.1 OpenMP

OpenMP, [10] which stands for Open Multi-Processing, is an API used to parallelize compute kernels on CPUs in a task/thread-based, shared-memory programming model. The OpenMP standard specifies a set of source-code pragmas, library functions and environment variables that allow developers to leverage underlying threading implementations (such as PThreads [33]) without the implementation-specific threading setup/cleanup code. Programmers annotate source code with OpenMP `parallel` pragmas to setup a *team of threads*, each of which executes the source code in the parallel region. Work-splitting among threads is specified by further annotation of for-loops and task-specific sections. The main OpenMP work-splitting construct used by the benchmarks in this work is the `for` construct, which splits the iterations of a given for-loop among the threads in a team using various strategies (the default of which is to give every consecutive iteration to consecutive threads). Developers can specify how data is shared between threads by appending shared or thread-private data clauses to the `for` pragma. The standard also includes support for various multithreading-specific constructs, such as critical/atomic regions, locks and barriers. The OpenMP runtime is configured through a set of library calls and environment variables to specify various parameters, such as the number of threads in a team. Since OpenMP is a standard and not an implementation, it includes no support for code refactoring to enable threading. However, many commercial and open-source compilers contain OpenMP support [28][26][15].

As of OpenMP 4.0, the standard includes support for offloading compute kernels to alternative devices using the OpenMP `target` pragmas. This breaks the shared-memory model of previous OpenMP standards, and requires the user to annotate offload pragmas with data movement clauses. Despite the standard being released in July 2013, there is no available implementation that supports the new `target` pragmas. However, several research-level source-to-source translators convert OpenMP to device-specific programming models. OpenMPC [40][39], developed by Lee et al. using the Cetus compiler infrastructure [38], is a tool that translates for-loops annotated with `omp for` pragmas into the CUDA parallel programming model [47], suitable for execution on NVidia GPUs. OpenMPC additionally applies some GPU-specific optimizations and has support for tuning of the generated applications.

2.2.2 OpenCL

OpenCL is a parallel programming standard for heterogeneous compute devices [24]. An OpenCL application consists of two interlocking pieces – a host and device. The host (usually a single CPU in the system) performs all setup, coordinates data transfer between the host & device, launches compute kernels, and performs all cleanup. The device is the parallel processing component and is solely responsible for executing compute kernels. The standard, which is strongly influenced by NVidia's CUDA programming model, specifies a functionally portable set of host-side APIs and a device-side C-like programming language for developing compute kernels. Compute kernels are executed by a global set of threads divided into a hierarchy of work-groups and work items. There are usually a limited number of fine-grained work items per work group (usually 64-256 work items), but a potentially large number of work groups. Compute kernels are specified in a single work item description, and are compiled at runtime into full compute kernel descriptions suitable for execution on a specific device. During execution, multiple work items are usually executed in parallel in a SIMD or SIMT fashion. Vendors who support OpenCL [30][16][47] provide a compiler and runtime that plugs into the OpenCL Installable Client Driver, which directs the generic OpenCL API to the user-specified implementation.

Chapter 3

Related Work

Research related to this work falls into several categories. The first section pertains to quantifying architectural characteristics of widely varying processor architectures (such as the functional units in processors, number of cores, memory subsystem, etc.). The next section discusses some of the recent successes in using machine learning to aid in the compilation process. Next, there have been many approaches to scheduling of compute kernels across heterogeneous systems that stretch in scope from simple compiler-level analysis tools to cluster-wide frameworks. The final section contains a body of work that addresses co-scheduling the parallel work of a single compute kernel across multiple architectures in a system. This body of work is not directly related the problems we address in that they assume complete access to all devices in a system, whereas we seek to cooperatively schedule multiple kernels in the same system. However, these works show the focus of the majority of heterogeneous scheduling research.

3.1 Architecture Characterization

Several previous works characterized processors based on various performance metrics. Thoman et al. developed a suite of microbenchmarks, named uCLbench, that stress various parts of OpenCL devices [55]. These benchmarks stress the various subsystems and runtime components of OpenCL devices, including the devices' basic and transcendental arithmetic throughput, their memory bandwidth & latency, their ability to handle control flow divergence and OpenCL runtime overheads[1]. The authors evaluate several CPU, CPU-like and GPU architectures using the benchmark suite; they additionally use the results to guide manual kernel optimization (for vectorization, branching elimination and caching optimizations) for a Ja-

[1]Our set of features extracted from OpenMP (section 4.2.2) and OpenCL (section 5.2.2) applications was strongly influenced by this work. uCLbench was also used to generate hardware features for OpenCL devices (section 5.2.3).

cobi kernel on each device, showing an overall 192% improvement over a standard OpenCL compiler.

Zhang et al. study the poor performance portability of OpenCL compute kernels across diverse architectures and how performance portability can be improved by a set of "tuning knobs" and OpenCL language extensions [58]. The problems they discuss with poor performance portability arise strongly from the differences in the memory subsystems of GPUs and CPUs (and the inability of the respective compilers to adapt the compute kernels appropriately)[2]. Most of the performance adjustments they suggest involve refactoring the memory access patterns of individual kernels to map more efficiently onto the memory subsystem of a specific architecture. The proposed language extensions specify ways to express parallelism differently from the work-group/work-item language features in OpenCL and several new memory abstractions, including switching between programmer-managed and hardware-managed caches and row/column-major layouts.

3.2 Machine Learning & Compilers

In recent years, machine learning has been studied and used extensively in compiler research for problems such as selecting optimizations and generating heuristics to guide various optimizations. One of the most recent successes is Milepost GCC, a compiler tool that uses machine learning to select the most effective set of compiler optimization flags for individual applications [19]. The authors propose machine learning as a compromise between using standard GCC optimization levels (which significantly improve execution speed but are sub-optimal) and iterative compilation (which exhaustively explores the search space of optimization flags by iteratively combining and improving performance). With over 100 optimization flags available, exhaustively searching (or even using iterative compilation) requires an intractible amount of execution and recompilation; hence, machine-learning is used in Milepost GCC to prune the search space. This framework, now integrated into mainline GCC, extracts features and specializes optimization heuristics per function in order to improve all facets of the compilation process (compilation time, code size & execution time). The framework collects 56 code features pertaining to all aspects of code execution, from simple counts of operations to abstract features such as control-flow graph structure. Using several types of machine learning models, the authors are able to demonstrate a 11% speedup over the default optimization heuristics in GCC in a suite of benchmarks, and using the production-level Berkeley DB library, are able to demonstrate a 17% speedup in execution time, 7% reduction in code size and a 12% speedup in compilation time.

The ability of the machine learning model to make accurate predictions hinges on the collection of a set of representative application features. While humans have an intuitive idea of what features should be selected to describe applications, oftentimes strange and esoteric

[2]This work stressed the importance of adding memory access pattern features to the feature sets

features may give the machine learning model more appropriate information. Leather et al. presented an approach to automatically generate the set of program feature needed to train and evaluate machine learning models to select the unroll factor for loop unrolling [37]. The authors describe the set of all possible program features in a *feature grammar*, which is similar to (and automatically derived from) an application's low-level IR. Features are then automatically generated from the feature grammar using a genetic programming algorithm. Although the up-front cost is significant (the authors mention that it took several days to generate the set of features), the approach yielded models for loop unrolling that achieved 79% of the total performance achieved by an oracle. GCC's heuristic achieved just 3% of the oracle's performance.

3.3 Heterogeneous Scheduling

A significant amount of work aims to efficiently scheduling compute kernels across compute devices in system or even in a cluster. The Merge framework, developed by Linderman et al., is a programming model, compiler and runtime for heterogeneous multi-core systems [42]. The framework utilizes a library-based programming model, whereby users embed architecture-specific compute kernels (denoted as *sequencers*) into regular C source code. During compilation, these architecture-specific kernels are integrated into the application so that the runtime scheduler can select the appropriate architecture during execution. Additionally, the Merge framework API can be used to generate device-specific implementations of a generic kernel. At runtime, the various kernel implementations are mapped to the available hardware, and the results are combined using the traditional map/reduce pattern. Using the framework, the authors are able to achieve up to a 8.5x speedup using an Intel X3000 media accelerator, and a 22x speedup using SMP Intel Xeon processors.

Jiménez et al. developed a scheduling framework for running compute kernels across devices in CPU/GPU systems using a round-robin distribution policy that adapts to performance history of compute kernels on individual devices [32]. The scheduler, which performs scheduling at the function level, schedules architecture-specific (and developer-provided) implementations of compute kernels using a task queue. Compute kernels are scheduled onto devices as the devices become available. Additionally, the runtime records execution history of each compute kernel on each device; if there is a significant performance difference between several architectures, the kernel is no longer scheduled onto the slower device. Using this architecture, the authors are able to achieve consistent speedups of 30% over using the GPU exclusively.

StarPU, presented by Augonnet et al., is another framework for scheduling tasks onto the compute resources in a system [7]. The framework includes a memory management library that uses software caching to minimize the number of data transfers between separate memory spaces in the system. Although the programmer is responsible for providing device-specific implementations of the compute kernel, the framework includes a queue-based scheduler with task priorities that distributes compute kernels to the devices in the system. Several policies, including "greedy" (where processors grab tasks from the queue as soon as possible) and hint-based (where the programmer specifies the affinity of a compute kernel to an architecture using weights) policies are evaluated. The authors demonstrate with the StarPU framework strong speedups, with performance approaching that of a hand-optimized library.

Kim et al. developed the SnuCL framework, a framework that retrofits the OpenCL API into a distributed setting [34]. SnuCL utilizes the data transfer and kernel execution semantics of OpenCL in order to launch compute kernels across the compute devices in a system. All CPU cores and each individual GPU in a node over all the nodes in the system represent the collective set of compute devices available in the cluster. The SnuCL framework includes a set of compiler tools to convert OpenCL to CUDA (for NVidia GPUs) and OpenCL to C (for CPUs) while optimizing for each individual architecture. Additionally, using a distributed runtime (with control threads on each node to supervise data transfer and kernel execution), kernel executions are load balanced across the compute devices in the cluster by distributing work-groups among the devices using a static (every compute device executes the same number of work-groups) and dynamic (devices request new work groups from the host after finishin execution of another work group). They explore the scalability of benchmarks from several suites, demonstrating how clusters can be used to speed up compute kernels.

Several other works use machine learning to make scheduling decisions. Emani et al. developed a technique to automatically adjust the number of OpenMP threads used by the underlying OpenMP implementation in the presence of external workload [17]. The authors argue that oversubscribing threads to CPUs increases scheduler conflict and limits speedups. Instead, the authors perform spatial scheduling (instead of temporal scheduling) to eliminate scheduling conflicts and reduce context-swapping. The authors collect a small set of compute kernel statistics and architecture/system information and train an artificial neural network to adjust the number of threads used by an application at various OpenMP parallel sections. Using this trained model, the authors demonstrate a 50% speedup over adjusting the number of OpenMP threads using a hill-climbing approach.

Grewe et al. present a framework that utilizes machine learning to automatically select the most efficient architecture for a compute kernel [22]. Their framework accepts as input an application parallelized with OpenMP pragmas and generates an application that can launch the compute kernel on the host CPUs (using OpenMP) or a GPU (using OpenCL)[3]. Compute kernels is statically mapped to an architecture using a decision tree trained from a set of applications. The training data used to construct the decision tree is based on a set of combined kernel features extracted from each compute kernel in every application. A source-to-source translator converts the OpenMP compute kernel to OpenCL and applies GPU-specific optimizations (including access & loop reordering and prefetching). At runtime, the application calls into a machine-learning library to use the trained decision

[3]This work strongly influenced the work presented in chapter 4.

tree to select an architecture on which the compute kernel executes. They show that for the NAS Parallel Benchmark suite [8] their framework is able to predict the best architecture for every application. Additionally, they show the performance improvement of their OpenMP-to-OpenCL translation over OpenMPC [39] and SnuCL [34].

3.4 Co-Scheduling

There has been much work in co-scheduling of compute kernels across multiple devices in a system, i.e. dividing the work to be done in a compute kernel across several architectures. Qilin, developed by Luk et al., consists of a programming API and a runtime to adaptively schedule compute kernels in various ratios onto a CPU/GPU heterogeneous system [43]. The API consists of a set of primitives used to divide and map varying chunks of work to the two devices. The work ratio between the two devices is determined and adapted at runtime based on previous runs of the compute kernel, stored in a database. The runtime determines the optimal work ratio by minimizing the expected execution time of the two devices given varying work ratios. Utilizing past history for performance prediction, the authors are able to show a 69% speedup over a CPU-only mapping, and a 32% speedup over a GPU-only mapping.

Scogland et al. presented several co-scheduling approaches for use with higher-level heterogeneous parallel programming models, collectively referred to as "Accelerated OpenMP"[4] [52]. Within this framework, the authors describe a set of compiler refactoring techniques to enable compute kernel execution across a CPU and GPU. Additionally the authors present several runtime schedulers to adaptively adjust the ratio of workload between the devices. The set of schedulers include a static scheduler (which calculates a ratio based on device core counts), a dynamic scheduler (which adjusts the static scheduler in subsequent executions of the same kernel), and several scheduler that adjust the ratio at a finer granularity by splitting a single compute kernel invocation into smaller pieces. Using these various policies, the authors are able to achieve speedups ranging from 1.5x-8x on four benchmarks.

Other research trains machine learning models to decide the workload split among devices. Kofler et al. present a framework for splitting the workload based on compute kernel features and input size [35]. The framework is comprised of a compile-time source-to-source translator (to allow multi-device kernel execution), a compile-time feature extractor, a run-time feature extractor and a run-time co-scheduler that enables model training and evaluation. This framework performs scheduling for multi-CPU/multi-GPU systems by using artificial neural networks (ANN) and support vector machines (SVM). Their approach uses a technique called *greedy feature selection* to select the most effective compute kernel and runtime features from a pool of available features, i.e. the features that serve as the best predictors of performance. The authors demonstrate, using two simple benchmarks, that different work ratios provide

[4]Accelerated OpenMP includes OpenACC [48] and OpenMP 4.0

better performance for different applications.

Finally, Grewe et al. present an offline and machine-learning based technique for dividing a compute kernel among devices in a CPU/GPU system [20]. Their technique involves a two-level prediction model. The first level predicts if the compute kernel should be executed solely on the CPU or solely on the GPU using a binary classifier SVM. The second level again uses an SVM, but to predict the optimal work ratio between the CPU and GPU. This approach follows the same template as other machine-learning based approaches – the authors extract compute kernel features (and use principal component analysis, PCA, to reduce the features set's dimensionality) and train the models using 47 separate applications. The models are then evaluated for their prediction accuracy. By co-scheduling onto the available resources, the authors are able to achieve a 57% speedup over a dynamic approach that breaks the kernel execution into smaller chunks and distributes them on an as-needed basis.

Grewe expanded this work to co-scheduling onto CPUs with integrated GPUs with external workload [21]. In this work, the authors again use machine learning to predict the optimal workload division for the CPU and integrated GPU. However, in this work they also include *contention features* that include the delay in launching a kernel on the GPU (only one compute kernel can be running on the integrated GPU at a time, other kernels are queued up for execution). By taking into account external workload on the GPU, the authors are able to demonstrate speedups of of 54% and 256% over competitors.

Chapter 4

Refactoring & Scheduling OpenMP Applications

The first part of this work addressed refactoring and scheduling of OpenMP applications across the available processing resources in a system. OpenMP [10] is a traditional multicore/shared-memory parallelism model that targets thread-based parallelism in CPUs[1]. Therefore, these OpenMP applications required significant automatic refactoring in order to run on various architectures, including extensive memory management to adapt OpenMP's shared-memory model to handle device-specific memory spaces and conversion from OpenMP to device-specific programming models. These applications were refactored and subsequently analyzed to characterize how parallel *compute kernels*[2] executed. Using a GCC plugin (referred to as the *feature extractor*), compute kernel features were extracted from compute kernels at compile time as a basis for analysis and subsequent scheduling decisions. These features include operations performed on data, memory accesses and control flow. The extracted features from this set of applications were used to build a machine learning model to make scheduling decisions, from the machine learning implementation in OpenCV [57]. At run-time, scheduling decisions were made by a central scheduling daemon (the *scheduler*) that combined program features and system load (or *external workload*) in order make scheduling decisions. Applications sent the scheduler the extracted features via inter-process communication (IPC), which subsequently returned a scheduling decision to the application.

We assumed a master-slave model for our execution model, hereafter referred to as the "co-processor execution model". Figure 4.1 shows how control flows in a refactored application. The left side of the figure, labeled "Host", represents non-compute kernel application processing on a single host CPU, while "Device" represents compute kernel execution on a compute

[1]The newest iteration of the standard includes annotations for executing on heterogeneous architectures, but no stable implementation is available

[2]Parallel work sections in OpenMP are denoted by "`#pragma omp parallel`" pragmas in the source code. In this chapter, we use parallel work sections denoted by this pragma and *compute kernel* interchangeably

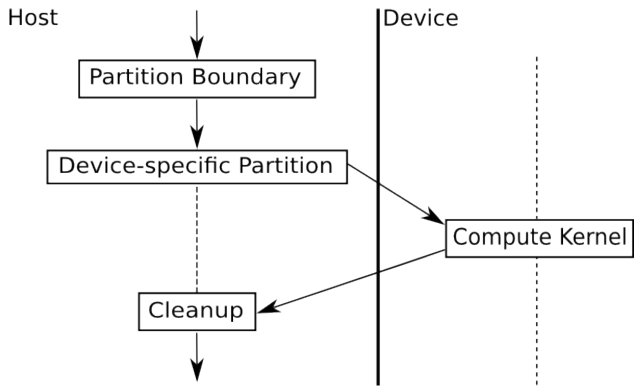


Figure 4.1: Coprocessor Execution Model

device (which could be the host CPUs, depending on the scheduler's decisions). Solid lines indicate that a processor is executing application code while dashed lines indicate that the processor is in a waiting state. The figure shows the flow of control between host and device when launching a compute kernel. In this model, coprocessors act as slaves – they are initialized when the application begins execution, but wait for commands from the host (master). Depending on scheduling decisions, some devices may not be used at all by the application (although other applications can still use the coprocessors). Application execution proceeds as follows – the host initializes all devices and begins executing the application as normal until it reaches the beginning of a compute kernel, called a *partition boundary*. The host requests a scheduling decision from the scheduler, and based on that decision, executes a device-specific *partition* which handles data transfer & execution on a particular coprocessor. This device-specific partition performs several steps:

- Notifies the device that execution is about to begin (not necessary for all devices)
- Transfers all input data to the coprocessor
- Launches the specified kernel on the coprocessor
- Waits for kernel execution to finish, then transfers all output data back to the host

All device partitions follow these steps, albeit in a device-specific order and using device-specific APIs. Once the host has transferred all data back from the device, it continues normal execution of the application while the coprocessor re-enters the waiting state. This continues until either another partition boundary is reached (in which the previous process repeats) or the application finishes execution; when it finishes, all devices are released. This model is similar to how kernels are executed on modern GPUs, with additional steps for runtime scheduling and cleanup of compute kernels.

The rest of the chapter discusses refactoring & scheduling of OpenMP applications and is structured as follows:

- Section 4.1 presents the design of heterogeneous OpenMP execution & scheduling
- Section 4.2 describes how OpenMP applications were refactored for heterogeneous execution & runtime scheduling, how features were extracted from the compute kernels, how applications were scheduled via the scheduler, and how machine learning models were used to make the scheduling decisions
- Section 4.3 evaluates the results of the refactoring & scheduling process

4.1 Design

This work stemmed from the desire to automatically refactor legacy code parallelized with OpenMP to be able to execute on heterogeneous-ISA devices. Once an application was refactored, it could be analyzed in order to be dynamically scheduled onto those devices for the best performance, taking into account compute kernel features and external workload. We chose to target three heterogeneous-ISA devices – an AMD Opteron 6376 16-core CPU, an NVidia Tesla C2075 GPU and a Tilera TILEncore-Gx36 coprocessor. There were several reasons why OpenMP was chosen:

1. OpenMP is a widely supported and has many mature implementations, including the free & open-source GNU OpenMP [15] implementation (GOMP). As such, OpenMP is a proven industry standard which continues to evolve in response to the changing processor landscape.
2. OpenMP is supported on two out of the three listed devices – the Opteron 6376 CPU, and the TILEncore-Gx36 coprocessor.
3. OpenCL [24] is an obvious alternative parallel programming model targeting heterogeneity. However, while OpenCL is widely supported by large vendors, smaller vendors may not provide an implementation for their architectures. In particular, our decision to target OpenMP applications arose from the lack of OpenCL support for the TILEncore-Gx36 coprocessor; using OpenCL would require implementing an entire OpenCL runtime, a daunting task.

Therefore, we selected OpenMP as our parallel programming model of choice. The main issue with using OpenMP is that there is no implementation available that allows heterogeneous execution; that is, there is no implementation that allows launching compute kernels on separate devices from the host. As mentioned above, OpenMP 4.0 [10] provides an API for executing compute kernels heterogeneously using OpenMP `target` pragmas, but even though this new standard has been available since since July 2013 there is no stable implementation available[3] (the Portland Group's commercial PGI compiler [25] supports an OpenMP 4.0-like model named OpenACC). Additionally, the standard requires programmers to utilize additional data movement pragmas to specify what data must be sent to and from the compute device to support the kernel. We wanted a solution that transparently handled data movement so that developers could ignore tedious and error-prone data movement code and instead focus on application logic. We decided to develop our own tool to satisfy these requirements.

While OpenMP is implemented by GCC on x86-64 and the TILEncore-Gx36 coprocessor, there is no OpenMP implementation for GPU architectures, which instead rely on single work-item compute kernel descriptions such as OpenCL and CUDA. Lee et al. produced a research tool called OpenMPC [40] [39], built on the Cetus compilation framework [38], that converts OpenMP parallel sections into CUDA source code, suitable for compilation and execution on NVidia GPUs. Rather than attempting to implement our own OpenMP-to-CUDA or OpenMP-to-OpenCL source-to-source translator, we chose to utilize this tool to convert the compute kernels of our benchmark applications into CUDA.

