Characterizing and Predicting Scientific Workloads for Heterogeneous Computing Systems

Beau Johnston

January 2019

A thesis submitted for the degree of Doctor of Philosophy of 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.

Finally, I would also like to thank Oracle and the ANU VC Travel Grants Office for providing additional funding, this was invaluable for conference attendance.

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 are fixed regardless of which accelerator is used for computation; for instance, a graph traversal program always exhibits the same high-branch and low-computation properties of graph traversal regardless of what device is used to execute it. To support efficient scheduling on HPC systems it is necessary to make accurate performance predictions for workloads on varied compute devices, which is challenging due to diverse computation, communication and memory access characteristics which result in varying performance between devices. This work presents a methodology to use device-independent characteristics of scientific codes to select the optimal accelerator device with regard to execution time or energy expenditure. On HPC systems a single node may feature a GPU, CPU, and an FPGA or MIC.

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.

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 OpenDwarfs OpenCL benchmark suite. Our extensions improve portability and robustness of applications, correctness of results and choice of problem size, and increase diversity through coverage of additional application patterns. This work manifests in performance measurements on a set 15 devices and over 11 applications.

We next present the Architecture Independent Workload Characterization (AIWC) tool which characterizes OpenCL kernels according to a set of architecture-independent features. Features are measured by counting target characteristics which are collected during program execution in a simulator. They are presented as 42 metrics that indicate performance bottlenecks in four categories: parallelism – how well an algorithm scales in response to core count; compute – the diversity of instructions; memory – working memory footprint and entropy measurements which correspond to caching characteristics; and control – 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 the differences in kernels to show the phase-transitional properties of application codes. We also discuss 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. We 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 heterogeneous.

Contents

Declaration

Acknowledgements

Abstract

Abbreviations **1**

1 Introduction **2**
1.1 Context . . . 3
1.2 Restatement of the problem . . . 4
1.3 Thesis Contributions . . . 5
1.4 Thesis Structure . . . 5

2 Background Information and Related Work **7**
2.1 The Dwarf Taxonomy . . . 7
2.2 Accelerator Architectures in HPC . . . 8
2.3 The Open Compute Language (OpenCL) . . . 14
2.4 Benchmark Suites . . . 16
2.4.1 Rodinia . . . 17
2.4.2 OpenDwarfs . . . 18
2.4.3 SHOC . . . 19
2.5 Hardware Performance and Scaling . . . 19
2.6 OpenCL Performance . . . 21
2.6.1 Autotuning . . . 21
2.6.2 Phase-Shifting . . . 22
2.6.3 Measurements . . . 23
2.7 Offline Ahead-of-Time Analysis . . . 23
2.8 Program Diversity Analysis and Characterization . . . 24
2.8.1 Microarchitecture-Independent Workload Characterization . . . 24
2.8.2 Architecture Independent Workload Characterization . . . 25
2.8.3 Workload Characterization for Benchmark Diversity Analysis . . . 26
2.9 Scheduling and Performance Prediction for Heterogeneous Architectures . . . 27
2.10 Predictions and Modelling . . . 27

3 Extending the OpenDwarfs Benchmark Suite **31**

3.1 Enhancing the OpenDwarfs Benchmark Suite . . . 31
3.2 Experimental Setup . . . 33
3.2.1 Hardware . . . 33
3.2.2 Software . . . 33
3.2.3 Measurements . . . 34
3.2.4 Setting Sizes . . . 34
3.2.4.1 kmeans . . . 35
3.2.4.2 lud, fft, srad, crc, nw . . . 36
3.2.4.3 dwt . . . 37
3.2.4.4 gem, swat, nqueens, hmm . . . 37
3.2.5 Summary of Benchmark Settings . . . 38
3.3 Results . . . 38
3.3.1 Time . . . 39
3.3.2 Energy . . . 47
3.4 Discussion . . . 48

4 AIWC: OpenCL based Architecture Independent Workload Characterization 50
4.1 Metrics . . . 51
4.2 Methodology – Workload Characterization by tooling Oclgrind . . . 53
4.3 Implementation . . . 53
4.4 Demonstration . . . 56
4.5 Detailed Analysis of LU Decomposition Benchmark . . . 57
4.6 Use Case: AIWC analysis on bioinformatics . . . 61
4.7 Summary . . . 64

5 Making Performance Predictions for Scheduling 65
5.1 Methodology . . . 65
5.1.1 Experimental Setup . . . 66
5.1.2 Constructing the Performance Model . . . 66
5.1.3 Choosing Model Parameters . . . 69
5.1.4 Performance Improvement with Increased Training Data . . . 69
5.2 Evaluation . . . 71
5.2.1 Making Predictions . . . 73
5.2.2 The benefits of this approach . . . 73
5.3 Discussion . . . 75

6 Conclusions and Future Directions 77
6.1 Extended OpenDwarfs – EOD . . . 78
6.2 AIWC . . . 80
6.3 Performance Prediction . . . 81
6.4 Future Directions . . . 82
6.4.1 Finding holes in benchmarks: Evaluating the coverage and corresponding performance predictions for conventional vs synthetic benchmarking 82

6.4.2 AIWC for the Masses: Towards language-agnostic architecture-independent workload characterization 82
6.4.3 Examining the Characteristics of Scientific Codes in Supercomputing with AIWC 83
6.4.4 Guiding Device Specific Optimization using Architecture-Independent Metrics 83
6.4.5 Faster FPGA development with AIWC and the Predictive Model 84

References **85**

List of Figures

2.1 The percentage of accelerators in use and the contributions of cores found on systems with accelerators in the Top500 supercomputers over time. 12
2.2 Power efficiency (GFlops/Watt) of using accelerators in the Top500 supercomputers over time. 14

3.1 Kernel execution times for the **crc** benchmark on different hardware platforms . 40
3.2 Kernel execution times for the **tiny** and **small** problem sizes of the `kmeans`, `lud`, `csr` and `dwt` benchmarks on different hardware platforms 41
3.3 Kernel execution times for the **medium** and **large** problem sizes of the `kmeans`, `lud`, `csr` and `dwt` benchmarks on different hardware platforms 42
3.4 Kernel execution times for the **tiny** and **small** problem sizes of the `fft`, `srad` and `nw` benchmarks on different hardware platforms 43
3.5 Kernel execution times for the **medium** and **large** problem sizes of the `fft`, `srad` and `nw` benchmarks on different hardware platforms 44
3.6 Single problem sized benchmarks of kernel execution times on different hardware platforms 45
3.7 Execution energy required to perform EOD benchmarks, presented on a linear (a) and logarithmic scale (b) from left to right respectively, on the (**large** problem size) on the Intel i7-6700K and Nvidia GTX1080 48

