Automatic program analysis and scheduling of scientific workloads for heterogeneous supercomputers

Beau Johnston

July 2018

A thesis submitted for the degree of Doctor of Philosophy
The Australian National University

Declaration

The work in this thesis is my own except where otherwise stated.

Beau Johnston

Acknowledgements

Thank you to our colleagues at The University of Bristol's High Performance Computing Research group for the use of `The Zoo` Research cluster for experimental evaluation which was critical to generating the runtime results.

Abstract

The next-generation of supercomputers will feature a diverse mix of accelerator devices. The increase of heterogeneity is explained by the nature of these devices – certain accelerators offer acceleration, or a shorter time to completion, for particular programs. Characteristics of these programs ~~is fixed, such that, say, a graph traversal program always exhibits the characteristics of a graph traversal program~~ regardless of which accelerator is used for computation. ~~Thus,~~ this work presents a methodology to collect these characteristics and use them to inform the selection of optimal accelerator device. ~~The usefulness of this work is more general, since the trend of having heterogeneous nodes is becoming increasingly applicable to general purpose high-performance computing systems, where currently, it is not uncommon for a GPU, a CPU co-processor and an FPGA or MIC to exist on a single node.~~ The focus of this work ~~then~~ is to schedule scientific codes to the most suitable device on a node which offers a more efficient system – shorter execution times and less energy expenditure.

OpenCL is an attractive programming model for high-performance computing systems, with wide support from hardware vendors it is a highly portable language – a single implementation can execute on CPU, GPU, MIC and FPGA alike. To support efficient scheduling on HPC systems it is necessary to perform accurate performance predictions for OpenCL workloads on varied compute devices, which is challenging due to diverse computation, communication and memory access characteristics which result in varying performance between devices.

The first focus of this work is to present a comprehensive benchmark suite for OpenCL in the heterogeneous HPC setting: an extended and enhanced version of the Open-Dwarfs OpenCL benchmark suite. ~~These~~ extensions ~~were developed to~~ improve portability and robustness ~~of applications~~, correctness of results and choice of problem size, and increase diversity through coverage of additional application patterns. The culmination in this work results in measurable performance on a set 15 devices and over 11 applications.

The Architecture Independent Workload Characterization (AIWC) tool ~~was then developed to characterize~~ OpenCL kernels according to a set of architecture-independent features. Features are measured by counting ~~desired~~ characteristics which are collected during program execution in a simulator. They are presented as a ~~series~~ of 42 metrics that indicate performance bottlenecks ranging from parallelism – how well an algorithm scales in response to core count, compute – such as the diversity of instructions, memory – working memory footprint and entropy measurements which correspond to caching characteristics and control – such as branching and program flow. The metrics collected are primarily used in the prediction of execution times, but since they are representative of structural characteristics of the underlying program and are free from architectural traits, they can be used in diversity analysis in benchmark suites, identifying program requirements which allows the automatic calculation of theoretical peak performance for a given device and examining phase-transitional properties of application codes. This work also discusses the design decisions made to collect AIWC features.

Finally, this work culminates in a methodology which uses AIWC features to form a model

capable of predicting accelerator execution times. I use this methodology to predict execution times for a set of 37 computational kernels running on 15 different devices representing a broad range of CPU, GPU and MIC architectures. The predictions are highly accurate, differing from the measured experimental run-times by an average of only 1.2%. A previously unencountered code can be instrumented once and the AIWC metrics embedded in the kernel, to allow performance prediction across the full range of modeled devices. The results suggest that this methodology supports correct selection of the most appropriate device for a previously unencountered code, and is highly relevant to efficiently scheduling codes to the emerging supercomputing systems where nodes are becoming increasingly heterogenous.

Given the need for more efficient super-computers it is believed that this research is well timed.

Contents

List of Figures

List of Tables

Abbreviations

HPC **H**igh **P**erformance **C**omputing
SC **S**uper **C**omputing
CPU **C**entral **P**rocessing **U**nit
GPU **G**raphics **P**rocessing **U**nit
FPGA **F**ield-**P**rogrammable **G**ate **A**rray
DSP **D**igital **S**ignal **P**rocessor
ASIC **A**pplication-**S**pecific **I**ntegrated **C**ircuit
MIC **M**any **I**ntegrated **C**ore
OpenCL **O**pen **C**omputing **L**anguage
SoC **S**ystem-**o**n-a-**C**hip
ISA **I**nstruction **S**et **A**rchitecture
PCA **P**rincipal **C**omponent **A**nalysis

Chapter 1

Introduction

Supercomputers are becoming increasingly heterogeneous. At an individual node, there is an increasing trend to use specialised hardware – known as accelerators – which can expedite the computation of codes from particular classes of scientific workloads. The use of accelerators for certain programs offers a shorter time to completion, and less energy expenditure, when compared to a conventional CPU architecture. The next generation of these systems has been designed to incorporate a greater number of accelerators, and of varying types per node. For instance, the CAPI and NVLINK technologies included in the latest IBM POWER9 processor offers a high-speed interconnect which allows the rapid movement data between processor and accelerator – NVIDIA Graphical Processing Unit (GPU) with NVLINK, and CAPI supporting Altera Field-Programmable Gate Array (FPGA), Intel Central Processing Unit (CPU) co-processors to AMD CPU and GPU devices. The POWER9 is featured in the latest Summit and forthcoming Sierra Supercomputers, and is configured such with two GPUs per CPU. High bandwidth, low latency interconnects such as the CRAY XC50 *Aries*, Fujitsu Post-K *Tofu* and IBM Power9 *Bluelink*, support tighter integration between compute devices on a node. Some interconnects support multiple different kinds of devices on a single node, for example *Bluelink* features both NVLINK support for Nvidia GPUs and CAPI for other emerging accelerators such as DSPs, FPGAs and MICs. The support from hardware vendors for a greater mix of heterogeneous devices indicates this is the new direction of supercomputing. However, this development is recent, and as such the scheduling of workloads to the suitable accelerator is a new problem. Without significant improvements in effectively using accelerators on these future systems, the cost of exascale computing and their corresponding energy efficiency will be prohibitive.

Independently, the characteristics of a scientific code, specifically around computation, memory, branching and parallelism, are independent of any particular device on which they may be finally executed. The metrics used to quantify each of these characteristics can be collected during program execution on a simulator. In other words, provided they are collected over a representative workload, a graph traversal program maintains the characteristics of a graph traversal program regardless of problem size. Moreover, these metrics can be used to accurately predict the execution time on each accelerator in a heterogeneous system.

This thesis outlines the methodology required to perform runtime predictions for any given code – provided the feature metrics are pre-generated – for any accelerator device. A benchmark suite is extended, a characterisation tool developed, and a model is generated to achieve the task. This research is of benefit to the scheduling of codes to the most appropriate device which in turn provides essential information for scheduling, to better utilise the next-generation of supercomputers.

1.1 Context

The Open Compute Language (OpenCL) allows programs to be written once and run anywhere on a range of accelerators. A majority of accelerator vendors ship products with an OpenCL supported runtime, many of which will be components on the next-generation of supercomputing nodes. Programs in the OpenCL setting are structured into two partitions, the host and the accelerator/device side. As such, the developer is manually responsible for allocating and transferring memory between the host and device. This requires programs to be restructured with computationally intensive regions of code – known as kernels – to be identified and partitioned – and rewritten in the OpenCL C kernel language. Kernels are viewed as indivisible functions and as such the nature of these kernels is fixed for all executions, and as such, a kernel does not suffer from the phase-transitions that are common when looking at larger scientific codes. The compositions of all kernels form a full accelerator agnostic implementation for any larger scientific code.

A benefit of the fixed/static nature of OpenCL kernels is that the collection of the characteristics is also constant. Such that, instrumentation of a kernel to measure computation, memory, branching and parallelism characteristics of a program are largely unchanged between run and are independent of data set. To this end, the Architecture Independent Workload Characterisation (AIWC) tool has been developed. This tool collects 40+ metrics that indicate computation, memory, branching and parallelism characteristics on a per OpenCL kernel basis. It simulates an OpenCL device using the Oclgrind tool and the AIWC plugin analyses the program trace, memory locations accessed, and thread-states to generate the metrics. Metrics can be collected quickly since it is a multi-threaded simulator. AIWC features, are generated for each kernel invocation and can be embedded as a comment into the header of OpenCL kernel codes – either in plain-text source or in the SPIR format.

Separately, additional work in this thesis comprises of the extension of a benchmark suite. This was needed since programs that are representative of scientific HPC applications which are capable of execution over a wide range of accelerators are few and far between, specifically with portable performance and reproducible results. Additionally, until this work was undertaken, the available OpenCL benchmark suites were not rich enough to adequately characterise performance across the diverse range of applications or accelerator devices of interest. Thus this thesis presents an enhanced version of the OpenDwarfs OpenCL benchmark suite – denoted the Extended OpenDwarfs Benchmark Suite (EOD) – which was developed with a strong focus placed on the robustness of applications, the curation of additional benchmarks with an increased emphasis on correctness of results and the selection of 4 problem sizes.

LibSciBench was added to EOD, this includes high precision timers along with support for the collection of PAPI – hardware performance counters – events and energy usage information. Runtime, or elapsed execution times, of all EOD benchmarks, were collected on 15 unique accelerator devices suitable for current HPC systems. Collection of these times occurs at a per kernel level along with instrumentation of other events common to the OpenCL setting, such as memory setup and timing data movement to accelerator devices. In addition to the higher level, total elapsed application execution time was also collected.

The final major contribution of this thesis is the development and use of a predictive model, using the random forest algorithm – a supervised learning algorithm and powerful pattern recognition technique – to show the link between AIWC features and execution times over all devices. Thus, the AIWC tool was run and the features collected from all the kernels of EOD. These AIWC metrics were used as predictor variables into the random forest, and the

time data of kernels from the experimental methodology was used as the response variables to indicate predictions. The accelerators examined in these predictions range from CPU, GPU and MIC, however, the methodology finally presented is expected to perform over DSP and FPGA also.

The final model performs very well and is capable of highly accurate predictions which on average differ from the measured experimental run-times by 1.1%, which correspond to actual execution time mispredictions of 8 μs to 1 secs according to problem size. The model is capable of predicting execution times for specific devices based on the computational characteristics captured by the AIWC tool, which in turn, provides a good prediction of an accelerator devices execution time needed for a real-world scheduler for nodes of future super-computing systems.

1.2 Problems in heterogeneous supercomputing

The future of supercomputing comprises several heterogeneous devices at the node level. Evaluating the suitability of any given device on a node requires a comprehensive benchmark suite which is capable of efficiently executing on all devices in a hardware agnostic way. Unfortunately, current benchmark suites are ill-suited to the task, either consisting of several different implementations per each device or lacking a comprehensive range of scientific applications to fully explore the performance characteristics of the device. Further, this suitability can be concerned with energy consumption, which is critical to the proposed exascale systems envisaged in the future, making performance-per-watt a fundamental concern. Additionally, examining the computation characteristics of scientific workloads is difficult, and this complexity only increases when considering the wide range of hardware in heterogeneous supercomputing – and the corresponding different implementations per device. Both the difficulties in identifying characteristics of scientific hardware agnostic codes, and the wider diversity of devices of the next-generation of HPC systems further compounds the issue of scheduling code within a node in order to fully utilise supercomputing facilities.

1.3 Thesis Contributions

A benchmark suite is extended to include a greater range of scientific applications and over a differing problem sizes. Additionally, the extended suite incorporates a high precision timing library which is capable of measuring energy usage and execution times on any OpenCL device. Examining the performance of the benchmark suite over a range of devices allows a direct evaluation to be made between these devices on a per application basis. From this evaluation, the suitably of OpenCL as a hardware agnostic language is shown.

Architecture Independent Workload Characterisation (AIWC) tool is capable of analysing kernels in order to extract a set of predefined features or characteristics. The benefits of AIWC include that it:

1) provides insights around the inclusion of an application via diversity analysis of the feature-space.
2) measures requirements in terms of FLOPs, memory movement and integer ops of any application kernel – which allows the automatic calculation of theoretical peak performance for a given device.

3) can be used to examine the phase-transitional properties of application codes – for instance if the instruction mix changes over time in terms of the balance between floating-point and memory operations. The tool can be used in diversity analysis – which is essential when assembling benchmark suites and justifying the inclusion of an application. Furthermore, these metrics are used for creating the prediction model to evaluate the performance of OpenCL kernels on different hardware devices and settings. Such a model is then applied as a prognosis tool to predict the performance of an application for any given platform without additional instrumentation. This prediction adds information that can be incorporated into existing HPC schedulers and has no run-time overhead – codes are examined one time by the developer when instrumenting with AIWC and these, in turn, are embedded into the header of each kernel code to be evaluated by the scheduler at the time of scheduling.

1.4 Thesis Structure

Chapter 2 canvasses the existing literature and current techniques used to schedule heterogeneous resources. Chapter 3 discusses the extensions added to the OpenDwarfs Benchmarking Suite in EOD. Chapter 4 highlights the construction, design decisions made and usage of the AIWC tool. Chapter 5 develops the prediction model and examines the accuracy of the final predictions. Chapter 6 discusses conclusions of this thesis and the future work required for the predictive model to be incorporated into scheduling on future Supercomputing systems.

Chapter 2

Background Information and Related Work

The chapter presents background information, terminology and the related work drawn upon in the rest of this thesis. It provides a background for readers who might not be familiar with workload characterisation of programs, the associated performance metrics or composition of current HPC systems and how their performance is evaluated. It begins with the definition of accelerators and a brief survey regarding their use in supercomputing and presents the hardware agnostic programming framework – OpenCL. The dwarf taxonomy is introduced along with a representative sample of benchmark suites which incorporate this taxonomy.

2.1 Accelerator Architectures ~~and their increasing diversity~~ in HPC

Accelerators, in this setting, refer to any form of specialised hardware which may accelerate a given application code. These include CPU, GPU, FPGA, DSP, ASIC and MIC devices. High-performance computing (HPC) hardware is becoming increasingly heterogeneous using a larger number of accelerators on a node. A major motivation for this is to reduce energy use; indeed, without significant improvements in energy efficiency, the cost of exascale computing will be prohibitive. From June 2016 to June 2017, the average energy efficiency of the top 10 of the Green500 supercomputers rose by 2.3x, from 4.8 to 11.1 gigaflops per watt [1]. For many systems, this was made possible by highly energy-efficient Nvidia Tesla P100 GPUs. In addition to GPUs, future HPC architectures are also likely to include nodes with FPGA, DSP, ASIC and MIC components. A single node may be heterogeneous, containing multiple different computing devices; moreover, an HPC system may offer nodes of different types. For example, the Cori system at Lawrence Berkeley National Laboratory comprises 2,388 Cray XC40 nodes with Intel Haswell CPUs, and 9,688 Intel Xeon Phi nodes [2]. The Summit supercomputer at Oak Ridge National Laboratory is based on the IBM Power9 CPU, which includes both NVLINK [3], a high bandwidth interconnect between Nvidia GPUs; and CAPI, an interconnect to support FPGAs and other accelerators [4]. Promising next-generation architectures include Fujitsu's Post-K [5], and Cray's CS-400, which forms the platform for the Isambard supercomputer [6]. Both architectures use ARM cores alongside other conventional accelerators, with several Intel Xeon Phi and Nvidia P100 GPUs per node.

what are all of these? How are they different in ways that are important to your target problem? 2018?

what does it all mean for programming, performance prediction & scheduling?

2.2 The Open Compute Language Setting

OpenCL (Open Compute Language) is a standard that allows computationally intensive codes to be written once and run efficiently on any compliant accelerator device. OpenCL is supported on a wide range of systems including CPU, GPU, FPGA, DSP and MIC devices. While it is possible to write application code directly in OpenCL, it may also be used as a base to implement higher-level programming models. This technique was shown by Mitra et al., [7] where an OpenMP runtime was implemented over an OpenCL framework for Texas Instruments Keystone II DSP architecture. Having a common back-end in the form of OpenCL allows a direct comparison of identical code across this diverse range of architectures.

OpenCL programs comprise of a host side and a device side and the program progression is always the same. The Host is responsible for querying the suitable platforms, vendor OpenCL runtime drivers, and establishing a context on the selected devices. Next, the host sets up memory buffers, compiles a kernel program for each device – the final compiled device binaries are generated for each specific device instruction set architecture (ISA).

On the device side, the developer code is en-queued for execution. Device-side code is typically small intensive sub-regions of programs and is known as the kernel. Kernel code is written in a subset of the C programming language. Special functions exist to determine a threads id, this can occur via getting a global index in a given dimension directly, with `get_group_id`, or determined using `get_group_id`, `get_local_size` and `get_local_id` in each dimension.

The host side is then notified once the device has completed execution – this takes the form of either the host waiting on the `clFinish` command or if the host does not the computed results yet, say for an intermediate result on which a second kernel will operate on the same data, a `clFlush` function call. Once all device execution has completed and the host has been notified the results are transferred back to the host from the device. Finally, the context established on the device is freed.

The selection of parameters on the host side can have a large impact on performance. One primary reason is that different accelerators benefit from different levels of parallelism – or how many threads are executed concurrently – for instance, GPU devices usually need a high degree of arithmetic intensive parallelism to offset the (relatively) narrow I/O pipeline, CPUs on the other hand are more general purpose and the switching of threads has a greater penalty on performance. The tuning of such parameters can positively impact performance, in the OpenCL setting by primarily influencing the workgroup size. In essence, the global work items can be viewed from the data-parallelism perspective. Global work indicates the number of threads or instances of a kernel to execute in total. Additionally, these work items can be run in teams – denoted local work groups. Each local work group has a given size, and as previously mentioned can be determined on the device side, in the kernel code, with `get_local_id`. Incorrectly setting the number of local work groups and therefore also the size of each work group can impact on performance directly. Thankfully recent work shows these parameters can be automatically optimised for any accelerator architecture and is discussed in the Autotuning Section 2.5 further on in this chapter. OpenCL codes can be written to be easily linked with auto-tuners – such as allowing the local work group size being set from the command line or as a macro in the pre-processor, these are set during execution and during compilation respectively.

Kernel compilation flags are an additional tuning argument which affects runtime performance of accelerator specific OpenCL kernel codes. These flags are set on the host side during the

Dwarf	Performance Limit
Dense Linear Algebra	Compute
Sparse Linear Algebra	Memory Bandwidth and Compute
Spectral Methods	Memory Latency
N-Body Methods	Compute
Structured Grid	Memory Bandwidth
Unstructured Grid	Memory Latency
Map Reduce	?
Combinational Logic	Memory Bandwidth and Compute
Graph Traversal	Memory Latency
Dynamic Programming	Memory Latency
Backtrack and Branch and Bound	?
Graphical Methods	?
Finite State Machines	?

Table 2.1: Dwarfs and their limits.

`clBuildProgram` procedure. Pre-processor macros can also be defined on the kernel side which allows various loop level parallelism constructs to be enabled or disabled. Mathematical intrinsic options can also be set to disable double floating point precision, and change how denormalised numbers are handled. Other optimisations include using the strictest aliasing rules, use of the fast fused multiply and add instruction (with reduced precision), ignoring the signedness of floating point zeros and relaxed, finite or unsafe math operations. ~~Thankfully,~~ these can also be corrected using autotuning for both kernel specific and device specific optimisations.

Really? Can an autotuner select settings that may affect the correctness of the program?

2.3 The Dwarf Taxonomy

This section jumps back and forth between Colella & Berkeley.

The Berkeley Dwarf taxonomy was initially performed by the Asanovic et. al. [8] and is grounded on the discussion on the seven dwarfs of high performance computing by Colella [9].

Rewrite to present an ordered exposition

It outlines that many applications in scientific computing share parallel patterns of communication and computation dispite being removed from specific implementations. From the original ~~of 7~~ dwarfs proposed by Colella, they were increased to 13 based on the study presented in this work. Clusters of applications with similar patterns of computation and communication are defined as being represented by a dwarf. There are 13 dwarfs in total. A summary of a diverse set of application benchmarks is presented and whilst it is believed that more dwarfs may be added to this list in the future all currently encountered scientific codes can be classified as belonging to one or more of these dwarfs. For each of the dwarfs presented the authors indicate the performance limit, in other words whether the dwarf is compute bound, memory latency limited or memory bandwidth limited. All dwarfs and the corresponding limiting factor are presented in Table 2.1. Note, the **?** symbol indicates the unknown performance limit at the time of publication.