Despite the availability of OpenMPC, applications still required significant refactoring to transparently handle execution and runtime scheduling onto the various coprocessors. We implemented a tool in order handle this application refactoring, which achieved several design goals:

1. Create device-specific partitions that handle execution of compute kernels for a particular device, including any necessary data transfers and kernel launches using device-specific APIs.
2. Track memory allocation & deallocation on the host-side so that data could be kept coherent on both the host & device, i.e. data transfers between host & device could be handled transparently by the compiler and a runtime memory management library.
3. Perform runtime scheduling of compute kernels across heterogeneous architectures by communicating with a centralized scheduler via a client-side scheduling library.

Once applications were refactored to enable runtime scheduling, we needed a mechanism to make the scheduling decisions themselves. Machine learning was chosen because of its recent successes in providing good decision making ability (especially in the related work discussed in chapter 3). In order to use machine learning, several additional design goals were established:

[3]Near the end of this work, Liao et al. published an initial OpenMP 4.0 implementation targeting NVidia GPUs & CUDA [41]

1. Compute kernels needed to be characterized. This was accomplished using a feature extractor tool, which quantified the types of operations being performed by the compute kernel. The feature extractor was implemented as a GCC plugin.
2. A machine learning model needed to be constructed from the extracted features. This was done using the machine learning implementation provided by OpenCV [57], an open-source computer vision library that provides generic machine learning algorithms.

There were several reasons why we chose to implement the feature extractor as a GCC plugin. First, GCC has a built-in, mature OpenMP implementation (while other common free and open source compilers, such as Clang/LLVM, do not). Although applications were being refactored to various parallel models for execution, features were extracted from the original source of the compute kernels which were written using OpenMP. In this way, the feature extractor could easily leverage GCC's OpenMP support to characterize kernels.

Another reason that GCC was chosen emerged from the problem of accurately characterizing compute kernels. We chose to incorporate profiling to obtain exact statement execution counts from the compute kernels (although others, such as [22], do not). GCC provides more functionality in this regard as well – it includes support for edge profiling of applications. This means that GCC can instrument the control flow graph of the functions in an application with counters so that a profiling run of the application records the exact number of times a basic block is executed. While GCC uses this capability to inform optimization heuristics, we leveraged this in order to accurately count the number of times a given statement is executed, and scale the features extracted from that statement accordingly. Note that using profiling is in general not portable as the features extracted correspond to a specific instantiation of the compute kernel; subsequent kernel executions could have vastly different features, depending on control flow, input data, etc. For our purposes, however, this profiling-based approach was sufficient as every run of a given benchmark used the same input and therefore resulted in the exact same features[4].

Finally, we needed a way to make scheduling decisions at runtime for applications. If applications were to be scheduled without considering external load, then no central scheduler was required and applications could simply evaluate the ML models individually. However, we wanted to incorporate external workload information into the scheduling decision. Because of this, we needed a central repository for external workload information, as there was no clean & easy way of querying workload on either the Tesla C2075 or the TILEncore-Gx36. Therefore, we needed a central scheduler to make scheduling decisions – applications would send the scheduler the set of features extracted by the feature extractor, and the scheduler (combining those features and external workload data) would make a scheduling decision. There were several design decisions that needed to be made in order to implement the scheduler:

[4]The second iteration of the feature extractor for OpenCL kernels (discussed in section 5.2.2) relaxed this constraint by using estimates to scale feature counts

1. How applications communicated with the scheduler
2. How the scheduler maintained external workload information

We considered two possible implementations for the scheduler: either as a kernel module (accessed via system calls), or as a scheduling daemon. We chose to implement it as a daemon, mainly for ease of use. Implementing the scheduler as a daemon eliminated a large class of errors that arise from kernel programming. In general, it was easier to iteratively develop the scheduler – it did not require continual unloading & reloading of a module for minute changes and there was better debugging visibility. Additionally, there was automatic synchronization by construction in the daemon. The scheduler was developed as a client-server architecture; applications called a client-side library which used IPC to communicate with the server. Using IPC allowed the scheduler to leverage a well-tested kernel capability and mandated a first-in-first-out ordering. While a scheduling daemon potentially incurred more overhead (several context switches between the application & scheduler, and between user-space/kernel-space), the ease of use afforded by a daemon meant more time evaluating the scheduling decisions and less time debugging. Section 4.3 discusses the overheads of this approach.

Additionally, we needed a way to maintain external workload state of the system, since it was not readily available for the coprocessors. We chose the simple approach of maintaining run-queues for each device in the system. Applications checked-in with the scheduler to request a scheduling decision and after execution of a compute kernel finished. In this way, the server maintains up-to-date information regarding the number of applications running on a given architecture. We could have potentially maintained the features of applications running on various architectures, because intuitively applications co-located on a given device could affect each other in many ways. However this would require extensive bookkeeping; we instead chose to keep the scheduler as lean as possible, spending time on scheduling decisions rather than updating internal state.

Finally, we needed a machine-learning algorithm to construct and evaluate the generated models. Because we used OpenCV as the core machine-learning driver, we could switch between implementations fairly easily. We tested both Artificial Neural Networks (ANN) and Support Vector Machines (SVM) and empirically determined that ANNs were a better match.

4.2 Implementation

There were several tools that needed to be developed to enable refactoring & scheduling:

1. A partitioner to refactor OpenMP applications for heterogeneous execution (4.2.1)

2. A feature extractor to characterize compute kernels (4.2.2)
3. A scheduling daemon that communicated with applications and made scheduling decisions using previously generated machine learning models (4.2.3 & 4.2.4)

4.2.1 Refactoring Applications: The Partitioner Tool

A partitioning tool (hereafter referred to as the *partitioner*) based on the ROSE source-to-source compilation framework [36] was developed to achieve the refactoring design goals. ROSE is a free source-to-source compilation & transformation framework developed by Lawrence Livermore National Laboratory to aid in compiler research by exposing the compiler internals for manipulation by translator passes; the partitioner is designed as a set of these translator passes. The partitioner consumes as input a set of C source files with OpenMP pragmas denoting the compute kernels and produces a set of C source files refactored from the original source such that they:

- Track all memory management
- Contain partition boundaries and partitions for launching compute kernels on devices in the system
- Contain device-specific code to be compiled for each device
- Interact with the scheduler to allow runtime scheduling onto the various architectures

The tool is structured as a series of passes, generally categorized into *analysis* and *refactoring* phases:

1. *Analysis* – determines if the kernel is legally able to be executed on specific device, and if so, what declarations and data are needed for execution.
2. *Refactoring* – generates the device-specific partitions, inserts calls to track memory in the application and inserts scheduling calls.

In general these passes operate on functions in an application's call graph using a post-order traversal (with special handling for loops). At the top level, a function containing an `#pragma omp parallel` pragma is designated as a compute kernel. The analysis phase investigates the compute kernel (and all functions called from within the compute kernel), collecting information. The refactoring phase changes the compute kernel as described above, and rewrites other parts of the application where necessary. Information is stored between passes in a `popcorn`[5] pragma - individual passes write specific clauses to this pragma. Because the partitioner is built as a source-to-source compiler, it works at the application source-code level and requires that the application be re-compiled after refactoring.

The following sections describe the two analysis and three refactoring passes within the partitioner.

Analysis – Find Compatible Architectures

This pass determines on which coprocessors a compute kernel can legally execute. Table 4.1 shows which characteristics are legally allowed on each architecture. Intuitively, the host is able to run all compute kernels, as the compute kernels were originally written to execute on this architecture. Because the TILEncore-Gx36 runs an operating system (and therefore has a wider capability than most GPUs), it is able to accommodate more types of compute kernels than most GPUs. However, there are many problems that arise from the complexity of refactoring OpenMP applications; in general, this pass attempts to preclude problematic kernels from refactoring.

	Undefined Functions	Higher-Order Pointers	Dynamic Memory	Undefined/ Complex Classes
Host (x86_64)	Yes	Yes	Yes	Yes
GPU	No	No	No	No
TILEncore-Gx36	No	No	Yes	No

Table 4.1: Compute kernel characteristics allowed on various architectures

In general, undefined functions (such as those contained in shared libraries) are not allowed on coprocessors because the partitioner must be able to include source code for any called function in a device's partition for re-compilation; this is an artifact of source-to-source refactoring. Notable exceptions to this rule are library math functions (with implementations available on NVidia GPUs and the TILEncore-Gx36) and standard library functions on the TILEncore-Gx36. Dynamic memory management (while generally not performed inside of compute kernels) is not allowed on GPUs but is allowed on the TILEncore-Gx36. Similarly to undefined functions, any undefined classes[6] (i.e. opaque class definitions) are not allowed on coprocessors because their definitions are not available for recompilation (the compiler cannot generate a memory layout for the structure or class).

[5]This work was originally part of the System Software Research Group's Popcorn Linux project [23]. The `popcorn` pragmas are an artifact from this initial goal, although we have since deviated from integration into Popcorn Linux.

[6]The partitioner tool accepts C source, not C++, as input; therefore there is no class language construct for these applications. However, ROSE is a C/C++ compiler; because of this it lumps structures and classes into the same representation in the AST. We will use the ROSE terminology throughout.

```
struct Mat {
        int** data;
        int x, y;
};
```

(a) Class containing lower-order pointers

```
struct Mat3 {
        int*** data;
        int x, y, z;
};
```

(b) Class containing higher-order pointers

Figure 4.2: Valid & invalid classes for the partitioner

Complex classes are also not supported on coprocessors, due to implementation limitations rather than any fundamental issue. Because the partitioner must transfer data to and from devices, it must be able to describe individual class objects to enable serialization. If all fields of a class are contiguous in memory, i.e. the class does not contain any pointers, then the partitioner can generate the necessary information to transfer the object between the host and coprocessor – for the GPU, it can simply hand the compiler the class' definition, and for the TILEncore-Gx36 it can generate a custom MPI datatype for use at runtime in MPI library functions. However, if the structure contains pointers, then its individual fields must be sent separately owing to the fact that the location and size must be resolved at runtime. Because so many compute kernels use structures with pointers, support was added such that pointers-to-pointers could be serialized (shown in listing 4.2a). However, if the structure contains pointers of order greater than three (shown in listing 4.2b), the compute kernel cannot be executed on the coprocessor. While this could be implemented, the benchmarks we used did not have higher-order pointers. Additionally, it could be argued that the runtime cost of serializing arbitrary levels of referencing could potentially outweigh any benefit obtained from executing on the coprocessor, as there would have to be several calls to the memory management library to serialize the structure and regenerate it on the coprocessor. Similarly, structures within structures are not currently supported in the partitioner.

```
#pragma popcorn compatibleArch(<architectures>)
void matMul(int dim, float *C, const float *A, const float *B) {
        ...
}
```

Figure 4.3: Pragmas generated by the "Find Compatible Architectures" pass

The results of this pass are stored in a pragma as shown in listing 4.3, and can contain the values `x86`, `gpu` and `mpi`, corresponding to the host, the Tesla C2075 and TILEncore-Gx36, respectively.

Analysis – Kernel Interface

This pass is responsible for gathering all information required for execution of a compute kernel on a coprocessor. In general, this pass must gather all declarations and definitions that are required for the kernel to be compiled for a specific device, and it must determine all inputs and outputs of the kernel. It gathers the following particular pieces of information from the compute kernel:

- *Inputs* – all the inputs for the compute kernel. This includes all formal arguments to the function containing the kernel.
- *Global Inputs* – all global variables that are read within the compute kernel.
- *Outputs* – all outputs from the kernel. This includes writes to any formal arguments that are pass-by-reference (i.e. any side-effects) and return values.
- *Global Outputs* – any writes to global variables. These changes must be visible in the host's memory space.
- *Functions* – any functions called within the compute kernel. Note that the definitions of these functions must be available for analysis and recompilation.
- *Classes* – class definitions of any objects used within the compute kernel.

The partitioner assumes that any variables passed as actual arguments to a function called within the compute kernel are being written within that sub-function (the partitioner does not do interprocedural analysis). This means that a variable passed by reference as a formal argument to the compute kernel which is subsequently passed to another function from within the compute kernel is considered to have side-effects and assumed to be an output. Also, note that any inputs, outputs and definitions needed by a function called from within the compute kernel are by necessity needed by the compute kernel itself. In other words, the kernel interface is dictated by the needs of the compute kernel and any functions called by the compute kernel.

The results of this pass are stored in pragmas, as shown in listing 4.4 (note: it does not show output from the first pass). Possible values include the symbol names of the variables, functions and classes discovered by the pass.

Refactoring – Partition Kernels

This pass is responsible for the bulk of the work in refactoring the code for runtime scheduling. It starts by loading an empty template for the device side of each architecture's partition, which will eventually be populated with compute kernels and accompanying class/function

```
#pragma popcorn inputs(<variables>)
#pragma popcorn globalInputs(<variables>)
#pragma popcorn outputs(<variables>)
#pragma popcorn globalOutputs(<variables>)
#pragma popcorn functionsNeeded(<functions>)
#pragma popcorn classesNeeded(<classes>)
void matMul(int dim, float *C, const float *A, const float *B) {
        ...
}
```

Figure 4.4: Pragmas generated by the "Kernel Interface" pass

definitions. While the GPU template is largely empty, the TILEncore-Gx36 partition requires some startup code for initialization. We chose OpenMPI [49] as our means of architecture-specific communication for the TILEncore-Gx36 because the coprocessor came with software to create a network bridge-over-PCIe between the host and device making MPI an attractive means for data transfer. Additionally, OpenMPI enables heterogeneous message passing, solving a myriad of ABI issues that arise when transferring data between ISA-heterogeneous architectures (such as x86 and the Tilera ISA). Thus we could leverage the stable and mature OpenMPI implentation of MPI.

Listing 4.5 shows an excerpt from the template TILEncore-Gx36 code. The application performs normal MPI initialization, then enters a command loop that waits for signals from the host. The host either sends a function code that signals the TILEncore-Gx36 to begin execution of a specific compute kernel (via the switch statement in the main loop), or sends a finish command which tells the TILEncore-Gx36 to cleanup and exit. Details about the device-side compute kernel are discussed below. Note that while this was implemented on the TILEncore-Gx36, this execution model could be used to launch compute kernels on any device that supports MPI.

The next step for this pass is to begin refactoring the host-side compute kernel. Listing 4.6 shows an excerpt from the result of refactoring a matrix multiply kernel. The original compute kernel is converted into a partition boundary that is used as a scheduling point for the compute kernel, while the compute kernel source code is moved to a separate function (`matmul_x86` in this example). Note that actual calls to the scheduler are inserted in a separate pass. Functions are generated for each device – on the GPU, a function which encapsulates GPU memory management & kernel launches is created in a separate source code file (named `kernels_gpu.c`) for separate compilation by NVidia's compiler. For the TILEncore-Gx36, a function containing MPI_Send & MPI_Recv calls is inserted into the host application source near the partition boundary. The device-side code for the TILEncore-Gx36 is added to a separate file (named `kernels_mpi.c`, which contains the template source code in listing 4.5) for compilation by Tilera's compiler. Within the partition boundary, a

```
int main(int argc, char** argv) {
        int finished = 0;
        MPI_Status __mpi_status;
        MPI_Init(&argc, &argv);

        while(!finished) {
                int function_code = INT_MAX;
                MPI_Recv(&function_code, 1, MPI_INT, HOST, 1,
                        MPI_COMM_WORLD, &__mpi_status);

                switch(function_code) {
                        case ...
                        case FUNCTION_CODE_END:
                                finished = 1;
                }
        }

        MPI_Finalize();
        return 0;
}
```

Figure 4.5: TILEncore-Gx36 initialization & wait loop (device-side)

```
void matMul(int dim, float *C, const float *A, const float *B) {
        int __partition_num = 0;

        /* Call to scheduler added in later pass */

        switch(__partition_num) {
        case 1:
                matMul_mpi(dim, c, a, b);
                break;
        case 2:
                matMul_gpu(dim, c, a, b);
                break;
        default:
                matMul_x86(dim, c, a, b);
                break;
        }
}
```

Figure 4.6: Result of partitioning a matrix-multiply kernel (host-side)

switch statement is inserted that switches to the different partitions based on the return value from a call to the scheduler. At this point, the application is able to call different functions which correspond to launching the compute kernel on a given device; however, there is still signficant work to be done.

The partitioner then consumes information generated by previous passes, and in particular, the kernel interface pass. It reads the `inputs` & `globalInputs` pragmas to generate a list of variables that must be transferred to the device, and the `outputs` & `globalOutputs` pragmas to generate a list of variables that must be transferred back to the host. In addition, it gathers the function & class definitions required by the kernel, stored in `functionsNeeded` & `classesNeeded` clauses. The partitioner can then begin device-specific code generation.

For the GPU, the partitioner copies all necessary function & class definitions into `kernels_gpu.c` so they are visible for compilation. In addition, it copies the compute kernel source into a function, representing the GPU's partition, to be transformed into CUDA (called `matMul_gpu` in this example). This entire source file is fed as input to OpenMPC, which generates all required GPU dynamic memory management (including allocation/deallocation & data transfers) and the compute kernel specification in CUDA. This file is given to `nvcc` and linked against the original application as an implementation of the compute kernel for GPUs.

Refactoring to provide an implementation of the compute kernel for the TILEncore-Gx36 is more complex – listings 4.7 and 4.8 show an example for matrix multiplication. The partitioner again copies all necessary function & class definitions into the device-side source

```
void matMul_mpi(int dim, float *C, const float *A, const float *B) {
        MPI_Status __mpi_status;
        int __func_num = 1UL;
        MPI_Send(&__func_num, 1, MPI_INT, DEVICE, 1,
                MPI_COMM_WORLD);
        unsigned long A__size = get_size(A) / sizeof(const float);
        MPI_Send(&A__size, 1UL, MPI_UNSIGNED_LONG, DEVICE, 1,
                MPI_COMM_WORLD);
        MPI_Send(A, A__size, MPI_FLOAT, DEVICE, 1,
                MPI_COMM_WORLD);
        unsigned long B__size = get_size(B) / sizeof(const float);
        MPI_Send(&B__size, 1UL, MPI_UNSIGNED_LONG, DEVICE, 1,
                MPI_COMM_WORLD);
        MPI_Send(B, B__size, MPI_FLOAT, DEVICE, 1,
                MPI_COMM_WORLD);
        unsigned long C__size = get_size(C) / sizeof(float);
        MPI_Send(&C__size, 1UL, MPI_UNSIGNED_LONG, DEVICE, 1,
                MPI_COMM_WORLD);
        MPI_Send(C, C__size, MPI_FLOAT, DEVICE, 1,
                MPI_COMM_WORLD);
        MPI_Send(&dim, 1UL, MPI_INT, DEVICE, 1,
                MPI_COMM_WORLD);
        MPI_Recv(C, C__size, MPI_FLOAT, DEVICE, 1,
                MPI_COMM_WORLD, &__mpi_status);
}
```

Figure 4.7: TILEncore-Gx36 partition (host-side)

```
void matMul_tilera(void) {
        MPI_Status __mpi_status;
        unsigned long A__size;
        const float *A;
        MPI_Recv(&A__size, 1UL, MPI_UNSIGNED_LONG, HOST, 1,
                MPI_COMM_WORLD, &__mpi_status);
        A = (const float*)(malloc(sizeof(const float) * A__size));
        MPI_Recv(A, A__size, MPI_FLOAT, HOST, 1,
                MPI_COMM_WORLD, &__mpi_status);
        unsigned long B__size;
        const float *B;
        MPI_Recv(&B__size, 1UL, MPI_UNSIGNED_LONG, HOST, 1,
                MPI_COMM_WORLD, &__mpi_status);
        B = (const float *)(malloc(sizeof(const float) * B__size));
        MPI_Recv(B, B__size, MPI_FLOAT, HOST, 1,
                MPI_COMM_WORLD, &__mpi_status);
        unsigned long C__size;
        float *C;
        MPI_Recv(&C__size, 1UL, MPI_UNSIGNED_LONG, HOST, 1,
                MPI_COMM_WORLD, &__mpi_status);
        C = (float *)(malloc(sizeof(float) * C__size));
        MPI_Recv(C, C__size, MPI_FLOAT, HOST, 1,
                MPI_COMM_WORLD, &__mpi_status);
        int dim;
        MPI_Recv(&dim, 1UL, MPI_INT, HOST, 1,
                MPI_COMM_WORLD, &__mpi_status);

        ... OpenMP compute kernel ...

        MPI_Send(C, C__size, MPI_FLOAT, HOST, 1,
                MPI_COMM_WORLD);
}
```

Figure 4.8: TILEncore-Gx36 compute kernel (device-side)

code file, `kernels_mpi.c`. The compute kernel is copied to the device-side application, but is augmented to handle data transfers. First, the partitioner assigns the compute kernel a function number that the host sends to the device to signal which compute kernel should be executed by the TILEncore-Gx36; this number is simply a running counter for the number of compute kernels that have been partitioned onto the device, and is incremented for each compute kernel moved to `kernels_mpi.c`. Then, the partitioner inserts calls to `MPI_Send` & `MPI_Recv` on the host & device, respectively, to transfer all inputs to the device. For data whose size is known at compile-time, the partitioner inserts transfer calls with the size hardcoded into the call site. For data whose size is unknown at compile-time (e.g. anything allocated on the heap) the partitioner inserts calls to a memory management runtime library to get the size of the data (seen as calls to `get_size` in figure 4.7). It then inserts calls to send the size of the data to the device, which receives the size and allocates memory for the data. Finally it sends the data to the device via another `MPI_Send`/`MPI_Recv` pair. For outputs, the reverse process occurs – the device sends data to the host through `MPI_Send` & `MPI_Recv`, respectively. Finally, the partitioner inserts calls to free the memory previously allocated on the device side; the device-side application then returns to the wait loop to wait for another command from the host. If at least one compute kernel is to be launched via MPI, the partitioner must add several things to the host application. First, it adds calls to `MPI_Init` and `MPI_Finalize` in the main function so that the MPI runtime initializes and exits normally. Additionally, a custom MPI datatype is generated (using the `MPI_datatype` capability to dynamically specify data types) for each class that does not contain pointers and is used in the compute kernel; classes that contain pointers are serialized and sent manually by the partitioner. This new datatype is inserted into both the host application and `kernels_mpi.c`. Finally, the partitioner inserts code in the host application that sends the exit signal (the value `FUNCTION_CODE_END` in figure 4.5) which notifies the device that it should exit.