4.1 Selected AIWC metrics from each category over all kernels and 4 problem sizes. 58
4.2 A) and B) show the AIWC features of the `diagonal` and `internal` kernels of the LUD application over all problem sizes. 59
4.3 A) shows the AIWC features of the `perimeter` kernel of the LUD application over all problem sizes. B) shows the corresponding Local Memory Address Entropy for the `perimeter` kernel over the tiny problem size. 60
4.4 Architecture-Independent Workload Characterization features for selected bioinformatics benchmarks 62
4.5 Architecture-Independent Workload Characterization features for the `hmm` bioinformatics benchmark 63

5.1 Full coverage of min.node.size with fixed tuning parameters: num.trees = 300 and mtry = 30. 67
5.2 Full coverage of num.trees and mtry tuning parameters with min.node.size fixed at 9. 68

5.3 Prediction error across all benchmarks for models trained with varying numbers of kernels. 72
5.4 Predicted vs. measured execution time for all kernels 72
5.5 Error in predicted execution time for each kernel invocation over four problem sizes 74
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. 75

List of Tables

2.1 The Berkeley Dwarfs and their limiting factors. 8

3.1 Hardware 33
3.2 List of Extended OpenDwarfs Applications and their respective dwarfs 35
3.3 OpenDwarfs workload scale parameters Φ 38
3.4 Program Arguments 39

4.1 Metrics collected by the **AIWC** tool ordered by type. 51

5.1 Optimal tuning parameters from the same starting location for all models omitting each individual kernel. 70

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
KNL **K**nights **L**anding
SMaC **S**calable **M**any **C**ore
VPU **V**ector **P**rocessing **U**nit
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 a trend towards 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 – where NVIDIA Graphical Processing Unit (GPU) use NVLink, whereas other accelerator devices such as Altera Field-Programmable Gate Array (FPGA), Digital Signal Processors (DSPs), Intel Many-Integrated-Core (MIC) devices, and both Intel and AMD Central Processing Unit (CPU) and AMD GPU devices can utilise the CAPI interconnect. 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 most suitable accelerator is a new problem. The cost of exascale computing and their corresponding energy efficiency will be prohibitive without significant improvements in effectively using accelerators on current and future supercomputers.

This thesis will argue that 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 or on what platform it is run. Moreover, these metrics can be used to accurately predict the execution time on each accelerator in a heterogeneous system.

This thesis also presents a methodology 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. We believe this research will be of benefit to the scheduling of codes to the most

appropriate device to achieve better performance and utilization on 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. The developer is responsible for allocating and transferring memory between the host and device. This requires programs to be structured with computationally intensive regions of code – known as kernels – to be identified and written 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 composition of all kernels forms a full accelerator agnostic implementation for a larger scientific code.

A benefit of the fixed/static nature of OpenCL kernels is that the collection of the characteristics is also constant. Instrumentation of the execution of a kernel measures computation, memory, branching and parallelism metrics – these form the characteristics of a program and 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 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 High Performance Computing (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 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 Restatement of the problem

The future of supercomputing comprises several heterogeneous devices at the node level. 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. This facilitates the usage of a mix of accelerators given the low penalty to move data between them. 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. The types of devices considered in this thesis and the benchmark suites examined can be broadly classified according to the Dwarf Taxonomy, as such, this Chapter begins with an introduction to the Dwarf Taxonomy. Next, we define accelerators and a provide brief survey regarding their use in supercomputing. The hardware agnostic programming framework OpenCL is presented. Finally, this section culminates in a discussion of benchmark suites, applications and where they are incorporated into the dwarf taxonomy.

2.1 The Dwarf Taxonomy

Phil Colella [1] identified seven motifs of numerical methods which he thought would be important for the next decade. Based on this style of analysis, The Berkeley Dwarf Taxonomy [2] was conceived to present the motifs commonplace in HPC. Initially performed by Asanovic et al. [3], the Dwarf Taxonomy claims that many applications in parallel computing share patterns of communication and computation. Applications with similar patterns are defined as being represented by a single dwarf. Dwarfs are removed from specific implementations and optimisations. Asanovic et al. [3] present a total of 13 dwarfs, 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 13 dwarfs the authors indicate the performance limit – in other words, whether the dwarf is compute bound, memory latency limited or memory bandwidth limited. The dwarfs and their limiting factors are presented in Table 2.1. Note, the **?** symbol indicates the unknown performance limit at the time of publication – none of these have been resolved since.

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: The Berkeley Dwarfs and their limiting factors.

Implementations of applications that are represented by the Dwarf Taxonomy are discussed in the benchmark evaluations presented in Section 2.4. Having familiarity with the division of applications and which of the dwarfs they lie within assists in motivating the variety of accelerators used in HPC and is discussed in the next section.

2.2 Accelerator Architectures in HPC

Accelerators, in this setting, refer to any form of hardware specialized to a particular pattern of computation; T specialized hardware may accelerate a given application code according to that codes characteristics. From The Dwarf Taxonomy previously presente is envisaged that all applications represented by a dwarf are are better suited to specific types of accelera er. Accelerators commonly include GPU, FPGA, DSP, ASIC and MIC devices. We define accelerators to include all compute devices, including CPUs, since their architecture is well suited to accelerate the computation of specific dwarfs; ionally, the heterogeneous configuration of side cores on modern CPUs presents a similar set of work-scheduling problems, that occur on other accelerators, primarily, these cores need to be given the appropriate work to ensure good system performance. The remainder of this section will present and describe each type of accelerator, its history and its uses.

Central Processing Units (CPU) have additional circuitry for branch control logic, and generally operate at a high frequency, ensuring this architecture is highly suited to sequential tasks or workloads with many divergent logical comparisons – corresponding to the finite-state machine, combinational logic, dynamic programming and backtrack branch and bound dwarfs of the Berkeley Dwarf Taxonomy. Additionally, CPUs are increasingly configured as two separate CPUs but provided on the same System-on-a-Chip (SoC) and strengthens the argument of defining accelerators to include CPUs. Comprised of two separate micro-architectures, a high-performance CPU – faster base clock speed with additional hardware for

branching – to support the irregular control and access behaviour of typical workloads; and a smaller CPU – commonly with a lower base-clock frequency but with many more cores and support for longer vector instructions – for the highly parallel workloads/tasks common in scientific computing.