Seven (use words for numbers less than ten)

What does this mean for accelerators?

2.4 Benchmark Suites

This section needs an opening paragraph

The NAS parallel benchmarks [10] follow a 'pencil-and-paper' approach, specifying the computational problems to be included in the benchmark suite but leaving implementation choices such as language, data structures and algorithms to the user. The benchmarks include varied kernels and applications which allow a nuanced evaluation of a complete HPC system, however, the unconstrained approach does not readily support direct performance comparison between different hardware accelerators using a single set of codes.

Martineau et al. [11] collected a suite of benchmarks and three mini-apps to evaluate Clang OpenMP 4.5 support for Nvidia GPUs. Their focus was on comparison with CUDA; OpenCL was not considered.

Barnes et al. [12] collected a representative set of applications from the current NERSC workload to guide optimization for Knights Landing in the Cori supercomputer. As it is not always feasible to perform such a detailed performance study of the capabilities of different computational devices for particular applications, the benchmarks described in this paper may give a rough understanding of device performance and limitations.

Rodinia and the original OpenDwarfs benchmark suite focused on collecting a representative set of benchmarks for scientific applications, classified according to dwarfs, with a thorough diversity analysis to justify the addition of each benchmark to the corresponding suite.

The Scalable Heterogeneous Computing benchmark suite (SHOC)[13], unlike OpenDwarfs and Rodinia, supports multiple nodes using MPI for distributed parallelism. SHOC supports multiple programming models including OpenCL, CUDA and OpenACC, with benchmarks ranging from targeted tests of particular low-level hardware features to a handful of application kernels.

?

All 3 benchmark suites discussed in this Section as they feature an OpenCL implementation and 2 of the 3 have been categorised according to the Dwarf Taxonomy.

Sun et al.[14] propose Hetero-Mark, a Benchmark Suite for CPU-GPU Collaborative Computing, which has five benchmark applications each implemented in HCC – which compiles to OpenCL, HIP – for a CUDA and Radeon Open Compute back-end, and a CUDA version. Meanwhile, Chai by Gómez-Luna et al.[15], offers 15 applications in 7 different implementations with the focus on supporting integrated architectures.

first time these are mentioned

These benchmark suites focus on comparison between languages and environments; whereas our work focuses on benchmarking for device specific performance limitations, for example, by examining the problem sizes where these limitations occur – this is largely ignored by benchmarking suites with fixed problem sizes. For these reasons, we introduce the enhanced OpenDwarfs benchmark suite in Chapter 3 which covers a wider range of application patterns by focusing exclusively on OpenCL using higher-level benchmarks.

2.4.1 Rodinia

Che et. al [16] initially proposed a benchmark suite ~~which~~ to cover a wide range of parallel communication patterns. ~~The selection of these patterns was inspired by the Berkeley dwarf taxonomy, as discussed in Section **??**, and the selection of these benchmarks are~~ The benchmarks were selected following the Berkeley dwarf taxonomy from real world high performance scientific computing applications. ~~Evaluated in the paper were~~ a NVIDIA GTX 280 GPU and an Intel Core 2 Extreme CPU. The diversity between selected

move to next page

benchmarks was shown by measuring execution times, communications overheads and energy usage of running each benchmark on the target architectures. Across the suite: speedups in execution times ranged from 5.5x to 80.8x, communication overheads vary from 2-76% and GPU power consumption overheads ranged from 38-83 Watts. From this, the resulting benchmarks were proven to be useful when illustrating important architectural differences between the CPU and GPU. However, all devices presented featured applications typical of select dwarfs which benefit from GPU architectures. At the time this paper was written the Rodinia Benchmark suite consisted of nine applications; namely, Leukocyte Tracking, Speckle Reducing Anisotropic Diffusion, HotSpot, Back Propagation, Needleman-Wunsch, K-means, Stream Cluster, Breadth-First Search and Similarity Score, but it has since been extended. This extension features a subset of the dwarfs, namely, Structured Grid, Unstructured Grid, Dynamic Programming, Dense Linear Algebra, MapReduce, and Graph Traversal. Diversity analysis was also performed and took the form of a Micro-Architecture independent analysis study. The MICA framework, discussed in Section 2.10, was used as the basis of the evaluation and the motivation was to justify each applications inclusion in the benchmark suite by showing deviations between applications in the corresponding kiviat diagrams. Separate implementations were developed for each application CUDA for the GPU, and OpenMP for the CPU, OpenCL was also included for both architecture types. Ultimately several applications from the Rodinia benchmark suite were added to the extended OpenDwarfs benchmark suite – developed in this thesis. However, ultimately, having several implementations caused fragmentation in development, where changes often resulted in the OpenCL version of each benchmark application being neglected, in some instances lacking an implementation of a given application entirely – or at the least, missing features offered in other implementations. For this reason, OpenDwarfs was selected as the benchmark suite on which to perform the extension work.

2.4.2 SHOC

The Scalable Heterogeneous Computing benchmark suite SHOC, presented by Danalis et. al. [17], offers an alternative benchmark suite and unlike OpenDwarfs and Rodinia, supports multiple nodes using MPI for distributed parallelism. It also has not been structured into the dwarf taxonomy but rather the benchmarks it encompasses have been categorised according according to two major sets, whether the application performs a stress test role or acts as a performance test. SHOC supports multiple programming models including OpenCL, CUDA and OpenACC, with benchmarks ranging from targeted tests of particular low-level hardware features to a handful of application kernels. The variety of language implementations for each benchmark application, was one of the original motivators for its construction. In this benchmark suite the OpenCL versions of each application have been designed to strongly mirror the CUDA counterparts, unfortunately this results in fixed tuning parameters such as local workgroup size that is well suited to GPU architectures but is not suited to CPU and other accelerator devices.

However, since this suite has not been classified according to the dwarf taxonomy and also if the classification were performed during this these, adding more applications would likely need to occur to fully encompass the dwarf taxonomy; the addition of applications is more expensive in SHOC, since it would require implementations for the same application into at least 3 other languages – which is not a motivating factor for this thesis. By focusing on application kernels written exclusively in OpenCL, our enhanced OpenDwarfs bench-mark suite is able to cover a wider range of application patterns.

2.4.3 OpenDwarfs

As with Rodinia, Feng et. al [18] introduce the OpenDwarfs (OpenCL and the 13 Dwarfs) as an OpenCL implementation of Berkeley's 13 computational dwarfs of scientific computing. In this work, the absolute execution times were collected over 11 benchmarks. In this paper 11 applications were evaluated on 1 CPU, an Intel Xeon E5405, and 3 GPUs, a low power AMD HD5450 with 25W TDP, and 2 high-power GPUs AMD HD5870 and an Nvidia GT520, for scale both high-end GPUs had an energy footprint of 228/238W TDP respectively. A larger range of dwarfs are covered by OpenDwarfs than Rodinia, however, one dwarf, MapReduce, is still not represented by any application. Additionally, several dwarfs currently have one representative application which may not expose the entire set of characteristics of that dwarf.

A potential criticism is that no diversity analysis was performed to justify the inclusion of each application – however since many applications where inherited from the Rodinia code-base these applications have a proven MICA diversity. Recently, this work was updated and evaluated on FPGA devices by Krommydas et. al. [19] – and adds relevancy to the OpenDwarfs benchmark suite. Given the focused effort of having all the dwarfs represented, the choice to have one implementation – and that being OpenCL, and the recent use of the benchmark suite for a new accelerator architecture all result in it being the selected benchmark suite to perform the extension. These efforts are discussed in Chapter 3.

2.5 Autotuning

When combined with autotuning, an OpenCL code may exhibit good performance across varied devices. Every application presented in the Rodinia Benchmark Suite [sec:rodinia] requires a local workgroup to be passed. In the OpenDwarfs set of benchmarks 9 out of 14 allow for local workgroup tuning. Therefore, given a majority of OpenCL programs use local workgroup tuning, serious considerations need be given regarding how to ensure an accurate depiction of execution times for all accelerators is given. Older literature on the subject also suggests autotuning will play an increasingly important role, in determining accelerator centric optimisations. Tasks such as compiler optimisations and kernel runtime tuning parameters are well suited to auto-tuners without requiring an exhaustive search in this search space.

This has been manifested in many auto-tuning libraries that use machine learning. Spafford et al. [20], Chaimov et al. [21] and Nugteren and Codreanu [22] all propose open source libraries capable of performing autotuning of dynamic execution parameters in OpenCL kernels. Additionally, Price and McIntosh-Smith [23] have demonstrated high performance using a general purpose autotuning library [24], for three applications across twelve devices.

One auto-tuning library of particular interest is OpenTuner [24] since it has already been employed by Price and McIntosh-Smith [23] to improve the performance of OpenCL applications. The OpenTuner library requires the search space to be defined in order to effect the runtime performance of the application. These take the forms of command line of compile time arguments – and are known as the configuration parameter when performing application execution. Next, machine learning techniques are used employing a black box mechanism to effectively search for the optimal configuration parameter arguments in the search space. Measurements are collected per run effectively updating a cost function. Both the objective of the search and the cost function are entirely flexible, since this framework takes the form of a modular python library.

In the Price [23] survey, OpenCL kernels are optimised across 9 current GPUs, 5 Nvidia and 4 AMD devices, and 3 high-end Intel CPUs. The experiment was performed over 3 benchmarks, the Jacobi Iterative Method, a Bilateral Filtering algorithm and BUDE – A general purpose molecular docking program. Presented results show the inefficiencies when auto-tuning for one target device and then execute this optimised program on the other systems. The usefulness of this multi-objective auto-tuning technique is demonstrated and shows that it is a useful tool to generate performance portable OpenCL kernels. Additionally the literature shows that over-optimisation hurts performance portability.

2.6 Offline Ahead-of-Time Analysis

The term offline analysis, in this setting, is defined as the detailed examination of the structure of code and that it requires the entire data set is given in advance. Ahead-of-time indicates that this analysis be done before the program is executed. The combination of theses two terms is directly applicable to OpenCL SPIR codes, which is based on LLVM, since LLVM is well suited to performing ahead-of-time optimised native code generation [25]. Additionally, since SPIR is hardware agnostic/ISA-independent the patterns of computation and communication as shown in the dwarf taxonomy it can be done once the OpenCL source is converted to SPIR and the dwarf represented by the kernel will always be the same. Therefore, analysis such as the classification of which dwarf a new code can be identified, can be performed before any actual device execution is performed. Additionally, these classification and other analysis metrics can be embedded into the SPIR code as a comment in the header, which in turn can be used by a scheduler to determine which device the kernel should be executed.

Closely related to the work performed in this thesis was independently performed by Muralidharan et. al. [26]. Wherein, they use offline ahead-of-time analysis with Oclgrind to collect an instruction histogram of each OpenCL kernel execution in order to generate an estimate of the roofline model analysis for each given accelerator. The resultant tool-flow methodology is used to analyse and track the performance over 3 distinct heterogeneous platforms, and results in a metric to characterise performance.

2.7 Phase-Shifting

Phase is defined as a set of intervals (or slices in time) within a programs execution that has similar behaviour. Therefore, the term phase-shifting refers to change of the execution of a program with temporal adjacency such that the program experiences time-varying effects. Sherwood et. al. [27] observe that common system design and optimisation focus heavily on the assume average system behaviour. They propose however instead programs should be modelled and optimised for phase-based program behaviour. The approach outlined states that phase-behaviour can be profiled quickly using block vector [28] profiles (a vector of per element counts, where each element is the number of times a code block has been entered over a given interval) and off-line classification.

An assumption in the literature is that OpenCL kernels are largely unaffected by program phase-shift. Rather, the program as a whole will doubtlessly experience phase-shifts, compiling an OpenCL kernel code which is an active component of all OpenCL programs will heavily utilise the host CPU device, and when a kernel is executed and the host waits for the device

to finish, CPU utilisation is low. However, the kernel in execution itself will experience very little differences in phases since by their very nature OpenCL kernels are designed small and compartmentalised sections of computation. Such that, if a kernel executed on a particular accelerator device is memory bound, it will consistently be memory bound. If the accelerator experiences consistent stalls on repeated branch mispredictions, this is consistent throughout the kernels entire execution.

2.8 Scaling

This sections discusses scaling with respect to clock frequency and core count respectively. Included in this summary of the relevant literature is the impact it has on energy consumption – namely the non-linear relationship between time and energy.

2.8.1 Frequency

Changing the clock frequency of a conventional CPU core ultimately change performance results – not solely just on execution times.

Choi, Soma and Pedram [29] present an intra-process dynamic voltage and frequency scaling with the goal of minimising energy consumption yet maximising performance by dynamically changing the clock frequency of the CPU. This is achieved by modelling the on-chip / off-chip ratio which is updated using runtime event monitoring. Hardware measurements showed that dynamically lowing the clock frequency for memory bound problems up to 70% energy was saved with a 12% performance loss, compute bound workloads 15-60% energy savings were had at a cost of losing 5-20% performance.

Meanwhile, Agarwal et. al. [30] show that wire latencies (which correspond to memory movement and chip-to-chip communication) have not matched the increase in the range of clock-frequency. As such the impact of increasing the clock frequency is having (and will continue to have) less of an impact on computational efficiency.

Recently, Brown [31] discusses how increasing the clock frequency to generate a result faster (known as race-to-idle or race-to-sleep) saves up to 95% of energy if the entire system can be put in a suspended state – as in embedded and mobile systems. In 2014, this was validated for hardware used in HPC provided it supports a sleep state. Albers and Antoniadis [32] present a framework to accurately approximate the energy cost of speed scaling with a sleep state. In this study, the authors show that the active state of a CPU is comparable to the dynamic energy needed for processing.

2.8.2 Core count

Taylor [33] surveys the transition of typical homogeneous cores to a potentially dark silicone – bright future for heterogeneous systems. This is not because of the lack of scale from increasing cores, indeed Taylor proposes this trend will continue, but from energy concerns of having the utilisation wall[34].

2.8.3 Time and Energy – a non-linear relationship

Additionally, there exist applications where the coupling between execution time and energy consumption is non-linear[lively2011energy], and as such, there should be dwarfs wherein this non-linear relationship holds.

2.9 Program Diversity Analysis and characterization

Program Diversity Analysis has occurred historically to justify the inclusion of an application into a benchmark suite for many years, Principal Component Analysis (PCA) has been used to demonstrate program diversity by others[35][36][37]. Often this work, is manually performed by those focused on assembling the benchmark suites. Indeed, much of the motivation for curating OpenCL applications in Rodinia [16], OpenDwarfs [18] and SHOC [17] was to have real-world scientific problems that represented regular workloads of HPC and SC systems. Determining the suitability of an application regarding these characterised metrics has been evaluated by others. This Section examines these combined efforts in characterising workloads that are less sensitive to the architectures on which they are executed, and concludes with how these characterisation techniques have been used when assembling benchmark suites.

The use of a vector-space or feature-space in order to classify the characteristics of parallel programs was performed by Meajil, El-Ghazawi and Sterling in 1997 [38]. The target of this work was to determine the major factors in modelling performance between parallel computer architectures in an architecture-independent manner. The remainder of this section introduces more recent developments in using a vector-space to characterise applications.

2.10 Microarchitecture-independent workload characterization

Hoste and Eeckhout [39] propose metrics to characterise an application independent to the corresponding microarchitectural characteristics. In this work, Hoste and Eeckout show that despite being useful when locating performance bottlenecks [40] [41], the conventional microarchitecture-dependent characteristics are misleading when used as a basis on which to differentiate benchmark applications. The dependent characteristics typically include instructions per cycle (IPC) and miss rates – cache, branch misprediction and translation look-aside buffer (TLB) – and are collected from hardware performance counter results, typically PAPI. However, the results generated from them is misleading as they either indicate two benchmark applications are similar if they have similar hardware performance counter results or different, since they have different counter results, and this analysis potentially hides the underlying, inherent program behaviour. Additionally, microarchitecture-dependent characteristics are increasingly limited as they are heavily variable between systems, results are machine dependent since CPU architectures significantly differ in pipeline depth and cache size.

Instead Hoste propose a higher level metric framework based on results which do not vary between microarchitecture – the Microarchitecture-independent workload characterization. Features in this metric include instruction mix, Instruction-level parallelism (ILP), Register traffic, Working-set size, Data stream strides and Branch predictability. These feature results

were collected using the PIN [42] binary instrumentation tool. In total 48 measurement characteristics are presented in order to classify an application in a microarchitecture agnostic manner. To reduce the variety of measurements presented, the authors use Principal Component Analysis to reduce the number of measurements in this feature-space to 8 dimensions – those with the largest variance and corresponding impact between applications.

This research is also released as the MICA software and is deployed as a PIN module.

A caveat in the MICA approach is that the results presented are not instruction set architecture independent nor independent from differences in compilers.

2.10.1 ISA-independent workload characterization

More recently, Shao and Brooks [43] have extended the generality of workload characterisation to be ISA independent. The primary motivation for this work was in evaluating the suitability of benchmark suites when targeted on general purpose accelerator platforms. This work was inspired by the MICA framework and collects similar features to those presented in the Hoste and Eeckhout [39] paper.

Instead of using PIN events on x86 systems, this technique uses a Just-In-Time (JIT) compiler to trace instrumented features over a compiler intermediate representation (IR) – in this instance the Low Level Virtual Machine (LLVM) representation. The proposed framework briefly evaluates eleven SPEC benchmarks and examines 5 ISA-independent features/metrics. Namely, number of opcodes (e.g., add, mul), the value of branch entropy – a measure of the randomness of branch behavior, the value of memory entropy – a metric based on the lack of memory locality when examining accesses, the unique number of static instructions, and the unique number of data addresses.

The branch entropy measure presented in the Shao and Brooks paper was initially proposed by Yokota [44] and uses Shannon's information entropy to determine a score of Branch History Entropy.

An additional metric, the average linear branch entropy metric , was recently suggested by De Pestel [45]. It is unique, in that the floating-point value presented linearly corresponds to the miss-rate flow of program execution; $p = 0$ for branches always taken or not-taken but $p = 0.5$ for the most unpredictable control flow. Thus, it offers a bounded value of $0 – 0.5$ and it additionally offers an averaging method that is also easily presented.

2.10.2 Using workload characterization for diversity analysis in the benchmark suites

Several benchmarks have performed characterisation of applications in the past, this has been primarily, at least historically motivated, for diversity analysis to justify the inclusion of an application into a benchmark suite. Rodinia used MICA as the diversity analysis framework. However, the OpenDwarfs benchmark suite have applications which have been manually classified as dwarfs and any characterisation into this taxonomy is based largely intuition. Some of the shared applications ported from the Rodinia Benchmark suite cluster microarchitecture-dependent characteristics of applications into dwarfs. Alas, the approach has the same limitations as those presented in Section 2.10.

For this reason Chapter 4 of this thesis apart from extending the OpenDwarfs Benchmark suite also adds formal verification of the diversity characterisation. To some extent Chapter 5 does this even more formally by generating and clustering the feature-space of all applications grouped as dwarfs. The evaluation on the feature-space is critical to the inclusion of particular extended OpenDwarfs applications and is performed in Chapter 4.

2.11 Oclgrind: Debugging OpenCL kernels via simulation

Oclgrind is an OpenCL device simulator developed by Price and McIntosh-Smith [46] capable of performing simulated kernel execution. It operates on a restricted LLVM IR known as Standard Portable Intermediate Representation (SPIR) established by the Khronos group consortium [47], thereby simulating OpenCL kernel code in a hardware agnostic manner. This architecture independence allows the tool to uncover many portability issues when migrating OpenCL code between devices. Additionally Oclgrind comes with a set of tools to detect runtime API errors, race conditions and locating invalid memory accesses but also comes with an ability to generate instruction histograms. These histograms show the computational composition of a kernel as a series of SPIR instructions with the corresponding count, these results can be used to directly infer the instruction mix similarly to the mechanisms presented in Section 2.10.1.