Refactoring – Add Memory Tracking

This pass is responsible for adding calls to track memory as it is allocated & deallocated within the application. This pass only needs to refactor host-side code; device memory management strictly depends on what is allocated on the host.

The memory management library is used to store allocated memory locations and their sizes. During memory allocation, a pointer to the start of the data and the size of that data are passed to the library either explicitly or implicitly. If the data is allocated on the stack, the partitioner inserts calls to `register_pointer` within the scope in which that variable is live. If the data is allocated on the heap, the data is tracked implicitly using a link-time trick. By using the `-wrap <func-name>` command-line argument, the linker to replaces all calls to `<func-name>` with calls to `__wrap_<func-name>`. The memory management library wraps calls to `malloc`, `calloc` and `realloc` with simple stubs that internally calls the associated memory management function and records the results via `register_pointer`

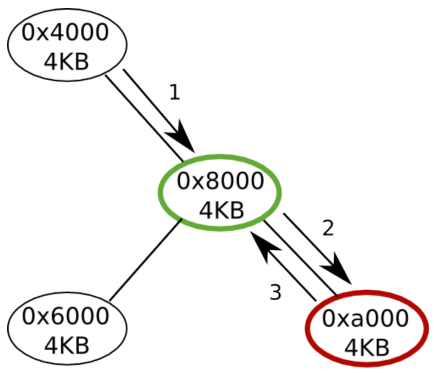


Figure 4.9: Example call to `get_size` and the red-black tree modification

before returning the pointer to the application. De-allocation is similar; the partitioner inserts calls to `unregister_pointer` when exiting the live scope of data in static memory, and a wrapped `free` stub calls `unregister_pointer`. for dynamic memory. Internally, the library tracks data using a modified red-black tree that solves problems created by the flexibility of pointers. While the data lives in a single location in memory, applications can use pointers to arbitrary locations within a memory block. For example, consider a recursive implementation of binary search. The function might contain pointer arguments for the low- and high-part of the range of data being searched. Depending on what value is being searched for, either the low or high pointer is updated and the function is called again. While these pointers are simply pointing to different parts of a single piece of data, the compiler cannot in general connect the pointers to the piece of data they are accessing (alias analysis is potentially unsolvable at compile time). The modification solves this problem by the fact that pointers to any location in memory within the range of a registered datum's specified size can be linked to that piece of data. Therefore when looking up sizes of data via the `get_size` API, the red-black tree is first searched for the specified memory location of the pointer. If the pointer is not found, the immediate predecessor of the previously checked leaf node (in the ordering imposed by the structure of the tree) is examined, and if the pointer is within the data's registered memory range, that datum's size is returned. Consider the sub-tree of a red-black tree in figure 4.9 being searched by a call `get_size(0x8800)`.

1. The node representing data at 0x4000 is searched, and the right child is taken.
2. The node representing data at 0x8000 is searched, and the right child is again taken.
3. The leaf node representing data at 0xa000 is searched, and it is determined that location 0x8800 is not in the tree. The search traverses back to the immediate predecessor of the leaf node (in this case, the node representing data at 0x8000).
4. The pointer is determined to be within that datum's range, and 4kB is returned.

```
int main(int argc, char** argv) {
        register_mm_wrappers();
        atexit(unregister_mm_wrappers());
        ...
}
```

Figure 4.10: Registering & unregistering the memory management runtime library

Finally, `get_pointer` can be used to get the correct version of a pointer (i.e. convert a pointer to the middle of some data to the head of that data) using the preceding modification.

This pass iterates through every function in the application and adds explicit tracking for the following scenarios:

- Array types allocated on the stack
- Scalar data that has its address taken (and thus can have side effects if the generated pointer is passed to a function)

For data on the stack, calls to `register_pointer` are added at the beginning of the functionin which they are live, and calls to `unregister_pointer` are added at every return point, since those pieces of data are only live for the duration of that function invocation. For dynamic memory functions, calls to `malloc`, `calloc`, `realloc` and `free` are wrapped using the linker wrapping mechanism. The wrapping mechanism swaps calls to the wrapped function (e.g. `malloc`) with calls to the wrapped function (e.g. `__wrap_malloc`). The wrapped versions of the functions register (for allocation functions) and unregister (for `free`) the data before calling the real version of the functions.

The last step of this pass is to insert initialization and teardown code to ensure that the memory management library does not interfere with C standard library process creation & teardown. Initially the memory management library is disabled, meaning calls to any of the memory management routines simply pass through the wrappers. At the start of main, the wrapper is enabled and a call is registered to be executed before teardown that disables the library. Listing 4.10 shows an example of this process.

Refactoring – Add Scheduler Calls

The final pass in the partitioner hooks the application into the scheduler, which is discussed in more detail in section 4.2.3. In each partition boundary, it inserts a call to `select_implementation`, an application-facing API from the client-side scheduling library. This call passes a set of compute kernel features (extracted using the feature extractor described in section 4.2.2) via IPC to the scheduler daemon. The daemon combines these features with external workload data and returns a device decision to the application (the return value from the call to `select_implementation`). This pass parses the feature files generated from the feature extractor and initializes a features object with the generated values. It also generates some runtime-only features, including the amount of data being transferred between the host & device, and the number of parallel work items (equivalent to the number of loop iterations for a `parallel omp for` loop). Finally, it inserts the call to the client-side library. Once this call returns a value, the application switches to the appropriate partition and executes the compute kernel on the associated device. After the compute kernel finishes and control is returned to the application, the application notifies the scheduler (with a call to `cleanup_kernel`) so that the scheduler can update its external workload statistics accordingly. Figure 4.11 shows the results from this pass for the matrix-multiply example.

At this point, the partitioner has finished refactoring the application. The GPU partition (completely self-contained in `kernels_gpu.c`) is run through OpenMPC to generate CUDA code. Individual partitions are then compiled for their device using a combination of `gcc` for x86 & TILEncore-Gx36 and `nvcc` for NVidia GPUs. This binary is launched normally if the TILEncore-Gx36 is not used or via `mpiexec` to execute on the TILEncore-Gx36.

4.2.2 Extracting Features from OpenMP Kernels

In order to use machine learning models to perform heterogeneous scheduling, it was necessary to characterize compute kernels based on what types of operations the kernels executed in addition to the context in which they were executed (i.e. the external workload on the co-processors in addition to the current application). The feature extractor was developed and implemented as a plugin pass for GCC to capture the former, while the latter was collected at runtime by the scheduler (discussed in section 4.2.3).

The pass implementing the feature extractor was inserted into GCC's internal pass scheduler after all optimizations had been performed on GCC's SSA-GIMPLE intermediate representation [14] so that the feature counts closely represented the code ultimately generated by GCC. Features were extracted on a per-function basis; the feature extractor iterated through every basic block in a function's control flow graph, and through each statement within the basic block. Table 4.2 shows the features extracted from the applications.

There are two categories of features collected for the machine learning model – kernel features and external workload features. Kernel features are features that are specific to a given compute kernel. These consist of the types of operations a compute kernel executes (features 1 - 9), the amount of data required by the kernel (features 10 & 11) and the amount of parallel work available within the compute kernel (feature 12). Features 1-9 were extracted by the feature extractor and scaled by a profiling run, while features 10-12 were collected during the profiling run. Note that during the refactoring phase of the partitioner, it reads the output from the feature extractor and adds them to the partition boundary, as mentioned

```
void matMul(int dim, float *C, const float *A, const float *B) {
        int __partition_num = 0;
        static int __init = 0;
        static kernel_features __kernel_features;
        if (!__init) {
                /*
                 * Compiler-extracted features are set only
                 * once because they do not change during
                 * execution
                 */
                ... compiler-extracted kernel features ...
                __init = 1;
        }
        __kernel_features.memory_tx = 0;
        __kernel_features.memory_rx = 0;
        __kernel_features.memory_tx =
                __kernel_features.memory_tx + get_size(A);
        __kernel_features.memory_tx =
                __kernel_features.memory_tx + get_size(B);
        __kernel_features.memory_tx =
                __kernel_features.memory_tx + get_size(C);
        __kernel_features.memory_tx =
                __kernel_features.memory_tx + 1;
        __kernel_features.memory_rx =
                __kernel_features.memory_rx + get_size(C);
        __kernel_features.work_items = dim;
        __partition_num = select_implementation(&__kernel_features);
        switch(__partition_num) {
        case 1:
                matMul_mpi(dim, c, a, b);
                break;
        case 2:
                matMul_gpu(dim, c, a, b);
                break;
        default:
                matMul_x86(dim, c, a, b);
                break;
        }
        cleanup_kernel(&__kernel_features);
}
```

Figure 4.11: Partition boundary with features & scheduler calls

#	Kernel Feature	Description
1	*num_instructions*	Number of instructions
2	*int_ops*	Number of integer math operations
3	*float_ops*	Number of floating-point math operations
4	*boolean_ops*	Number of bitwise boolean operations
5	*load_ops*	Number of memory load operations
6	*store_ops*	Number of memory store operations
7	*func_calls*	Number of function calls
8	*intrinsic_math_ops*	Number of intrinsic math operations
9	*cond_branches*	Number of conditional branches
10	*memory_tx*	Number of bytes transferred to device
11	*memory_rx*	Number of bytes transferred back from device
12	*work_items*	Number of parallelizable work items

Table 4.2: Compute kernel features collected from OpenMP applications

in listing 4.11.

We chose these features because they represent the major components of any modern microprocessor. The integer, floating-point and boolean operations represent the major computing functional units in a processor – different processors have different numbers of functional units to handle these types of operations, resulting in differing compute throughputs (built-in functions stress the transcendental functional units of processors, or the software implementations in the absence of those units). The loads and stores represent, at a very basic level, the memory hierarchy of a system. Different processors access memory in different ways and have different memory bandwidth. Function calls and conditional branches represent the processor's ability to mitigate the impact of control flow and divergence in an application. Finally, the last features represent the runtime's efficiency in handling data transfers and the differing levels of parallelism in the applications. Thus, these features allow the machine learning model to evaluate the mapping of compute kernel to architecture based on what the compute kernel is actually executing.

4.2.3 Heterogeneous Scheduling Daemon

As mentioned in section 4.1, the scheduler was implemented as a daemon process that waits in the background for applications to request scheduling decisions. On startup, the scheduler opens up a Unix socket for IPC using the file `/var/run/het_sched.sock` and waits for incoming connections. When an application wants to request a scheduling decision, it connects to the server through the Unix socket file and sends a message containing the extracted features to the server, which makes a scheduling decision and notifies the client of

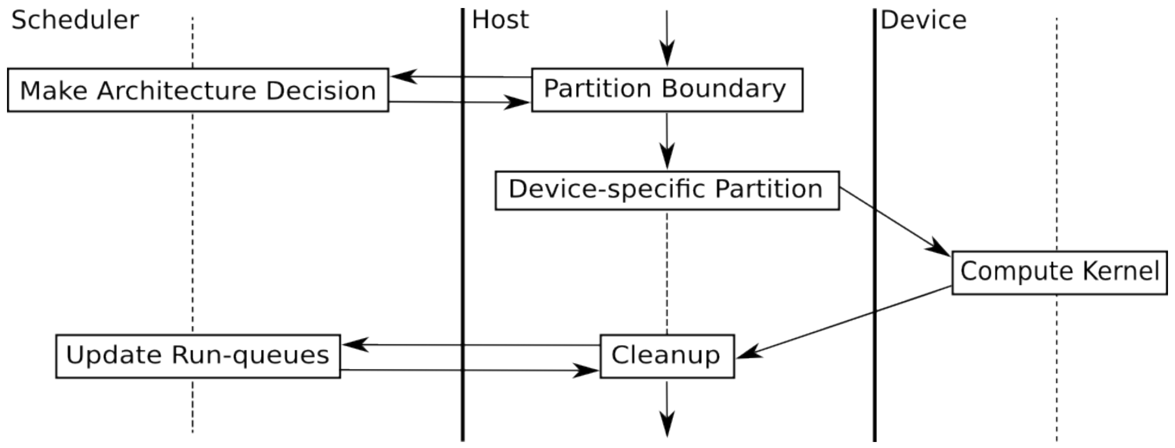


Figure 4.12: Scheduling compute kernels

the decision in a return message. The connection is then closed and the server re-enters the waiting state. Similarly, when a compute kernel finishes execution it re-opens the connection to the server and sends a message that notifies the server that it has finished execution on a particular architecture. The server updates its internal accounting information and closes the connection, re-entering the waiting state. The daemon exits either when it receives a message instructing the server to exit, or receives the exit signal `SIGINT`. Figure 4.12 shows the general flow of scheduling for the refactored OpenMP applications.

The server is responsible for maintaining external workload state of the system, since there is no simple way to query current load from either the Tesla C2075 or the TILEncore-Gx36. Thus the scheduler maintains the external workload features for the machine learning model with cooperation from the applications themselves. External workload features, shown in table 4.3, are the features that represent the current state of the system in regards to workload on the individual compute resources. Feature 1 (*load_avg*) represents the load average of the system as reported by Linux in `/proc/loadavg`. From the Linux manual page for `proc`, this number represents "the number of jobs in the run queue (state R) or waiting for disk I/O (state D) averaged over 1, 5, and 15 minutes" – we use only the load averaged over the past minute. Features 2-4 are internal run queues for each of the architectures in the system maintained by the scheduler. Applications who request a scheduling decision for a compute kernel (via the client-side API `select_implementation`) are added to the run queue for the architecture on which they are scheduled. When the compute kernel completes, the application notifies the scheduler (via `cleanup_kernel`), which subsequently removes the application from the appropriate run queue. If an application does not want a scheduling decision from scheduler but prefers to make its own scheduling decisions, it can notify the server using a call to `notify_server`, so that the server can still make decisions based on accurate external workload information (these applications should also still call `cleanup_kernel` following compute kernel execution). Finally, applications can request the run-queues from the server using a call to `get_table`.

#	External Workload Feature	Description
1	*load_avg*	Average host load
2	*runq_x86*	Number of kernels running on the host (Opteron 6376)
3	*runq_gpu*	Number of kernels running on the Tesla C2075
4	*runq_tilera*	Number of kernels running on the TILEncore-Gx36

Table 4.3: External workload features maintained by the scheduler

4.2.4 Machine Learning

Machine learning has gained popularity in recent years as a means for generating models that achieve good accuracy in analysis and decision making. OpenCV [57] is an open-source computer vision library that contains a machine-learning component with several supervised machine-learning algorithms. We leveraged this library to build and analyze our machine learning models. The only development required for using this library was to build a wrapper to get data in and out of the machine learning algorithms.

There are two phases in the machine learning process – a *training* phase and a *testing* phase. Generally speaking, for supervised machine learning the training phase involves generating a set of training data (consisting of input features and the desired output from the model) which is then handed to the machine-learning algorithm to train the model. The testing phase involves checking the predictions of the trained models by handing the model the input features and checking the output decisions. For this work, our training data consisted of compute kernel features from the applications, external workload features from the scheduler, and normalized runtimes on all three architectures (ranking them in relative efficiency for a compute kernel), generated from the following equation:

$$E = \frac{R_{x86}}{R_i},$$

Where E is the relative efficiency, R_{x86} is the runtime on the Opteron 6376 and R_i is the runtime for architecture i. Therefore we rolled the scheduling process, including mapping of compute kernels to an architecture and minimizing workload interference, into a single decision making model. The models generated from this training data produced an output for the predicted relative efficiency of the compute kernel on each architecture given an architecture's external workload.

Additionally, in order to eliminate features within the machine learning algorithm we used principle component analysis (PCA) to reduce the set of input features to the machine learning algorithm. The PCA process projects the set of all input features (all compute kernel & external workload features) down to a smaller set of *principle components*. PCA projects features in such a way as to remove linearly correlated features, or features that provide redundant information – the goal is to create a minimal set of principle components while maintaining as much variability as possible in the feature set. A trivial example might be that it may be able to statistically determine that function calls correspond to an increase in loads & stores (for the precall/postcall sequence inserted by the compiler) and can therefore combine them into a single feature without a loss of information. This process is applied to all combinations of input features to reduce the feature set, speeding up training/evaluation of models by focusing only on the variability in the feature set. The PCA process for our feature sets is discussed further in 4.3.

We denoted the following terms in the training & testing phases:

- *Training application* – the application for which training data is being generated (training phase only)
- *Testing application* – the application currently being evaluated (testing phase only)
- *External applications* – the set of applications used to generate external workload (both phases). For the training phase, this is the set of all applications minus the training application. For the testing phase, this is the set of all applications minus the testing application.

The first phase of the machine learning process involved generating training data from our set of applications to build the model. Each OpenMP application was selected as the training application exactly once. Training data was generated for each given training application by running the training application on each of the three architectures at varying workload levels. To generate external workload, a background script was run that looped infinitely through a list of all external applications, placing them on architectures randomly. Each instance of this background script generated an extra level of external workload. For example, if two of these background scripts were running, then the system experienced an external workload of two (owing to the fact that each script was running a single benchmark application at a time). So, each training application was run 50 iterations for every combination of architecture and external workload, up to an external workload level of three (i.e. three background scripts launching external applications).

There is a problem that arises when generating training data with external workload. Because we are using supervised machine learning to generate our models, we must have exhaustive training data to cover all cases that can arise. This is very hard to achieve in the case of external workload; this requirement specifies that we must have training data for every combination of training application and external workload (i.e. run-queue length)[7].

[7]This problem would be exacerbated if we had additionally chosen to track the features of the external

Because generating exhaustive data in this fashion is not scalable, we chose to instead generate a large amount of *targeted* data and use the k-nearest neighbors to fill in gaps. To generate this targeted data, we ran each application on a single architecture at a time and then varied the external workload on the same architecture; for example, we ran a training application on the Tesla with no other external applications, one external application on the Tesla, two external applications and three external applications. This process was repeated for each benchmark on each architecture (on the TILEncore-Gx36, it was performed up to only a single external application). Then, this data was stitched together to create the training data set – all possible permutations of external external workload were populated by by the targeted training data, i.e. for a specific external workload of two applications running on the Opteron and one on the Tesla, all data targeted data points with an external workload of two on the Opteron (from all applications) were averaged and used to populate the relative efficiency output for the Opteron, while all targeted data points with an external workload of one on the Tesla were used to populated the relative efficienty output for the Tesla.

The testing process was similar to the training process. Each testing application was run in conjunction with varying levels of external workload, and the speedup was determined from the application's runtime. Applications were tested using leave-one-out cross validation, a standard machine learning methodology for testing the effectiveness of the generated models. Leave-one-out cross validation specifies that training data from all applications except the testing application is used to generate a machine-learning model, i.e. training data is generated for the external applications. Then during testing, the scheduler is responsible for scheduling the external applications in addition to the testing application (whose features it has not seen during training). The effectiveness of the model is evaluated on unseen features to show that it can extrapolate to new applications. This was used for our testing process, which is discussed in more detail in 4.3.

We used an artificial neural network (ANN), a standard machine-learning algorithm, to generate and evaluate our models. Other approaches have used decision trees [22] and support-vector machines [35], but we found that artificial neural networks worked better in practice. ANNs, illustrated in figure 4.13, are an abstract model of a brain represented as a graph. In the graph, vertices are *neurons* and edges are *synapses*. Neural networks consist of several layers, and depending on the neural network, are fully connected between layers (i.e. every neuron in a given layer is connected to every neuron in the layer before it and after it). The *input layer*, or the first layer, contains a neuron for every input feature f_j (compute kernel & external workload features). The *output layer*, i.e. the last layer, contains a neuron for the relative efficiency E_i of every architecture. There are zero or more *hidden* (or middle) layers between the input and output layers, containing varying numbers of neurons. Values flow through the network from the input to output layer by propagating neuron values along synapses, each of which has an associated weight. Neurons are "activated" using some

workload, as we would have to have training data for every combination of training application, run-queue length and external workload application.

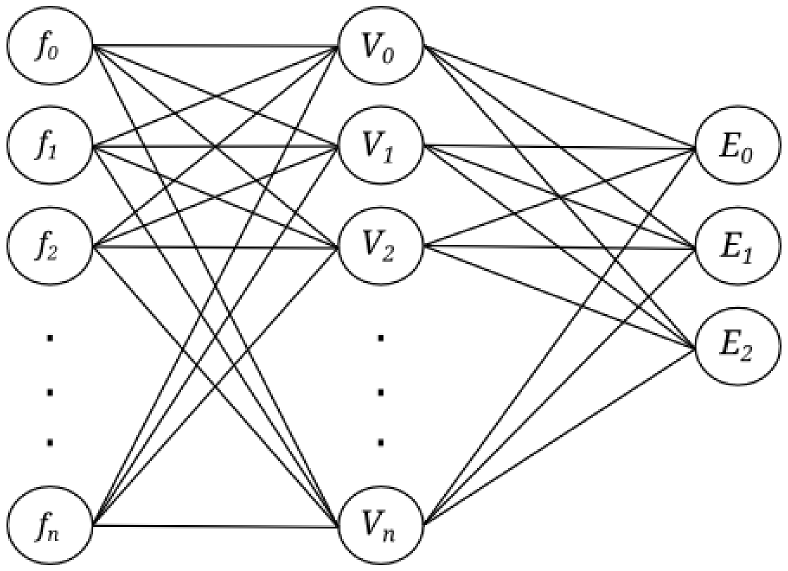


Figure 4.13: Artificial Neural Netowrk

activation function[8] $A(x)$ that describes how the output from the neuron scales with the inputs. The value for a node y in layer i, V_y, is the sum of the activation function output of all nodes in the previous layer N_{i-1} multiplied by the edge weight e of the synapse connecting the two nodes:

$$V_y = \sum_{x \in N_{i-1}} A(V_x) * e_{xy}$$

The input layer's value is the exception, as the values of these neurons are set by the input features. Training the ANN consists of setting the edge of weights of all synapses to a random value and adjusting them until the outputs generated by the network are within some specified error range compared to the outputs specified in the training data. The ANN is trained using the famous backpropagation algorithm [51], which calculates the errors throughout the network and updates them based on gradients between the activation function and the output error. The process of propagating values forward, back-propagating calculated errors and adjusting weights based on the gradients is repeated until the errors are within a specified range. Neural networks with a single hidden layer have been proven to be able to model any function [44]; hence, we chose to use ANNs to make scheduling decisions.