The SW26010 and ARM big.LITTLE type processors are current examples of how CPUs are treated as accelerators to achieve performance on modern supercomputers. The SW26010 CPU deployed in the Sunway TaihuLight supercomputer, contains high-performance cores known as Management Processing Elements (MPE), and low-powered Computer Processing Elements (CPE). The CPE are arranged in an 8x8 mesh of cores, supports only user mode, and each core sports a small 16 KB L1 instruction cache and 64 KB scratch memory. Both MPE and CPE are of 64-bit Reduced Instruction-Set Computers (RISC) and support 256-bit vector instructions. This configuration shows the intent of the architecture, that the smaller CPEs need be used effectively to achieve good performance [4]. In other words, the host or primary core contributes only a small part of the maximum theoretical FLOPs on modern heterogeneous supercomputers.

ARM processors with big.LITTLE and dynamIQ configurations have been proposed to meet the power needs of exascale supercomputers [5]–[8]. big.LITTLE is an heterogeneous configuration of CPU cores on the same die and memory regions. The big cores have higher clock frequencies and are generally more powerful than the LITTLE cores, which creates a multi-core processor that suites a dynamic workload more than clock scaling. Tasks can migrate to the most appropriate core, and unused cores can be powered down. CPUs can be considered accelerators since many heterogenous configurations including the SW26010 and big.LITTLE devices, have side cores, which, with careful work scheduling, can accelerate workloads and achieve high FLOPs.

Graphics Processing Units (GPU) were originally designed to accelerate manipulating computer graphics and image processing, which is achieved by having circuit designs to apply the same operation to many values at once. This highly parallel structure makes them suitable for applications which involve processing large blocks of data. Many of the dwarfs of scientific computation are directly suited to GPUs for acceleration, including dense [9][10] and sparse linear algebra and N-Body methods. There has been an active effort to migrate applications from less suited dwarfs, such as spectral methods [11], structured grids [12] and graph traversal [13] for GPU acceleration. Efforts are primarily algorithmic, such as reordering of operations and the padding of shared memory, and have been used with various success on GPU architectures [14]. Avoiding bank-conflicts and non-coalesced memory accesses thus increasing the use of private and shared memory are critical to performance of these dwarfs on GPUs. They are the most common type of accelerator in supercomputer systems. The recent adoption of the NVIDIA Volta GV100 GPU as the primary accelerator into the Summit and Sierra supercomputers [15] is attributed to its performance [16] and energy efficiency [17] on workloads fundamental to scientific computing.

Many Integrated Core (MIC) architectures are an Intel Corporation specific accelerator. Xeon

Phi formerly known as Knights Landing (KNL) is the last series of the MIC accelerators, and was discontinued in July 2018. It is significantly different to a GPU, it relies heavily on Single Instruction Multiple Data (SIMD) parallelism as opposed to the Single Instruction Multiple Thread (SIMT) needed for GPUs. It has many low frequency in-order cores sharing the same bus and each core is based on conventional CPU x86 architectures. There are 72 cores with a layout based on a 2D mesh topology – comprised of 38 tiles, each tile features two CPU cores, and each core contains two Vector Processing Units (VPU). [18]; four cores are reserved for host-side system control and orchestration of work to the other cores. A 2D cache-coherent interconnect between tiles is included provide high-bandwidth pathways to match the memory access patterns on the core and mesh layout – cores on the same tile have a shared 1 MB L2 cache. Each core supports a 512-bit vector instruction to utilize a large amount of SIMD parallelism. Dwarfs such as Dense and Sparse Linear Algebra are high-intensity and throughput-oriented workloads suited to the Xeon Phi accelerator [19]. The Xeon Phi is the primary accelerator in the Trinity [20] and Cori [21] supercomputer systems – currently in the top 10 of the Top500.

Field-Programmable Gate Arrays (FPGA) are accelerators which allow the physical hardware to be reconfigured for any specific task. They are composed of a high number of logic-gates organised into logic-blocks with fast I/O rates and bi-directional communication between them. FPGAs are suitable for workloads which require simple operations on very large amounts of data with a fast I/O transfer. Specifically, they are well suited to accelerating applications from spectral methods dwarf, specifically stream/filter processing on temporal data, and the combinational logic dwarf, which exploit bit-level parallelism to achieve high throughput. An example of the combinational logic dwarf is in the computing of checksums which is commonly required for network processing and ensuring data archive integrity. The configurablity of these devices may make them well suited to the characteristics of many dwarfs, however, the compilation or configuring the hardware for an application takes many orders of magnitude longer than any of the other examined accelerator architectures. Akram et al.[22] present a prototype FPGA supercomputer comprised of 5 compute nodes, each with an ARM CPU and Xilinx 7 FPGA. The benchmark application was of a Finite Impulse Response Filter – an application typical of the Spectral Methods dwarf – and presents 8.5× performance improvement over direct computation on the ARM CPU alone. Unfortunately, energy efficiency or a comparison between GPU accelerators is not presented. Fujita et al. [23] present a comparison between a P100 GPU and BittWare A10PL4 FPGA over a Authentic Radiation Transfer scientific application and show that the performance is comparable, however an energy efficiency comparison between these two accelerators is not presented. Given the increasing need for high-throughput devices from applications in combinational logic and other dwarfs, FPGA devices are likely to be included in future HPC systems.

An integrated circuit designed solely for a specific task is known as an Application-Specific Integrated Circuit (ASIC). In this regard, they are akin to FPGAs without the ability to be reconfigured. They have been used to accelerate the hashing workloads from the combinational logic dwarf for bitcoin mining tasks. Google's Tensor Processing Units (TPU) are another

example of ASICs, and support the TensorFlow [24] framework. TPUs perform convolutions for Machine Learning applications, which require many large matrix operations and are encapsulated by both the dense and sparse linear algebra dwarfs [25].

Digital Signal Processors (DSP) have their origins in audio processing – specifically in telephone exchanges and more recently in mobile phones – where streams of data are constantly arriving and an identical operation must be applied to each element. Audio compression and temporal filtering are examples of the Spectral Methods dwarf and are best suited to the DSP architecture. DSP cores operate on a separate clock to the host CPU and have circular memory buffers which allow a host device – using shared memory – to provide and remove data for processing without ever interrupting the DSP. Mitra et al. [26] evaluate a prototype nCore Brown-Dwarf system where each node contains an ARM Cortex-A15 host CPU, a single Texas Instruments Keystone II DSP and two Keystone I DSPs. They compare the performance and energy-efficiency of dense matrix multiplication and a real-world scientific code for biostructure based drug design against conventional x86 based HPC systems with attached accelerators. They show a Brown-Dwarf node is competitive with contemporary systems for memory-bound computations and show the C66x multi-core DSP is capable of running floating-point intensive HPC application codes.