2.12 Schedulers and Predicting the most Appropriate Accelerator

Lyerly [48] demonstrates by executing a subset of applications from OpenDwarfs it becomes apparent that not one accelerator has the fastest execution time for all benchmarks. This contribution focuses on developing a scheduler to delegate the most appropriate accelerator for a given program. This was achieved by developing a partitioning tool to separate computationally intensive OpenMP regions from C, extracting to and building a predictive model based on past history of the programs executing on the accelerators. We broaden this analysis by claiming that all benchmarks encompassing a dwarf will perform optimally on one accelerator type, but identify that one type of accelerator is non-optimal for all dwarfs.

Hoste et. al. [49] show that the prediction of performance can be based on inherent program similarity. In particular, they show that the metrics collected from a program executing on a particular instruction set architecture (ISA) with a specific compiler offers a relatively accurate characterization of workload for the same application on a totally different micro-architecture. [16] broadens this finding with an assumption that performing analysis on a single threaded CPU version of a benchmark application maintains the underlying set of instructions and the composition of the application. Therefore, it is intuitive that the composition of a program collected using a simulator (such as Oclgrind discussed in Section 2.11, which operates on the most common intermediate form for the OpenCL runtime) regardless of accelerator to which it is ultimately mapped, offers a more accurate architecture agnostic set of metrics around an applications workload. This, in turn, can be used as a basis for performance prediction on general accelerators.

2.13 Formal measurements

Many studies presented in this thesis use tools developed by others in order to perform the necessary measurements. Time measurements have primarily used LibSciBench as the default performance measurement tool. It allows high precision timing events to be collected for statistical analysis [50]. Additionally, it offers a high resolution timer in order to measure short running kernel codes, reported with one cycle resolution and roughly 6 ns of overhead. Throughout Chapter 4 LibSciBench was intensively used to record timings in conjunction with hardware events, which it collects via PAPI [51] counters.

Chapter 3

Dwarfs on Accelerators

Given the heterogeneity of hardware and the wide diversity of scientific application codes, workload characterization, performance prediction and scheduling are all becoming more challenging. To evaluate different approaches requires a representative benchmark suite which is portable to a wide variety of devices. We focus on the OpenCL programming model as it is supported on a wide range of systems including CPU, GPU and FPGA devices. While it is possible to write application code directly in OpenCL, it may also be used as a base to implement higher-level programming models. This technique was shown by Mitra et al. [7] where an OpenMP runtime was implemented over an OpenCL framework for Texas Instruments Keystone II DSP architecture. Having a common back-end in the form of OpenCL allows a direct comparison of identical code across diverse architectures.

Alternative benchmark suites are discussed in Section 2.4 the work presented in this Chapter focuses on the OpenDwarfs benchmark suite, a set of OpenCL benchmarks for heterogeneous computing platforms.[19] Instead, we present an extended and enhanced version of the OpenDwarfs OpenCL benchmark suite (EOD), with a strong focus placed on the robustness of applications, curation of additional benchmarks with an increased emphasis on correctness of results and choice of problem size. Results and analysis are reported for eight benchmark codes on a diverse set of architectures – three Intel CPUs, five Nvidia GPUs, six AMD GPUs and a Xeon Phi. EOD focuses on adding additional benchmarks to better represent each Dwarf along with supporting a range of 4 problem sizes for each application. The rationale for the latter is to survey the range of applications over a diverse set of HPC accelerators across increasing amounts of work, which allows for a deeper analysis of the memory subsystem on each of these devices. The corresponding analysis directly addresses the sub-question around: *Does problem size affect the optimality of a dwarf and its suitability for an accelerator type?*

3.1 Enhancing the OpenDwarfs Benchmark Suite

The OpenDwarfs benchmark suite comprises a variety of OpenCL codes, classified according to patterns of computation and communication known as the 13 Berkeley Dwarfs [8]. The original suite focused on collecting representative benchmarks for scientific applications, with a thorough diversity analysis to justify the addition of each benchmark to the corresponding suite. We aim to extend these efforts to achieve a full representation of each dwarf, both by integrating other benchmark suites and adding custom kernels.

Marjanović et al. [52] argue that the selection of problem size for HPC benchmarking critically affects which hardware properties are relevant. We have observed this to be true across a

wide range of accelerators, therefore we have enhanced the OpenDwarfs benchmark suite to support running different problem sizes for each benchmark. To improve reproducibility of results, we also modified each benchmark to execute in a loop for a minimum of two seconds, to ensure that sampling of execution time and performance counters was not significantly affected by operating system noise.

For the Spectral Methods dwarf, the original OpenDwarfs version of the FFT benchmark was complex, with several code paths that were not executed for the default problem size, and returned incorrect results or failures on some combinations of platforms and problem sizes we tested. We replaced it with a simpler high-performance FFT benchmark created by Eric Bainville [53], which worked correctly in all our tests. We have also added a 2-D discrete wavelet transform from the Rodinia suite [16] (with modifications to improve portability), and we plan to add a continuous wavelet transform code.

To understand benchmark performance, it is useful to be able to collect hardware performance counters associated with each timing segment. LibSciBench is a performance measurement tool which allows high precision timing events to be collected for statistical analysis [50]. It offers a high resolution timer in order to measure short running kernel codes, reported with one cycle resolution and roughly 6 ns of overhead. We used LibSciBench to record timings in conjunction with hardware events, which it collects via PAPI [51] counters. We modified the applications in the OpenDwarfs benchmark suite to insert library calls to LibSciBench to record timings and PAPI events for the three main components of application time: kernel execution, host setup and memory transfer operations. Through PAPI modules such as Intel's Running Average Power Limit (RAPL) and Nvidia Management Library (NVML), LibSciBench also supports energy measurements, for which we report preliminary results in this paper.

3.2 Experimental Setup

3.2.1 Hardware

The experiments were conducted on a varied set of 15 hardware platforms: three Intel CPU architectures, five Nvidia GPUs, six AMD GPUs, and one MIC (Intel Knights Landing Xeon Phi). Key characteristics of the test platforms are presented in Table~5.2. The L1 cache size should be read as having both an instruction size cache and a data cache size of equal values as those displayed. For Nvidia GPUs, the L2 cache size reported is the size L2 cache per SM multiplied by the number of SMs. For the Intel CPUs, hyper-threading was enabled and the frequency governor was set to `performance`.

3.2.2 Software

OpenCL version 1.2 was used for all experiments. On the CPUs we used the Intel OpenCL driver version 6.3, provided in the 2016-R3 opencl-sdk release. On the Nvidia GPUs we used the Nvidia OpenCL driver version 375.66, provided as part of CUDA 8.0.61, AMD GPUs used the OpenCL driver version provided in the amdappsdk v3.0.

The Knights Landing (KNL) architecture used the same OpenCL driver as the Intel CPU platforms, however, the 2018-R1 release of the Intel compiler was required to compile for the architecture natively on the host. Additionally, due to Intel removing support for OpenCL on the KNL architecture, some additional compiler flags were required. Unfortunately, as Intel

Table 3.1: Hardware

Name	Vendor	Type	Series	Core Count	Clock Frequency (MHz) (min/max/turbo)	Cache (KiB) (L1/L2/L3)	TDP (W)	Launch Date
Xeon E5-2697 v2	Intel	CPU	Ivy Bridge	24∗	1200/2700/3500	32/256/30720	130	Q3 2013
i7-6700K	Intel	CPU	Skylake	8∗	800/4000/4300	32/256/8192	91	Q3 2015
i5-3550	Intel	CPU	Ivy Bridge	4∗	1600/3380/3700	32/256/6144	77	Q2 2012
Titan X	Nvidia	GPU	Pascal	3584†	1417/1531/–	48/2048/–	250	Q3 2016
GTX 1080	Nvidia	GPU	Pascal	2560†	1607/1733/–	48/2048/–	180	Q2 2016
GTX 1080 Ti	Nvidia	GPU	Pascal	3584†	1480/1582/–	48/2048/–	250	Q1 2017
K20m	Nvidia	GPU	Kepler	2496†	706/–/–	64/1536/–	225	Q4 2012
K40m	Nvidia	GPU	Kepler	2880†	745/875/–	64/1536/–	235	Q4 2013
FirePro S9150	AMD	GPU	Hawaii	2816‖	900/–/–	16/1024/–	235	Q3 2014
HD 7970	AMD	GPU	Tahiti	2048‖	925/1010/–	16/768/–	250	Q4 2011
R9 290X	AMD	GPU	Hawaii	2816‖	1000/–/–	16/1024/–	250	Q3 2014
R9 295x2	AMD	GPU	Hawaii	5632‖	1018/–/–	16/1024/–	500	Q2 2014
R9 Fury X	AMD	GPU	Fuji	4096‖	1050/–/–	16/2048/–	273	Q2 2015
RX 480	AMD	GPU	Polaris	4096‖	1120/1266/–	16/2048/–	150	Q2 2016
Xeon Phi 7210	Intel	MIC	KNL	256‡	1300/1500/–	32/1024/–	215	Q2 2016

∗ HyperThreaded cores
† CUDA cores
‖ Stream processors
‡ Each physical core has 4 hardware threads per core, thus 64 cores

has removed support for AVX2 vectorization (using the `-xMIC-AVX512` flag), vector instructions use only 256-bit registers instead of the wider 512-bit registers available on KNL. This means that floating-point performance on KNL is limited to half the theoretical peak.

GCC version 5.4.0 with glibc 2.23 was used for the Skylake i7 and GTX 1080, GCC version 4.8.5 with glibc 2.23 was used on the remaining platforms. OS Ubuntu Linux 16.04.4 with kernel version 4.4.0 was used for the Skylake CPU and GTX 1080 GPU, Red Hat 4.8.5-11 with kernel version 3.10.0 was used on the other platforms.

As OpenDwarfs has no stable release version, it was extended from the last commit by the maintainer on 26 Feb 2016. [54] LibSciBench version 0.2.2 was used for all performance measurements.

3.2.3 Measurements

We measured execution time and energy for individual OpenCL kernels within each benchmark. Each benchmark run executed the application in a loop until at least two seconds had elapsed, and the mean execution time for each kernel was recorded. Each benchmark was run 50 times for each problem size (see §**??**) for both execution time and energy measurements. A sample size of 50 per group – for each combination of benchmark and problem size – was used to ensure that sufficient statistical power $\beta = 0.8$ would be available to detect a significant difference in means on the scale of half standard deviation of separation. This sample size was computed using the t-test power calculation over a normal distribution.

To help understand the timings, the following hardware counters were also collected:

- total instructions and IPC (Instructions Per Cycle);
- L1 and L2 data cache misses;

- total L3 cache events in the form of request rate (requests / instructions), miss rate (misses / instructions), and miss ratio (misses/requests);
- data TLB (Translation Look-aside Buffer) miss rate (misses / instructions); and
- branch instructions and branch mispredictions.

For each benchmark we also measured memory transfer times between host and device, however, only the kernel execution times and energies are presented here.

Energy measurements were taken on Intel platforms using the RAPL PAPI module, and on Nvidia GPUs using the NVML PAPI module.

3.2.4 Setting Sizes

For each benchmark, four different problem sizes were selected, namely **tiny**, **small**, **medium** and **large**. These problem sizes are based on the memory hierarchy of the Skylake CPU. Specifically, **tiny** should just fit within L1 cache, on the Skylake this corresponds to 32 KiB of data cache, **small** should fit within the 256 KiB L2 data cache, **medium** should fit within 8192 KiB of the L3 cache, and **large** must be much larger than 8192 KiB to avoid caching and operate out of main memory.

The memory footprint was verified for each benchmark by printing the sum of the size of all memory allocated on the device.

For this study, problem sizes were not customized to the memory hierarchy of each platform, since the CPU is the most sensitive to cache performance. Also, note for these CPU systems the L1 and L2 cache sizes are identical, and since we ensure that **large** is at least 4× larger than L3 cache, we are guaranteed to have last-level cache misses for the **large** problem size.

Caching performance was measured using PAPI counters. On the Skylake L1 and L2 data cache miss rates were counted using the `PAPI_L1_DCM` and `PAPI_L2_DCM` counters. For L3 miss events, only the total cache counter event (`PAPI_L3_TCM`) was available. The final values presented as miss results are presented as a percentage, and were determined using the number of misses counted divided by the total instructions (`PAPI_TOT_INS`).

The methodology to determine the appropriate size parameters is demonstrated on the k-means benchmark.

3.2.4.1 kmeans

K-means is an iterative algorithm which groups a set of points into clusters, such that each point is closer to the centroid of its assigned cluster than to the centroid of any other cluster. Each step of the algorithm assigns each point to the cluster with the closest centroid, then relocates each cluster centroid to the mean of all points within the cluster. Execution terminates when no clusters change size between iterations. Starting positions for the centroids are determined randomly. The OpenDwarfs benchmark previously required the object features to be read from a previously generated file. We extended the benchmark to support generation of a random distribution of points. This was done to more fairly evaluate cache performance, since repeated runs of clustering on the same feature space (loaded from file) would deterministically generate similar caching behavior. For all problem sizes, the number of clusters is fixed at 5.

Given a fixed number of clusters, the parameters that may be used to select a problem size are the number of points P_n, and the dimensionality or number of features per point F_n. In the kernel for k-means there are three large one-dimensional arrays passed to the device, namely **feature**, **cluster** and **membership**. In the **feature** array which stores the unclustered feature space, each feature is represented by a 32-bit floating-point number, so the entire array is of size $P_n \times F_n \times \text{sizeof}(\text{float})$. **cluster** is the working and output array to store the intermediately clustered points, it is of size $C_n \times F_n \times \text{sizeof}(\text{float})$, where C_n is the number of clusters. **membership** is an array indicating whether each point has changed to a new cluster in each iteration of the algorithm, it is of size $P_n \times \text{sizeof}(\text{int})$, where sizeof(int) is the number of bytes to represent an integer value. Thereby the working kernel memory, in KiB, is:

$$\frac{\text{size}(\textbf{feature}) + \text{size}(\textbf{membership}) + \text{size}(\textbf{cluster})}{1024} \tag{3.1}$$

Using this equation, we can determine the largest problem size that will fit in each level of cache. The tiny problem size is defined to have 256 points and 30 features; from Equation~3.1 the total size of the main arrays is 31.5 KiB, slightly smaller than the 32 KiB L1 cache. The number of points is increased for each larger problem size to ensure that the main arrays fit within the lower levels of the cache hierarchy, measuring the total execution time and respective caching events. The **tiny**, **small** and **medium** problem sizes in the first row of Table~3.2 correspond to L1, L2 and L3 cache respectively. The **large** problem size is at least four times the size of the last-level cache – in the case of the Skylake, at least 32 MiB – to ensure that data are transferred between main memory and cache.

For brevity, cache miss results are not presented in this paper but were used to verify the selection of suitable problem sizes for each benchmark. The procedure to select problem size parameters is specific to each benchmark, but follows a similar approach to k-means.

3.2.4.2 lud, fft, srad, crc, nw

The LU-Decomposition `lud`, Fast Fourier Transform `fft`, Speckle Reducing Anisotropic Diffusion `srad`, Cyclic Redundancy Check `crc` and Needleman-Wunsch `nw` benchmarks did not require additional data sets. Where necessary these benchmarks were modified to generate the correct solution and run on modern architectures. Correctness was examined either by directly comparing outputs against a serial implementation of the codes (where one was available), or by adding utilities to compare norms between the experimental outputs.

3.2.4.3 dwt

Two-Dimensional Discrete Wavelet Transform is commonly used in image compression. It has been extended to support loading of Portable PixMap (.ppm) and Portable GrayMap (.pgm) image format, and storing Portable GrayMap images of the resulting DWT coefficients in a visual tiled fashion. The input image dataset for various problem sizes was generated by using the resize capabilities of the ImageMagick application. The original gum leaf image is the large sample size has the ratio of 3648×2736 pixels and was down-sampled to 80×60.

3.2.4.4 gem, nqueens, hmm

For three of the benchmarks, we were unable to generate different problem sizes to properly exercise the memory hierarchy.

Gemnoui `gem` is an n-body-method based benchmark which computes electrostatic potential of biomolecular structures. Determining suitable problem sizes was performed by initially browsing the National Center for Biotechnology Information's Molecular Modeling Database (MMDB)[55] and inspecting the corresponding Protein Data Bank format (pdb) files. Molecules were then selected based on complexity, since the greater the complexity the greater the number of atoms required for the benchmark and thus the larger the memory footprint. **tiny** used the Prion Peptide 4TUT[56] and was the simplest structure, consisting of a single protein (1 molecule), it had the device side memory usage of 31.3 KiB which should fit in the L1 cache (32 KiB) on the Skylake processor. **small** used a Leukocyte Receptor 2D3V[57] also consisting of 1 protein molecule, with an associated memory footprint of 252KiB. **medium** used the nucleosome dataset originally provided in the OpenDwarfs benchmark suite, using 7498 KiB of device-side memory. **large** used an X-Ray Structure of a Nucleosome Core Particle[58], consisting of 8 protein, 2 nucleotide, and 18 chemical molecules, and requiring 10 970.2 KiB of memory when executed by `gem`. Each `pdb` file was converted to the `pqr` atomic particle charge and radius format using the `pdb2pqr`[59] tool. Generation of the solvent excluded molecular surface used the tool `msms` [60]. Unfortunately, the molecules used for the **medium** and **large** problem sizes contain uninitialized values only noticed on CPU architectures and as such further work is required to ensure correctness for multiple problem sizes. The datasets used for `gem` and all other benchmarks can be found in this paper's associated GitHub repository [61].

The `nqueens` benchmark is a backtrack/branch-and-bound code which finds valid placements of queens on a chessboard of size n×n, where each queen cannot be attacked by another. For this code, memory footprint scales very slowly with increasing number of queens, relative to the computational cost. Thus it is significantly compute-bound and only one problem size is tested.

The Baum-Welch Algorithm Hidden Markov Model `hmm` benchmark represents the Graphical Models dwarf and did not require additional data sets, however validation of the correctness of results has not occurred apart from over the **tiny** problem size, as such, it is the only size examined in the evaluation.

3.2.5 Summary of Benchmark Settings

The problem size parameters for all benchmarks are presented in Table~3.2.

Each **Device** can be selected in a uniform way between applications using the same notation, on this system **Device** comprises of `-p 1 -d 0 -t 0` for the Intel Skylake CPU, where `p` and `d` are the integer identifier of the platform and device to respectively use, and `-p 1 -d 0 -t 1` for the Nvidia GeForce GTX 1080 GPU. Each application is run as **Benchmark Device - Arguments**, where **Arguments** is taken from Table 3.3 at the selected scale of Φ. For reproducibility the entire set of Python scripts with all problem sizes is available in a GitHub repository [61]. Where Φ is substituted as the argument for each benchmark, it is taken as the respective scale from Table~3.2 and is inserted into Table~3.3.

Table 3.2: OpenDwarfs workload scale parameters Φ

Benchmark	tiny	small	medium	large
kmeans	256	2048	65600	131072
lud	80	240	1440	4096
csr	736	2416	14336	16384
fft	2048	16384	524288	2097152
dwt	72x54	200x150	1152x864	3648x2736
srad	80,16	128,80	1024,336	2048,1024
crc	2000	16000	524000	4194304
nw	48	176	1008	4096
gem	4TUT	2D3V	nucleosome	1KX5
nqueens	18	–	–	–
hmm	8,1	900,1	1012,1024	2048,2048

Table 3.3: Program Arguments

Benchmark	Arguments
kmeans	`-g -f 26 -p` Φ
lud	`-s` Φ
csr†	`-i` Ψ
	Ψ = `createcsr -n` Φ `-d 5000` △
fft	Φ
dwt	`-l 3` Φ`-gum.ppm`
srad	Φ_1 Φ_2 `0 127 0 127 0.5 1`
crc	`-i 1000_`Φ`.txt`
nw	Φ 10
gem	Φ 80 1 0
n-queens	Φ
hmm	`-n` Φ_1`-s` Φ_2`-v s`

△ The `-d 5000` indicates density of the matrix in this instance 0.5% dense (or 99.5% sparse).

† The `csr` benchmark loads a file generated by `createcsr` according to the workload size parameter Φ; this file is represented by Ψ.