[8]Most neural networks, including ours, use the sigmoid function as the activation function

4.3 Results

The evaluation process consisted of generating a set of training data, training machine learning models (using leave-one-out cross validation), and testing the resulting scheduling models in live execution using varying levels of external workload. We evaluated our scheduling technique on 14 refactored applications (refactored with the partitioner and OpenMPC, with manual modifications when necessary) from Rodinia and Parboil. Applications that were written in C++ were ported to C and all applications were compiled with aggressive optimizations enabled, i.e. `-O3`. We evaluated the trained models for the three architectures, shown in table 4.4, in two setups. In the first setup, we scheduled applications onto the Opteron CPU and the Tesla GPU (ignoring the TILEncore-Gx36), while in the second setup we scheduled onto the Opteron CPU and the TILEncore-Gx36 coprocessor (ignoring the Tesla)[9]. For each setup, we trained a complete set of models using leave-one-out cross validation and performed a set of experiments to evaluate the ability of the trained models to address two aspects of the scheduling problem:

1. Map a compute kernel to the most appropriate architecture, i.e. the architecture that executed the compute kernel the fastest (without external workload)
2. Adjust the mapping in the presence of varying levels of external workload

	Opteron 6376	Tesla C2075	TILEncore-Gx36
Vendor	AMD	NVidia	Tilera
# of Cores	16	448	36
Frequency (GHz)	2.3	1.15	1
Core Design	Superscalar/ OoO	SIMT	VLIW
Memory (GB)	32	6	8
Connection	N/A	PCIe 2.0 x16	PCIe 2.0 x8
Compiler	GCC 4.4.7	NVCC 5.5.0	Tilera GCC 4.4.6
Operating System	CentOS 6.4		

Table 4.4: Architectures used for evalation of OpenMP scheduling

We evaluated two types of models for the first setup – models that did not incorporate external workload (*static* models) and models that did incorporate external workload (*dynamic* models). We only evaluated dynamic models for the second setup, as all applications were more efficient on the Opteron versus the TILEncore-Gx36.

[9]We divided the three architectures into two setups based on the relatively poor compute performance of the TILEncore-Gx36. Even with external workload, the models were likely to forego scheduling onto the TILEncore-Gx36 in favor of the Opteron or Tesla

Value	Time (μs)
Request Scheduling Decision	62
Cleanup After Kernel	29
Machine Learning Evaluation (static)	10
Machine Learning Evaluation (dynamic)	12

Table 4.5: Scheduling overheads for OpenMP applications

4.3.1 General Overheads

Table 4.5 shows the overheads associated with various aspects of the scheduling process, including communication with the scheduler and evaluation of the machine learning model (reducing input features by PCA and generating output values from the ANN)[10]. As mentioned in section 4.2.4, PCA was used to reduce the dimensionality of the input feature set without loss of information (i.e. maintaining variability). For the static models we reduced the input feature set down to five dimension while retaining 93% of variance; for the dynamic models, we reduced the feature set down to ten dimensions while retaining 96% of variance. The dimensionality of the reduced feature set dictated the number of neurons in the input layer for the generated ANNs. There were three output neurons for every generated ANN, corresponding to the predicted relative efficiency for each architecture. Each ANN also contained a single hidden layer. The number of neurons in the hidden was determined empirically – there were five neurons for the static models and 17 for the dynamic model. It should be noted that the larger size of the ANN for the dynamic models reflects the fact that the additional workload features provide a significant amount of extra information, and thus a more complex network is required to accurately model the run-time architectural interactions.

Overall, there was approximately 100μs of overhead for scheduling & cleanup. Communication consumed the majority of that time, requiring several context switches to perform IPC (as mentioned in 4.2.3). This could possibly be reduced by converting the scheduler into a kernel module, invoked via system call; however, it could be argued that for any compute kernel where 100μs of overhead is unacceptable should simply be run on the host and should not use the scheduler.

4.3.2 Setup 1: Opteron 6376 & Tesla C2075

This setup, comprised of a server-class CPU & server-class GPU, was similar to most heterogeneous setups in that it paired together latency- and throughput-based architectures to handle a diverse set of applications. We wanted to evaluate the ability of the models to first,

[10]These times were obtained on the Opteron 6376

predict the relative efficiency of a given compute kernel on the two architectures and second, schedule in a workload-aware fashion to avoid overloading an architecture when possible.

Benchmark	Best Architecture	Predicted Correctly?
backprop	Opteron	✓
bfs	Opteron	✓
cutcp	Tesla	✓
hotspot	Opteron	✓
lavaMD	Opteron	✓
lbm	Tesla	✓
lud	Tesla	✓
mri-q	Tesla	✓
pathfinder	Opteron	✓
sad	Opteron	✓
sgemm	Tesla	✓
spmv	Opteron	✓
srad	Opteron	✓
stencil	Tesla	×

Table 4.6: Best architecture and predictions from our model for benchmarks

We first evaluated the ability of the models to predict the best architecture for the compute kernels in an application. Table 4.6 shows the list of benchmarks, on which architecture the application's compute kernels executed most efficiently, and whether our model was able to predict the best architecture. The table indicates that the model was able to learn the correct mappings for all benchmarks except the `stencil` computation benchmark. We investigated this benchmark further to understand why the models were not able to predict the best architecture.

Case Study: Stencil

The `stencil` benchmark from Parboil embodies a traditional stencil computation – it is an iterative Jacobi/7-point stencil performed on a 3D grid. We investigated this benchmark further to understand the model's misprediction. Figure 4.14 shows the structure of the computational kernel for this implementation. There are 3 nested for-loops, with the outer loop parallelized using OpenMP. According to the semantics of OpenMP, in this compute kernel consecutive threads are assigned consecutive iterations (i.e. consecutive values of `i`) and are responsible for performing the stencil computation to be stored in the grid `Anext`. The write pattern to `Anext` varies based on all 3 loop indices; however, writes by consecutive threads are to adjacent elements in `Anext`, due to the assignment of loop iterations to threads. This has a striking impact on performance on CPUs versus GPUs. On GPUs, memory

```
#pragma omp parallel for
for(int i = 1; i < (nx-1); i++)
  for(int j = 1; j < (ny-1); j++)
    for(int k = 1; k < (nz-1); k++)
      //Perform stencil computation
      Anext[i + (nx * (j + (ny * k)))] = ...
```

Figure 4.14: Computational kernel for `stencil`

	L1 Load Misses	% of Total Loads	L1 Store Misses	% of Total Stores
Original	12,742	82%	1,625	79%
Fixed	419	2.7%	104	5.2%

Table 4.7: Cache statistics for `stencil` from `perf`

coalescing plays an important part of kernel optimization – reads and writes from multiple threads can be combined into a single access, increasing memory bandwidth utilization. On CPUs, however, the case is drastically different. Because caches operate at cache line granularity, threads on separate cores writing to different elements within the same cache line cause false conflicts, decreasing bandwidth utilization. The cache coherency protocol must constantly transfer exclusive write access of a cache line between separate cores, creating a ping-pong effect and drastically reducing the cache hit rate. Performing a simple loop-index reordering (a valid optimization since all writes happen to independent elements) alleviates this issue. We implemented this fix and profiled both the original and fixed version of `stencil` to quantify this cache behavior. Table 4.7 shows the numbers from the cache counters obtained using `perf`, where each number is in millions of events counted. Performing this simple loop-index re-ordering results in a drastic reduction in false cache conflicts and greatly improves performance. This modified version of `stencil` showed a 4x speedup over the original CPU version, and even surpassed the performance of the GPU version. This highlighted a hole in our feature-set – we did not consider any form of memory access patterns in our feature set, and were unable to predict that the original implementation of `stencil` was more suitable for the GPU. We added memory coalescing features to the features extractor for OpenCL kernels, discussed in section 5.2.2.

Next, we analyzed the ability of the models to adjust the scheduling decisions in the midst of external workload. Figures 4.15-4.18 show the speedup obtained by the individual benchmarks, scheduled onto either the 16 Opteron cores or the 448 Tesla cores, over the baseline of a single Opteron core with no external workload. The graphs contain the results for the static mapping scheme and for the dynamic mapping scheme using a logscale to display the

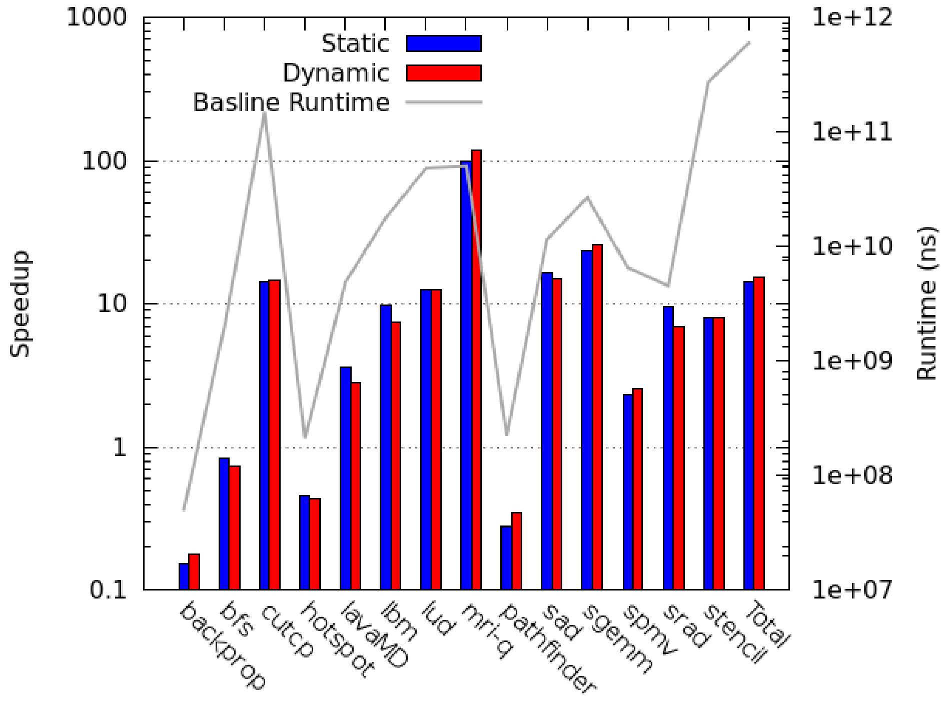


Figure 4.15: Setup 1 – Results of scheduling OpenMP kernels without external workload

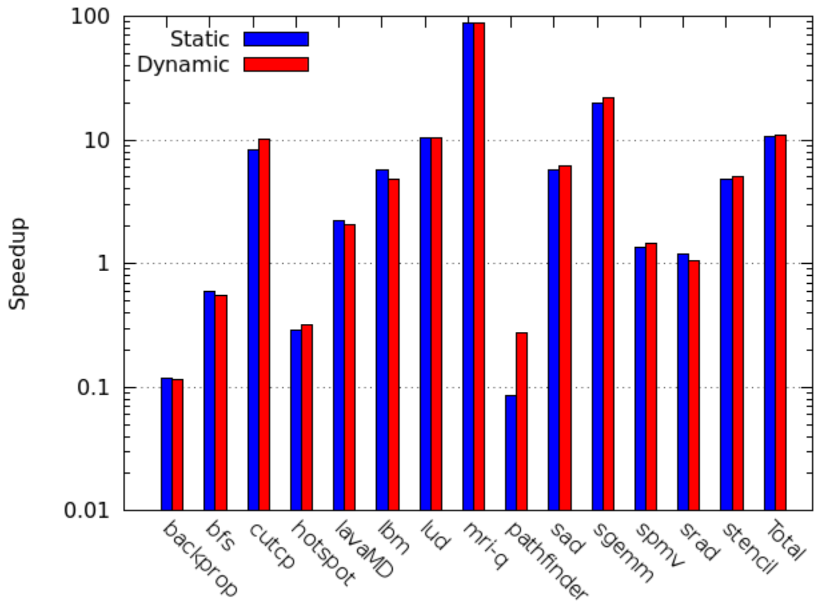


Figure 4.16: Setup 1 – Results of scheduling OpenMP kernels with one external application

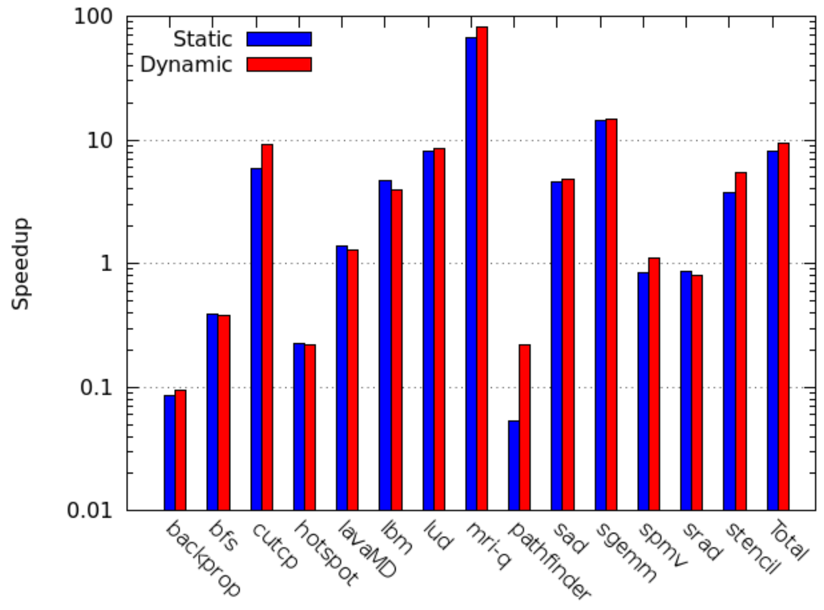


Figure 4.17: Setup 1 – Results of scheduling OpenMP kernels with two external applications

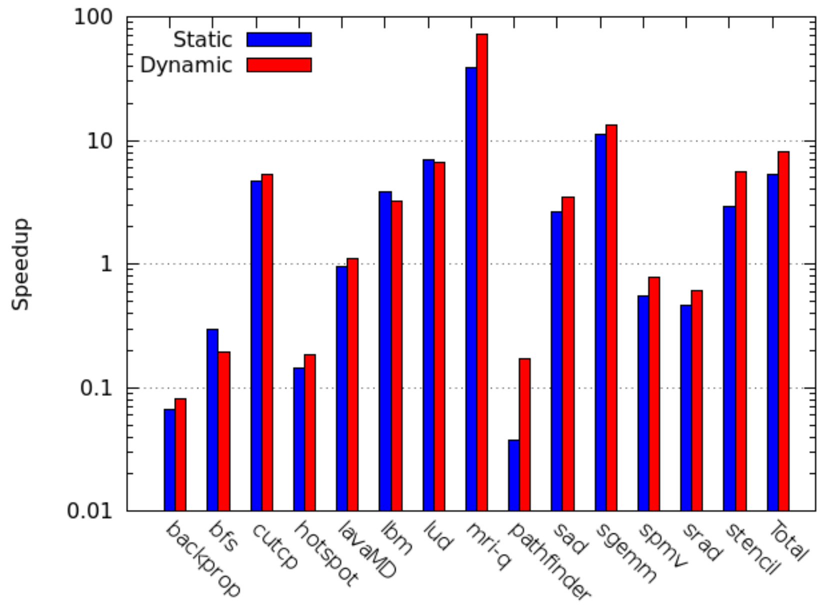


Figure 4.18: Setup 1 – Results of scheduling OpenMP kernels with three external applications

speedups. The figures show speedups for external workload levels 0-3, respectively (recall that an external workload level of 1 means a single background script launched external applications during evaluation of the testing application). Figure 4.15 additionally has a line superimposed on the graph which shows the running time of the application for the baseline execution. The scale for the running time of the applications is shown to the right of the graph, and is in nanoseconds.

With no external workload, we were able to achieve an overall 11x speedup over the baseline due to the introduction of parallelism and heterogeneity. There were minor differences in speedups obtained by the static and dynamic schemes due to variability in application runtimes, but the speedups obtained are comparable in both types of models. Interestingly, `backprop`, `bfs`, `hotspot` and `pathfinder` experienced slowdowns in our testing. The superimposed line shows that `backprop`, `hotspot` and `pathfinder` are very short-running. The combination of small compute kernel execution times (which leads to scalability issues with increasing numbers of threads) and memory-management/scheduling overheads caused a net slowdown. However, `bfs` has a relatively longer total running time. The slowdown for this application was caused not by the running time of the application, but by the fact that `bfs` launches a short-running compute kernel multiple times, causing non-negligible scheduling overhead. However, longer-running benchmarks experienced large speedups by enjoying the benefits of the two architectures. The applications scheduled onto the GPU, such as `lud` and `mri-q` experienced extremely large speedups; this highlighted the fact that even a naive translation from OpenMP to CUDA allows fantastic performance gains for some highly-parallel applications (the compute kernel for `mri-q` contains 250,000 work items).

As expected, when the external workload increases the speedup per benchmark decreases because the system executes additional benchmarks, each of which competes for compute resources in the system. However, the speedups obtained using the dynamic scheme degrade more gracefully compared to the speedups obtained using the static scheme (recall that speedups are shown on a logarithmic scale), to the point of a 4x increase over the static model in figure 4.18. Occasionally, as in `lbm` and `lud` in figure 4.18, the static mapping scheme might outperform the dynamic mapping scheme due to the model seeing the "wrong" external workload. For example, application A (which is statically mapped to the Tesla) might request a scheduling decision from the scheduler, which has application B in the Tesla run-queue. However, application B has finished execution and is waiting for cleanup, but is behind application A in the server queue. Application A could be scheduled onto the Opteron because the scheduler believes there are more applications in the Tesla run-queue. Thus, it might make a bad decision from outdated accounting. Notice that speedups obtained for `stencil` spike as external workload increases; this is an artifact from the model misprediction discussed above. As external workload increases, the scheduler moves `stencil` to the Tesla more frequently, subsequently increasing the speedup.

One interesting trend emerges as the amount of workload increases – the performance of applications that were mapped to the Tesla degrades less than applications that were mapped to the Opteron. This is because of the way threading and multi-tasking occurs on the two architectures. For the Opteron, scheduling of threads is handled by the Linux scheduler. During every scheduling epoch, the scheduler is responsible for giving threads "fair" execution time. This means that threads are swapped on and off the processor more frequently as the number of threads increases, especially if those threads are compute-intensive. The default number of threads used by the OpenMP runtime without any direction by the application is equal to the number of cores in the system. Therefore, if there are four applications, each inside of a compute kernel, 64 threads are vying for time on the 16 processors. This chokes the Linux scheduler, causing it to repeatedly swap processes in and out of the cores, giving each thread a smaller share of execution time and ultimately decimating performance. This is indicated by the performance of several benchmarks, such as `lavaMD`, `sad`, `spmv` and `srad`. These benchmarks go from achieving healthy speedups of 3-10x to ultimately being slower than the baseline for the highest level of workload. For GPU applications, however, slowdowns are minimal. This is due to the run-to-completion model of the NVidia driver; there is no multi-tasking on the GPU itself. Commands (possibly from multiple applications) are multiplexed into a single command-queue for the device. Compute kernels, once started, cannot be interrupted and must finish before other kernels or data transfers can be performed. This eliminates much of the scheduling overhead caused by choking the Linux scheduler – compute kernels must simply wait their turn in the command queue, but have access to all compute resources within the device for the length of time necessary to finish kernel execution (rather than being given decreasing time share on a processor). This problem with the Linux scheduler indicates that spatial rather than temporal scheduling should be investigated on the CPU. In other words, applications should cooperatively reduce the number of CPU threads used for compute kernels to allow threads longer compute time on a CPU and limit the context-swap storm that ensues when oversubscribing threads onto processors.

The peculiarities of `pathfinder`'s performance are somewhat related to the previously discussed issue of threading overhead. Initially `pathfinder` runs poorly on the Opteron as mentioned above. However, with increasing workload the dynamic model performs much better than the static model. This is because `pathfinder` exhibits inner-loop parallelism, i.e. the OpenMP parallel region is nested inside of an outer for-loop. This causes a large amount of threading overhead to initiate a parallel for-loop section and to end one (OpenMP parallel for-loops have an implicit barrier at the end). Thus, as the system becomes more and more choked by the threading overhead, the scheduler places more instantiations of `pathfinder` on the Tesla, mitigating the bottlenecks.

4.3.3 Setup 2: Opteron 6376 & TILEncore-Gx36

The second setup, containing an Opteron 6376 CPU and the TILEncore-Gx36 coprocessor, was different from most heterogeneous systems. In this system, we combined a traditional CPU with a "smart network card" to show that there may be situations where non-traditional compute resources could be leveraged to alleviate some of the workload of a host processor. As mentioned previously, none of the compute kernels ran fastest on the TILEncore-Gx36

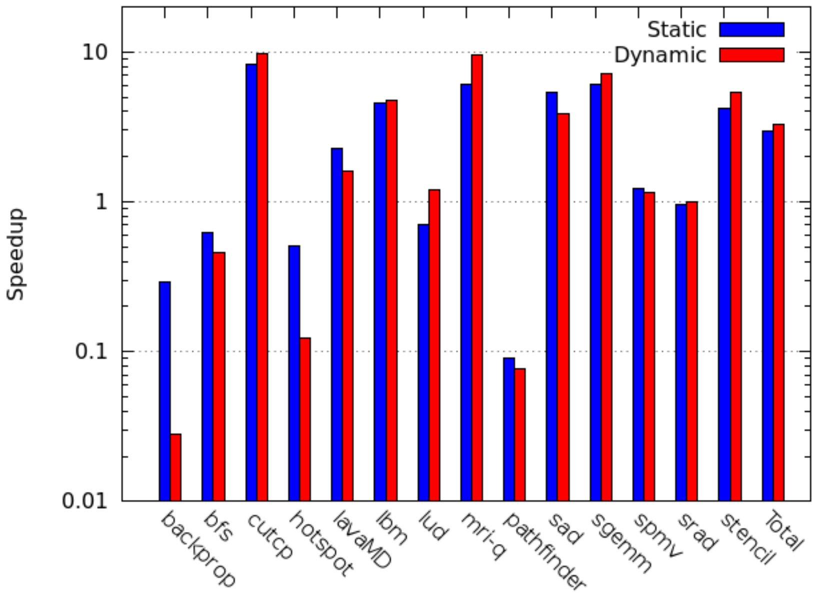


Figure 4.19: Setup 2 – Results of scheduling OpenMP kernels with one external application

– therefore, we were only interested in seeing if there were possible performance gains for offloading to the TILEncore-Gx36 when the host Opteron was oversubscribed.