Research around the suitability of ARM CPUs for HPC systems is highly active, with comparisons against the conventional Intel and AMD CPUs being made and the potential strengths of ARM systems when striving for energy efficiency [27][28][29]. Isambard[30] and Astra[31] systems use the Cavium ThunderX2 CPU accelerator, where each ThunderX2 accelerator consists of 32 high-end ARM cores operating at 2.1 GHz [32]. Separately, Fujitsu propose using ARMv8-A cores for the Post-K supercomputer [33]. In a similar layout to the ThunderX2 the FX100 is a Scalable Many Core (SMaC) with the memory model – Core Memory Group – and core configuration – Compute Engine – also in a grid-layout.

Currently, only 25 of the Top500 systems are based on ARM technologies, but these experimental systems may indicate the way forward for exascale supercomputing. The most compelling reason for this transition to ARM is improved energy efficiency. ARM processors were originally targeted for embedded and mobile computing markets, where energy efficiency is a major constraint, and may explain that while time-to-completion time are higher on these systems verses conventional x86 architectures, the energy usage is much lower. 34 [illegible]ate ARM processors against conventional x86 processors on real-time cortical simulations and consider the energy and interconnect scaling over distributed systems. They show joules per synaptic event on a network of ARM based Jetson systems use 3× less energy than the Intel solution, whilst being 5× slower. The benchmark identifies an interesting bottleneck on current HPC x86 based systems: as the problem sizes grow larger more nodes and a larger network is required, thus, it is the lack of a low-latency, energy-efficient interconnect that is the primary concern. However, since ARM based HPC systems can be populated more densely and offer a lower baseline energy profile, it is an architecture better suited to bio-inspired artificial intelligence applications and scientific investigations of the cognitive functions of the brain.

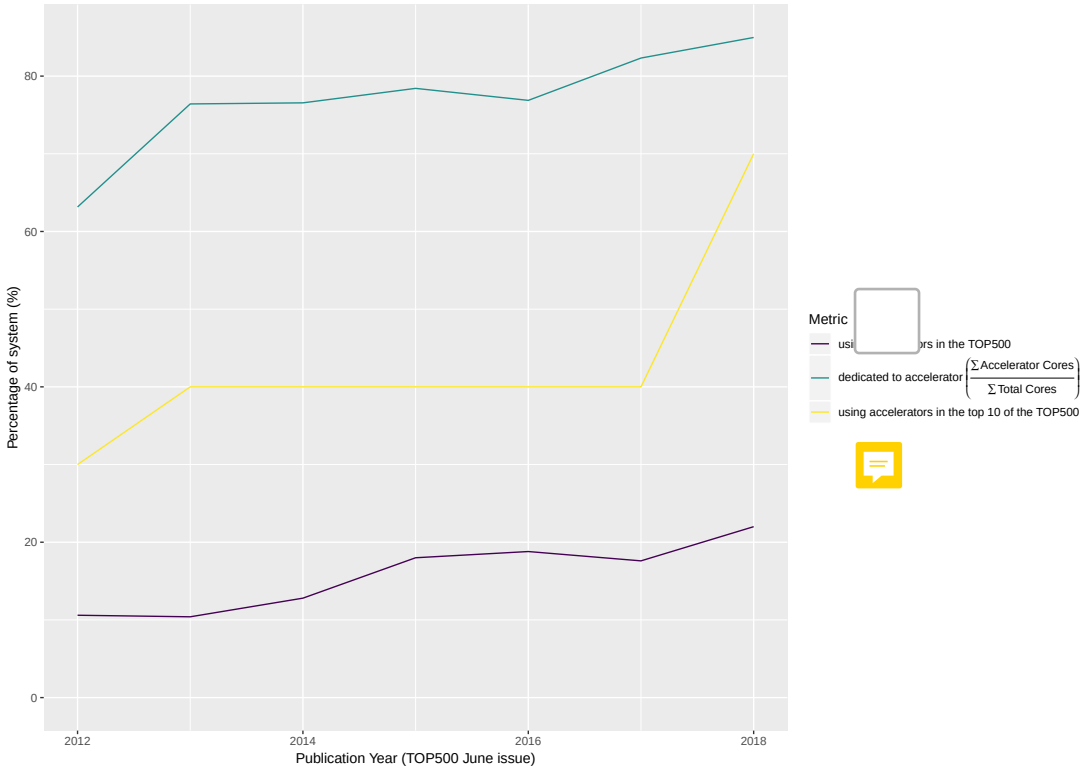


Figure 2.1: The percentage of accelerators in use and the contributions of cores found on systems with accelerators in the Top500 supercomputers over time.

A major motivation for the increasing use of heterogeneous architectures is to reduce energy use; indeed, without significant improvements in energy efficiency, the cost of exascale computing will be prohibitive [35]. The diversity of accelerators in this space is best shown in a survey of accelerator usage and energy consumption in the worlds leading supercomputers. The complete results from the TOP500 and Green500 lists [36] were examined, over consecutive years from 2012 to 2018. 2012 was selected as the starting year since it was the first occurrence in the TOP500 spreadsheets to provide both accelerator name and accelerator core count information. Each dataset was taken from the June editions of the yearly listings.

Figure 2.1 shows steady increase in the use of accelerators in supercomputers depicted as the purple line. This is presented as a percentage of the number of systems using accelerators in the TOP500 divided by 500 – the total number of systems listed in the TOP500 every year. In 2012 and 2013 11% systems in the TOP500 used accelerators, this increased by roughly 2% per year. As of 2018 22% of the TOP500 use accelerators. Note, from 2016 the Sunway TaihuLight system was introduced and is in the top 10, however due to the reliance on the CPE side-core to achieve the FLOPs for its rank, the data was adjusted to be listed as containing an accelerator [4]. Also shown in Figure 2.1 is the average percentage of cores in the TOP500 every year dedicated to accelerators, presented as the teal line. This measure indicates how much of the TOP500 compute is dependent on the accelerator – for systems that contain accelerators.

The rationale for this metric is that systems in the TOP500 which use accelerators are not only accelerator based systems – they contain conventional x86 CPU architectures as a host-side device which mirror the non-accelerator HPC systems, the teal line indicates what percentage of compute resources are attributed to the accelerator. Unsurprisingly, every year from 2012 to 2018, we see that a greater contribution of system resources – cores – are dedicated for accelerator devices and fewer resources for systems with accelerators are provided for the host. In 2012 63% of supercomputer cores were located on the accelerator, by 2013 it jumped to 76%, this increased on average by 1.5% per year to 85% of compute cores being accelerator based in 2018.