3.3 Results

The primary purpose of including these time results is to demonstrate the benefits of the extensions made to the OpenDwarfs Benchmark suite. The use of LibSciBench allowed high resolution timing measurements over multiple code regions. To demonstrate the portability of the Extended OpenDwarfs benchmark suite, we present results from 11 varied benchmarks running on 15 different devices representing four distinct classes of accelerator. For 12 of the benchmarks, we measured multiple problem sizes and observed distinctly different scaling patterns between devices. This underscores the importance of allowing a choice of problem size in a benchmarking suite.

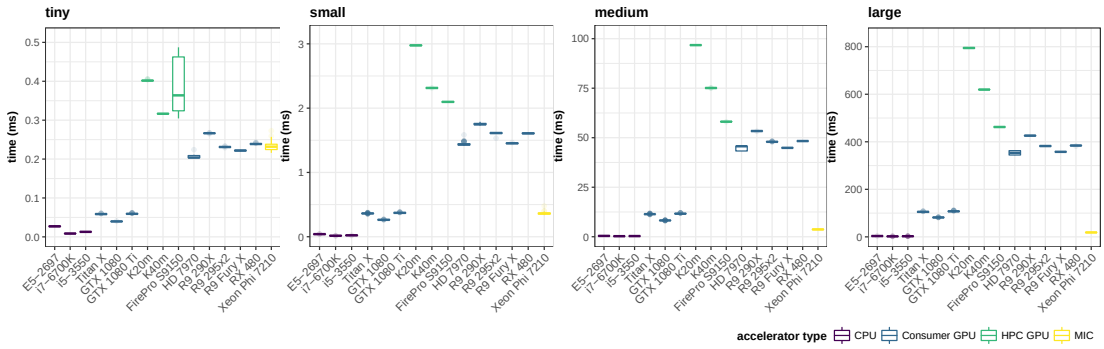

Figure 3.1: Kernel execution times for the **crc** benchmark on different hardware platforms

3.3.1 Time

We first present execution time measurements for each benchmark, starting with the Cyclic Redundancy Check `crc` benchmark which represents the Combinational Logic dwarf.

Figure~3.1 shows the execution times for the `crc` benchmark over 50 iterations on each of the target architectures, including the KNL.

The results are colored according to accelerator type: purple for CPU devices, blue for consumer GPUs, green for HPC GPUs, and yellow for the KNL MIC. Execution times for `crc` are lowest on CPU-type architectures, probably due to the low floating-point intensity of the CRC computation[Ch. 6][62]. Excluding `crc`, all the other benchmarks perform best on GPU type accelerators; furthermore, the performance on the KNL is poor due to the lack of support for wide vector registers in Intel's OpenCL SDK. We therefore omit results for KNL for the remaining benchmarks.

Figures~3.2 and~3.3 shows the distribution of kernel execution times for the remaining benchmarks. Some benchmarks execute more than one kernel on the accelerator device; the reported iteration time is the sum of all compute time spent on the accelerator for all kernels. Each benchmark corresponds to a particular dwarf: Figure~3.2a (`kmeans`) represents the MapReduce dwarf, Figure~3.2b (`lud`) represents the Dense Linear Algebra dwarf, Figure~3.2c (`csr`) represents Sparse Linear Algebra, Figure~3.2d (`dwt`) and Figure~3.2e (`fft`) represent Spectral Methods, Figure~3.3a (`srad`) represents the Structured Grid dwarf and Figure~3.3b (`nw`) represents Dynamic Programming.

Finally, Figure~3.4 presents results for the three applications with restricted problem sizes and only one problem size is shown. The N-body Methods dwarf is represented by (`gem`) and the results are shown in Figure~3.4a, the Backtrack & Branch and Bound dwarf is represented by the (`nqueens`) application in Figure~3.4b and (`hmm`) results in Figure~3.4c represent the Graphical Models dwarf.

Examining the transition from tiny to large problem sizes (from left to right) in Figure~3.3a shows the performance gap between CPU and GPU architectures widening for `srad` – indicating codes representative of structured grid dwarfs are well suited to GPUs.

In contrast, Figure~3.3b shows Dynamic Programming problems have performance results tied to micro-architecture or OpenCL runtime support and can not be explained solely by considering accelerator type. For instance, the Intel CPUs and NVIDIA GPUs perform comparably over all problem sizes, whereas all AMD GPUs exhibit worse performance as size

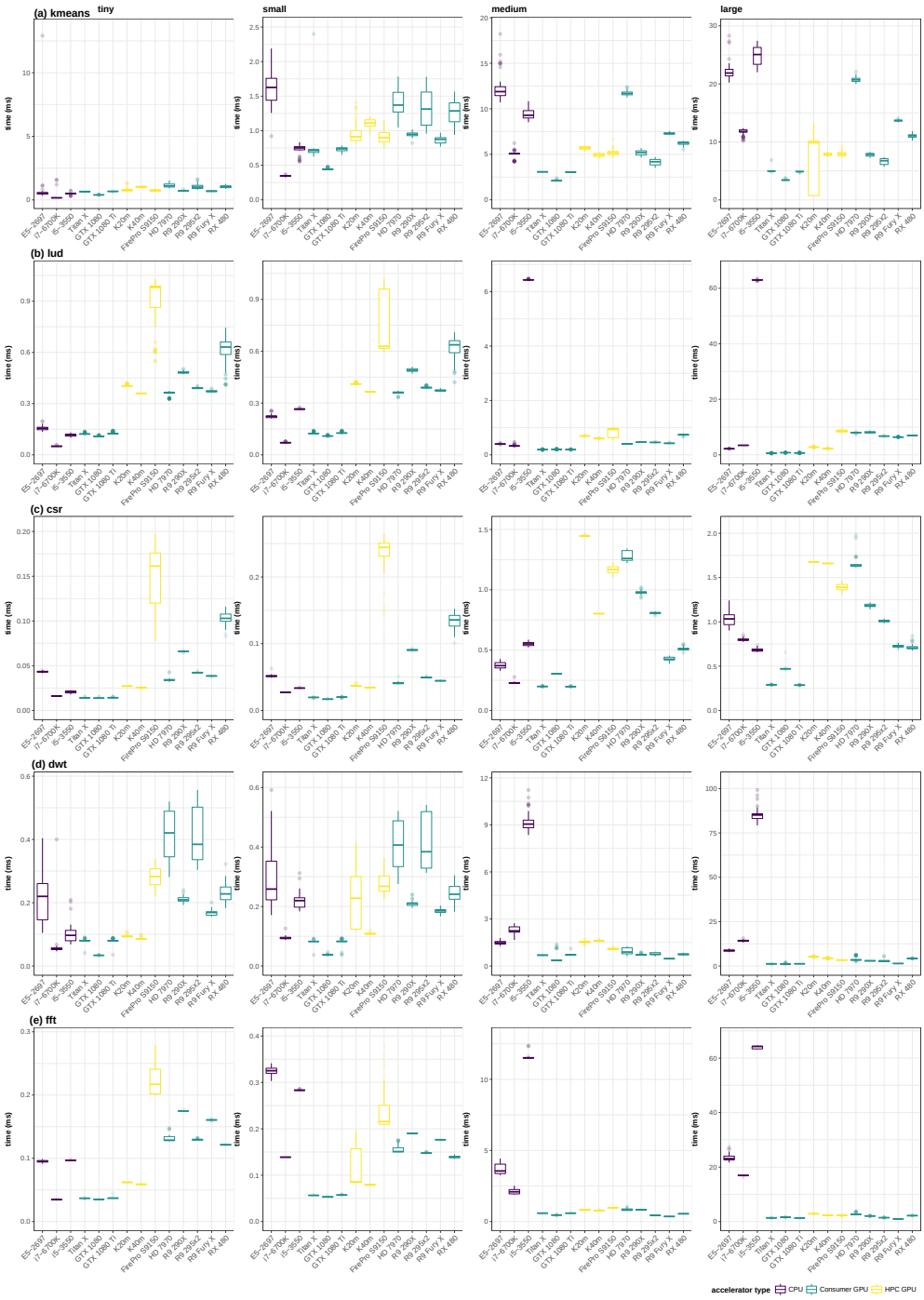

Figure 3.2: Benchmark kernel execution times on different hardware platforms

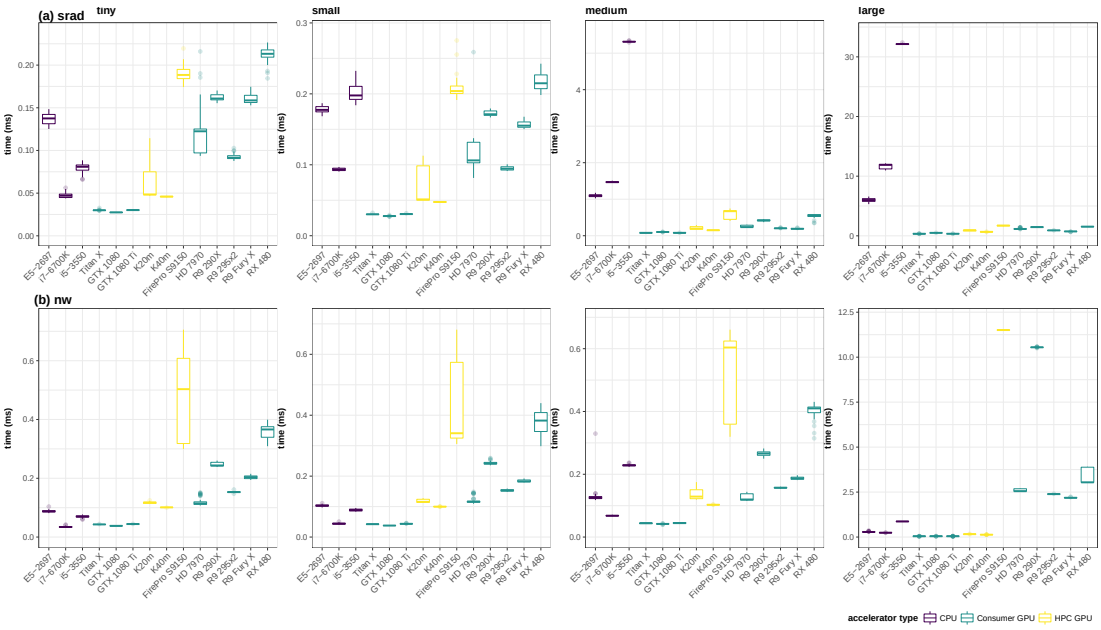

Figure 3.3: Benchmark kernel execution times on different hardware platforms (continued)

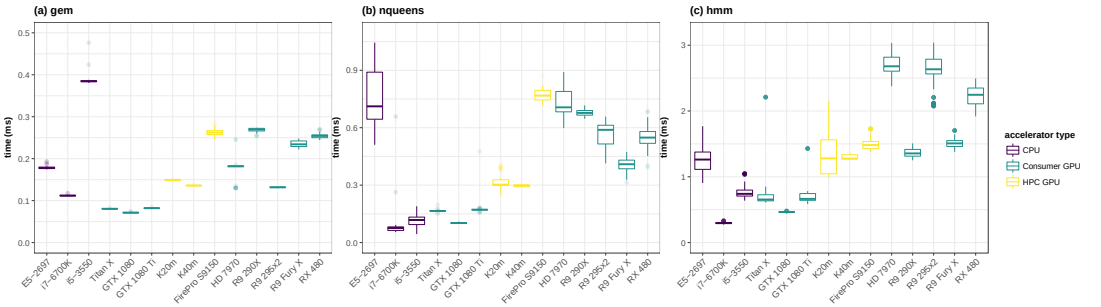

Figure 3.4: Single problem sized benchmarks of kernel execution times on different hardware platforms

increases.

For most benchmarks, the coefficient of variation in execution times is much greater for devices with a lower clock frequency, regardless of accelerator type. While execution time increases with problem size for all benchmarks and platforms, the modern GPUs (Titan X, GTX1080, GTX1080Ti, R9 Fury X and RX 480) performed relatively better for large problem sizes, possibly due to their greater second-level cache size compared to the other platforms. A notable exception is `k-means` for which CPU execution times were comparable to GPU, which reflects the relatively low ratio of floating-point to memory operations in the benchmark.

Generally, the HPC GPUs are older and were designed to alleviate global memory limitations amongst consumer GPUs of the time. (Global memory size is not listed in Table~5.2.) Despite their larger memory sizes, the clock speed of all HPC GPUs is slower than all evaluated consumer GPUs. While the HPC GPUs (devices 7-9, in yellow) outperformed consumer GPUs of the same generation (devices 10-13, in green) for most benchmarks and problem sizes, they were always beaten by more modern GPUs. This is no surprise since all selected problem

sizes fit within the global memory of all devices.

A comparison between CPUs (devices 1-3, in purple) indicates the importance of examining multiple problem sizes. Medium-sized problems were designed to fit within the L3 cache of the i7-6700K system, and this conveniently also fits within the L3 cache of the Xeon E5-2697 v2. However, the older i5-3550 CPU has a smaller L3 cache and exhibits worse performance when moving from small to medium problem sizes, and is shown in Figures$_{3.2b,}$3.2d,~3.2e and ~3.3a,

Increasing problem size also hinders the performance in certain circumstances for GPU devices. For example, Figure~3.3b shows a widening performance gap over each increase in problem size between AMD GPUs and the other devices.

Predicted application properties for the various Berkeley Dwarfs are evident in the measured runtime results. For example, Asanović et al. [8] state that applications from the Spectral Methods dwarf is memory latency limited. If we examine `dwt` and `fft` – the applications which represent Spectral Methods – in Figure~3.2d and Figure~3.2e respectively, we see that for medium problem sizes the execution times match the higher memory latency of the L3 cache of CPU devices relative to the GPU counterparts. The trend only increases with problem size: the large size shows the CPU devices frequently accessing main memory while the GPUs' larger memory ensures a lower memory access latency. It is expected if had we extended this study to an even larger problem size that would not fit on GPU global memory, much higher performance penalties would be experienced over GPU devices, since the PCI-E interconnect has a higher latency than a memory access to main memory from the CPU systems. As a further example, Asanović et al. [8] state that the Structured Grid dwarf is memory bandwidth limited. The Structured Grid dwarf is represented by the `srad` benchmark shown in Figure~3.3a. GPUs exhibit lower execution times than CPUs, which would be expected in a memory bandwidth-limited code as GPU devices offer higher bandwidth than a system interconnect.

3.3.2 Energy

In addition to execution time, we are interested in differences in energy consumption between devices and applications. We measured the energy consumption of benchmark kernel execution on the Intel Skylake i7-6700k CPU and the Nvidia GTX1080 GPU, using PAPI modules for RAPL and NVML. These were the only devices examined since collection of PAPI energy measurements (with LibSciBench) requires superuser access, and these devices were the only accelerators available with this permission. The distributions were collected by measuring solely the kernel execution over a distribution of 50 runs. RAPL CPU energy measurements were collected over all cores in package 0 `rapl:::PP0_ENERGY:PACKAGE0`. NVML GPU energy was collected using the power usage readings `nvml:::GeForce_GTX_1080:power` for the device and presents the total power draw (+/-5 watts) for the entire card – memory and chip. Measurements results converted to energy J from their original resolution nJ and mW on the CPU and GPU respectively.

From the time results presented in Section 3.3.1 we see the largest difference occurs between CPU and GPU type accelerators at the **large** problem size. Thus we expect that the **large** problem size will also show the largest difference in energy.

Figures~3.5 and~3.5 show the kernel execution energy for several benchmarks for the **large** size. All results are presented in joules. The box plots are coloured according to device: red

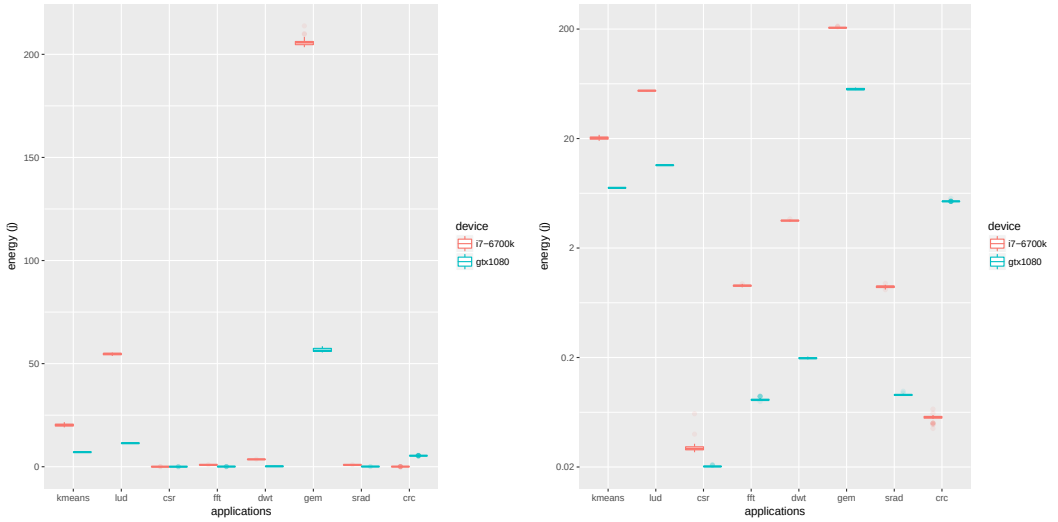

Figure 3.5: Benchmark kernel execution energy presented on a linear and logarithmic scale from left to right respectively, on the (**large** problem size) on Core i7-6700K and Nvidia GTX1080

for the Intel Skylake i7-6700k CPU and blue for the Nvidia GTX1080 GPU. The logarithmic transformation has been applied to Figure~3.5 to emphasise the variation at smaller energy scales (< 1 J), which was necessary due to small execution times for some benchmarks. In future this will be addressed by balancing the amount of computation required for each benchmark, to standardize the magnitude of results.

All the benchmarks use more energy on the CPU, with the exception of `crc` which as previously mentioned has low floating-point intensity and so is not able to make use of the GPU's greater floating-point capability. Variance with respect to energy usage is larger on the CPU, which is consistent with the execution time results.

3.4 Conclusions

We have performed essential curation of the OpenDwarfs benchmark suite. We improved coverage of spectral methods by adding a new Discrete Wavelet Transform benchmark, and replacing the previous inadequate `fft` benchmark. All benchmarks were enhanced to allow multiple problem sizes; in this paper we report results for four different problem sizes, selected according to the memory hierarchy of CPU systems as motivated by Marjanović's findings [52]. These can now be easily adjusted for next generation accelerator systems using the methodology outlined in Section~**??**.

We ran many of the benchmarks presented in the original OpenDwarfs [19] paper on current hardware. This was done for two reasons, firstly to investigate the original findings to the state-of-the-art systems and secondly to extend the usefulness of the benchmark suite. Re-examining the original codes on range of modern hardware showed limitations, such as the fixed problem sizes along with many platform-specific optimizations (such as local work-group size). In the best case, such optimizations resulted in sub-optimal performance for newer systems (many problem sizes favored the original GPUs on which they were originally

run). In the worst case, they resulted in failures when running on untested platforms or changed execution arguments.

Finally a major contribution of this work was to integrate LibSciBench into the benchmark suite, which adds a high precision timing library and support for statistical analysis and visualization. This has allowed collection of PAPI, energy and high resolution (sub-microsecond) time measurements at all stages of each application, which has added value to the analysis of OpenCL program flow on each system, for example identifying overheads in kernel construction and buffer enqueuing. The use of LibSciBench has also increased the reproducibility of timing data for both the current study and on new architectures in the future.

3.5 Future Work

We plan to complete analysis of the remaining benchmarks in the suite for multiple problem sizes. In addition to comparing performance between devices, we would also like to develop some notion of "ideal" performance for each combination of benchmark and device, which would guide efforts to improve performance portability. Additional architectures such as FPGA, DSP and Radeon Open Compute based APUs – which further breaks down the walls between the CPU and GPU – will be considered.

Each OpenCL kernel presented in this paper has been inspected using the Architecture Independent Workload Characterization (AIWC). Analysis using AIWC helps understand how the structure of kernels contributes to the varying runtime characteristics between devices that are presented in this work, and will be published in the future.

Certain configuration parameters for the benchmarks, e.g. local workgroup size, are amenable to auto-tuning. We plan to integrate auto-tuning into the benchmarking framework to provide confidence that the optimal parameters are used for each combination of code and accelerator.