Figures 4.19-4.21 show the results of scheduling with 1-3 external applications over the same previous baseline of execution on a single core. There is no graph for scheduling without external workload – without external workload, all applications were scheduled onto the Opteron meaning the static and dynamic speedups were essentially equal. The same applications that previously experienced slowdowns experienced slowdowns in this setup for the same reasons. However, some applications (such as `lud` and `mri-q`) that previously experienced significant speedups when executing on the Tesla saw smaller or non-existant speedups in this setup. This is due to the fact that the Opteron is less efficient than the Tesla for the compute kernels in these applications. Some applications tolerate this switch from Tesla to Opteron better than others – `mri-q` sustains large speedups as workload increases, but `lud` suffers dramatic slowdowns as the scheduler becomes a bottleneck (it exhibits the same inner-loop parallelism that plagued `pathfinder`). Overall these results highlighted the effects of the constant context swapping in the Linux scheduler. Rather than seeing 10x overall speedups, overall speedups drop to 1.5-3.5x, an awful result considering these applications use either the 16 cores on the Opteron or 36 cores on the TILEncore-Gx36.

Similarly to the first setup, as the external workload increased the dynamic model adjusted scheduling decisions to put more work onto the TILEncore-Gx36 as the Opteron became overloaded. The dynamic model continued to increase its speedup over the static model

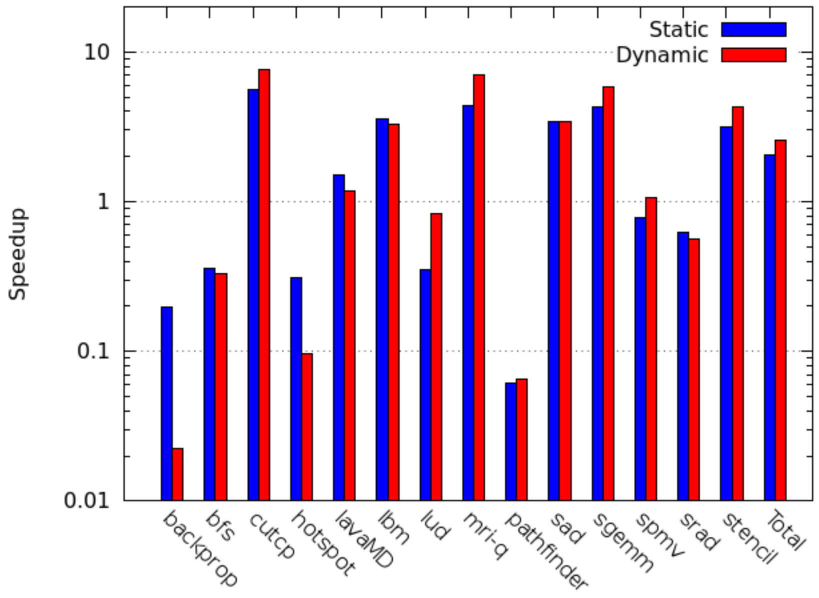


Figure 4.20: Setup 2 – Results of scheduling OpenMP kernels with two external applications

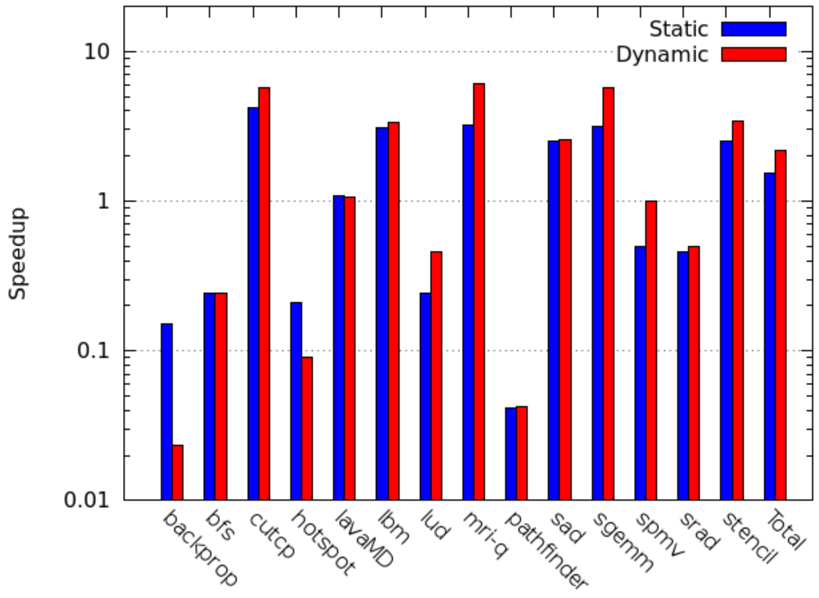


Figure 4.21: Setup 2 – Results of scheduling OpenMP kernels with three external applications

as workload increased, up to a 50% speedup with three external applications. Short-lived applications (especially `backprop` and `hotspot`) experienced especially poor performance as external workload increased. This is due to the previously mentioned "wrong" workload experienced by the scheduler, which caused bad scheduling decisions (launching short-running applications on the coprocessor). Overall, the dynamic model was able to make better scheduling decisions than the static model, meaning that it was able to utilize a non-compute device for performance gains in a non-traditional setup.

Chapter 5

Scheduling OpenCL Applications

The partitioner automatically refactored OpenMP applications to be able to execute heterogeneously with little-to-no modifications by the developer. It allowed rapid conversion from a general programming model to device-specific models, letting developers focus on hardware-software co-design instead of tedious porting work between models; indeed we were able to work on a first iteration of the heterogeneous scheduling problem and it gave us valuable insight into the problems that arise with automatic refactoring approaches.

While the partitioner achieved the stated design goals, it was sub-optimal in a variety of situations:

- The partitioner created unacceptable overheads for short running applications – automatically tracking memory and mandatory scheduling at partition boundaries can dominate the running time of some applications with small compute kernels
- The partitioner created unnecessary overheads because of the design decision to schedule at every partition boundary. The semantics of this decision required that all inputs and outputs be transferred to and from a device for every invocation of every compute kernel. Applications that invoke multiple kernels in succession[1] that operate on the same data incur overhead from unnecessary data transfers. This could be potentially mitigated by lazy data transfers, but would require additional overhead to determine what data is "dirty" and where the most up-to-date copy of the data resides.
- The partitioner inserted calls that transfer data in their entirety – all memory allocated for a given piece of data is sent to and from compute kernels that may operate only on a subset of that data.
- OpenMPC, while sufficient as a research tool for transforming OpenMP kernels to CUDA, was unusable without significant manual modifications to the code, effectively

[1]That is, without any intervening host code that modifies the data

eliminating any benefits obtained from the tool. Additionally, much of the code generated for GPUs was sub-optimal, with few adjustments from a CPU implementation to a GPU implementation.

- Similarly the partitioner is a research tool and although it is stable, it has not been tested with large applications.
- Our benchmarks, all part of high-performance computing benchmark suites, were well suited for the massive, fine-grained parallelism model of OpenCL.

Because of these issues, we chose to switch focus to OpenCL applications. Our OpenCL application suite, consisting of OpenCL benchmarks from the OpenDwarfs [18], Rodinia [11] [12] & Parboil [53] benchmark suites, have been written by programmers in a portable programming model, solving most of the problems mentioned above. In addition, it gave us a much larger set of applications on which to train our machine-learning models. Thus we chose to shift our efforts away from the engineering work required by the partitioner and instead focus on the problems of heterogeneous scheduling. This chapter presents heterogeneous scheduling of OpenCL applications across devices in a system, and is structured similarly to chapter 4:

1. Section 5.1 discusses the design in our implementation of scheduling for OpenCL compute kernels
2. Section 5.2 presents the implementation details for scheduling OpenCL applications, including the new feature extractor for OpenCL compute kernels[2]
3. Section 5.3 presents the evaulation of our scheduling techniques

5.1 Design

While we switched focus from OpenMP to OpenCL, the main task remained – we wanted to address the problem of heterogeneous scheduling of compute kernels. However, there were several changes in direction:

1. With OpenMP applications, we wanted to attack the problem of scheduling in the midst of changing system workload. Incorporating external workload into the machine-learning model showed good initial results but was somewhat superficial. We had doubts as to whether this approach could be applied universally and in a scalable fashion. We decided the external workload problem should be analyzed separately from determining the efficiency of a compute kernel on a given architecture.

[2]In this chapter compute kernel is used interchangeably with OpenCL kernel functions, denoted by the `__kernel` function qualifier. Additionally, we use work item and thread interchangeably.

2. Previous scheduling approaches, including our own, use machine learning to generate models for systems that contain only 2 architectures (generally a CPU and a GPU). We wanted to test the accuracy of our scheduling approach in systems with 3 architectures, and systems where the architectures are similar. For example, we wanted to test a system with two throughput-oriented architectures from different vendors.

3. Previous scheduling approaches generate models on a per-system basis, meaning the heavy training & model generation process for the scheduling model must be repeated for every new combination of processors in a system (which is especially problematic given the proliferation of highly heterogeneous embedded systems, such as smartphones). We sought to generate a single model for all possible systems, making the training & model generation phase a true "at-the-factory" cost.

We decided to put the external workload aspect of scheduling in the background as it requires its own thorough design and analysis. Focusing on the latter two goals, we needed a means for specifying architectures to the machine learning model, making it generic for all architectures. We arrived at the idea of combining both software and hardware features into the machine learning model so that a given compute kernel (characterized by compute kernel features) could be evaluated against a given processor (characterized by hardware features) by the produced machine-learning model. Previous approaches did not incorporate hardware features into the model, as models were generated per system and hardware features did not change.

We needed a way to characterize the hardware in the same fashion as the compute kernels. Thoman et al. developed uCLbench, a set of microbenchmarks that characterize various aspects of OpenCL implementations & their associated architectures [55]. We leveraged this set of microbenchmarks (and added several of our own) to generate a list of hardware features for each architecture that could be fed into the machine learning model. These hardware features are are discussed in more detail in 5.2.3. It should be noted that while the inputs & outputs of the machine-learning model changed, we still used ANNs as our model of choice. The training phase was almost identical (minus the varying levels of external workload) and the testing phase involved leave-one-out cross validation over all applications and all architectures. The scheduler from section 4.2.3 could be reused with almost no modification – the OpenCL scheduling models were swapped in place of the OpenMP ones.

This switch to OpenCL slightly changed the scheduling semantics. Instead of scheduling on a per-kernel basis, scheduling was performed on a per-benchmark basis due to the fact that these benchmarks were written as self-contained OpenCL applications ready for execution on a single device and would require significant refactoring to be scheduled on a per-kernel basis. Scheduling on a per-benchmark basis meant that features used for machine learning no longer corresponded to a single compute kernel within the application, but instead corresponded to all compute kernels within the application (essentially, the sum of all features of all compute kernels within an application). However, applications still followed the coprocessor execution model, where scheduling was performed at the partition boundary (i.e. the start of the application), and the entire benchmark was the partition. Because the benchmarks consisted of full OpenCL applications and did not require any refactoring, most of the problems mentioned in the introduction to this chapter were eliminated:

- No tracking of data locations and sizes was required (the benchmarks contained all required information) and scheduling happened once per application instead of once per kernel. This eliminated a lot of the overhead experienced by short-lived applications.
- Data was transferred to and from a device only when necessary. If the benchmark contains multiple compute kernels that operate on the same data, the benchmarks were written to let the data stay on the device between compute kernel executions.
- The applications sent only the required data to the specified device as opposed to always data transferring data in its entirety, even when only a subset is required.
- Finally, because these applications were written using OpenCL, a functionally portable parallel programming model, all refactoring and programming model translation issues were avoided. Note that we chose to drop support for the TILEncore-Gx36 because of its disappointing performance, noted in section 4.3.

The only overheads carried over from the previous approach were those required for scheduling (IPC & machine learning analysis), and even those were reduced by scheduling on a per-application rather than a per-kernel basis.

Another set of tools was developed in order to perform scheduling for OpenCL applications, including an OpenCL runtime support library (5.2.1) and another feature extractor built on top of Clang/LLVM (5.2.2). We switched to a heuristic-based approach rather than a profiling-based approach to scale features in the new implementation, because as mentioned in section 4.1 using profiling information is in general non-portable. The feature extractor is meant to gather trends in compute kernel execution rather than exact operation counts; this approach sufficed for our needs.

5.2 Implementation

Most of the implementation details for OpenCL applications involved hooking the applications into the client-side scheduling library (5.2.1), creating a new feature extractor for OpenCL compute kernels (5.2.2) and generating hardware features with the uCLbench microbenchmarks (5.2.3). There were very few changes to the scheduler & machine learning process (5.2.4).

5.2.1 OpenCL Runtime Support Library

The OpenCL runtime support library was developed as a means for providing naming consistency between the application and scheduler. It provided a structured approach to accessing devices in the system, based on how they are queried at runtime. OpenCL specifies a hierarchy of resources in the system to enable support for a variety of devices. Vendors of compute resources provide *platforms* that correspond to that vendor's OpenCL implementation; examples include the AMD Accelerated Parallel Processing (APP) SDK [16] for CPUs & AMD GPUs, the CUDA SDK [47] for NVidia GPUs, and the Intel OpenCL SDK [30] for Intel CPUs. The platforms provide OpenCL implementations for *devices*, including a compiler and implementations of the OpenCL API. Devices represent the actual processors in the system and are supported by one or more platforms. For example, Intel CPUs can use either the AMD APP SDK or the Intel OpenCL SDK for their OpenCL implementation.

Listing 5.1 shows an example of using the support library & scheduling API (note that the `cl_runtime` & `cl_exec` objects are part of the support library and not the OpenCL API). To query the system for platforms and devices, applications normally use the `clGetPlatformIDs` and `clGetDeviceIDs` calls. The runtime support library structures access to devices through platform and device numbers by the order in which they are returned from these calls. For example, our evaluation machine "bob" contains three compute devices – 4 AMD Opteron 6376 CPUs (exposed as a single device), an NVidia GTX Titan GPU and an AMD Radeon R290x GPU. A call to `clGetPlatformIDs` returns two platform IDs – the CUDA SDK (platform 0) and the AMD APP SDK (platform 1). Calling `clGetDeviceIDs` with platform 0 returns one device – the GTX Titan. Calling `clGetDeviceIDs` with platform 1 returns two devices – the Radeon R290x and the Opteron CPUs. Therefore, if an application wanted to get a compute context and command queue for the Radeon R290x, the application would call `get_device(rt, 1, 0, 0)`.

In listing 5.1, the `cl_runtime` object is a handle that contains OpenCL information queried from the system while the `cl_exec` object contains fields for the platform, device and number of work units in order to succinctly specify an *execution environment* for a compute kernel. The application starts by initializing the runtime handle by calling `new_ocl_runtime`. It proceeds to do any other sort of initialization required before beginning device setup. In order to schedule, the application first initializes a `cl_kernel_features` struct with features from the feature extractor (run on the compute kernel source). Then, it calls the client-side scheduling library using `clSchedule` to interact with the scheduling daemon, which returns a execution environment in a `cl_exec` struct. The application uses the individual fields of the struct to initialize the environment – the call to `get_context_by_platform` asks the library to construct a context and the call to `get_device` asks the library to return a device ID. For devices that support device fission (an extension in OpenCL 1.1 and a part of the standard in version 1.2), the last argument to `get_device` can be used to specify the number of compute units out of the total available number of compute units to use. If 0 is specified, then the maximum number of available compute units in the device is used. After a compute

```
/* OpenCL runtime support library API */
struct cl_runtime* new_ocl_runtime();
cl_context get_context_by_platform(struct cl_runtime* rt, int platform);
cl_device_id get_device(struct cl_runtime* rt, int platform, int device,
        int compute_units);
void delete_ocl_runtime(struct cl_runtime* rt);

/* Client-side scheduling API */
struct cl_exec clSchedule(struct cl_kernel_features* features);
void clCleanupExecution();
/* end all API */

int main(int argc, char** argv)
{
        struct cl_runtime* rt = new_ocl_runtime();

        ... (application initialization) ...

        struct cl_kernel_features* feats = ... (extracted features)
        struct cl_exec exec = clSchedule(feats);
        cl_context ctx = get_context_by_platform(rt, exec.platform);
        cl_device_id dev = get_device(rt, exec.platform, exec.device,
                exec.work_units);

        ... (compute kernel execution) ...

        clCleanupExecution();

        ... (remaining cleanup) ...

        delete_ocl_runtime(rt);
        return 0;
}
```

Figure 5.1: Using OpenCL support library & scheduling API

has finished execution, it notifies the scheduler with a call to `clCleanupExecution`[3]. Finally the application performs any remaining cleanup and frees the runtime handle.

This library is used to maintain naming consistency between the application and the scheduler by passing `cl_exec` execution environments. If, for example, the scheduler decides that an application should run a compute kernel on the GTX Titan, it sends a `cl_exec` struct with platform 0 and device 0 to the client, which uses the runtime library to get the appropriate context and command queue.

Listing 5.1 shows the extent of the changes made to applications to enable scheduling. The only other minor changes enabling profiling when command queues were created and timing individual OpenCL actions (namely data transfers & kernel executions).

5.2.2 Extracting Features from OpenCL Kernels

A separate feature extractor was developed for OpenCL applications based on Clang/LLVM, simply because GCC cannot parse OpenCL kernel code. The feature extractor was implemented as an `opt` pass (the LLVM middle-end optimizer) over the LLVM bitcode IR generated by Clang after optimizations. Rather than utilizing profiling to scale feature counts, we chose to use static heuristics. There were several reasons for this – first, other work ([22]) did not use profiling to scale extracted features but managed to obtain ideal mappings. Second, the Clang/LLVM toolchain does not have the ability to instrument OpenCL kernel code to record basic block counts, although other tools such as Intel VTune Amplifier [31], can record edge profiling information (which requires licensing fees and is closed source). We decided against using VTune Amplifier because Clang/LLVM are part of a free & open-source toolchain that is easy to obtain, is portable and is pluggable. Finally, as mentioned in 4.2.2, using profiling information is in general non-portable because profiling information becomes obsolete for each different invocation of a compute kernel. Although using a heuristic is potentially innaccurate, it is acceptable as long as it is able to capture compute kernel trends.

The following heuristics were used to estimate the number of times a given basic block executed:

1. Each compute kernel was executed by 10,000 separate work items
2. Each loop performed 1,000 iterations
3. Every control flow path through the code executed with equal probability

Consider an excerpt from the sparse matrix-vector multiplication compute kernel from the OpenDwarfs benchmark suite [18] in listing 5.2. At the beginning, 10,000 work items begin

[3]This is for future work, when we reconsider external workload when scheduling OpenCL kernels

```
__kernel void csr(...) {
        unsigned int row = get_global_id(0);
        if(row < num_rows) {
                ...
                for(jj = row_start, jj < row_end; jj++)
                        sum += Ax[jj] * x[Aj[jj]];
                y[row] = sum;
        }
}
```

Figure 5.2: Estimates for basic block counts based on heuristics

execution of the kernel and call `get_global_id`, according to rule (1). Then, the first `if`-conditional checks to make sure that the current work item is within the bounds of the multiplication. According to rule (3) above, half of the work-items evaluate the conditional as true, and half evaluate it as false – thus, 5,000 of the work items enter the body of the `if`-conditional, and 5,000 finish execution of the kernel. Those 5,000 threads that enter the conditional then perform the `for`-loop initialization and evaluate its condition. Again, according to rule (3), half branch to the end of the loop, while half enter the loop body. According to rule (2), each work item that enters the loop executes the loop body 1,000 times, meaning that the body is evaluated a total of 2,500,000 times. Finally, all threads that entered the `if`-conditional execute the basic block containing the store to array `y`. The features extracted within each of these basic blocks, including the entry block of the function, the beginning block in the `if`-conditional, the body of the `for`-loop and the store statement after the for-loop, execute 10,000; 5,000; 2,500,000 and 5,000 times, respectively. This heuristic was used to extract features from all compute kernels in an application.

Table 5.1 lists the features extracted from the OpenCL compute kernels. While many of these features are similar to before, there are several differences. First, vector operations are collected for the main types of operations on data (floating-point, integer & bitwise) since OpenCL supports vector types while traditional C does not (without special compiler intrinsics). Additionally, memory accesses to the dedicated local memory (`__local` memory region in OpenCL) are collected, as different architectures handle this memory differently. Also, conditional branches are collected in addition to unconditional branches.