A closer inspection of the top 10 of the TOP500 systems over the same time period is presented as the yellow line in Figure 2.1 and shows a greater dependence on accelerators and a corresponding increase in heterogeneity. In 2012 3 [illegible] the top 10 supercomputers used accelerators to secure a position. From 2013 to 2017 the use of accelerators in these systems was consistently at 40% however in 2018 it jumped to 70%. Since the use of accelerators in the top 10 is much higher than in the rest of the TOP500 (purple line), we can conclude that the use of accelerators gives an edge to the ranking of these systems. The general trend of increased use of accelerators throughout all of the TOP500 continues to increase and reinforces the importance of accelerators in this space.

Another benefit from the increasing dependence on a heterogeneous mix of accelerator devices is improved energy efficiency on these systems.

Figure 2.2 presents a comparison of the energy efficiency – the rate of computation that can be delivered by a computer for every watt of power consumed – in terms of billions of floating point operations per second per watt, of supercomputers which use accelerators, presented as the purple line, and systems which do not use accelerators – shown as the yellow line. Generally, we see that the mean energy efficiency of all systems improves over time. However, it is apparent that the use of accelerators in supercomputers has always offered better energy efficiency than using conventional x86 architectures as the primary means of computation. Systems without accelerators had a mean energy efficiency of 500 MFlops/Watt in 2012 and have increased on average by 200 MFlops/Watt every year, in 2018 these systems achieved 2 GFlops/Watt. These results are modest when compared to the gains in efficiency when using accelerators in supercomputing systems. In contrast, in 2012 the mean energy efficiency of supercomputers with accelerators was 900 MFlops/Watt and reached 5.9 GFlops/Watt in 2018, growing non-linearly by 750 MFlops/Watt per year. The efficiency of systems using accelerators is improving faster than supercomputers which rely on homogeneous CPU architectures.

Similar efficiencies ha[illegible]so been shown in the most energy efficient supercomputing list – the Green500. 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 [36]. 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,

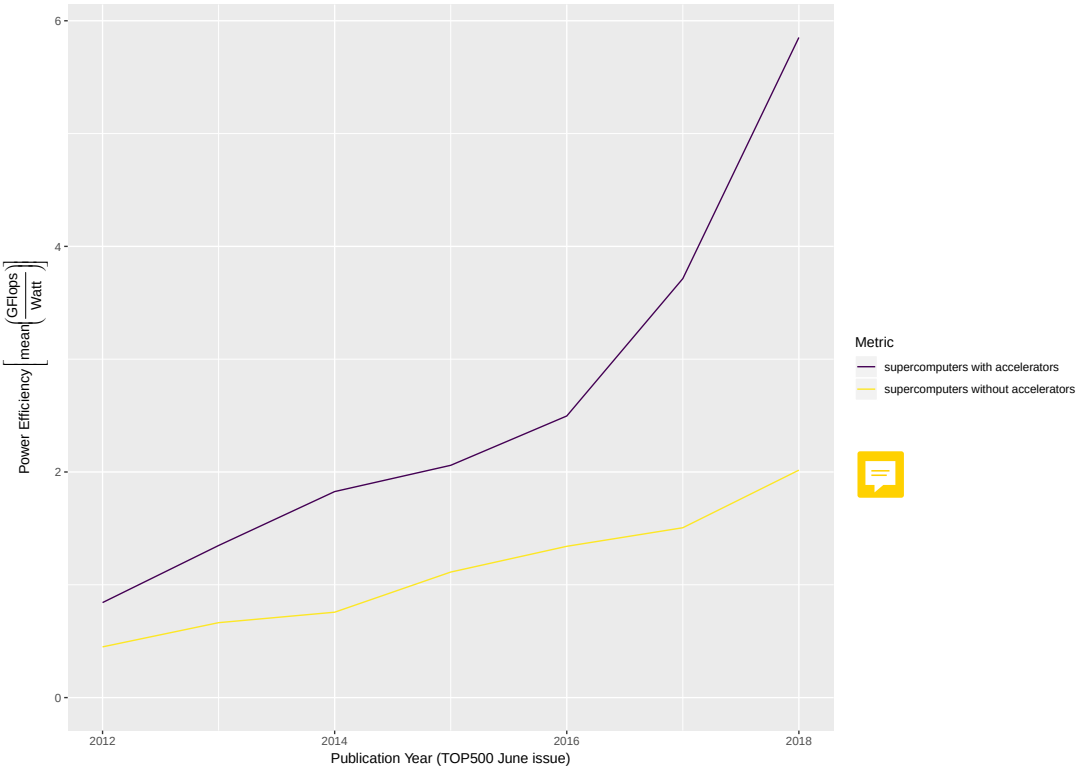


Figure 2.2: Power efficiency (GFlops/Watt) of using accelerators in the Top500 supercomputers over time.

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 [37]. The Summit supercomputer at Oak Ridge National Laboratory is based on the IBM Power9 CPU, which includes both NVLINK [38], a high bandwidth interconnect between Nvidia GPUs; and CAPI, an interconnect to support FPGAs and other accelerators [39]. Promising next-generation architectures include Fujitsu's Post-K [33], and Cray's CS-400, which forms the platform for the Isambard supercomputer [30]. Both architectures use ARM cores alongside other conventional accelerators, with several Intel Xeon Phi and Nvidia P100 GPUs per node. The Tianhe-2A uses a Matrix2000 DSP accelerator [40]; so will the future system, the Tianhe-3, which is due to be operational in 2020 and will use ARM CPU cores as the primary compute hardware [41].

2.3 The Open Compute Language (OpenCL)

OpenCL 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., [42] 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 consist of a host and a device side, which cooperate to perform a computation using a standard sequence of steps. 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 enqueued 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 thread's 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 surrounding how work should be partitioned – such as how many threads to use and how many threads are in a workgroup – can have a large impact on performance. One primary reason is that different accelerators benefit from different levels of parallelism, for instance, GPU devices usually need a high degree of arithmetic intensive parallelism to offset the (relatively) narrow I/O pipeline, while CPUs are 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 reduce performance however, recent work shows these parameters can be automatically optimised for any accelerator architecture as will be discussed in Section 2.6.1.

The OpenCL programming framework is well-suited to heterogeneous computing environments, as a single OpenCL code may be executed on multiple different device types. When

combined with autotuning, an OpenCL code may exhibit good performance across varied devices. [43] OpenCL has been used for DSP programming since 2012 [47]. Furthermore, Mitra et al. [26] propose a hybrid programming environment that combines OpenMP, OpenCL and MPI to utilize a nCore Brown-Dwarf system where each node contains an ARM Cortex-A15 host CPU, a single Texas Instruments Keystone II DSP and two Keystone I DSPs. OpenCL codes can be written to be easily linked with auto-tuners, by allowing the local work group size to be set from the command line or as a macro in the pre-processor at 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 `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 for less critical codes can 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. These can also be corrected using autotuning for both kernel specific and device specific optimisations.