The original goal of this research was to discover methods for choosing the best device for a particular computational task, for example to support scheduling decisions under time and/or energy constraints. Until now, we found the available OpenCL benchmark suites were not rich enough to adequately characterize performance across the diverse range of applications and computational devices of interest. Now that a flexible benchmark suite is in place and results can be generated quickly and reliably on a range of accelerators, we plan to use these benchmarks to evaluate scheduling approaches.

3.6 Summary

The work presented in this chapter does not address the optimality of the OpenCL programming language for accelerator devices, nor does it need to, instead, it presents the culmination of ground work and the associated considerations required to evaluate the performance of heterogenous devices and introduces a final benchmarking suite – EOD – which serves this purpose. It serves as a platform which is essential to perform workload scheduling of scientific workloads on accelerator devices which will be common to next-generation scientific HPC nodes.

Separately, three major points become apparent when examining the results presented in this chapter. Generally, energy is correlated to execution time for most applications. Secondly,

particular accelerator types do not perform best under all applications encompassing a dwarf. Finally, all dwarfs are not suited to one type of accelerator – for instance GPU type accelerators are unsuited to the combinational-logic dwarf.

These last two points reinforce the assumption that there is a most appropriate accelerator for any particular OpenCL code, this in turn raises an interesting research question, "can the automatic classification of a program binary allow the efficient scheduling of work to the most appropriate accelerator", the automatic classification tool is introduced in the next chapter whilst the broader question is addressed in Chapter 5.

Chapter 4

AIWC: OpenCL based Architecture Independent Workload Characterization

OpenCL is an attractive programming model for high performance computing systems composed of heterogeneous compute devices, with wide support from hardware vendors allowing portability of application codes. For accelerator designers and HPC integrators, understanding the performance characteristics of scientific workloads is of utmost importance. However, if these characteristics are tied to architectural features that are specific to a particular system, they may not generalize well to alternative or future systems. A architecture-independent method ensures an accurate characterization of inherent program behavior, without bias due to architectural-dependent features that vary widely between different types of accelerators. This work presents the first architecture-independent workload characterization framework for heterogeneous compute platforms. The Architecture Independent Workload Characterization (AIWC) tool, is capable of characterizing OpenCL workloads currently in use in the supercomputing setting, and is deployed as part of the open source Oclgrind simulator. AIWC simulates execution of OpenCL kernels to collect architecture-independent features which characterize each code, and may also be used in performance prediction. We demonstrate the use of AIWC, and the associated metrics, to characterize a variety of codes over a subset from the EOD Benchmark Suite presented in Chapter 3.

4.1 AIWC's Back story

Oclgrind is an OpenCL device simulator developed by Price and McIntosh-Smith [46] capable of performing simulated kernel execution. It operates on a restricted LLVM IR known as Standard Portable Intermediate Representation (SPIR) [47], thereby simulating OpenCL kernel code in a hardware agnostic manner. This architecture independence allows the tool to uncover many portability issues when migrating OpenCL code between devices. Additionally Oclgrind comes with a set of tools to detect runtime API errors, race conditions and invalid memory accesses, and generate instruction histograms. AIWC is added as a tool to Oclgrind and leverages its ability to simulate OpenCL device execution using LLVM IR codes.

AIWC relies on the selection of the instruction set architecture (ISA)-independent features determined by Shao and Brooks [43], which in turn builds on earlier work in microarchitecture-

independent workload characterization. Hoste and Eeckout [39] show that although conventional microarchitecture-dependent characteristics are useful in locating performance bottlenecks [40], they are misleading when used as a basis on which to differentiate benchmark applications. Microarchitecture-independent workload characterization and the associated analysis tool, known as MICA, was proposed to collect metrics to characterize an application independent of particular microarchitectural characteristics. Architecture-dependent characteristics typically include instructions per cycle (IPC) and miss rates – cache, branch misprediction and translation look-aside buffer (TLB) – and are collected from hardware performance counter results, typically PAPI. However, these characteristics fail to distinguish between inherent program behavior and its mapping to specific hardware features, ignoring critical differences between architectures such as pipeline depth and cache size. The MICA framework collects independent features including instruction mix, instruction-level parallelism (ILP), register traffic, working-set size, data stream strides and branch predictability. These feature results are collected using the Pin [42] binary instrumentation tool. In total 47 microarchitecture-independent metrics are used to characterize an application code. To simplify analysis and understanding of the data, the authors combine principal component analysis with a genetic algorithm to select eight metrics which account for approximately 80% of the variance in the data set.

A caveat in the MICA approach is that the results presented are not ISA-independent nor independent from differences in compilers. Additionally since the metrics collected rely heavily on Pin instrumentation, characterization of multi-threaded workloads or accelerators are not supported. As such, it is unsuited to conventional supercomputing workloads which make heavy use of parallelism and accelerators.

Shao and Brooks have since extended the generality of the MICA to be ISA independent. The primary motivation for this work was in evaluating the suitability of benchmark suites when targeted on general purpose accelerator platforms. The proposed framework briefly evaluates eleven SPEC benchmarks and examines 5 ISA-independent features/metrics. Namely, number of opcodes (e.g., add, mul), the value of branch entropy – a measure of the randomness of branch behavior, the value of memory entropy – a metric based on the lack of memory locality when examining accesses, the unique number of static instructions, and the unique number of data addresses.

Related to the paper, Shao also presents a proof of concept implementation (WIICA) which uses an LLVM IR Trace Profiler to generate an execution trace, from which a python script collects the ISA independent metrics. Any results gleaned from WIICA are easily reproducible, the execution trace is generated by manually selecting regions of code built from the LLVM IR Trace Profiler. Unfortunately, use of the tool is non-trivial given the complexity of the tool chain and the nature of dependencies (LLVM 3.4 and Clang 3.4). Additionally, WIICA operates on `C` and `C++` code, which cannot be executed directly on any accelerator device aside from the CPU. Our work extends this implementation to the broader OpenCL setting to collect architecture independent metrics from a hardware-agnostic language – OpenCL.

The branch entropy measure used by Shao and Brooks [43] was initially proposed by Yokota [44] and uses Shannon's information entropy to determine a score of Branch History Entropy. De Pestel, Eyerman and Eeckhout [63] proposed an alternative metric, average linear branch entropy metric, to allow accurate prediction of miss rates across a range of branch predictors. As their metric is more suitable for architecture-independent studies, we adopt it for this work.

Caparrós Cabezas and Stanley-Marbell [64] present a framework for characterizing

Table 4.1: **AIWC** tool metrics.

Type	Metric	Description
Compute	opcode	# of unique opcodes required to cover 90% of dynamic instructions
Compute	Total Instruction Count	Total # of instructions executed
Parallelism	Work-items	# of work-items or threads executed
Parallelism	Total Barriers Hit	maximum # of instructions executed until a barrier
Parallelism	Min ITB	minimum # of instructions executed until a barrier
Parallelism	Max ITB	maximum # of instructions executed until a barrier
Parallelism	Median ITB	median # of instructions executed until a barrier
Parallelism	Max SIMD Width	maximum number of data items operated on during an instruction
Parallelism	Mean SIMD Width	mean number of data items operated on during an instruction
Parallelism	SD SIMD Width	standard deviation across the number of data items affected
Memory	Total Memory Footprint	# of unique memory addresses accessed
Memory	90% Memory Footprint	# of unique memory addresses that cover 90% of memory accesses
Memory	Global Memory Address Entropy	measure of the randomness of memory addresses
Memory	Local Memory Address Entropy	measure of the spatial locality of memory addresses
Control	Total Unique Branch Instructions	# unique branch instructions
Control	90% Branch Instructions	# unique branch instructions that cover 90% of branch instructions
Control	Yokota Branch Entropy	branch history entropy using Shannon's information entropy
Control	Average Linear Branch Entropy	branch history entropy score using the average linear branch entropy

instruction- and thread-level parallelism, thread parallelism, and data movement, based on cross-compilation to a MIPS-IV simulator of an ideal machine with perfect caches and branch prediction and unlimited functional units. Instruction- and thread-level parallelism are identified through analysis of data dependencies between instructions and basic blocks. The current version of AIWC does not perform dependency analysis for characterizing parallelism, however, we hope to include such metrics in future versions.

4.2 Metrics

For each OpenCL kernel invocation the Oclgrind simulator **AIWC** tool collects a set of metrics, which are listed in Table 5.1.

The **Opcode**, **total memory footprint** and **90% memory footprint** measures are simple counts. Likewise, **Total Instruction Count** is the number of instructions achieved during a kernel execution. The **global memory address entropy** is an positive real number that corresponds to the randomness of memory addresses accessed. The **local memory address entropy** is computed as 10 separate values according to increasing number of Least Significant Bits (LSB), or low order bits, omitted in calculation. The number of bits skipped ranges from 1 to 10, and a steeper drop in entropy with increasing number of bits indicates greater spatial locality in the address stream.

Both **unique branch instructions** and the associated **90% branch instructions** are counts indicating the count of logical control flow branches encountered during kernel execution. **Yokota branch entropy** ranges between 0 and 1, and offers an indication of a program's predictability as a floating point entropy value. The **average linear branch entropy** metric is proportional to the miss rate in program execution; $p = 0$ for branches always taken or not-taken but $p = 0.5$ for the most unpredictable control flow. All branch-prediction metrics were computed using a fixed history of 16-element branch strings, each of which is composed of 1-bit branch results (taken/not-taken).

As the OpenCL programming model is targeted at parallel architectures, any workload characterization must consider exploitable parallelism and associated communication and

synchronization costs. We characterize thread-level parallelism (TLP) by the number of **work-items** executed by each kernel, which indicates the maximum number of threads that can be executed concurrently.

Work-item communication hinders TLP, and in the OpenCL setting takes the form of either local communication (within a work-group) using local synchronization (barriers) or globally by dividing the kernel and invoking the smaller kernels on the command queue. Both local and global synchronization can be measured in **instructions to barrier** by performing a running tally of instructions executed per work-item until a barrier is encountered under which the count is saved and resets; this count will naturally include the final (implicit) barrier at the end of the kernel. **Min**, **Max** and **Median ITB** are reported to understand synchronization overheads as well as load imbalance, as a large difference between min and max ITB may indicate an irregular workload.

To characterize data parallelism, we examine the number and width of vector operands in the generated LLVM IR, reported as **Max SIMD Width**, **Mean SIMD Width** and **SD SIMD Width**. Some of the other metrics are highly dependent on workload scale, so **work-items** may be used to normalize between different scales. For example, **total memory footprint** can be divided by **work-items** to give the total memory footprint per work-item, which indicates the memory required per processing element.

4.3 Methodology – Workload Characterization by tooling Oclgrind

AIWC verifies the architecture independent metrics since they are collected on a tool chain and in a language actively executed on a wide range of accelerators – the OpenCL runtime supports execution on CPU, GPU, DSP, FPGA, MIC and ASIC hardware architectures. The intermediate representation of the OpenCL kernel code is a subset of LLVM IR known as SPIR-V – Standard Portable Intermediate Representation. This IR forms a basis for Oclgrind to perform OpenCL device simulation which interprets LLVM IR instructions.

Migrating the metrics presented in the ISA-independent workload characterization paper [43] to the Oclgrind tool offers a accessible, high-accuracy and reproducible method to acquire these AIWC features. Namely:

- Accessibility: since the Oclgrind OpenCL kernel debugging tool is one of the most adopted OpenCL debugging tools freely available to date, having AIWC metric generation included as a Oclgrind plugin allows rapid workload characterization.
- High-Accuracy: evaluating the low level optimized IR does not suffer from a loss of precision since each instruction is instrumented during its execution in the simulator, unlike with the conventional metrics generated by measuring architecture driven events – such as PAPI and MICA analysis.
- Reproducibility: each instruction is instrumented by the AIWC tool during execution, there is no variance in the metric results presented between OpenCL kernel runs.

The caveat with this approach is the overhead imposed by executing full solution HPC codes on a slower simulator device. However, since AIWC metrics do not vary between runs, this is still a shorter execution time than the typical number of iterations required to get a reasonable statistical sample when compared to a MICA or architecture dependent analysis.

4.4 Implementation

AIWC is implemented as a plugin for Oclgrind, which simulates kernel execution on an ideal compute device. OpenCL kernels are executed in series, and Oclgrind generates notification events which AIWC handles to populate data structures for each workload metric. Once each kernel has completed execution, AIWC performs statistical summaries of the collected metrics by examining these data structures.

The **Opcode** diversity metric updates a counter on an unordered map during each `workItemBegin` event, the type of operation is determined by examining the opcode name using the LLVM Instruction API.

The number of **work-items** is computed by incrementing a global counter – accessible by all work-item threads – once a `workItemBegin` notification event occurs.

TLP metrics require barrier events to be instrumented within each thread. Instructions To Barrier **ITB** metrics require each thread to increment a local counter once every `instructionExecuted` has occurred, this counter is added to a vector and reset once the work-item encounters a barrier. The **Total Barriers Hit** counter also increments on the same condition. Work-items are executed sequentially within all work-items in a work-group, if a barrier is hit the queue moves onto all other available work-items in a ready state. Collection of the metrics post barrier resumes during the `workItemClearBarrier` event.

ILP **SIMD** metrics examine the size of the result variable provided from the `instructionExecuted` notification, the width is then added to a vector for the statistics to be computed once the kernel execution has completed.

Total Memory Footprint 90% Memory Footprint and Local Memory Address Entropy **LMAE** metrics require the address accessed to be stored during kernel execution and occurs during the `memoryLoad`, `memoryStore`, `memoryAtomicLoad` and `memoryAtomicStore` notifications.

Branch entropy measurements require a check during `instructionExecuted` event on whether the instruction is a branch instruction, if so a flag indicating a branch operation has occurred is set and both LLVM IR labels – which correspond to branch targets – are recorded. On the next `instructionExecuted` the flag is queried and reset while the current instruction label is examined and is stored around which of the two targets were taken. The branch metrics can then be computed. The **Total Unique Branch Instructions** is a count of the absolute number of unique locations that branching occurred, while the **90% Branch Instructions** indicates the number of unique branch locations that cover 90% of all branches. **Yokota** from Shao [43], and **Average Linear Branch Entropy**, from De Pestel [63] and have been computed and are also presented based on this implementation. `workGroupComplete` events trigger the collection of the intermediate work-item and work-group counter variables to be added to the global suite, while `workGroupBegin` events reset all the local/intermediate counters.

Finally, `kernelBegin` initializes the global counters and `kernelEnd` triggers the generation and presentation of all the statistics listed in Table 5.1. The source code is available at the GitHub Repository [65].

4.5 Demonstration

We now demonstrate the use of **AIWC** with a few example scientific application kernels selected from the Extended OpenDwarfs Benchmark Suite [66]. These benchmarks were extracted from and are representative of general scientific application codes. Our selection is not intended to be exhaustive, rather, it is meant to illustrate how key properties of the codes are reflected in the metrics collected by **AIWC**.

AIWC is run on full application codes, but it is difficult to present an entire summary due to the nature of OpenCL. Computationally intensive kernels are simply selected regions of the full application codes and are invoked separately for device execution. As such, the AIWC metrics can either be shown per kernel run on a device, for all kernel runs on the device, or as the summation of all metrics for a kernel for a full application at a given problem size – we chose the latter. Additionally, given the number of kernels presented we believe AIWC will generalise to full codes in other domains.

We present metrics for 11 different application codes – which includes 37 kernels in total. Each code was run with four different problem sizes, called **tiny**, **small**, **medium** and **large** in the Extended OpenDwarfs Benchmark Suite; these correspond respectively to problems that would fit in the L1, L2 and L3 cache or main memory of a typical current-generation CPU architecture. As simulation within Oclgrind is deterministic, all results presented are for a single run for each combination of code and problem size.

In a cursory breakdown 4 selected metrics are presented in Figure~4.1. Each of the 4 metrics were chosen as one of each of the main categories, namely, Opcode, Barriers Per Instruction, Global Memory Address Entropy, Branch Entropy (Linear Average). Each category has also been segmented by colour: blue results represent *compute* metrics, green represent metrics that indicate *parallelism*, beige represents *memory* metrics and purple bars represent *control* metrics. Median results are presented for each metric – while there is no variation between invocations of AIWC, certain kernels are iterated multiple times and over differing domains / data sets. Each of the 4 sub-figures shows all kernels over the over 4 different sized problems.

All kernels are presented along the x-axis, whereas the normalized percentage of each category is presented in the y-axis.

Generally, problem size primarily affects the memory category – where global memory address entropy increases with problem size – while the generic shape of the other categories is fixed per kernel. Note, there are fewer kernels presented over the **medium** and **large** problem sizes due to the difficulties in determining arguments and the ability of some benchmark applications to run correctly over the larger problem sizes.

Following Shao and Brooks [43], we present the AIWC metrics for a kernel as a kiviat or radar diagram, for each of the problem sizes. Unlike Shao and Brooks, we do not perform any dimensionality reduction, but choose to present all collected metrics. The ordering of the individual spokes is not chosen to reflect any statistical relation between the metrics, however, they have been grouped into four main categories: green spokes represent metrics that indicate *parallelism*, blue spokes represent *compute* metrics, beige spokes represent *memory* metrics and purple spokes represent *control* metrics. For clarity of visualization, we do not present the raw AIWC metrics, but instead normalize or invert the metrics to produce a scale from 0 to 1. The parallelism metrics presented are the inverse values of the metrics collected by AIWC, i.e. **granularity** = 1/**work-items** ; **barriers per instruction** = 1/**mean ITB** ; **instructions per operand** = $1/\sum$ **SIMD widths**. All other values are normalized according to

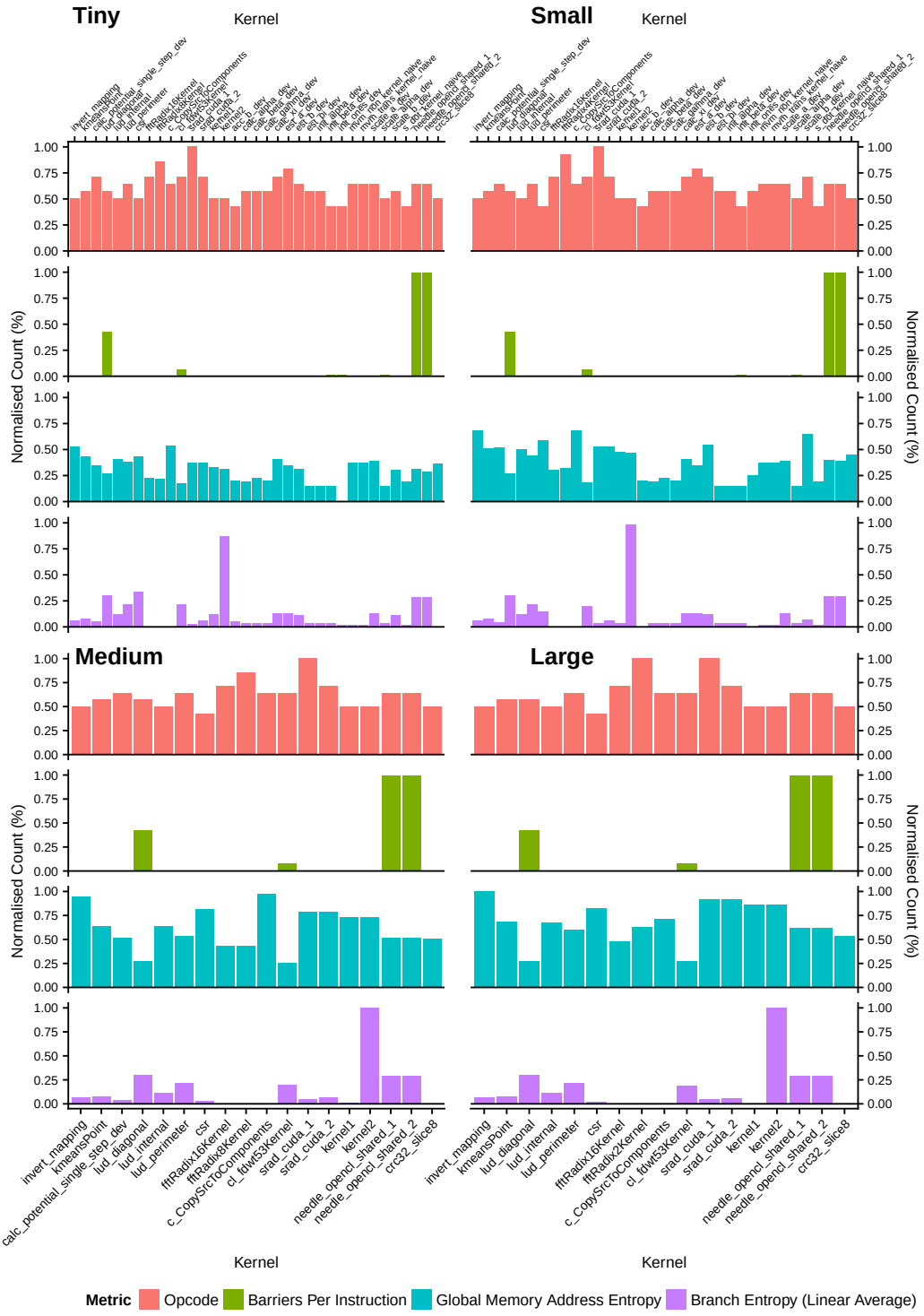

Figure 4.1: Selected AIWC metrics from each category over all kernels and 4 problem sizes.