The last change to the set of features pertain to memory accesses performed by work items in a single group. Memory accesses by adjacent work items in a work group are considered *coalesced* if they can all be serviced by a single load operation; increasing coalescing increases the effective memory bandwidth of the architecture by batching together several individual accesses into a single operation. However, uncoalesced accesses are tolerated differently by different architectures especially in terms of hardware (CPU) vs. software (GPU) caching and prefetching. Consider the two vector addition kernels in listing 5.3, executed by an

```
__kernel vectorAdd(
        __global const int *a,
        __global const int *b,
        __global int *c,
        int size)
{
        int tid = get_global_id(0);
        c[tid] = a[tid] + b[tid];
}
```

(a) Coalesced memory accesses

```
__kernel vectorAdd(
        __global const int *a,
        __global const int *b,
        __global int *c,
        int size)
{
        int tid = get_global_id(0);
        int i;
        for(i = 0; i < 32; i++)
                c[tid * 32 + i] =
                        a[tid * 32 + i] +
                        b[tid * 32 + i];
}
```

(b) Uncoalesced memory accesses

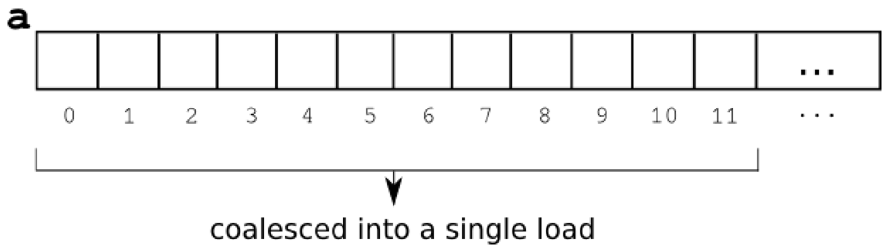
Figure 5.3: Coalesced memory

#	Kernel Feature	Description
1	*num_instructions*	Number of instructions
2	*float_ops*	Number of floating-point math operations
3	*vector_float_ops*	Number of vector floating-point math opterations
4	*int_ops*	Number of integer math operations
5	*vector_int_ops*	Number of vector integer math operations
6	*boolean_ops*	Number of bitwise and boolean operations
7	*vector_boolean_ops*	Number of vector bitwise and boolean operations
8	*load_ops*	Number of memory load operations
9	*store_ops*	Number of memory store operations
10	*local_load_ops*	Number of local-memory load operations
11	*local_store_ops*	Number of local-memory store operations
12	*coa_mem_ops*	Number of coalesced memory accesses
13	*maybe_coa_mem_ops*	Number of potentially coalesced memory accesses
14	*uncoa_mem_ops*	Number of uncoalesced memory accesses
15	*func_calls*	Number of function calls
16	*intrinsic_math_ops*	Number of intrinsic math operations
17	*cond_branches*	Number of conditional branches
18	*uncond_branches*	Number of unconditional branches
19	*work_items*	Number of parallelizable work items
20	*memory_tx*	Number of bytes transferred to device
21	*memory_rx*	Number of bytes transferred back from device

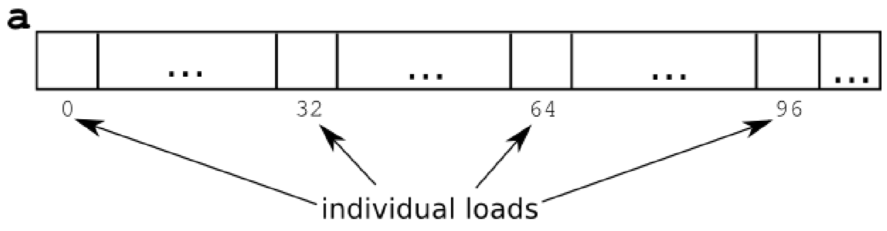
Table 5.1: Compute kernel features collected from OpenCL applications

NVidia Tesla C2075 GPU. In 5.3a, each work item is responsible for calculating the addition for a single item in the vector, while in 5.3b, each work item is responsible for calculating the addition for 32 items in the vector, the size of an individual warp (or cluster of scheduled threads) on NVidia Fermi architectures [45]. In 5.3a, all work items in a work group access adjacent memory locations – thread 0 accesses index 0, thread 1 accesses index 1, etc. Because of this, loads from multiple work items are *coalesced* into a single load operation[4] issued by the memory unit in the Tesla C2075's streaming multiprocessor (SM); similar coalescing occurs when storing results. In 5.3b, however, for the first iteration of the for-loop thread 0 accesses index 0, thread 1 accesses index 32, thread 2 accesses index 64, etc. Thus, the SM's memory unit cannot batch those loads together into a single load operation. The SM now must issue 2 load & 1 store operation *per work item* rather than for every 12 work items. Additionally, without any caching mechanism the rest of the fetched memory line is discarded and must be accessed with subsequent memory operations (luckily most

[4]The memory bus width for the NVidia Tesla C2075 is 384 bits, meaning that it can combine up to 12 32-bit integer memory operations at a time



(a) Diagram of a coalesced memory access



(b) Diagram of an uncoalesced memory access

Figure 5.4: Diagram of memory access from compute kernels in 5.3

modern architectures, including GPUs, use some form of caching). Figure 5.4 illustrates how individual threads load data from array "`a`" in the corresponding kernel code in listing 5.3. Thus, the patterns in which compute kernels access memory can have a drastic affect on performance from architecture to architecture, especially because of the "memory wall" discussed in section 1.1.

Memory coalescing is an implicit feature of compute kernels because the kernel code specifies the execution of a single work item; it is the runtime compiler's responsibility to convert the single work item description into a multi-work item implementation that can be executed in parallel by the device. In general, threads access input & output data based on their thread ID; we used this fact to track coalescing within the kernel. The feature extractor keeps track of a list of values[5]that are in some way based on a work item's thread ID (called *thread ID values*), starting with calls to `get_local_id` or `get_global_id` (`get_group_id` is most often used as a common constant offset for all work items in a work group and does not affect coalescing, according to rule (3) below). The feature extractor tracks how values initially obtained from calls to these OpenCL built-in functions evolve during execution. Memory coalescing analysis depends on the *stride* of an access between adjacent work items. The initial thread ID values obtained from calls to `get_local_id` and `get_global_id` have a stride of 1 – any work items that access memory using a value with stride 1 access adjacent elements in memory, resulting in perfect coalescing (thread 0 accesses element 0, thread 1 accesses element 1, etc.). Operations that change this stride produce values that affect

[5]In LLVM nomenclature, a *value* is any constant or the result of any operation, and can be used in other operations. Note that a value has no assigned storage since the code is in target-independent form.

coalescing between adjacent work items. If, for example, threads access memory locations based on $2 * threadID$ (i.e. thread 0 accesses location 0, thread 1 accesses location 2, etc.), then only half as many accesses can be serviced by a single operation and the effective memory bandwidth is halved. So, the feature extractor tracks how thread ID values and their associated strides change during kernel execution. The following rules determine how the access stride changes as various operations are performed on thread ID values:

1. A value that is the result of a call to `get_local_id` or `get_global_id` has a stride of 1. These values are the initial thread ID values, and all subsequent thread ID values flow from these values.
2. Any operation that involves a thread ID value and an unknown value (such as the value stored at some memory location, or some runtime-dependent value) produces an unknown stride.
3. Addition or subtraction of a constant value and a thread ID value corresponds to an offset for all work items and does not affect the access stride.
4. Addition or subtraction with a non-constant value results in an unknown stride.
5. Multiplication or division of a constant and a thread ID value scales the stride of that thread ID value by the constant value. Note that shift-left and shift-right by a constant value c correspond to multiplying and dividing the stride by 2^c, respectively.
6. Multiplication or division with a non-constant value results in an unknown stride.
7. Bitwise operations (and, or, xor) between any value and a thread ID value produces an unknown stride.
8. Any operation that does not use a thread ID value in one of its operands is considered orthogonal to memory coalescing analysis and is ignored.

Note that any interaction with a value of unknown stride produces a value of unknown stride in a "viral" fashion. When the compute kernel actually performs a memory access (either a `load` or `store` in LLVM bitcode IR), the following rules are applied to classify the access as either `coalesced`, `maybe-coalesced` or `uncoalesced`:

1. If the access stride between adjacent work items is less than or equal to 2, then the access is considered *coalesced*
2. If the access stride between adjacent work items is greater than 2, then the access is considerd *uncoalesced*
3. If the access stride is unknown, then the access is considered *maybe coalesced*

A stride of 2 was chosen as the cutoff point because very few applications exhibit perfect coalescing, and compute kernels with an access stride of 2 still exhibit high memory bandwidth utilization. However when increasing the stride beyond 2, most architectures experience a dramatic dropoff in terms of memory bandwidth.

Returning to the compute kernel in 5.3a, value `tid` is initialized by a call to `get_global_id`; according to rule (1), its access stride is 1. The subsequent uses of `tid` to reference into `a`, `b` and `c` are all perfectly coalesced, meaning that every memory access in this compute kernel exhibits perfect coalescing. However, in listing 5.3b, accesses are based on a the thread ID multiplied by 32, meaning the access stride between adjacent work items is 32 according to rule (3), but it is also also added to a runtime value `i`, meaning that the stride is unknown and therefore considered `maybe-coalesced`.

In general, accesses that are based on constants are either coalesced or uncoalesced, while everything else depends on runtime values and is considered maybe-coalesced.

5.2.3 Extracting Hardware Features

As mentioned in section 5.1, we used uCLbench, a set of OpenCL microbenchmarks, to generate hardware features for each architecture. The benchmark suite generates the following information per architecture:

Arithmetic Throughput

This benchmark tests the throughput of various types of arithmetic operations (add, subtract, multiply, divide) and built-in math functions (`exp`, `log`, `sqr`, `sqrt`, `cos`, `sin` and `tan`). It tests these operations using both scalar and OpenCL vector types for vectors of length 2, 4, 8 and 16. The results (in millions of operations per second, MOp/s) are the throughput for an individual work item per compute unit and for the maximum number of work-items per work group for all compute units a given architecture. We used the latter number, as it corresponded to the maximum throughput of the entire device (which corresponds more closely to how benchmarks execute); rarely do applications use only a single work-item per compute device. Higher throughput for each operation means the architecture can execute compute kernels more quickly.

Branch Penalty

This benchmark tests the ability of the architecture to handle control flow divergence. This varies significantly between architectures – CPUs have advanced branch prediction hardware[6]

[6]Depending on the OpenCL compiler, the compute kernel may be converted to a SIMD-format for CPUs in order to handle multiple work items, limiting the ability of the branch predictor and forcing refactoring

while GPUs generally favor simpler cores and use some form of predication or stalling to handle divergence. The compute kernel for the benchmark contains 128 branches that can be taken by work items; which branches are taken per work item depends on values computed on the host. The host precomputes these values and launches the compute kernel with a single work group containing 128 work items. The amount of branching experienced at runtime varies based on the number of individual branches in the kernel taken by the work items (up to 128 separate branches) and the width of those branches, which corresponds to the number of work items that execute the same branch (i.e. if the branch width is 4, work items 0, 1, 2 and 3 execute branch 0, work items 4, 5, 6 and 7 execute branch 1, etc). The benchmark reports amount of time taken to finish executing the compute kernel in nanoseconds; the lower the time, the better the architecture is able to handle divergence.

Buffer Bandwidth

This benchmark tests the data transfer bandwidth from host to device, from device to device and from device to host in megabytes per second (MB/s). This quantifies the bandwidth over the PCIe bus (for coprocessors) as well as the for main memory in the system[7]. This is important for applications that have different compute-to-transfer ratios; although a given kernel may be faster on a specific device, the data transfer overheads may outweigh any benefits. The higher the buffer bandwidth, the less of an impact data transfers between host and device have on performance.

Kernel Overheads

This benchmark tests the various OpenCL runtime overheads associated with kernel compilation & invocation. It varies both the number of arguments (from 4 to 32) and the number of lines of source code (from 24 to 3000) for a given compute kernel, then measures the compilation & invocation times in nanoseconds. The lower the kernel overheads, the less time is spent preparing for execution, meaning less of a performance penalty.

Memory Latency

This benchmark tests the access latency to different OpenCL memory regions (global, constant & local memory) in a sequential or cache-line-stepping pattern. It reports the access latency in nanoseconds for the combinations of these parameters. The less access latency,

to enable predication within the kernel

[7]One semantic peculularity of OpenCL on CPUs is that unless explicitly disabled during construction, OpenCL device memory buffers live in separate locations from host memory buffers. Therefore, data is actually copied between different locations in main memory when using the host CPUs as a device

the more quickly data can be moved between different regions of memory and the more time can be spent performing operations on that data.

Additionally, we augmented the suite of microbenchmarks to include memory coalescing features and other static architecture features.

Memory Coalescing

This benchmark tests the streaming memory bandwidth of accesses with varying strides (as mentioned in section 5.2.2) between consecutive work items. It tests accesses with varying strides to global memory (strides of 1 to 256 in powers of 2), constant memory (strides of 1 to 256 in powers of 2) and to local memory (strides of 1 to 16 in powers of two). It tests both reads and writes for the global & local memory spaces, but only reads for the constant memory space. It reports the results in gigabytes per second (GB/s) for each combination of parameters. The higher the bandwidth, the better the architecture is able to handle various types of access patterns to memory, increasing compute kernel performance.

Static Architecture Features

This benchmark simply prints out some simple features of each architecture as obtained from the OpenCL runtime. These features include device type (CPU, GPU, accelerator or custom), number of compute units, clock frequency and sizes of the individual OpenCL memory regions on the device.

These features correspond to the extracted kernel features, in the hopes that the machine learning model could correlate the abilities of the hardware with the operations performed by the compute kernel. Each of these benchmarks were run for every architecture to gather hardware features, which were paired with compute kernel features to form the basis for the training & testing data.

5.2.4 Changes to the Scheduler & Machine Learning

There were very few changes to be made to the scheduler in order to adapt it for scheduling of OpenCL applications. First, the messages passed between the client and server were modified slightly to allow passing the new OpenCL features (inside a `struct cl_kernel_feature` object) to the server and an execution environment (inside a `struct cl_exec` object) back to the client. Additionally, any notion of external workload was removed – the scheduler maintained no information about which applications were running on which devices, although it could given that the applications notified the scheduler when they had finished execution. As mentioned before, this was purely future-proofing the work. Finally, the scheduler loaded the models trained for scheduling OpenCL compute kernels rather than the ones trained for

OpenMP compute kernels. The rest of the scheduler, including the IPC wrapper layer which formed the basis of communication between application and scheduler, was untouched.

The machine learning process was also largely identical. We implemented decision trees in order to test our approach against a competitor, but this was handled internally by OpenCV. The process of generating training data was much shorter by avoiding having to run an application with various levels of external workload. The input training data included the updated set of kernel features from the new feature extractor as well as the hardware features from uCLbench; the external workload features were removed. Finally, the generated model produced a single output rather than outputs for all architectures in the system.

5.3 Results

For our evaluation of our unified prediction model, we tested the speedups obtained from the scheduling decisions made by our model versus the speedups obtained using the state-of-the-art technique presented in [22]. The speedups in the graphs correspond only to the OpenCL-specific portions of applications (data transfers & kernel executions), as we cannot affect the running times of other parts of applications. In this evaluation, we tested our prediction model on eight separate architectures (distributed between three systems) and 34 benchmarks from the OpenDwarfs, Rodinia and Parboil benchmark suites. All speedups presented are relative to the time it took the CPU in the system to execute the OpenCL portions of the code, i.e. a speedup of 1 indicates the system's CPU was chosen, while other values indicate another architecture was chosen. The results are presented for each system and broken down into graphs corresponding to the individual benchmark suites for readability.

As presented in [22], a decision tree model was generated per-system using the features in table 5.2. These features consist of various combinations of several of the raw features listed in table 5.1. The decision trees were evaluated using the leave-one-out cross validation technique applied to our models.

#	Kernel Feature	Description
1	*transfer/(comp+mem)*	communication/computation ratio
2	*coalesced/mem*	% coalesced memory accesses
3	*(localmem/mem)*avgws*	ratio of local/global memory accesses multiplied by the average # of work items/kernel
4	*comp/mem*	computation/memory access ratio

Table 5.2: Competitor features

As previously mentioned, we used leave-one-out cross validation to evaluate our models.

However because we developed a unified model, we performed leave-one-out cross validation per-system and per-benchmark. In other words, we the tested the ability of the model to predict the relative efficiency of an unseen application on a system containing unseen architectures. For example, when testing the models for `astar` on "bob", we generated a model with training data from the 33 external applications on the five architectures not present in "bob".

5.3.1 General Overheads

Because the scheduler implementation was re-used as described in section 4.2.3, the same scheduling overheads presented in section 4.3.1 remain for these models. However the model evaluation costs, shown in table 5.3, were slightly different[8]:

Value	Time (μs)
Machine Learning Evaluation (competitor)	1.3
Machine Learning Evaluation (unified model)	14.5

Table 5.3: Model evaluation overheads

Decision trees are classifiers, in that the predictions output from the tree fall into one of several classes (in this case, architectures with highest predicted relative efficiency). As indicated by table 5.3, evaluation using decision trees requires much less work than artificial neural networks because a single factor (e.g. number of loads) is evaluated at a node. Depending on the value of that factor, either the left or right child of the current node is taken, until a leaf node is reached. Each leaf node corresponds to prediction of a certain class, i.e. making an architecture prediction. The generated decisions trees had on average five levels, meaning that evaluation for a given set of input features was very efficient. For our unified model, evaluation was much slower (although still fast relative to kernel execution times). The unified model required more evaluation time due to the added hardware features used to make the model portable.

For the rest of the section, we will denote the *competitor* as the technique presented in [22].

5.3.2 Setup 1

The first setup consisted of the architectures listed in table 5.4. This setup is similar to many desktop systems, containing a high-end desktop-grade multicore CPU (Intel's Haswell microarchitecture) and a discrete gaming GPU (NVidia's Fermi microarchitecture). This setup is similar to the setups tested in the competitor's work, where the system combines

[8]These times were obtained on the Core i7-4770

	Core i7-4770	GTX 560 Ti
Vendor	Intel	NVidia
# of Cores (Logical)	4 (8)	448
Frequency (GHz)	3.4	1.46
Core Design	Superscalar/OoO	SIMT
Memory (GB)	16	1.2
Connection	N/A	PCIe 3.0 x16
Operating System	Ubuntu Desktop 12.04 LTS	

Table 5.4: Architectures in evaluation system "hulk"

latency- and throughput-based processors. Making decisions for this setup was somewhat easier than other cases, as there are only two very different architectures. It should be noted that the GPU in this system was connected to the CPU using PCIe version 3.0, meaning that it benefitted from higher data transfer bandwidth than the accelerators in other systems (which used PCIe version 2.0).

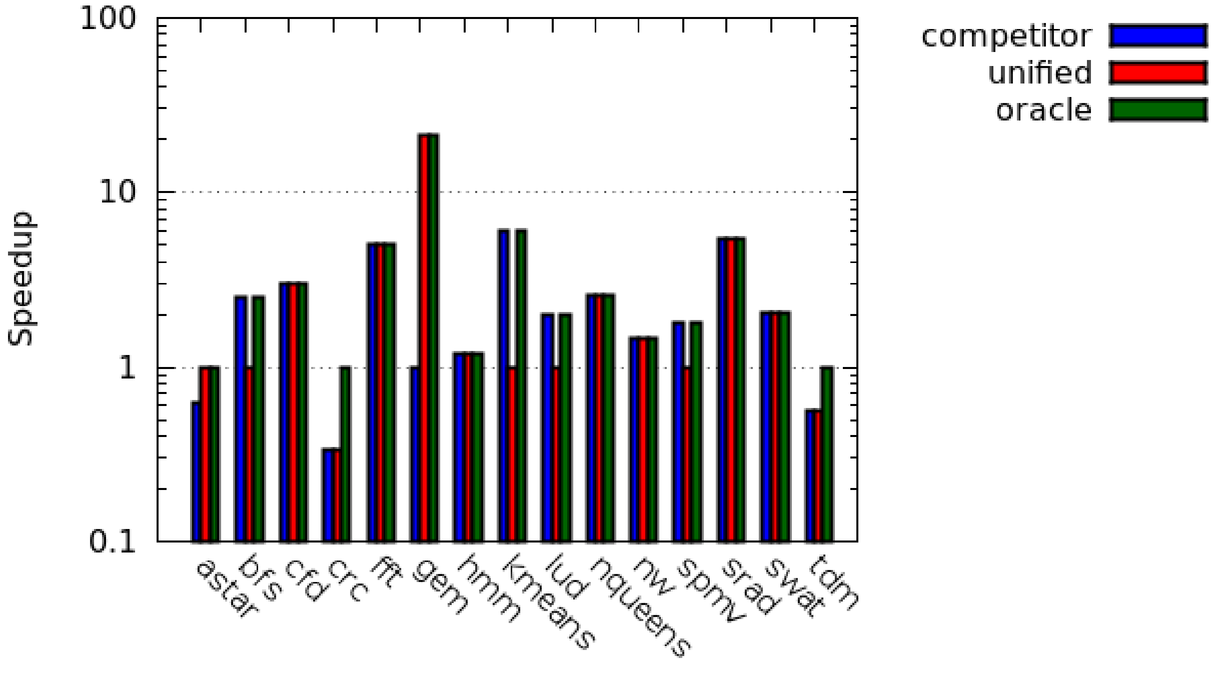


Figure 5.5: OpenDwarfs benchmarks on "hulk"

Figures 5.5, 5.6 and 5.7 show the speedups obtained over the Core i7-4770 by the benchmarks in the OpenDwarfs, Parboil and Rodinia benchmark suites, respectively (overall results per

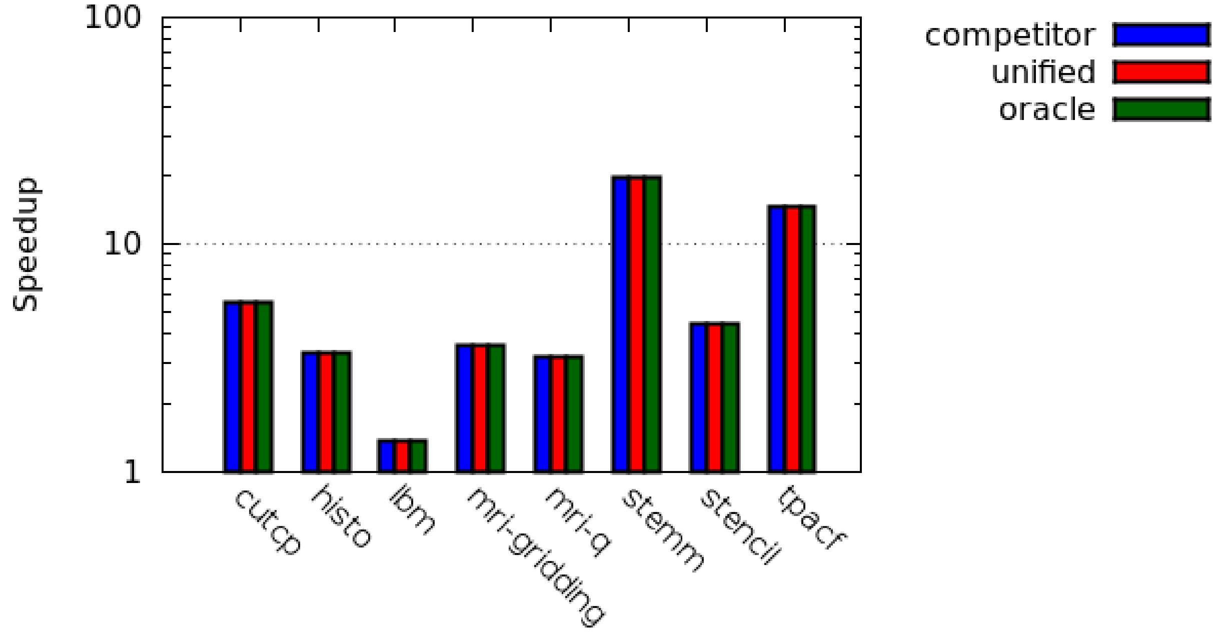


Figure 5.6: Parboil benchmarks on “hulk”