2.4 Benchmark Suites

Benchmarking forms the basis on which comparisons between languages and environments are made. Benchmark suites are large sets of benchmark codes used to reliably compare and measure realistic problems under realistic settings. 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 Extended OpenDwarfs benchmark suite in Chapter 3 which covers a wider range of application patterns by focusing exclusively on OpenCL using higher-level benchmarks. Before jumping into this work, existing benchmark suites are considered in the remainder of this section.

The NAS parallel benchmarks [48] 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. [49] 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. [50] 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.

51 ose Hetero-Mark, a Benchmark Suite for CPU-GPU Collaborative Computing, which has five benchmark applications each implemented in the Heterogeneous Compute Compiler (HCC) – which compiles to OpenCL and HIP which converts CUDA codes to the AMD Radeon Open Compute back-end. Meanwhile, Chai by Gómez-Luna et al.[52], offers 15 applications in 7 different implementations with the focus on supporting integrated architectures.

The Princeton Application Repository for Shared-Memory Computers (PARSEC) is a benchmark suite proposed by 53. It curates a set of real-world benchmarks from recognition, mining, synthesis and systems applications which mimic large-scale multithreaded commercial programs instead of the conventional types of HPC benchmark applications that achieve a high-performance. Its primary focus is to have a general purpose suite that assesses performance of multiprocessor CPUs over realistic application domains. Additionally, they identify CPU performance is tied to problem size, as such, one of the features of PARSEC is that it includes multiple problem sizes for the benchmark simulations – **simsmall**, **simmedium** and **simlarge**. Since accelerators are not considered in this work – and as such, all applications are written in C – it is not included in our evaluation, however, the fundamental principals of having a general purpose and portable set of applications that assess real-world workloads over multiple problem sizes, forms the basis of our extensions and are presented in Chapter 3.

Rodinia and the o nal 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)[54] also features an OpenCL implementation of several scientific applications. We considered Rodinia, OpenDwarfs and SHOC as the potential basis for our extended benchmark suite – the strengths and weaknesses of three are presented independently in the following subsections.

2.4.1 Rodinia

Che et al. [55] proposed the Rodinia benchmark suite to cover a wide range of parallel communication patterns to examine the performance of heterogeneous platforms free from language and device specific optimizations. The benchmarks were selected following the Berkeley Dwarf Taxonomy and are from real world high performance computing applications. The diversity between selected benchmarks was shown by measuring execution times, communications overheads and energy usage of running each benchmark on an NVIDIA GTX 280 GPU and an Intel Core 2 Extreme CPU. Across the suite: speedups in execution times ranged from 5.5x to 80.8x, communication overheads varied from 2-76% and GPU power consumption

overheads ranged from 38-83 Watts, illustrating important architectural differences between the CPU and GPU. At the time this thesis 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. [56] This extension features a subset of the dwarfs, namely, Structured Grid, Unstructured Grid, Dynamic Programming, Dense Linear Algebra, MapReduce, and Graph Traversal all of which may be expected to benefit from GPU acceleration. Diversity analysis was also performed and took the form of a Micro-Architecture independent analysis study. The MICA framework, discussed in section 2.8.1, was used as the basis of the evaluation and the motivation was to justify each application's inclusion in the benchmark suite by showing deviations between applications in the corresponding kiviat diagrams. Three separate implementations were developed for each application using CUDA for the GPU, OpenMP for the CPU and OpenCL for both architecture types. Several implementations caused fragmentation in development, which often resulted in the OpenCL version of each benchmark application being neglected; missing features offered in other implementations and in some instances lacking an implementation of a given application entirely. For this reason, Rodinia is not a suitable base for an OpenCL benchmark suite, however, we were able to incorporate the dwt2d benchmark into our extended version of the OpenDwarfs benchmark suite as will be discussed in Chapter 3. Many of the benchmarks were added from Rodinia into the original OpenDwarfs suite, in our extended evaluation many of the datasets were generated by analyse the original Rodinia source.

2.4.2 OpenDwarfs

As with Rodinia, Feng et al. [57] 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 a CPU, an Intel Xeon E5405, and three GPUs, a low power AMD HD5450 with 25W TDP, and two high-power GPUs: AMD HD5870 and an Nvidia GT520 with energy footprints of 228 and 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 only one representative application which may not expose the entire set of characteristics of that dwarf.

A potential criticism is that diversity analysis was performed to justify the inclusion of each application – however since many applications were 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. [58]. We selected OpenDwarfs as the basis for our extensions, this was a good place to start given it had the largest number of dwarfs already represented, the sole implementation was OpenCL, and had already been tested on a wide range of accelerators. These efforts are discussed in Chapter 3.

2.4.3 SHOC

The Scalable Heterogeneous Computing benchmark suite SHOC, presented by Danalis et al. [59], is an alternative benchmark suite to test the performance and stability of these scalable heterogeneous computing systems – primarily GPU and multi-core CPU accelerators. 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: the micro-benchmarks perform a stress test role to assess the device capabilities and assess the architectural features of each accelerator, and application kernels which measure entire system performance on real world applications. Some application kernels also supports multiple nodes using MPI to assess distributed parallelism of the system – intranode and internode communication among devices.

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 – aside from testing the performance and stability of scalable heterogeneous computing systems it also seeks to provide a comparison of programming models. 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.

There are two caveats of SHOC if it were used for our purposes. Firstly, there is a lack of classification according to the dwarf taxonomy, much of the work towards using micro-benchmarks to stress-test the system falls outside of the taxonomy and the higher level application benchmarks are too few to adequately cover a wide range of dwarfs – indeed only a few are represented. Secondly, the addition of applications is more expensive in SHOC, since it would require implementations for the same application into at least three other languages. There are additional difficulties to ensure each implementation is identical in order to adequately compare the programming models.

By focusing on application kernels written exclusively in OpenCL, our enhanced OpenDwarfs benchmark suite – presented in Chapter 3 – is able to represent a wider range of dwarfs while minimising development effort required when duplicating the functionality of applications between languages.

2.5 Hardware Performance and Scaling

The performance of heterogeneous devices is often evaluated against a theoretical upper-bound. Computing this limit requires an understanding of a couple of important hardware characteristics. This section discusses scaling with respect to clock frequency and core count.

Also included is a discussion on the impact frequency has on energy consumption.