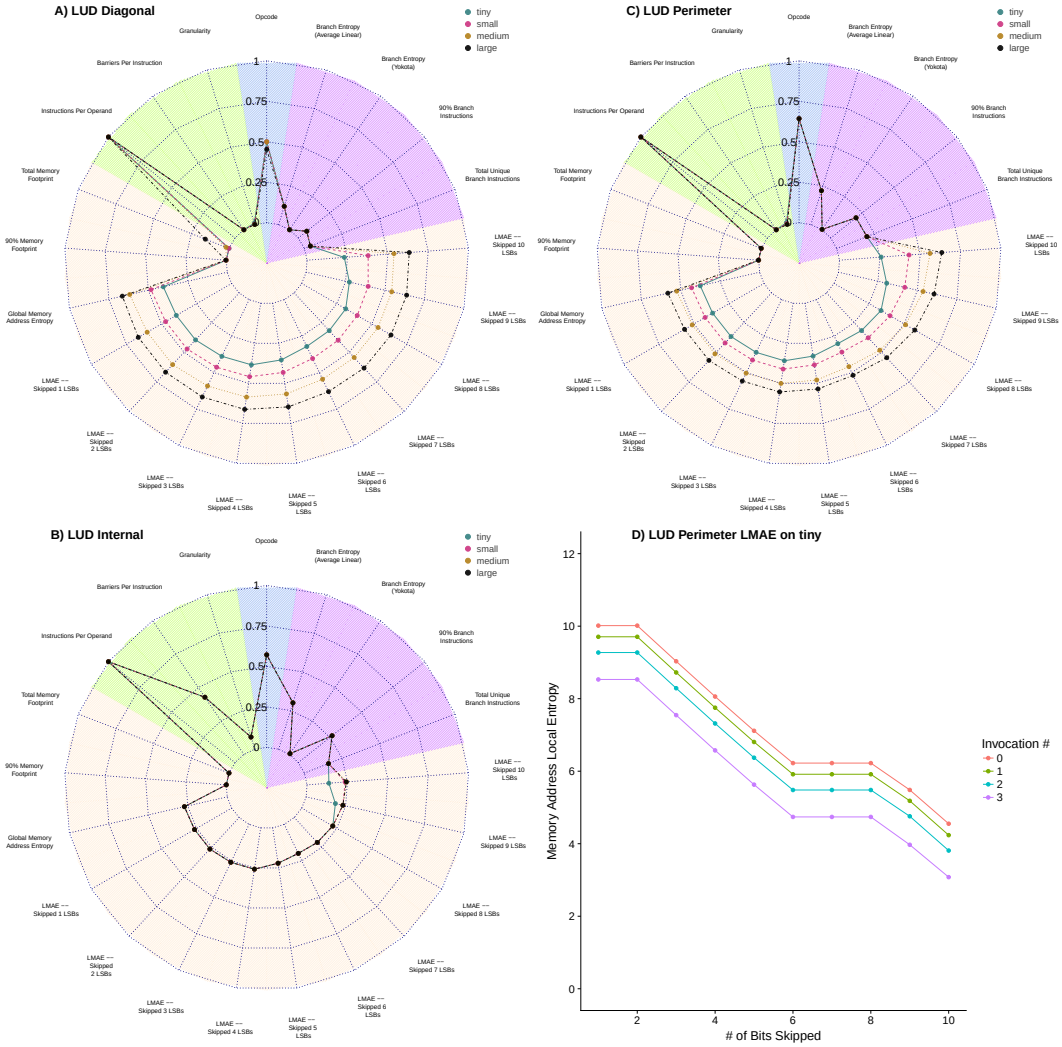

Figure 4.2: A) B) and C) show the AIWC features of the `diagonal`, `internal` and `perimeter` kernel of the LUD application over all problem sizes. D) shows the corresponding Local Memory Address Entropy for the `perimeter` kernel over the tiny problem size.

the maximum value measured across all kernels examined – and on all problem sizes. This presentation allows a quick value judgement between kernels, as values closer to the center (0) generally have lower hardware requirements, for example, smaller entropy scores indicate more regular memory access or branch patterns, requiring less cache or branch predictor hardware; smaller granularity indicates higher exploitable parallelism; smaller barriers per instruction indicates less synchronization; and so on.

The **lud** benchmark application comprises three major kernels, **diagonal**, **internal** and **perimeter**, corresponding to updates on different parts of the matrix. The AIWC metrics for each of these kernels are presented – superimposed over all problem sizes – in Figure~4.2 A) B) and C) respectively. Comparing the kernels, it is apparent that the diagonal and perimeter kernels have a large number of branch instructions with high branch entropy, whereas the internal kernel has few branch instructions and low entropy. This is borne out through inspection of the OpenCL source code: the internal kernel is a single loop with fixed bounds, whereas diagonal and perimeter kernels contain doubly-nested loops over triangular bounds and branches which depend on thread id. Comparing between problem sizes (moving across the page), the large problem size shows higher values than the tiny problem size for all of the memory metrics, with little change in any of the values.

The visual representation provided from the kiviat diagrams allows the characteristics of OpenCL kernels to be readily assessed and compared.

Finally, we examine the linear memory access entropy (LMAE) presented in the kiviat diagrams in greater detail. Figure~4.2 D) presents a sample of the linear memory access entropy, in this instance of the LUD Perimeter kernel collected over the tiny problem size. The kernel is launched 4 separate times during a run of the tiny problem size, this is application specific and in this instance each successive invocation operates on a smaller data set per iteration. Note there is steady decrease in starting entropy, and each successive invocation of the LU Decomposition Perimeter kernel the lowers the starting entropy. However the descent in entropy – which corresponds to more bits being skipped, or bigger the strides or the more localized the memory access – shows that the memory access patterns are the same regardless of actual problem size.

4.6 Conclusions and Future Work

We have presented the Architecture-Independent Workload Characterization tool (AIWC), which supports the collection of architecture-independent features of OpenCL application kernels. These features can be used to predict the most suitable device for a particular kernel, or to determine the limiting factors for performance on a particular device, allowing OpenCL developers to try alternative implementations of a program for the available accelerators – for instance, by reorganizing branches, eliminating intermediate variables et cetera. The additional architecture independent characteristics of a scientific workload will be beneficial to both accelerator designers and computer engineers responsible for ensuring a suitable accelerator diversity for scientific codes on supercomputer nodes.

Caparrós Cabezas and Stanley-Marbell [64] examine the Berkeley dwarf taxonomy by measuring instruction-level parallelism, thread parallelism, and data movement. They propose a sophisticated metric to assess ILP by examining the data dependency graph of the instruction stream. Similarly, Thread-Level-Parallelism was measured by analysing the block dependency graph. Whilst we propose alternative metrics to evaluate ILP and TLP – using the max, mean

and standard deviation statistics of SIMD width and the total barriers hit and Instructions To Barrier metrics respectively – a quantitative evaluation of the dwarf taxonomy could be performed by examining these metrics. This evaluation could be used to examine the legitimacy of the Dwarf Taxonomy. For instance, it is envisaged that if the original proposal of 13 dwarfs is correct, such that all scientific applications can be identified by one or more of these dwarfs, this implies that each individual kernel is demonstrated by one dwarf. One would then expect that each AIWC metric would have – at the most – 13 unique clusters corresponding to each of the 13 dwarfs for any individual kernel analysis. If this is true then the legitimacy is confirmed, however if not, perhaps AIWC metrics are a better and more direct measure of application diversity when assembling a benchmark suite. This is left as future work, however, we expect that the additional AIWC metrics will generate a comprehensive feature-space representation which will permit cluster analysis and comparison with the dwarf taxonomy.

The coverage of characteristics and the suitability AIWC metrics can now be assessed. This is performed in in the next Chapter 5 where these AIWC metrics – from the collection over all EOD applications and over all problem sizes – are used as predictor variables to form a model with the aim of performing execution time predictions. Which could in turn be directly used to schedule devices in the HPC mult-accelerator node setting. The feature-space collected from AIWC is also evaluated – if accurate model predictions are achieved, relative to the actual measured execution times presented in Chapter 3, then the metrics selected during the design of AIWC are valid – since all significant components that depict an applications execution time on any accelerator have been measured.

Chapter 5

Application-Accelerator Performance Prediction

remove old abstract? It's the next 2 paragraphs

OpenCL is an attractive programming model for heterogeneous high-performance computing ~~systems~~, with wide support from hardware vendors and significant performance portability. To support efficient scheduling on HPC systems it is necessary to perform accurate performance predictions for OpenCL workloads on varied compute devices, which is challenging due to diverse computation, communication and memory access characteristics which result in varying performance between devices.

The Architecture Independent Workload Characterization (AIWC) tool can be used to characterize OpenCL kernels according to a set of architecture-independent features. This work presents a methodology where AIWC features are used to form a model capable of predicting accelerator execution times. We used this methodology to predict execution times for a set of 37 computational kernels running on 15 different devices representing a broad range of CPU, GPU and MIC architectures. The predictions are highly accurate, differing from the measured experimental run-times by an average of only 1.2%, and correspond to actual execution time mispredictions of 9 μs to 1 sec according to problem size. A previously unencountered code can be instrumented once and the AIWC metrics embedded in the kernel, to allow performance prediction across the full range of modelled devices. The results suggest that this methodology supports correct selection of the most appropriate device for a previously unencountered code, which is highly relevant to the HPC scheduling setting.

This Chapter uses the performance results presented in Chapter 4 and the feature-spaces presented in Chapter 5 to develop a model. The feature-space first undergoes a reduction to simplify the space from the application domain to one focused on the superset of applications – dwarfs. This model is then used to predict an optimal accelerator type for new OpenCL kernels, ones unused for the model development. The evaluation of this model is presented and the corresponding feasibility addresses the primary question raised by this thesis, namely: *Can the structure of OpenCL kernels be used determine the type of accelerator on which it should be run?*

5.1 Introduction

HPC architectures are becoming increasingly heterogeneous. This trend is increasingly apparent at the node level in supercomputer systems. For instance, the Cori system at Lawrence Berkeley National Laboratory comprises 2,388 Cray XC40 nodes with Intel Haswell CPUs, and 9,688 Intel Xeon Phi nodes [2]. The Summit supercomputer at Oak Ridge National Laboratory is based on the IBM Power9 CPU, which includes both NVLINK [3], a high bandwidth interconnect between Nvidia GPUs; and CAPI, an interconnect to support FPGAs and other accelerators [4]. Promising next-generation architectures include Fujitsu's Post-K [5], and Cray's CS-400, which forms the platform for the Isambard supercomputer [6]. Both architectures use ARM cores alongside other conventional accelerators, with several Intel Xeon Phi and Nvidia P100 GPUs per node.

The OpenCL programming framework is well-suited to such heterogeneous computing environments, as a single OpenCL code may be executed on multiple different device types including most CPU, GPU and FPGA devices. Predicting the performance of a particular application on a given device is challenging due to complex interactions between the computational requirements of the code and the capabilities of the target device. Certain classes of application are better suited to a certain type of accelerator [67], and choosing the wrong device results in slower and more energy-intensive computation [68]. Thus accurate performance prediction is critical to making optimal scheduling decisions in a heterogeneous supercomputing environment.

The Architecture-Independent Workload Characterization (AIWC) tool [69] was previously introduced in order to collect architecture-independent features of OpenCL application workload. AIWC operates on OpenCL kernels by simulating an OpenCL device and performing instrumentation to collect various features to characterize parallelism, compute complexity, memory and control that are independent of the target execution architecture. In this paper, we propose a model that employs the AIWC features to make accurate predictions over a range of current accelerators. These features are used to build a model which accurately predicts the execution times of a previously unseen OpenCL code over the range of available devices. The performance predictions from this model may serve as input to scheduling decisions on heterogeneous supercomputing systems.

A major benefit of this approach is that the developer need only instrument a kernel once and the AIWC metrics can be embedded as a comment in the kernel's source code or Standard Portable Intermediate Representation (SPIR). A scheduler system could be augmented to use the performance model with very low overhead, since querying the model is computationally inexpensive. The model need only be retrained when a new accelerator type is added. The methodology to develop the model is outlined in the following sections. All tools used are open source, and all code is available in the respective repositories: [61] and [65].

5.2 Related Work

Augonnet et al. [70] propose a task scheduling framework for efficiently issuing work between multiple heterogeneous accelerators on a per-node basis. They focus on the dynamic scheduling of tasks while automating data transfers between processing units to better utilise GPU-based HPC systems. Much of this work is placed on evaluating the scaling of two applications over multiple nodes – each of which are comprised of many GPUs. Unfortunately,

the presented methodology requires code to be rewritten using their MPI-like library. OpenCL, by comparison, has been in use since 2008 and supports heterogeneous execution on most accelerator devices. The algorithms presented to automate data movement should be reused for scheduling of OpenCL kernels to heterogeneous accelerator systems.

Existing works, [71], [72], [73] and [74], have addressed heterogeneous distributed system scheduling and in particular the use of Directed Acyclic Graphs to track dependencies of high priority tasks. Provided the parallelism of each dependency is expressed as OpenCL kernels, the model proposed here can be used to improve each of these scheduler algorithms by providing accurate estimates of execution time for each task for each potential accelerator on which the computation could be performed.

Our work is most closely related to efforts to enable low-cost performance estimates over a wide range of execution platforms. One such approach uses partial execution, as introduced by Yang et al. [75]. Here a short portion of a parallel code is executed and, since parallel codes are iterative behave predictably after the initial startup portion. An important restriction for this approach is it requires execution on each of the accelerators for a given code, which may be complicated to achieve using common HPC scheduling systems.

An alternative performance prediction approach is given by Carrington et al. [76]. Their solution generates two separate models each requiring two fundamental components: firstly, a machine profile of each system generated by running micro-benchmarks to probe simple performance attributes of each machine; and secondly, application signatures generated by instrumented runs which measure block information such as floating-point utilization and load/store unit usage of an application. This is akin to our proposed solution using AIWC to generate each application signature and the generation of a random forest model to learn each machine profile. However, in their method, no training takes place and the micro-benchmarks were developed with CPU memory hierarchy in mind, thus it is unsuited to a broader range of accelerator devices. There are also many components and tools in use, for instance, network traffic is interpreted separately and requires the communication model to be developed from a different set of network performance capabilities, which needs more micro-benchmarks. In comparison, our proposed solution uses a plugin to the Oclgrind tool, which is already widely used by OpenCL developers.

5.3 Methodology

The AIWC tool [69] is a plugin to the Oclgrind [46] OpenCL device simulator, debugging and instrumentation tool. AIWC simulates the execution of OpenCL kernels to collect architecture-independent features which characterize each code. It operates on a restricted LLVM IR known as Standard Portable Intermediate Representation (SPIR) [47], thereby simulating OpenCL kernel code in a hardware agnostic manner. The AIWC metrics are shown in Table 5.1. We collected these metrics for a suite of benchmarks representative of scientific codes, which cover a wide spectrum of computation, communication and memory access patterns. For each benchmark, we also collected detailed performance measurements on a varied set of compute devices.

Table 5.1: **AIWC** tool metrics.

Type	Metric	Description
Compute	opcode	# of unique opcodes required to cover 90% of dynamic instructions
Compute	Total Instruction Count	Total # of instructions executed
Parallelism	Work-items	# of work-items or threads executed
Parallelism	Total Barriers Hit	maximum # of instructions executed until a barrier
Parallelism	Min ITB	minimum # of instructions executed until a barrier
Parallelism	Max ITB	maximum # of instructions executed until a barrier
Parallelism	Median ITB	median # of instructions executed until a barrier
Parallelism	Max SIMD Width	maximum number of data items operated on during an instruction
Parallelism	Mean SIMD Width	mean number of data items operated on during an instruction
Parallelism	SD SIMD Width	standard deviation across the number of data items affected
Memory	Total Memory Footprint	# of unique memory addresses accessed
Memory	90% Memory Footprint	# of unique memory addresses that cover 90% of memory accesses
Memory	Global Memory Address Entropy	measure of the randomness of memory addresses
Memory	Local Memory Address Entropy	measure of the spatial locality of memory addresses
Control	Total Unique Branch Instructions	# unique branch instructions
Control	90% Branch Instructions	# unique branch instructions that cover 90% of branch instructions
Control	Yokota Branch Entropy	branch history entropy using Shannon's information entropy
Control	Average Linear Branch Entropy	branch history entropy score using the average linear branch entropy

5.3.1 Experimental Setup

AIWC was used to characterize a variety of codes in the Extended OpenDwarfs (EOD) Benchmark Suite [66], and the corresponding AIWC metrics were used as predictor variables in to fit a random forest regression model. The metrics were generated over 4 problem sizes for each of the 11 applications – and 37 computationally regions known as kernels in the OpenCL setting. Response variables were collected following the same methodology outlined in [66] – where the details for each of the applications is also presented. Execution times were measured for at least 50 iterations and a total runtime of at least two seconds for each combination of device and benchmark. Each application was run over 15 different accelerator devices, and are presented in Table 5.2. The L1 cache size should be read as having both an instruction cache and a data cache of the stated size. For Nvidia GPUs, the L2 cache size reported is the size L2 cache per SM multiplied by the number of SMs. For the Intel CPUs, Hyper-threading was enabled and the frequency governor was set to `performance`.

5.3.2 Constructing the Performance Model

The R programming language was used to analyse the data, construct the model and analyse the results. In particular, the ranger package by Wright and Ziegler [77] was used for the development of the regression model. The ranger package provides computationally efficient implementations of the Random Forest model [78] which performs recursive partitioning of high dimensional data.

The ranger function accepts three main parameters, each of which influences the fit of the model to the data. In optimizing the model, we searched over a range of values for each parameter including:

- num.trees, the number of trees grown in the random forest: over the range of 10 – 10,000 by 500
- mtry, the number of features tried to possibly split within each node: ranges from 1 – 34, where 34 is the maximum number of input features available from AIWC,

Table 5.2: Experimental hardware for generating runtime response data

Name	Vendor	Type	Series	Core Count	Clock Frequency (MHz) (min/max/turbo)	Cache (KiB) (L1/L2/L3)	TDP (W)	Launch Date
Xeon E5-2697 v2	Intel	CPU	Ivy Bridge	24∗	1200/2700/3500	32/256/30720	130	Q3 2013
i7-6700K	Intel	CPU	Skylake	8∗	800/4000/4300	32/256/8192	91	Q3 2015
i5-3550	Intel	CPU	Ivy Bridge	4∗	1600/3380/3700	32/256/6144	77	Q2 2012
Titan X	Nvidia	GPU	Pascal	3584†	1417/1531/–	48/2048/–	250	Q3 2016
GTX 1080	Nvidia	GPU	Pascal	2560†	1607/1733/–	48/2048/–	180	Q2 2016
GTX 1080 Ti	Nvidia	GPU	Pascal	3584†	1480/1582/–	48/2048/–	250	Q1 2017
K20m	Nvidia	GPU	Kepler	2496†	706/–/–	64/1536/–	225	Q4 2012
K40m	Nvidia	GPU	Kepler	2880†	745/875/–	64/1536/–	235	Q4 2013
FirePro S9150	AMD	GPU	Hawaii	2816‖	900/–/–	16/1024/–	235	Q3 2014
HD 7970	AMD	GPU	Tahiti	2048‖	925/1010/–	16/768/–	250	Q4 2011
R9 290X	AMD	GPU	Hawaii	2816‖	1000/–/–	16/1024/–	250	Q3 2014
R9 295x2	AMD	GPU	Hawaii	5632‖	1018/–/–	16/1024/–	500	Q2 2014
R9 Fury X	AMD	GPU	Fuji	4096‖	1050/–/–	16/2048/–	273	Q2 2015
RX 480	AMD	GPU	Polaris	4096‖	1120/1266/–	16/2048/–	150	Q2 2016
Xeon Phi 7210	Intel	MIC	KNL	256‡	1300/1500/–	32/1024/–	215	Q2 2016

∗ HyperThreaded cores
† CUDA cores
‖ Stream processors
‡ Each physical core has 4 hardware threads per core, thus 64 cores

- min.node.size, the minimal node size per tree: ranges from 1 – 50, where 50 is the number of observations per sample.