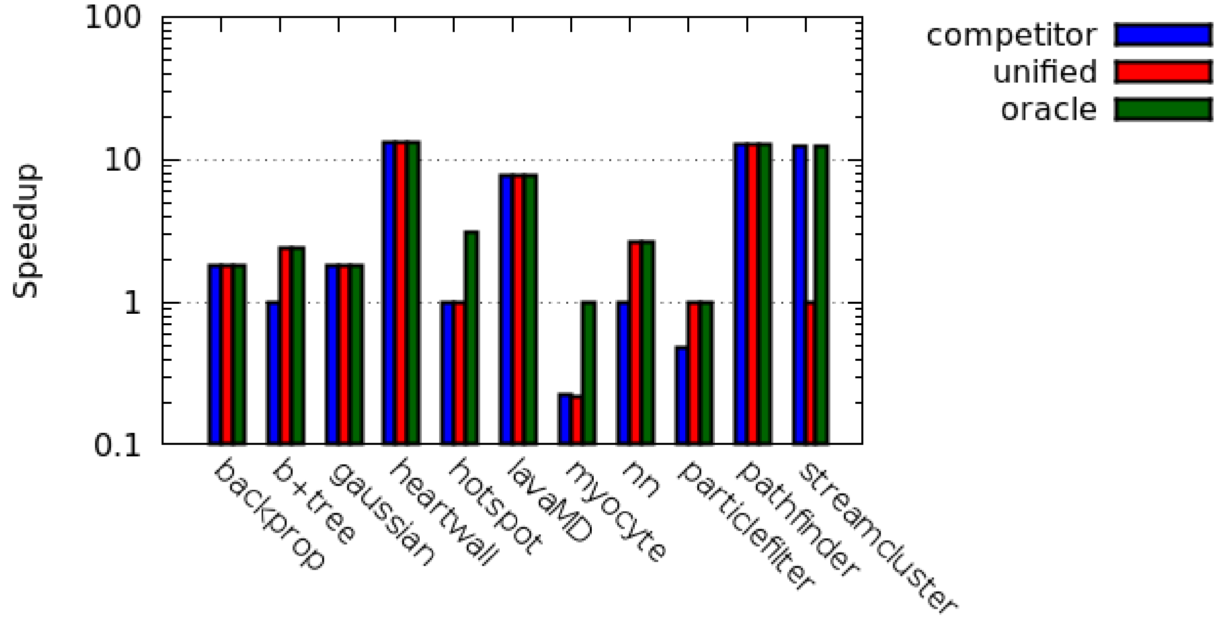


Figure 5.7: Rodinia benchmarks on “hulk”

system are listed in figure 5.14). For each benchmark, there are three bars – one bar for each of the competitor's model, our model and an oracle (that predicts the correct architecture for every benchmark). Both models mispredict benchmarks with high GPU speedups – the competitor mispredicts `gem` while our model mispredicts `streamcluster`. Interestingly, both models are able to predict the correct architecture for all Parboil benchmarks. In general, however, both models produce overall speedups close to the oracle, as shown in figure 5.14. The competitor's model produces a speedup of 4.36x over a CPU-only solution, while our model produces a speedup of 4.46x over the baseline. The maximum speedup obtainable with an oracle is 5.2x.

5.3.3 Setup 2

	Opteron 6376 (x4)	GeForce GTX Titan	Radeon R9 290X
Vendor	AMD	NVidia	AMD
# of Cores	64	2688	2816
Frequency	2.3 GHz	837 MHz	1 GHz
Core Design	Superscalar/OoO	SIMT	SIMD
Memory (GB)	128	6	4
Connection	N/A	PCIe 2.0 x16	PCIe 2.0 x16
Operating System	Ubuntu Server 12.04 LTS		

Table 5.5: Architectures in evaluation system "bob"

The second setup consisted of an interesting mix of three architectures – a highly-multicore set of CPUs and two high-end gaming GPUs (which rival server-grade GPUs). Architectural details are listed in table 5.5. The system contained 4 of the 16-core Opteron 6376 CPUs used for evaluation in section 4.3. It additionally used an NVidia GeForce GTX Titan and an AMD Radeon R9 290X, two GPUs competing for the high-end gaming market. We used this setup to test the ability of the model to distinguish fine-grained architectural details (instead of a simple latency vs. throughput prediction).

Figures 5.8, 5.9 and 5.10 show the speedups obtained using the generated models. It is easy to see that prediction accuracies dropped significantly from "hulk" to "bob"; however, a misprediction in this scenario might still provide speedups, as the Radeon and Titan tend to excel at the same sorts of applications. For many of the benchmarks where the two models disagree (e.g. all Parboil benchmarks except `tpacf`), the two models predicted one of the two GPUs. In general, the Titan was the faster of the two GPU because of data transfer latency between the host and device (discussed further in more detail below). Our unified model seemed to be able to pick up on this fact moreso than the competitor's model as our model predicted the Radeon fewer times than the Titan. Both models mispredicted `myocyte`, resulting in drastic slowdowns for the benchmark. This is due to the benchmark

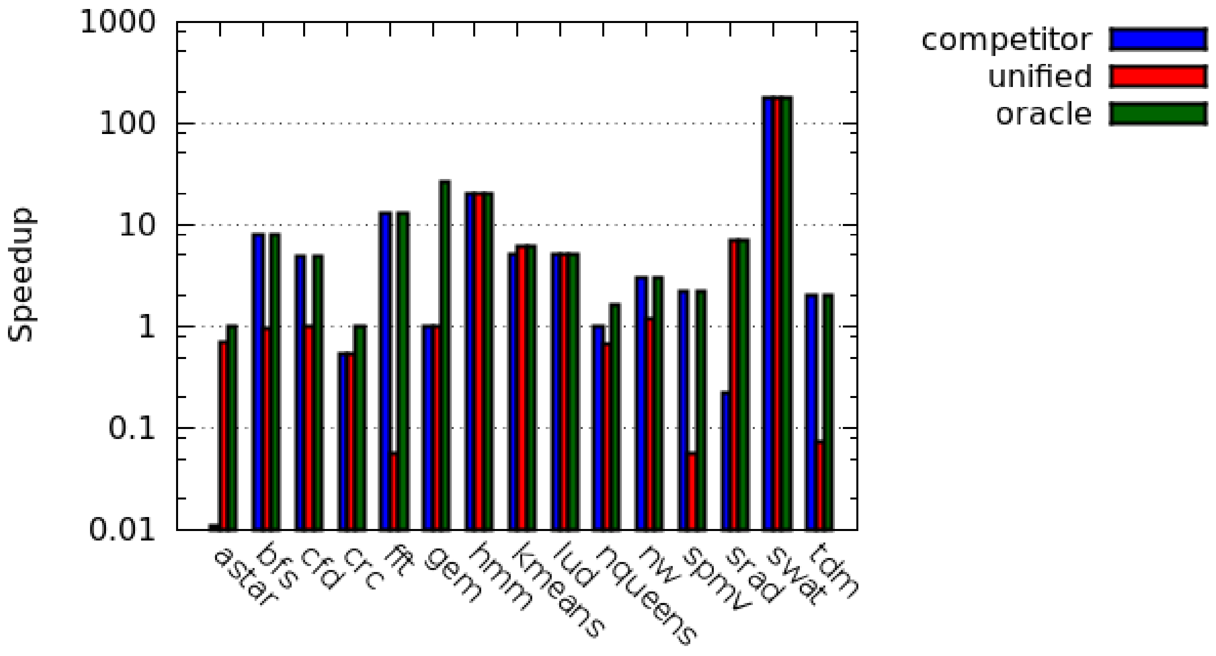


Figure 5.8: OpenDwarfs benchmarks on “bob’

using only four work items per kernel invocation, a characteristic not indicated in either feature set. Adding a feature that explicitly states the average number of work items per kernel invocation would help alleviate this misprediction. Overall, both models were able to obtain about 75%-80% of the possible speedup. The competitor obtained an overall speedup of 12.83x while our model obtained a speedup of 12.93x, out of a possible 16.2x speedup.

5.3.4 Setup 3

The third system, named “whitewhale”, contained the architectures listed in table 5.6. This setup was interesting in that it contained a spectrum of devices – it contained traditional CPUs, a server-grade GPU and a manycore coprocessor that represents an attempt to combine the best features of the other two.

Figures 5.11, 5.12 and 5.13 show the speedups obtained using the two models on “whitewhale”. On this sytem, there was less obtainable speedup compared to the other two systems. This is due to the increased performance of the Xeon CPUs relative to the Core i7-4770 and AMD Opteron 6376. Although there are many more CPU cores in the Opteron CPUs, the Xeon cores are better able to hide memory latencies than the Opteron cores because they use simultaneous multithreading to quickly swap between thread contexts when stalled on memory accesses. Additionally there is less of a perfomance gap between the CPU and GPU

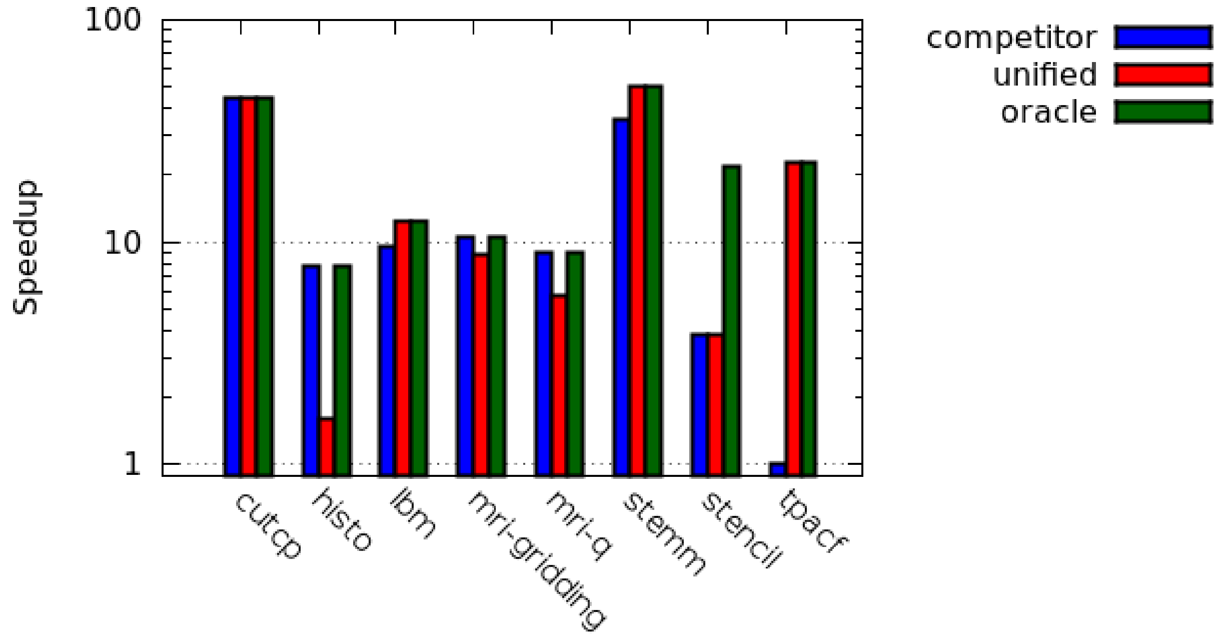


Figure 5.9: Parboil benchmarks on “bob”

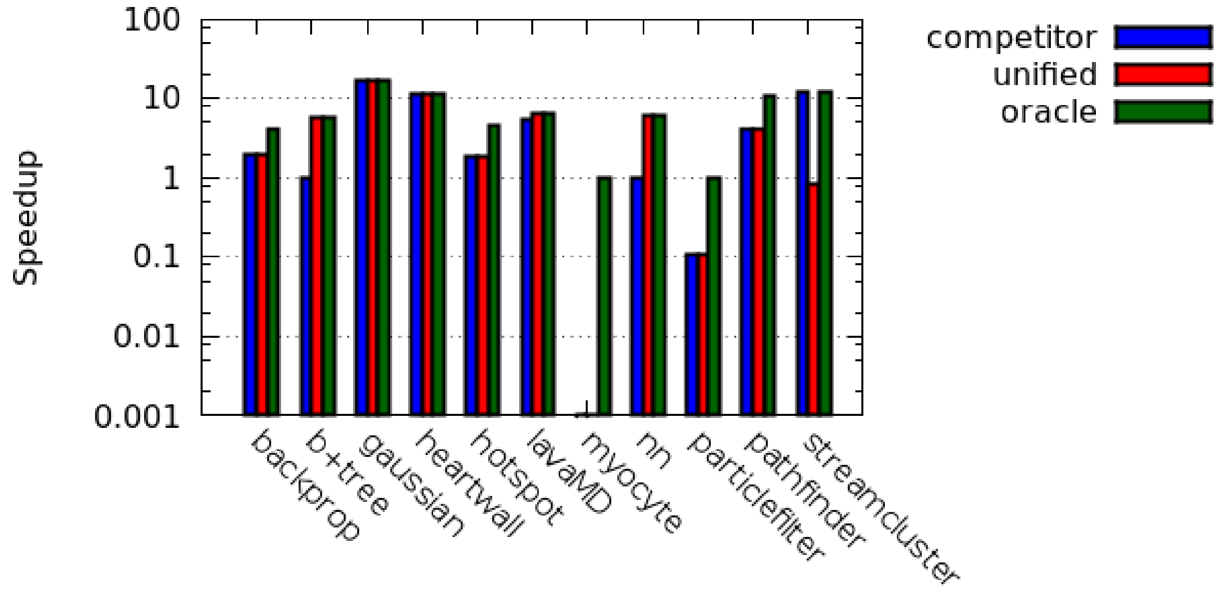


Figure 5.10: Rodinia benchmarks on “bob”

in this system because the Xeon CPUs are Intel’s newest Haswell micro-architecture while the Tesla contains the older Fermi microarchitecture. For the OpenDwarfs benchmarks, both

	Xeon E5-2695 (x2)	Xeon Phi 3120A	Tesla C2075
Vendor	Intel	Intel	NVidia
# of Cores (Logical)	24 (48)	57 (228)	448
Frequency (GHz)	2.4	1.1	1.15
Core Design	Superscalar/OoO	Superscalar/In-order	SIMT
Memory (GB)	64	6	6
Connection	N/A	PCIe 2.0 x16	PCIe 2.0 x16
Operating System	CentOS 6.5		

Table 5.6: Architectures in evaluation system "whitewhale"

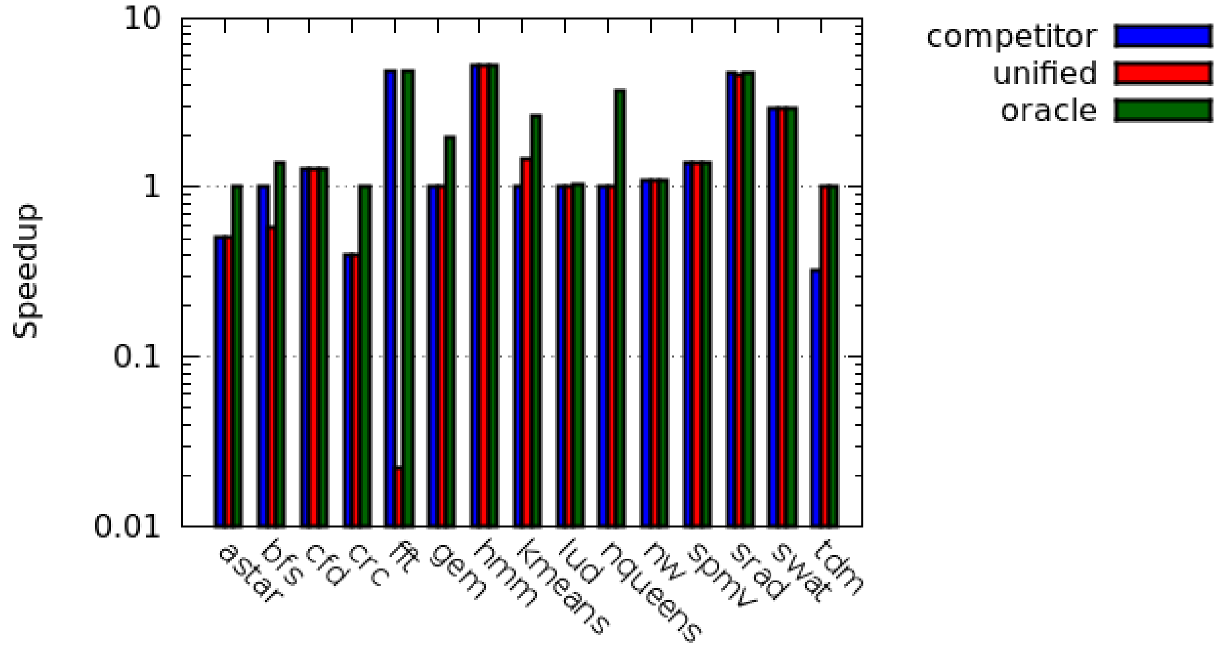


Figure 5.11: OpenDwarfs benchmarks on "whitewhale"

models predict architectures that produce speedups relatively close to the maximum possible speedup.

The unified model mispredicts `fft`, due to the relatively small number of work items used per kernel invocation (1024 and 512, for the two kernels). It schedules `fft` onto the Xeon Phi instead of the Tesla, resulting in bad performance degradation. The `fft` kernels perform a large number of floating-point operations. Based on the hardware features from uCLbench, the Xeon Phi is the fastest at arithmetic operations, reaching nearly a teraflop of floating-point performance. However, this feature is generated using a large number of work items. The Intel OpenCL SDK implicitly auto-vectorizes the work items in a work group so that 16

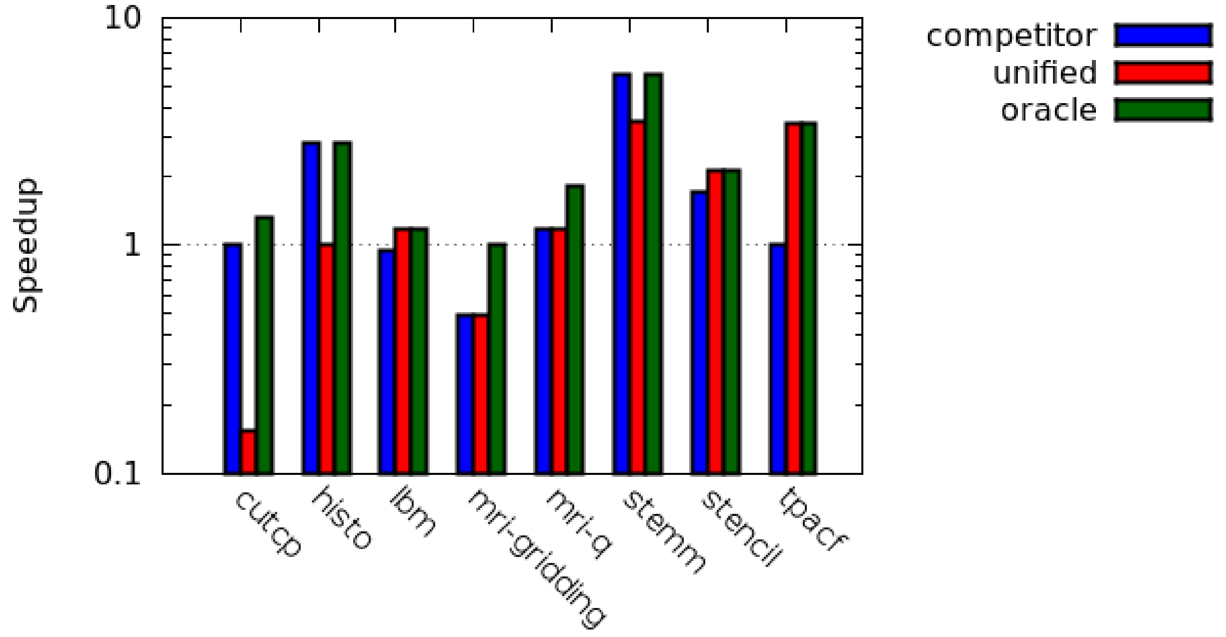


Figure 5.12: Parboil benchmarks on "whitewhale"

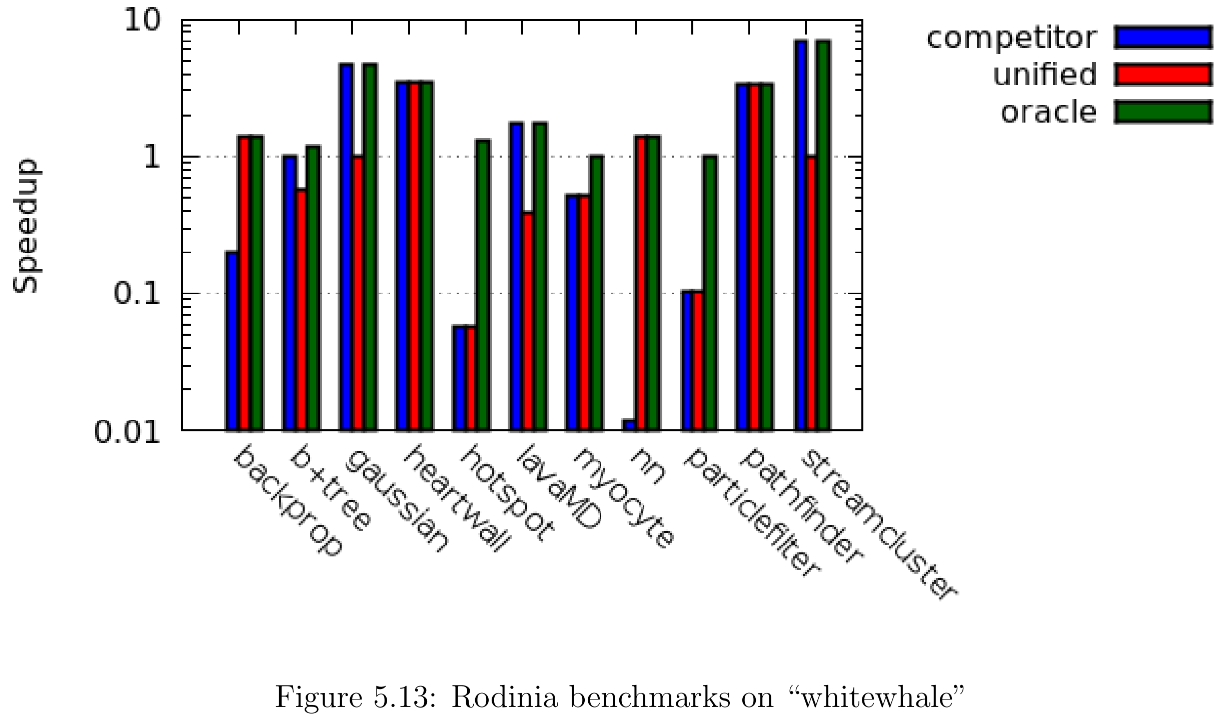


Figure 5.13: Rodinia benchmarks on "whitewhale"

work items of a work group are executed by a single thread on the device [27]. The kernels do not provide enough work items to reach the hardware-multithreading limits of the device, meaning that the obtained arithmetic throughput is greatly reduced. We could add features that indicate the amount of work per work item, similarly to the competitor's feature set, to alleviate this problem.