Changing the clock frequency of a conventional CPU core ultimately changes performance results, where execution times are impacted but the energy efficiency of the device is also affected. Choi, Soma and Pedram [60] present an intra-process dynamic voltage and frequency scaling approach with the goal of minimising energy consumption yet maximising performance. This is achieved by modelling the on-chip / off-chip ratio using runtime event monitoring. Hardware measurements showed that dynamically lowering 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 a performance drop of 5-20%.

Recently, Brown [61] showed that 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 by Albers and Antoniadis [62] for hardware used in HPC provided it supports a sleep state. They present a framework to approximate the energy cost of frequency 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.

Meanwhile, Agarwal et al. [63] 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. The bottle-neck on many of these workloads is also moving from being compute-bound to memory or communication bound, since the imbalance of hardware improvements shift application requirements to wait on communication and memory transfers. As such, the impact of increasing the clock frequency is having (and will continue to have) less of an impa computational efficiency. This trend has been reinforced in current work by 64 and 65; M rn processors increasingly rely on both latency minimisation and latency hiding to conceal the widening gap between processor and memory clock frequencies. To this end, both 64 and 65 introduce techniques to model parallelism and opportunistically steal work during interrupt events which result in hiding the latency in the processor pipeline and reducing the latency in the memory hierarchy.

Since wire latencies have not matched the increase in the range of clock-frequency, the coupling between execution time and energy consumption is non-linear [66]. As such, the impact of increasing the clock frequency on applications that are compute-bound will result in a proportional reduction in execution time to having a higher clock-frequency, however, there are applications that are memory or communication bound, where increasing the frequency of a core does not also increase the speed of the memory bus and thus will experience little to no benefit. Applications and dwarfs may benefit from an accelerator with a memory clock which matches the core clock.

A good indication of a successful implementation of a parallel algorithm is performance scalability in response to core availability [67][68][69][70]. However, the trend of achieving good performance scaling by increasing the number of homogeneous cores on a system will

cease, primarily, due to the power limitations of having arrived at the utilisation wall – a limitation of the fraction of a chip that can run at full speed at one time [71][72].

Taylor [73] surveys the transition of typical homogeneous cores to a potentially dark silicon The primary factor is the percentage of a silicon chip that can switch at full frequency is dropping with each generator of processor, known as Dennard scaling – that as transistors get smaller, their power density stays constant, so that the power use stays in proportion with area – and ensures that large fractions of chips are either idle or operating at a lower clock frequency. Limitations from hitting this power-wall has meant specialized architectures are increasingly employed to "buy" energy efficiency by "spending" more on die area – thus increasing heterogeneity of the entire system. Indeed, the increasing utilization of accelerators as seen in today's leading supercomputers indicates an accurate prediction by Taylor – a bright future for heterogeneous systems. Taylor also notes that a by-product of adding specialized architectures – or accelerators – is massive increases in complexity. Introducing a methodology to direct codes to the most appropriate accelerator is one of the goals of this thesis.

2.6 OpenCL Performance

The performance of OpenCL kernels is affected by runtime p eters determining the allocation and partitioning of work changes between devices. Much of the partitioning can occur automatically using autotuning. Autotuning and tools and techniques used to measure device performance are summarised in this subsection. Also discussed is the common issue of phase shifting and how it relates to measuring OpenCL performance.

2.6.1 Autotuning

Whilst OpenCL is hardware-portable it is not inherently also performance-portable, autotuning is important when evaluating the performance of OpenCL codes on systems. 74 migrated CUDA versions of level 3 BLAS routines to OpenCL and measured the direct performance on GPU accelerator devices. They show low-level languages achieve 80% of peak performance on multicores and accelerators whilst OpenCL only achieves 50% of peak performance. They propose the use of auto-tuning to improve the performance of OpenCL kernels. They conclude that OpenCL is fairly competitive with CUDA on Nvidia hardware in terms of performance, and if architecture specifics are unknown, autotuning is an effective way to generate tuned kernels that deliver acceptable levels of performance with little programmer effort.

When combined with autotuning, an OpenCL code may exhibit good performance across varied devices – yielding accelerator device specific optimizations with no user or developer input. 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. [43],

Chaimov et al. [44] and Nugteren and Codreanu [45] all propose open source libraries capable of autotuning dynamic execution parameters in OpenCL kernels.

Additionally, Price and McIntosh-Smith [46] have demonstrated high performance using a general purpose autotuning library [75], for three applications across twelve devices. The OpenTuner library requires the search space to be defined the form of command line or compile time arguments – which are used as configuration parameters 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 46 paper, 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, Price shows that over-optimisation hurts performance portability.

Of the benchmarks presented in section 2.4, every application presented in the Rodinia Benchmark Suite requires a local workgroup to be passed. In the OpenDwarfs set of benchmarks 9 out of 14 allow for local workgroup tuning. Auto-tuning frameworks could be readily used with the Extended OpenDwarfs Benchmark Suite along with the other suites mentioned, however, since performance portability has been shown by others it is not the goal of this thesis and thus is left as future work.

2.6.2 Phase-Shifting

A program phase is defined as a set of intervals (or slices in time) during execution that have 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. [76] 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 [77] profiles (a vector of per element counts, where each element is the number of times a code block has been entered over a given in) 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. The kernel in execution itself will experience very little differences in phases since by their very nature OpenCL kernels are small compartmentalised sections of computation. For example, 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.

2.6.3 Measurements

The studies presented in this thesis require the use of tools to perform high-accuracy and low-overhead measurements. We use LibSciBench fo formance measurements of OpenCL kernels. It allows high precision timing events to be collected for statistical analysis [78]. 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 3 LibSciBench was intensively used to record timings, energy usage and hardware events, which it collects via PAPI [79 nters.

2.7 Offline Ahead-o me Analysis

Offline Analysis it does not operate on a running code, for our purposes, the analysis provides a detailed examination of the structure of code. Ahead-of-time indicates that this analysis be done before the program is executed – in the real-world usage of the code. The combination of theses two terms is directly applicable to OpenCL SPIR code, which is based on LLVM, since LLVM is well suited to performing ahead-of-time optimised native code generation [80]. Additionally, SPIR is hardware agnostic and ISA-independent as these features can be computed directly on the intermediate representation, that is, before a binary for an e is generated. Our analysis, presented with AIWC in Chapter 4 outlines a methodology to collect features of programs before they are deployed. These features are embedded into the header of the SPIR code – as a comment – which can be evaluated at runtime on supercomputing systems to be used by the scheduler to provide useful information around scheduling, specifically, determining on which device the kernel should be executed.