Given the size of the data set, it was not computationally viable to perform an exhaustive search of the entire 3-dimensional range of parameters. Auto-tuning to determine the suitability of these parameters has been performed by Ließ et al. [79] to determine the optimal value of mtry given a fixed num.trees. Instead, to enable an efficient search of all variables at once, we used Flexible Global Optimization with Simulated-Annealing, in particular, the variant found in the R package *optimization* by Husmann, Lange and Spiegel [80]. The simulated-annealing method both reduces the risk of getting trapped in a local minimum and is able to deal with irregular and complex parameter spaces as well as with non-continuous and sophisticated loss functions. In this setting, it is desirable to minimise the out-of-bag prediction error of the resultant fitted model, by simultaneously changing the parameters (num.trees, mtry and min.node.size). The *optim_sa* function allows defining the search space of interest, a starting position, the magnitude of the steps according to the relative change in temperature and the wrapper around the ranger function (which parses the 3 parameters and returns a cost function — the predicted error). It allows for an approximate global minimum to be detected with significantly fewer iterations than an exhaustive grid search.

Figure 5.1 shows the relationship between out-of-bag prediction error and min.node.size, with the num.trees = 300 and mtry = 30 parameters fixed. In general, the min.node.size has the smallest prediction error for values less than 15 and variation in prediction error is similar throughout this range. As such, the selection to fix min.node.size = 9 was made to reduce the search-space in the remainder of the tuning work. We assume conditional (relative) independence between min.node.size and the other variables.

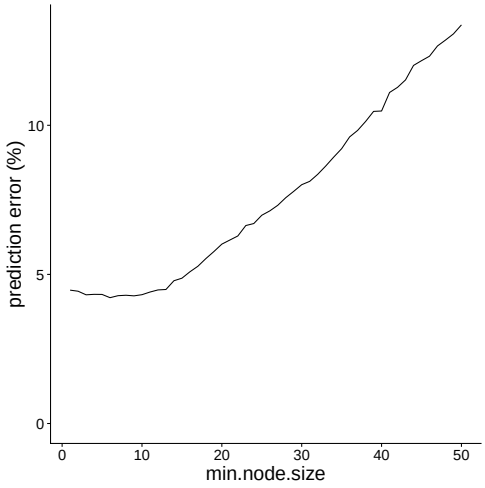

Figure 5.1: Full coverage of min.node.size with fixed tuning parameters: num.trees = 300 and mtry = 30.

Figure 5.2 shows how the prediction error of the random-forest ranger model changes over a wide range of values for the two remaining tuning parameters, mtry and num.trees. Full coverage was achieved by selecting starting locations in each of the 4 outer-most points of the search space, along with 8 random internal points — to avoid missing out on some critical internal structure. For each combination of parameter values, the *optim_sa* function was allowed to execute until a global minimum was found. At each step of optimization a full trace was collected, where all parameters and the corresponding out-of-bag prediction error value were logged to a file. This file was finally loaded, the points interpolated using the R package akima, without extrapolation between points, using the mean values for duplication between points. The generated heatmap is shown in Figure 5.2.

A lower out-of-bag prediction error is better. For values of mtry above 25, there is good model fit irrespective of the number of trees. For lower values of mtry, fit varies significantly with different values of num.trees. The worst fit was for a model with a value of 1 num.trees, and 1 for mtry, which had the highest out-of-bag prediction error at 194%. In general, the average prediction error across all choices of parameters is very low at 16%. Given these results, the final ranger model should use a small value for num.trees and a large value for mtry, with the added benefit that such a model can be computed faster given a smaller number of trees.

5.3.3 Choosing Model Parameters

The selected model should be able to accurately predict execution times for a previously unseen kernel over the full range of accelerators. To show this, the model must not be over-fitted, that is to say, the random forest model parameters should not be tuned to the particular set of kernels in the training data, but should generate equally good fits if trained on any other reasonable selection of kernels.

We evaluated how robust the selection of model parameters is to the choice of kernel by repeatedly retraining the model on a set of kernels, each time removing a different kernel. The procedure used is presented in Algorithm 1. For each selection of kernels, *optima_sa* was

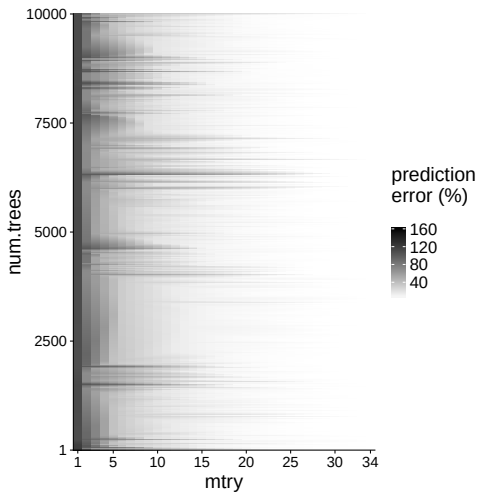

Figure 5.2: Full coverage of num.trees and mtry tuning parameters with min.node.size fixed at 9.

Algorithm 1: Find the suitability of the optimal parameters for random forest models for future kernels

for *each unique kernel* **do**
    construct a full data frame with all but the current kernel;
    run optimization *optim_sa* with the full data frame at selected starting location;
    record the final optimal parameters

run from the same starting location – num.trees=500, mtry=32 – and the final optimal values were recorded. min.node.size was fixed at 9.

The optimal – and final – parameters for each omitted kernel are presented in Table 5.3. Regardless of which kernel is omitted, the R-squared values – or explained variance – is very high at 0.99, indicating a good model fit. The optimal parameters are very similar regardless of which kernel was omitted. As such, the median value of each of the parameters was selected for the final model: num.trees = 505, mtry = 30 and min.node.size = 9. These parameters were used for all further model training.

5.3.4 Performance Improvement with Increased Training Data

For a model to be useful in predicting execution times for previously unseen kernels, it needs to be trained on a representative sample of kernels i.e. a sample that provides good coverage of the AIWC feature space of all possible application kernels.

We measured how model fit improves with the number of kernels used in training, following the method presented in Algorithm 2. The set of unique kernels available during model development is denoted by k (37 kernels in this study), s is the maximum number of sample models (including different combinations of kernels) to evaluate for each number of kernels $1..|k|$, ϕ is a data frame of the combined AIWC feature-space with measured runtime results. The parameters to the random forest model were fixed at num.trees = 505, mtry = 30 and min.node.size = 9, according to the methodology in Section 5.3.3.

Algorithm 2: Compute average fit of random forest models trained on different numbers of kernels.

```
s ← 500
k ←unique(kernel)
for i ← 1to length(k) do
    v_p ← []
    v_m ← []
    for j ← 1 to s do
        x ←shuffle(k)
        y ← x[1 .. i]
        training data ← subset(φ, kernel == y)
        test data ← subset(φ, kernel ! = y)
        discard variables unavailable during real-world training from training data e.g. size,
          application, kernel name and measured total application time
        build ranger model r using training data
        generate prediction responses p from r using test data
        append predicted execution times p to v_p
        append measured execution times from test data to v_m
    compute the mean absolute error e from vector of p relative to vector m
    store(e)
```

The results presented in Figure 5.3 show the mean absolute error of models trained on varying numbers of kernels. As expected, the model fit improves with increasing number of kernels. In particular, larger improvements occur with each new kernel early in the series and tapers off as a new kernel is added to an already large number of kernels. The gradient is still significant until the largest number of samples examined ($k = 37$) suggesting that the model could benefit from additional training data. However, the model proposed is a proof of concept and suggests that a general purpose model is attainable and may not require many more kernels.

5.4 Evaluation

Figure 5.4 presents the measured kernel execution times against the predicted execution times from the trained model. Each point represents a single combination of kernel and problem size. The plot shows a strong linear correlation indicating a good model fit. Under-predictions typically occur on four kernels over the medium and large problem sizes, while over-predictions occur on the tiny and small problem sizes. However, these outliers are visually over-represented in this figure as the final mean absolute error is low, at ~0.1.

5.5 Making Predictions

In this section, we examine differences in accuracy of predicted execution times between different kernels, which is of importance if the predictions are to be used in a scheduling setting.

The four heat maps presented in Figure 5.5 show the difference between mean predicted and measured kernel execution times as a percentage of the measured time. Thus, they depict the

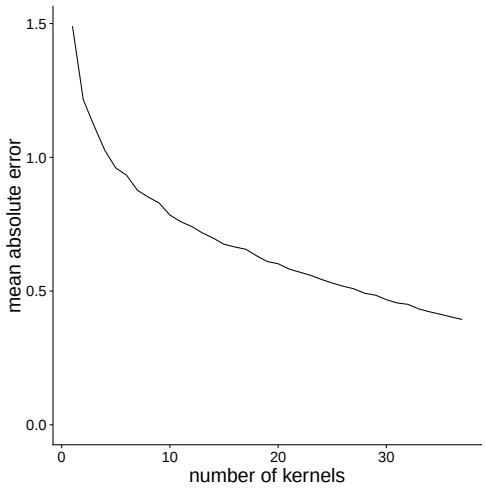

Figure 5.3: Prediction error across all benchmarks for models trained with varying numbers of kernels.

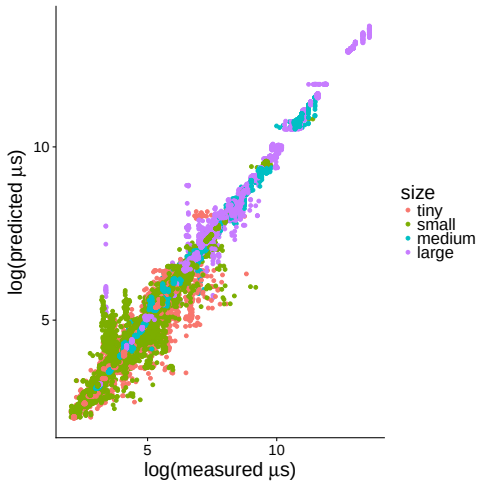

Figure 5.4: Predicted vs. measured execution time for all kernels

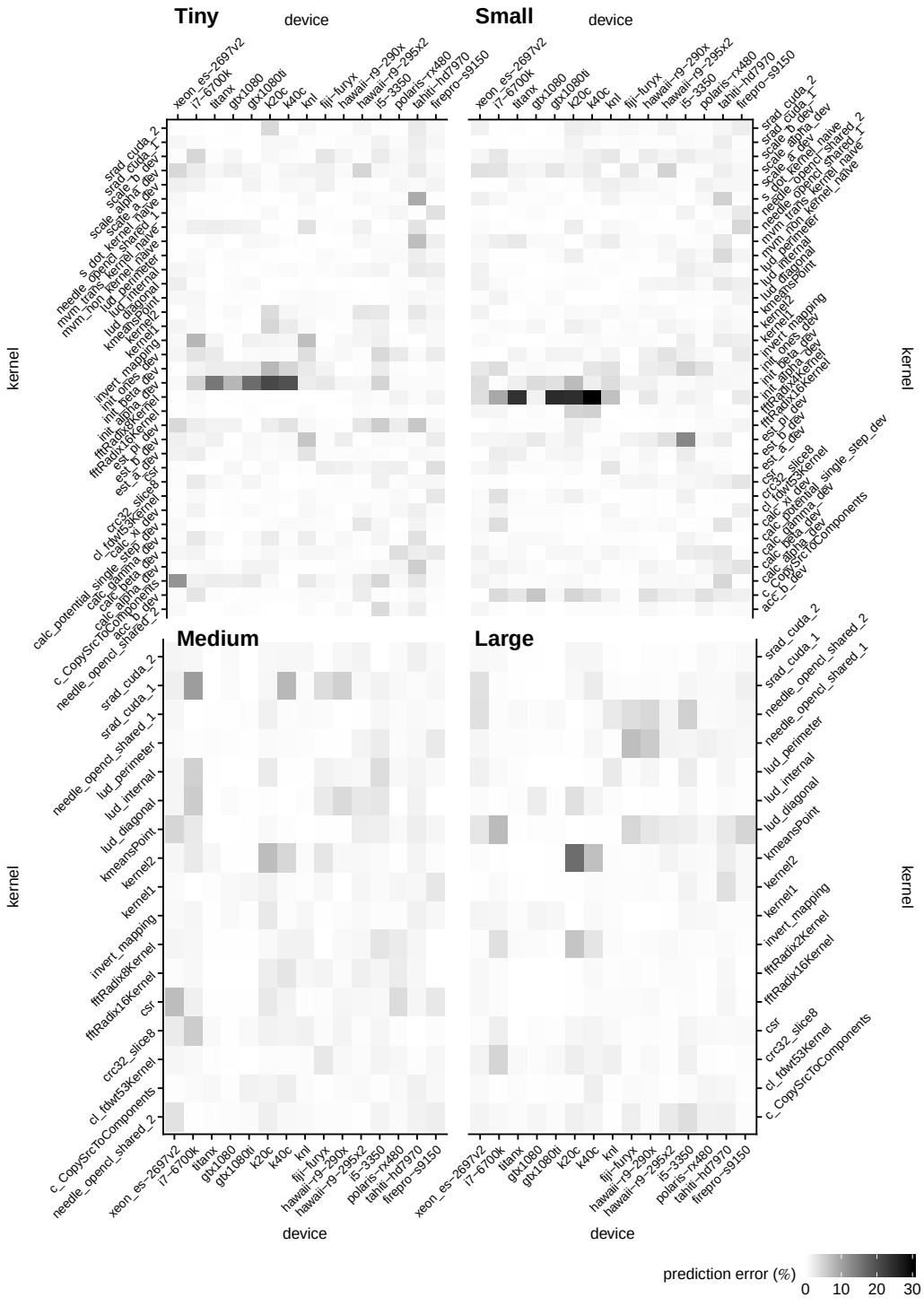

Figure 5.5: Error in predicted execution time for each kernel invocation over four problem sizes

relative error in prediction – lighter indicates a smaller error. Four different problem sizes are presented: tiny in the top-left, small in the top-right, medium bottom-left, large bottom-right.

In general, we see highly accurate predictions which on average differ from the measured experimental run-times by 1%, which correspond to actual execution time mispredictions of 8 μs to 1 secs according to problem size.

The `init_alpha_dev` kernel is the worst predicted kernel over both the tiny and small problem sizes, with mean misprediction at 7.6%. However, this kernel is only run once per application run – it is used in the initialization of the Hidden Markov Model – and as such there are fewer response variables available for model training.

5.6 The benefits of this approach

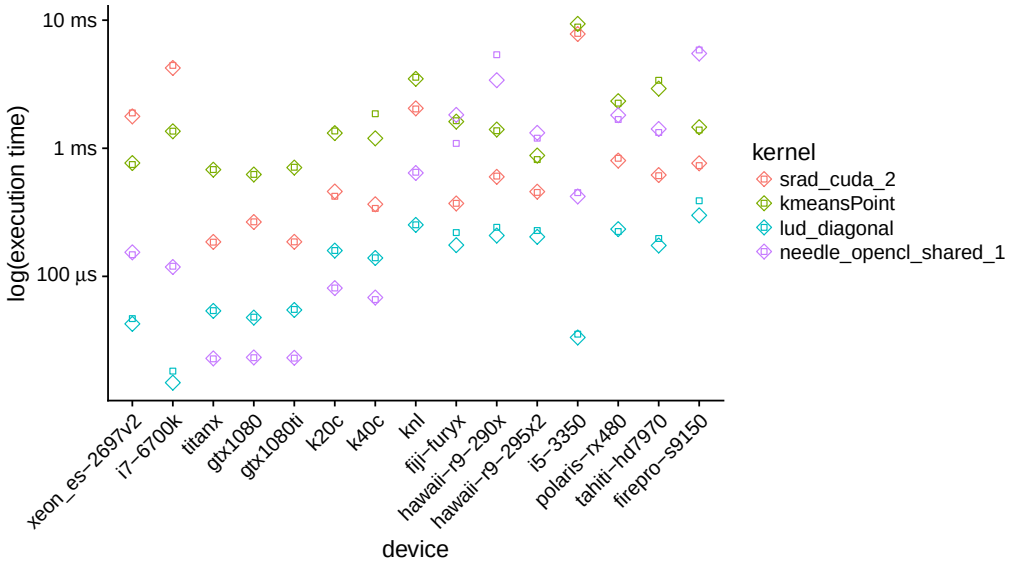

Figure 5.6: Mean measured kernel execution times compared against mean predicted kernel execution times to perform a selection of kernels on large problem sizes across 15 accelerator devices.

To demonstrate the utility of the trained model to guide scheduling choices, we focus on the accuracy of performance time prediction of individual kernels over all devices. The model performance in terms of real execution times is presented for four randomly selected kernels in Figure 5.6. The shape denotes the type of execution time data point, a square indicates the mean measured time, and the diamond indicates the mean predicted time. Thus, a perfect prediction occurs where the measured time – square – fits perfectly within the predicted – diamond – as seen in the legend.

The purpose of showing these results is to highlight the setting in which they could be used – on the supercomputing node. In this instance, it is expected a node to be composed of any combination of the 15 devices presented in the Figure 5.6. Thus, to be able to advise a scheduler which device to use to execute a kernel, the model must be able to correctly predict on which of a given pair of devices the kernel will run fastest. For any selected pair of

devices, if the relative ordering of the measured and predicted execution times is different, the scheduler would choose the wrong device. In almost all cases, the relative order is preserved using our model. In other words, our model will correctly predict the fastest device in all cases – with one exception, the `kmeansPoint` kernel. For this kernel, the predicted time of the fiji-furyx is lower than the hawaii-r9-290x, however the measured times between the two shows the furyx completing the task in a shorter time. For all other device pairs, the relative order for the `kmeansPoint` kernel is correct. Additionally, the `lud_diagonal` kernel suffers from systematic under-prediction of execution times on AMD GPU devices, however the relative ordering is still correct. As such, the proposed model provides sufficiently accurate execution time predictions to be useful for scheduling to heterogeneous compute devices on supercomputers.

5.7 Conclusions and Future Work

A highly accurate model has been presented that is capable of predicting execution times of OpenCL kernels on specific devices based on the computational characteristics captured by the AIWC tool. A real-world scheduler could be developed based on the accuracy of the presented model.

We do not suppose that we have used a fully representative suite of kernels, however, we have shown that this approach can be used in the supercomputer accelerator scheduling setting, and the model can be extended/augmented with additional training kernels using the methodology presented in this paper.

We expect that a similar model could be constructed to predict energy or power consumption, where the response variable can be directly swapped for an energy consumption metric – such as joules – instead of execution time. However, we have not yet collected the energy measurements required to construct such a model. Finally, we show the predictions made are accurate enough to inform scheduling decisions.

Table 5.3: Optimal tuning parameters from the same starting location for all models omitting each individual kernel.