The unified model also mispredicts `cutcp`, scheduling its kernels onto the Xeon Phi when they should be scheduled onto the Tesla. This is due to the extensive use of local memory and barriers within the `cutcp` kernel. As mentioned in the OpenCL optimization guide for the Xeon Phi, there is no hardware support for barriers in the Xeon Phi (while NVidia GPUs contain special hardware for barriers) [27]. Additionally, there is no local memory for the Xeon Phi – it is emulated in global memory and causes performance degredation. The kernels in `cutcp` use barriers extensively, constantly flushing data to the emulated local memory and hammering the weaknesses of the Xeon Phi. We could better inform our model of this problem by including features for synchronization APIs in our feature set.

The third setup is the only setup where the unified model fails to outperform the competitor's model. The unified model obtains a speedup of 1.49x, while the competitor obtains a speedup of 1.92x out of a possible 2.42x. This is due to the failure of the feature set to describe the architectural characteristics of the Xeon Phi relative to the Tesla.

5.3.5 Discussion

Figure 5.14 shows the overall speedups obtained by the two models relative to the possible speedup (represented by the oracle). In general, we believe our feature set better quantifies the OpenCL applications than the competitor's feature set, as we are able to obtain comparable speedups to the competitor while evaluating the model's predictions on unseen benchmarks and unseen architectures (while the competitor only make predictions for unseen benchmarks). Any speedup over the competitor is significant, as the competitor's predictions are based on models generated from hours worth of training data on a particular system, giving those models a chance to learn implicit architectural details. Alternatively, our model makes predictions for the same system without any training data, which also explains why our models fail to learn some of the minute architectural details of the Xeon Phi, making the predictions less accurate on "whitewhale".

There were some perfomance quirks of several architectures in the system:

1. The Radeon has a comparable buffer transfer bandwidth (i.e. host-to-device or device-host transfers) to the Titan, but has a significantly higher transfer latency (or at least transfer startup cost). When transferring four bytes (the smallest size allowed by the buffer bandwidth benchmark) between the host and device 100,000 times, the Titan had a latency of 2.5μs per transfer, whereas the Radeon had a latency of 31.8μs per transfer. This manifests itself negatively in applications that have small runtimes (meaning data transfers take up a larger percentage of execution time, such

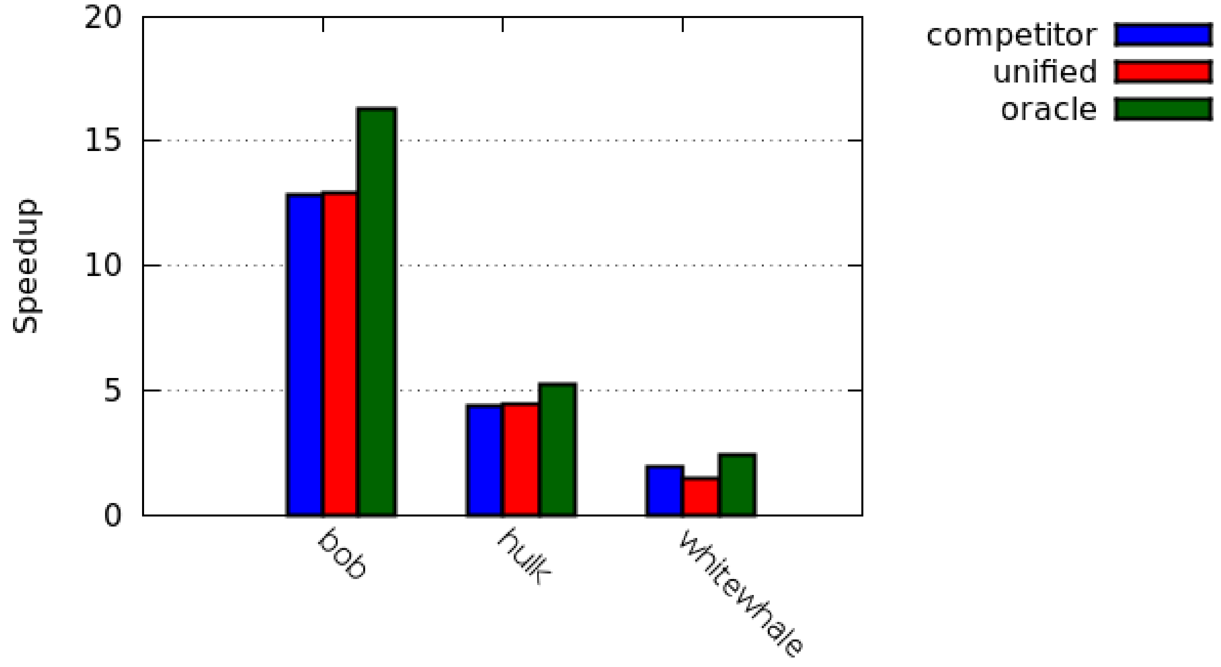


Figure 5.14: Overall speedups per system

as `bfs`), whereas applications that have large runtimes see an overall benefit to using the Radeon, as it has a higher overall arithmetic throughput (such as `gem`).

2. Although the Intel Xeon Phi's peak memory throughput on the device is 240 GB/s, our hardware benchmarking could only reach a maximum of 29.88 GB/s. This indicates that the OpenCL implementation allocates memory to consecutive memory controllers out of the 12 memory controllers on-chip, limiting the overall available memory bandwidth. Because OpenCL has no means of specifying memory in NUMA architectures, the memory bandwidth of OpenCL kernels is inherently limited by the OpenCL implementation. Other architectures reached near-theoretical memory bandwidth in our testing.

While both models are able to obtain a majority of the possible speedups, we believe several additions would further improve our technique. We predict that using more systems would improve the prediction accuracy of the unified model as it would give the model more information on which to train – it is hard to train any model of sufficient complexity using only 8 datapoints. Additionally, we believe the added features from above (buffer transfer latency, computation per work-item and synchronization) would increase the accuracy of the model and improve some of the main sources of performance degredation. Finally, we believe introducing some form of profile-guided scaling (i.e. profiling the kernel would to get accurate counts of basic block execution) would greatly improve the prediction accuracy of the model.

Chapter 6

Conclusion

In this thesis, we have shown that when scheduled correctly, utilizing parallelism & heterogeneity for high-performance applications can lead to drastic performance improvements. Making this scheduling decision is a non-trivial problem, especially with the proliferation of architectures that target a variety of design points. Mapping applications to architectures requires some form of quantitative characterization and decision making. We developed several feature extractors for OpenMP & OpenCL compute kernels, and used these extracted features to schedule compute kernels onto different architectures in a system using trained machine learning models. Additionally we demonstrated the ability of these models to adapt the scheduling decision in the presence of varying external workload on the system.

6.1 Contributions

This work consists of the following contributions:

1. **We developed a partitioner, which automatically refactored applications to be scheduled and executed on heterogeneous architectures.** The partitioner analyzed the code to determine on which architectures a compute kernel could execute, and then refactored compute kernels into a set of partition boundaries and device-specific partitions to execute the compute kernel. The scheduling decision was made by an external scheduling daemon, callable from a client-side API. The partitioner additionally handled all data transfer to and from the device automatically, with the help of a runtime memory management library.

2. **We developed feature extractors to extract features from both OpenMP & OpenCL compute kernels.** These extractors, built on top of GCC and LLVM, extracted compute-kernel features such as the number and types of operations executed

by the compute kernel, in addition to higher-level pattern features such as cyclomatic complexity and memory access patterns.

3. **We developed a central scheduling daemon to make scheduling decisions for individual applications using trained machine learning models.** Applications sent the scheduler a set of compute kernel features using IPC, which in combination with other features (external workload features for OpenMP applications and hardware features for OpenCL applications) were used to make a scheduling decision. The logic for the scheduling decision was implemented using an ANN, trained using supervised machine learning. The generated machine-learning model was generated on a per-system basis for OpenMP applications, but extended into a unified model for all architectures for OpenCL appliations. The scheduling decision was subsequently returned to the applications, which executed the device-specific implementation specified by the scheduler.

4. **Using this framework, we were able to accurately predict the relative efficiency of a compute kernel on an architecture, and were able to adjust this prediction in the presence of external workload.** For OpenMP applications, when using a workload-aware model we were able to see a 4x speedup over using a static machine learning model. For OpenCL applications, we generated models that were !!TODO!!% more accurate than state-of-the-art models, and incorporated hardware features that made the model truly portable for previously unseen architectures and systems.

5. **We additionally discovered the poor performance of the Linux scheduler when oversubscribing CPU threads to cores.** Because of the enormous number of context switches caused by an overload of CPU threads, compute kernels should be scheduled onto CPU cores using some form of spatial scheduling to eliminate these conflicts.

Chapter 7

Future Work

Although we demonstrated several strong research contributions, there are further implementation details that can be addressed, in addition to solidifying the research.

7.1 Partitioner

As mentioned in section 4.2.1, the partitioner could be made more robust and several usability features could be added:

- The partitioner as its currently structured depends on the programmer to annotate the source code with OpenMP pragmas, directing the partitioner to potential compute kernels. Ideally the partitioner could take arbitrary C source code and determine where the compute kernels were inside to the code, meaning that refactor an application from being single threaded to being multi-threaded and able to be executed on heterogeneous architectures.
- Several of the restrictions enforced in the "Find Compatible Architecture" analysis pass could be lifted by implementating or utilizing a structure serialization library that can handle arbitrary-dimension pointers in addition to more complex abstract data types, such as lists or trees. Although this might be to the detriment of performance, it would allow usability on a wider variety of applications.
- ROSE has limited support for preprocessor macro/pragma handling due to using a discrete frontend that lowers the parse tree into an AST for use by ROSE; this requires the programmer to sometimes manually add `#include` directives for the memory management and the client-side scheduler library (additionally, the programmer must manually specify which device-specific partitions must be generated via a `popcorn` pragma, a step that could be removed).

- The partitioner currently supports a CUDA backend (which essentially copies the compute kernel into a separate file for OpenMP-to-CUDA refactoring) and an MPI backend. Support could be added for other backends, such as OpenCL.
- The partitioner currently handles all data movement automatically. Some of the problems introduced by this process could be eliminated by user-specified data movement. Additionally, some form of lazy data movement could be implemented to eliminate redundant data transfers so that data is only moved when absolutely necessary.
- The partitioner currently schedules at every compute-kernel boundary. This should be a configurable parameter to allow adjusting the scheduling granularity and frequency to mitigate some of those scheduling costs.
- The partitioner should be tested on larger applications to further expose issues that did not arise with the smaller benchmark applications.

7.2 Scheduling with External Workload

Although we were able to generate a system-specific model to adapt scheduling decisions to external workload, this solution is not portable to other systems. In addition, it could be extended to handle a wider variety of situations and to actively counteract scheduler thrashing introduced by oversubscribing cores by using spatial scheduling:

- Some of the related work utilized other scheduling techniques, such as actual task queues used to regulate compute kernel execution on individual devices (instead of the simple counters utilized by our scheduler). This would avoid the problem of oversubscribing individual devices, especially devices where applications are able to run concurrently. With these task queues, further load balancing techniques could be applied (such as work stealing).
- Another possible approach to spatial scheduling would be to utilize OpenCL *device fission*, a feature introduced as an extension to OpenCL 1.1 and integrated into the core specification in version 1.2, allows the application to create *sub-devices* that contain a subset of the compute units in a device. This could be used by the scheduler to allocate non-overlapping sets of compute units to applications to avoid scheduler conflicts (e.g. give applications A and B four CPUs of an eight CPU processor). We initially considered this idea for the OpenCL work, but this feature is unfortunately only supported on CPUs at this point.
- Applications that are too small to benefit from parallelization or heterogeneity should be filtered out by analysis and removed from the scheduling process.

7.3 Feature Selection/Extraction

Currently our set of features consists of a hand-picked characteristics that we believe identify how compute kernels map to the underlying architecture. Other machine learning work ([37]) has shown, however, that seemingly strange features may allow machine learning models to make better predictions. Implementing a feature grammar and subsequently utilizing some algorithm (such as genetic programming) to automatically generate the feature set may lead to better predictions, or predictions for non-traditional architectures not evaluated in this work (such as FPGAs or DSPs).

Additionally, the static heuristics used by the feature extractor in section 5.2.2 to estimate basic block counts do not and cannot accurately predict runtime performance of the kernel. Therefore, re-introducing profiling-scaled features would significantly improve our prediction accuracy for OpenCL benchmarks. Additionally, some form of hybrid approach could be used to make the scaling of features more portable – a single work group from a kernel invocation could be executed to generate a scaling factor which could subsequently be used to make the scheduling decision.

7.4 Further Evaluation

While we have evaluated the unified model on server and desktop-grade architectures, it would be beneficial to also evaluate the model on embedded and SoC architectures (many of which leverage higher levels of integrated heterogeneity for increased performance). Having a single unified model for architectures on a broad spectrum of size and design increases the portability of the model to a wide variety of settings.

Additionally, we would like to evaluate implementations of OpenMP 4.0 as mature and robust implementations become available. The performance of the generated device code largely depends on the ability of the compiler to generate kernels that map well to the underlying device architecture. This can have drastic effects on the mapping decision of the scheduler.

Bibliography

[1] Intel previews future 'knights landing' xeon phi x86 coprocessor with integrated memory - the register, June 2013. http://www.theregister.co.uk/2013/06/17/intel_knights_landing_xeon_phi_fabric_interconnects/.

[2] Samsung teams up with mozilla to build a browser engine for multicore machines — ars technica, April 2013. http://arstechnica.com/information-technology/2013/04/samsung-teams-up-with-mozilla-to-build-browser-engine-for-multicore-machines/.

[3] Top 500 list – november 2013 — top500 supercompute sites, November 2013. http://www.top500.org/list/2013/11/.

[4] Qualcomm announces the ultimate connected computing next-generation snapdragon 810 and 808 processors, April 2014. http://www.qualcomm.com/media/releases/2014/04/07/qualcomm-announces-ultimate-connected-computing-next-generation-snapdragon.

[5] AMD. Amd graphics cores next (gcn) architecture. Technical report, AMD, 2012. http://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf.

[6] AMD. Software optimization guide for amd family 15h processors. Technical report, AMD, January 2012. http://developer.amd.com/wordpress/media/2012/03/47414_15h_sw_opt_guide.pdf.

[7] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

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

[9] Mateusz Berezecki, Eitan Frachtenberg, Mike Paleczny, and Kenneth Steele. Many-core key-value store. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–8. IEEE, 2011.

[10] The OpenMP Architecture Review Board. Openmp.org, April 2014. http://www.openmp.org.

[11] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, 2009.

[12] Shuai Che, Jeremy W Sheaffer, Michael Boyer, Lukasz G Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–11. IEEE, 2010.

[13] George Chrysos. Intel(r) xeon phi(tm) coprocessor – the architecture. Technical report, Intel, 2012. http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner.

[14] GNU Compiler Collection. Gimple - gnu compiler collection (gcc) internals, April 2014. http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html.

[15] GNU Compiler Collection. Gomp - an openmp implementation for gcc - gnu project - free software foundation (fsf), April 2014. http://gcc.gnu.org/projects/gomp/.

[16] Advanced Micro Devices. App sdk — amd, April 2014. http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/.

[17] Murali Krishna Emani, Zheng Wang, and Michael FP O'Boyle. Smart, adaptive mapping of parallelism in the presence of external workload. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10. IEEE, 2013.

[18] Wu-chun Feng, Heshan Lin, Thomas Scogland, and Jing Zhang. Opencl and the 13 dwarfs: a work in progress. In *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering*, pages 291–294. ACM, 2012.

[19] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, et al. Milepost gcc: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39(3):296–327, 2011.

[20] Dominik Grewe and Michael FP OBoyle. A static task partitioning approach for heterogeneous systems using opencl. In *Compiler Construction*, pages 286–305. Springer, 2011.

[21] Dominik Grewe, Zheng Wang, and Michael FP OBoyle. Opencl task partitioning in the presence of gpu contention.

[22] Dominik Grewe, Zheng Wang, and Michael FP O'Boyle. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10. IEEE, 2013.

[23] Systems Software Research Group. Popcorn linux, April 2014. http://www.popcornlinux.org/.

[24] The Khronos Group. Opencl - the open standard for parallel programming of heterogeneous systems, April 2014. http://www.khronos.org/opencl/.

[25] The Portland Group. Pgi(r) compilers & tools, April 2014. http://www.pgroup.com/index.htm.

[26] IBM. Ibm xl c/c++ for aix and linux, v12.1, May 2012. http://www-01.ibm.com/support/docview.wss?uid=swg27027518&aid=1.

[27] Intel. Opencl design and programming guide for the intel(r) xeon phi(tm) coprocessor. Technical report.

[28] Intel. Intel open source openmp runtime — intel openmp* runtime, April 2014. http://www.openmprtl.org/.

[29] Intel. Intel(r) 64 and ia-32 architectures optimization reference manual. Technical report, Intel, March 2014. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[30] Intel. Intel(r) sdk for opencl(tm) applications xe 2013 r3, April 2014. https://software.intel.com/en-us/vcsource/tools/opencl-sdk-xe.

[31] Intel. Intel(r) vtune(tm) amplifier xe 2013, April 2014. https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[32] Víctor J Jiménez, Lluís Vilanova, Isaac Gelado, Marisa Gil, Grigori Fursin, and Nacho Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *High Performance Embedded Architectures and Compilers*, pages 19–33. Springer, 2009.

[33] Andrew Josey. The single unix specification version 4. *Open Group*, 2013.

[34] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. Snucl: an opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 341–352. ACM, 2012.

[35] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 149–160. ACM, 2013.

[36] Lawrence Livermore National Laboratory. Rose compiler infrastructure, April 2014. http://rosecompiler.org/.

[37] Hugh Leather, Edwin Bonilla, and Michael O'Boyle. Automatic feature generation for machine learning based optimizing compilation. In *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, pages 81–91. IEEE, 2009.

[38] Sang-Ik Lee, Troy A Johnson, and Rudolf Eigenmann. Cetus–an extensible compiler infrastructure for source-to-source transformation. In *Languages and Compilers for Parallel Computing*, pages 539–553. Springer, 2004.

[39] Seyong Lee and Rudolf Eigenmann. Openmpc: Extended openmp programming and tuning for gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[40] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, 44(4):101–110, 2009.

[41] Chunhua Liao, Yonghong Yan, Bronis R de Supinski, Daniel J Quinlan, and Barbara Chapman. Early experiences with the openmp accelerator model. In *OpenMP in the Era of Low Power Devices and Accelerators*, pages 84–98. Springer, 2013.

[42] Michael D Linderman, Jamison D Collins, Hong Wang, and Teresa H Meng. Merge: a programming model for heterogeneous multi-core systems. In *ACM SIGOPS operating systems review*, volume 42, pages 287–296. ACM, 2008.

[43] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 45–55. IEEE, 2009.

[44] Marvin L Minsky and Seymour A Papert. *Perceptrons - Expanded Edition: An Introduction to Computational Geometry.* MIT press Boston, MA:, 1987.

[45] NVidia. Nvidia's next generation cuda(tm) compute architecture: Fermi. Technical report, NVidia, 2009. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_W

[46] NVidia. Nvidia's next generation cuda(tm) compute architecture: Kepler gk110. Technical report, NVidia, 2012. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[47] NVidia. Parallel programming and computing platform — cuda — nvidia — nvidia, April 2014. http://www.nvidia.com/object/cuda_home_new.html.

[48] OpenACC. Openacc home — openacc.org, April 2014. http://www.openacc.org/.

[49] The Open MPI Project. Open mpi : Open source high performance computing, April 2014. http://www.open-mpi.org/.

[50] Reza Rahman. Intel(r) xeon phi(tm) core micro-architecture. Technical report, Intel, May 2013. http://https://software.intel.com/en-us/articles/intel-xeon-phi-core-micro-architecture.

[51] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.

[52] Thomas RW Scogland, Barry Rountree, Wu-chun Feng, and Bronis R de Supinski. Heterogeneous task scheduling for accelerated openmp. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 144–155. IEEE, 2012.

[53] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and W-m Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[54] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3):202–210, 2005.

[55] Peter Thoman, Klaus Kofler, Heiko Studt, John Thomson, and Thomas Fahringer. Automatic opencl device characterization: guiding optimized kernel design. In *Euro-Par 2011 Parallel Processing*, pages 438–452. Springer, 2011.

[56] Tilera. Tile processor architecture overvew for the tile-gx series. Technical report, Tilera, 2012. http://www.tilera.com/products/processors/TILE-Gx_Family.

[57] Open Source Computer Vision. Opencv — opencv, April 2014. http://www.opencv.org.

[58] Yao Zhang, Mark Sinclair II, and Andrew A Chien. Improving performance portability in opencl programs. In *Supercomputing*, pages 136–150. Springer, 2013.