Kernel omitted	num.trees	mtry	prediction error (%)
invert_mapping	521	31	4.3
kmeansPoint	511	30	4.1
lud_diagonal	527	29	4.4
lud_internal	488	31	4.5
lud_perimeter	480	31	4.4
csr	507	30	4.4
fftRadix16Kernel	484	29	4.4
fftRadix8Kernel	529	34	4.3
fftRadix4Kernel	463	30	4.2
fftRadix2Kernel	443	28	4.4
calc_potential_single_step	502	24	4.8
c_CopySrcToComponents	529	31	4.1
cl_fdwt53Kernel	499	26	4.7
srad_cuda_1	504	32	4.7
srad_cuda_2	500	29	4.6
kernel1	536	30	4.5
kernel2	469	31	4.6
acc_b_dev	576	28	4.4
calc_alpha_dev	469	30	4.3
calc_beta_dev	498	30	4.3
calc_gamma_dev	517	28	4.4
calc_xi_dev	439	33	4.3
est_a_dev	524	30	4.2
est_b_dev	533	28	4.3
est_pi_dev	450	31	4.3
init_alpha_dev	558	32	2.6
init_beta_dev	467	30	4.1
init_ones_dev	566	32	4.1
mvm_non_kernel_naive	514	30	4.3
mvm_trans_kernel_naive	449	32	4.4
scale_a_dev	508	31	4.3
scale_alpha_dev	530	30	3.8
scale_b_dev	565	31	4.2
s_dot_kernel_naive	509	30	4.5
needle_opencl_shared_1	499	30	4.4
needle_opencl_shared_2	504	29	4.5
crc32_slice8	511	29	4.3

Chapter 6

Conclusions and Future Work

is this list summarised in this section? Can we think of additional contributions?

- extension of Dwarfs
- development of AIWC analysis tool
- addition of new metrics
- feature space analysis of all dwarfs and potential clustering of application types
- detailed data gathering for dwarfs on range of hardware
- analysis to determine if feature space types map to particular hardware

The development of EOD results in a reliable benchmark suite with multiple problem sizes and high precision measurements. This allows for reproducible results to be generated quickly, and over a range of heterogeneous accelerator devices. For this thesis 15 devices were used – and tested on – to produce a full set of execution times and other performance metrics over all 12 applications and 42 kernels. The performance metrics allow direct evaluation of devices.

Examining the performance of the benchmark suite over a range of devices allows a direct comparison to be made between these devices on a per application basis. As a by-product of this comparison, the suitably of OpenCL is shown as a hardware agnostic language.

Separately, the Architecture Independent Workload Characterisation was developed and is capable of identifying the fundamental characteristics of programs free from any specific device. Architecture Independent Workload Characterisation (AIWC) tool is capable of analysing kernels in order to extract a set of predefined features or characteristics. The tool can be used in diversity analysis – which is essential when assembling benchmark suites and justifying the inclusion of an application. Furthermore, these metrics are used for creating the prediction model to evaluate the performance of OpenCL kernels on different hardware devices and settings. Such a model is then applied as a prognosis tool to predict the performance of an application for any given platform without additional instrumentation. This prediction adds information that can be incorporated into existing HPC schedulers and has no run-time overhead – codes are examined one time by the developer when instrumenting with AIWC and these, in turn, are embedded into the header of each kernel code to be evaluated by the scheduler at the time of scheduling.

6.1 EOD

We plan to complete analysis of the remaining benchmarks in the suite for multiple problem sizes. In addition to comparing performance between devices, we would also like to develop some notion of "ideal" performance for each combination of benchmark and device, which would guide efforts to improve performance portability. Additional architectures such as FPGA, DSP and Radeon Open Compute based APUs – which further breaks down the walls between the CPU and GPU – will be considered.

Each OpenCL kernel presented in this paper has been inspected using the Architecture Independent Workload Characterization (AIWC). Analysis using AIWC helps understand how the structure of kernels contributes to the varying runtime characteristics between devices that are presented in this work, and will be published in the future.

Certain configuration parameters for the benchmarks, e.g. local workgroup size, are amenable to auto-tuning. We plan to integrate auto-tuning into the benchmarking framework to provide confidence that the optimal parameters are used for each combination of code and accelerator.

The original goal of this research was to discover methods for choosing the best device for a particular computational task, for example to support scheduling decisions under time and/or energy constraints. Until now, we found the available OpenCL benchmark suites were not rich enough to adequately characterize performance across the diverse range of applications and computational devices of interest. Now that a flexible benchmark suite is in place and results can be generated quickly and reliably on a range of accelerators, we plan to use these benchmarks to evaluate scheduling approaches.

6.2 AIWC

We have presented the Architecture-Independent Workload Characterization tool (AIWC), which supports the collection of architecture-independent features of OpenCL application kernels. These features can be used to predict the most suitable device for a particular kernel, or to determine the limiting factors for performance on a particular device, allowing OpenCL developers to try alternative implementations of a program for the available accelerators – for instance, by reorganizing branches, eliminating intermediate variables et cetera. The additional architecture independent characteristics of a scientific workload will be beneficial to both accelerator designers and computer engineers responsible for ensuring a suitable accelerator diversity for scientific codes on supercomputer nodes.

Caparrós Cabezas and Stanley-Marbell [64] examine the Berkeley dwarf taxonomy by measuring instruction-level parallelism, thread parallelism, and data movement. They propose a sophisticated metric to assess ILP by examining the data dependency graph of the instruction stream. Similarly, Thread-Level-Parallelism was measured by analysing the block dependency graph. Whilst we propose alternative metrics to evaluate ILP and TLP – using the max, mean and standard deviation statistics of SIMD width and the total barriers hit and Instructions To Barrier metrics respectively – a quantitative evaluation of the dwarf taxonomy using these metrics is left as future work. We expect that the additional AIWC metrics will generate a comprehensive feature-space representation which will permit cluster analysis and comparison with the dwarf taxonomy.

6.3 Performance Prediction

A highly accurate model has been presented that is capable of predicting execution times of OpenCL kernels on specific devices based on the computational characteristics captured by the AIWC tool. A real-world scheduler could be developed based on the accuracy of the presented model.

We do not suppose that we have used a fully representative suite of kernels, however, we have shown that this approach can be used in the supercomputer accelerator scheduling setting, and the model can be extended/augmented with additional training kernels using the methodology presented in this paper.

We expect that a similar model could be constructed to predict energy or power consumption, where the response variable can be directly swapped for an energy consumption metric – such as joules – instead of execution time. However, we have not yet collected the energy measurements required to construct such a model. Finally, we show the predictions made are accurate enough to inform scheduling decisions.

References

1. M. Feldman, "TOP500 meanderings: Supercomputers take big green leap in 2017," *TOP500 Supercomputer Sites*, Sep. 2017.
2. T. Declerck *et al.*, "Cori - a system to support data-intensive computing," *Proceedings of the Cray User Group*, p. 8, 2016.
3. T. Morgan, "NVLink takes GPU acceleration to the next level," *The Next Platform*, May 2016.
4. T. Morgan, "The Power9 rollout begins with Summit and Sierra supercomputers," *The Next Platform*, Sep. 2017.
5. T. Morgan, "Inside Japan's future exascale ARM supercomputer," *The Next Platform*. Stackhouse Publishing Inc., Jun-2016.
6. M. Feldman, "Cray to deliver ARM-powered supercomputer to UK consortium," *TOP500 Supercomputer Sites*, Jan. 2017.
7. G. Mitra, E. Stotzer, A. Jayaraj, and A. P. Rendell, "Implementation and optimization of the OpenMP accelerator model for the TI Keystone II architecture," in *International workshop on openmp*, 2014, pp. 202–214.
8. K. Asanović *et al.*, "The landscape of parallel computing research: A view from Berkeley," EECS Department, University of California, Berkeley, UCB/EECS-2006-183, 2006.
9. P. Colella, "Defining software requirements for scientific computing, 2004," *DARPA HPCS presentation*.
10. D. H. Bailey *et al.*, "The NAS parallel benchmarks," *International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.
11. M. Martineau *et al.*, "Performance analysis and optimization of Clang's OpenMP 4.5 GPU support," in *International workshop on performance modeling, benchmarking and simulation of high performance computer systems (pmbs)*, 2016, pp. 54–64.
12. T. Barnes *et al.*, "Evaluating and optimizing the NERSC workload on Knights Landing," in *International workshop on performance modeling, benchmarking and simulation of high performance computer systems (pmbs)*, 2016, pp. 43–53.
13. M. G. Lopez, J. Young, J. S. Meredith, P. C. Roth, M. Horton, and J. S. Vetter, "Examining recent many-core architectures and programming models using SHOC," in *International workshop on performance modeling, benchmarking and simulation of high performance computer systems (pmbs)*, 2015, p. 3.
14. Y. Sun *et al.*, "Hetero-mark, a benchmark suite for cpu-gpu collaborative computing," in *IEEE international symposium on workload characterization (iiswc)*, 2016.

15. J. Gómez-Luna *et al.*, "Chai: Collaborative heterogeneous applications for integrated-architectures," in *IEEE international symposium on performance analysis of systems and software (ispass)*, 2017.

16. S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE international symposium on workload characterization (iiswc)*, 2009, pp. 44–54.

17. A. Danalis *et al.*, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units*, 2010, pp. 63–74.

18. W.-c. Feng, H. Lin, T. Scogland, and J. Zhang, "OpenCL and the 13 dwarfs: A work in progress," in *Proceedings of the 3rd acm/spec international conference on performance engineering*, 2012, pp. 291–294.

19. K. Krommydas, W.-C. Feng, C. D. Antonopoulos, and N. Bellas, "OpenDwarfs: Characterization of dwarf-based benchmarks on fixed and reconfigurable architectures," *Journal of Signal Processing Systems*, vol. 85, no. 3, pp. 373–392, 2016.

20. K. Spafford, J. Meredith, and J. Vetter, "Maestro: Data orchestration and tuning for OpenCL devices," *Euro-Par 2010-Parallel Processing*, pp. 275–286, 2010.

21. N. Chaimov, B. Norris, and A. Malony, "Toward multi-target autotuning for accelerators," in *IEEE international conference on parallel and distributed systems (ICPADS)*, 2014, pp. 534–541.

22. C. Nugteren and V. Codreanu, "CLTune: A generic auto-tuner for OpenCL kernels," in *IEEE international symposium on embedded multicore/many-core systems-on-chip (MCSoC)*, 2015, pp. 195–202.

23. J. Price and S. McIntosh-Smith, "Analyzing and improving performance portability of opencl applications via auto-tuning," in *Proceedings of the 5th international workshop on opencl*, 2017, p. 14.

24. J. Ansel *et al.*, "OpenTuner: An extensible framework for program autotuning," in *International conference on parallel architectures and compilation techniques*, 2014.

25. C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on code generation and optimization: Feedback-directed and runtime optimization*, 2004, p. 75.

26. S. Muralidharan, K. O'Brien, and C. Lalanne, "A semi-automated tool flow for roofline anaylsis of opencl kernels on accelerators," in *First international workshop on heterogeneous high-performance reconfigurable computing (h2rc'15)*, 2015.

27. T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," *IEEE micro*, vol. 23, no. 6, pp. 84–93, 2003.

28. T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ACM sigarch computer architecture news*, 2002, vol. 30, pp. 45–57.

29. K. Choi, R. Soma, and M. Pedram, "Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 24, no. 1, pp. 18–28, 2005.

30. V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus ipc: The end of the road for conventional microarchitectures," in *Proceedings of the 27th annual*

international symposium on computer architecture, 2000, pp. 248–259.

31. D. J. Brown and C. Reams, "Toward energy-efficient computing," *Communications of the ACM*, vol. 53, no. 3, pp. 50–58, 2010.

32. S. Albers and A. Antoniadis, "Race to idle: New algorithms for speed scaling with a sleep state," *ACM Trans. Algorithms*, vol. 10, no. 2, pp. 9:1–9:31, Feb. 2014.

33. M. B. Taylor, "Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse," in *Design automation conference (dac), 2012 49th acm/edac/ieee*, 2012, pp. 1131–1136.

34. G. Venkatesh *et al.*, "Conservation cores: Reducing the energy of mature computations," in *ACM sigarch computer architecture news*, 2010, vol. 38, pp. 205–218.

35. S. M. Blackburn *et al.*, "The dacapo benchmarks: Java benchmarking development and analysis," in *ACM sigplan notices*, 2006, vol. 41, pp. 169–190.

36. Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis," *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.

37. A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the spec cpu2006 benchmark suite," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 412–423, 2007.

38. A. I. Meajil, T. El-Ghazawi, and T. Sterling, "An architecture-independent workload characterization model for parallel computer architectures," in *Parallel algorithms/architecture synthesis, 1997. proceedings., second aizu international symposium*, 1997, pp. 143–150.

39. K. Hoste and L. Eeckhout, "Microarchitecture-independent workload characterization," *IEEE Micro*, vol. 27, no. 3, 2007.

40. K. Ganesan, L. John, V. Salapura, and J. Sexton, "A performance counter based workload characterization on blue gene/p," in *Parallel processing, 2008. icpp'08. 37th international conference on*, 2008, pp. 330–337.

41. T. K. Prakash and L. Peng, "Performance characterization of spec cpu2006 benchmarks on intel core 2 duo processor," *ISAST Trans. Comput. Softw. Eng*, vol. 2, no. 1, pp. 36–41, 2008.

42. C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, 2005, vol. 40, pp. 190–200.

43. Y. S. Shao and D. Brooks, "ISA-independent workload characterization and its implications for specialized architectures," in *Performance analysis of systems and software (ispass), 2013 ieee international symposium on*, 2013, pp. 245–255.

44. T. Yokota, K. Ootsu, and T. Baba, "Introducing entropies for representing program behavior and branch predictor performance," in *Proceedings of the 2007 workshop on experimental computer science*, 2007, p. 17.

45. S. De Pestel, S. Eyerman, and L. Eeckhout, "Linear branch entropy: Characterizing and optimizing branch behavior in a micro-architecture independent way," *IEEE Trans. Comput.*, vol. 66, no. 3, pp. 458–472, Mar. 2017.

46. J. Price and S. McIntosh-Smith, "Oclgrind: An extensible opencl device simulator," in *Proceedings of the 3rd international workshop on opencl*, 2015, p. 12.

47. J. Kessenich, "A Khronos-Defined Intermediate Language for Native Representation of Graphical Shaders and Compute Kernels." 2015.

48. R. F. Lyerly, "Automatic scheduling of compute kernels across heterogeneous architectures," 2014.

49. K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere, "Performance prediction based on inherent program similarity," in *Parallel architectures and compilation techniques (pact), 2006 international conference on*, 2006, pp. 114–122.

50. T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2015, p. 73.

51. P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *Proceedings of the department of defense hpcmp users group conference*, 1999, vol. 710.

52. V. Marjanović, J. Gracia, and C. W. Glass, "HPC benchmarking: Problem size matters," in *International workshop on performance modeling, benchmarking and simulation of high performance computer systems (pmbs)*, 2016, pp. 1–10.

53. E. Bainville, "OpenCL fast Fourier transform." 2010.

54. "OpenDwarfs (base version)." https://github.com/vtsynergy/OpenDwarfs/commit/31c099aff5343e93ba9e8c3cd42bee5ec536aa93, 26-Feb-2017.

55. T. Madej *et al.*, "MMDB and VAST+: Tracking structural similarities between macromolecular complexes," *Nucleic Acids Research*, vol. 42, no. D1, pp. D297–D303, 2013.

56. L. Yu, S.-J. Lee, and V. C. Yee, "Crystal structures of polymorphic prion protein β1 peptides reveal variable steric zipper conformations," *Biochemistry*, vol. 54, no. 23, pp. 3640–3648, 2015.

57. M. Shiroishi, M. Kajikawa, K. Kuroki, T. Ose, D. Kohda, and K. Maenaka, "Crystal structure of the human monocyte-activating receptor, 'Group 2' leukocyte Ig-like receptor A5 (LILRA5/LIR9/ILT11)," *Journal of Biological Chemistry*, vol. 281, no. 28, pp. 19536–19544, 2006.

58. C. A. Davey, D. F. Sargent, K. Luger, A. W. Maeder, and T. J. Richmond, "Solvent mediated interactions in the structure of the nucleosome core particle at 1.9Å resolution," *Journal of Molecular Biology*, vol. 319, no. 5, pp. 1097–1113, 2002.

59. T. J. Dolinsky, J. E. Nielsen, J. A. McCammon, and N. A. Baker, "PDB2PQR: An automated pipeline for the setup of Poisson–Boltzmann electrostatics calculations," *Nucleic Acids Research*, vol. 32, no. suppl_2, pp. W665–W667, 2004.

60. M. F. Sanner, A. J. Olson, and J.-C. Spehner, "Reduced surface: An efficient way to compute molecular surfaces," *Biopolymers*, vol. 38, no. 3, pp. 305–320, 1996.

61. B. Johnston, "OpenDwarfs," *GitHub repository*. https://github.com/BeauJoh/OpenDwarfs; GitHub, 2017.

62. A. S. Joshi, "A performance focused, development friendly and model aided parallelization strategy for scientific applications," Master's thesis, Clemson University, 2016.

63. S. De Pestel, S. Eyerman, and L. Eeckhout, "Linear branch entropy: Characterizing and optimizing branch behavior in a micro-architecture independent way," *IEEE Transactions on Computers*, vol. 66, no. 3, pp. 458–472, Mar. 2017.

64. V. Caparrós Cabezas and P. Stanley-Marbell, "Parallelism and data movement characterization of contemporary application classes," in *Proceedings of the twenty-third annual ACM symposium on parallelism in algorithms and architectures*, 2011, pp. 95–104.

65. B. Johnston *et al.*, "BeauJoh/Oclgrind: Adding AIWC – An Architecture Independent Workload Characterisation Plugin." https://doi.org/10.5281/zenodo.1134175, Dec-2017.

66. B. Johnston and J. Milthorpe, "Dwarfs on accelerators: Extending OpenCL benchmarking for heterogeneous computing architectures," *unpublished*, 2017.

67. S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with gpus and fpgas," in *Application specific processors, 2008. sasp 2008. symposium on*, 2008, pp. 101–107.

68. M. B. Yildirim and G. Mouzon, "Single-machine sustainable production planning to minimize total energy consumption and total completion time using a multiple objective genetic algorithm," *IEEE transactions on engineering management*, vol. 59, no. 4, pp. 585–597, 2012.

69. B. Johnston and J. Milthorpe, "AIWC: OpenCL based Architecture Independent Workload Characterisation," *ArXiv e-prints*, May 2018.

70. C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, "Data-aware task scheduling on multi-accelerator based platforms," in *IEEE international conference on parallel and distributed systems (ICPADS)*, 2010, pp. 291–298.

71. H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Task scheduling algorithms for heterogeneous processors," in *Heterogeneous computing workshop (HCW)*, 1999, pp. 3–14.

72. R. Bajaj and D. P. Agrawal, "Improving scheduling of tasks in a heterogeneous environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 2, pp. 107–118, 2004.

73. T. Xiaoyong, K. Li, Z. Zeng, and B. Veeravalli, "A novel security-driven scheduling algorithm for precedence-constrained tasks in heterogeneous distributed systems," *IEEE Transactions on Computers*, vol. 60, no. 7, pp. 1017–1029, 2011.

74. O. Sinnen and L. Sousa, "List scheduling: Extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures," *Parallel Computing*, vol. 30, no. 1, pp. 81–101, 2004.

75. L. T. Yang, X. Ma, and F. Mueller, "Cross-platform performance prediction of parallel applications using partial execution," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 2005, p. 40.

76. L. Carrington, A. Snavely, and N. Wolter, "A performance prediction framework for scientific applications," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 336–346, 2006.

77. M. Wright and A. Ziegler, "ranger: A Fast Implementation of Random Forests for High Dimensional Data in C++ and R," *Journal of Statistical Software, Articles*, vol. 77, no. 1, pp. 1–17, 2017.

78. L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

79. M. Ließ, M. Hitziger, and B. Huwe, "The sloping mire soil-landscape of southern Ecuador: Influence of predictor resolution and model tuning on random forest predictions," *Applied and environmental soil science*, vol. 2014, 2014.

80. K. Husmann, A. Lange, and E. Spiegel, "The R package optimization: Flexible global optimization with simulated-annealing," 2017.