Muralidharan et al. [81] 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 three distinct heterogeneous platforms, and results in a metric to characterise performance.

Oclgrind is an OpenCL device simulator developed by Price and McIntosh-Smith [82] capable of performing simulated kernel execution. It operates on a restricted LLVM IR known as

Standard Portable Intermediate Representation (SPIR) [83], 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; this allows selected metrics to be collected by monitoring events during simulation, these metrics then indicate Architecture-Independent Workload Characteristics. Our work on AIWC is built on offline ahead-of-time analysis techniques and is presented in Chapter 4.

2.8 Program Diversity Analysis and Characterization

Program Diversity Analysis has been used to justify the inclusion of an application into a benchmark suite. Principal Component Analysis (PCA) on virtual machine and hardware (PAPI) events has been used to demonstrate program diversity [84][85]. Often this work is manually performed by those assembling the benchmark suite, indeed, much of the motivation for curating OpenCL applications in Rodinia [55], OpenDwarfs [57] and SHOC [59] was to have real-world scientific problems that represented regular workloads of HPC and SC systems.

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 [86]. The target of this work was to determine the major factors in modelling performance between parallel computer architectures in an architecture-independent manner. The focus of this section is examining the existing literature around the characterisation of an application in terms of dwarf and metrics, and concludes with how these characterisation techniques have been used when assembling benchmark suites.

2.8.1 Microarchitecture-Independent Workload Characterization

Hoste and Eeckout [87] show that although conventional microarchitecture-dependent characteristics are useful in locating performance bottlenecks [88], 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. These characteristics fail to distinguish between inherent program behaviour 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 [90] 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.

Lee et al. [91] present an evaluation of the performance of OpenCL applications on modern on out-of-order multicore CPUs. They collect CPU specific metrics around API and scheduling overheads, instruction-level parallelism, address space, data location, data locality, and vectorization which may serve as an indication of performance optimization metrics. These metrics could potentially be used by a developer to modify codes to achieve better performance on CPUs.

2.8.2 Architecture Independent Workload Characterization

Recently, Shao and Brooks [92] 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 five ISA-independent features/metrics. Namely, number of opcodes (e.g., add, mul), the value of branch entropy – a measure of the randomness of branch behaviour, 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 p[illegible]nts 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 toolchain 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 on Architecture Independent Workload Characterisation or known as (AIWC) is presented in Chapter 4, and extends Shao's work to the broader OpenCL setting to collect architecture independent metrics from a hardware-agnostic language – OpenCL. We also added metrics such as Instructions To Barrier (ITB), Vectorization (SIMD) indicators and Instructions Per Operand (SIMT) in order to perform a similar analysis for concurrent and accelerator workloads.

AIWC relies on the selection of the instruction set architecture (ISA)-independent features determined by Shao and Brooks [92], which in turn builds on earlier work in microarchitecture-independent workload characterization discussed in section 2.8.1.

The branch entropy measure used by Shao and Brooks [92] was initially proposed by Yokota [93] and uses Shannon's information entropy to determine a score of Branch History Entropy. De Pestel, Eyerman and Eeckhout [94] 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 our work on AIWC.

Caparrós Cabezas and Stanley-Marbell [95] present a framework for characterizing 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-level 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.

2.8.3 Workload Characterization for Benchmark Diversity Analysis

In contrast to our proposed multidimensional workload characterization, models such as Roofline [96] and Execution-Cache-Memory [97] seek to characterize an application based on one or two limiting factors such as memory bandwidth. The advantage of these approaches is the simplicity of analysis and interpretation. We view these models as capturing a 'principal component' of a more complex performance space; we claim that by allowing the capture of additional dimensions, AIWC supports performance prediction for a greater range of applications. In other words, there is less bias introduced when used for prediction since there is no cherry-picking of features and all are provided directly into a model. However this is discussed in greater detail in the next section.

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. 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. Unfortunately, this approach has the same limitations as those pre d in Section 2.8.1.

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.

2.9 Scheduling and Performance Prediction for Heterogeneous Architectures

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 [98], and choosing the wrong device results in slower and more energy-intensive computation [99]. Thus accurate performance prediction is critical to making optimal scheduling decisions in a heterogeneous supercomputing environment.

Lyerly [100] execute a subset of applications from OpenDwarfs to demonstrate 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 their scheduling analysis in Chapter 5 and claim 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. [101] 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. 55 broadens this finding by performing analysis on a single threaded CPU version and find that a benchmark application maintains the underlying set of instructions – the composition of the application is largely the same.

Therefore, it is intuitive that the composition of a program collected using a simulator (such as Oclgrind discussed in Section ??, 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 for an application workload. This, in turn, can be used as a basis for performance prediction on general accelerators.

2.10 Predictions and Modelling

Augonnet et al. [102] 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. The algorithms presented to automate data movement should be reused for scheduling of OpenCL kernels to heterogeneous accelerator systems.

Existing works [103], [104], [105], [106], 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.

One such approach uses partial execution, as introduced by Yang et al. [107] enables low-cost performance estimates over a wide range of execution platforms. 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. [108]. 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. 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.

Karami et al. [109] design a performance model for NVIDIA GPUs from OpenCL kernels to aid developers to locate GPU specific performance bottlenecks in their codes. This model depends on the collection of GPU performance counters over a range of benchmarks, these counters are then provided to a regression model with principle component analysis to develop a model to show how different GPU parameters account for applications performance bottlenecks. The model predicts application behavior with a 91% accuracy and when coupled with a larger database of collections can be used to predict their likely performance bottlenecks of unknown applications based on similarities with those previously collected. A caveat of this approach is that collecting performance counters as a basis for a model is microarchitecture specific – where counters collected from a system can range wildly between generation of processor and is not portable between vendors.

A GPU power-estimation model was developed by Wu et al. [110] which also uses hardware

performance counter values to train a machine learning model. Values for a new application are provided to a neural network at runtime to predict a scaling curve and corresponding estimates around performance and power of the application under different GPU configurations. OpenCL kernels are examined over different AMD GPUs throughout this investigation and the major factors contributing to the scaling curve was determined to be performance counters collected over varying core frequencies, memory bandwidths, and compute unit (CU) counts. The models performance was accurate to within 15% compared to real hardware and power estimates to within 10%. These models are based on AMD vendor specific counters which limits the scope of this work, however, the hardware configurations should be considered in estimating accelerator performance and power usage.

The X-MAP tool is proposed by Shetty [111] to achieve performance prediction when porting applications to accelerators. A Machine Learning based inference model is presented to predict the performance of a application on accelerator and programming language – either CUDA or OpenCL. Hardware counters are collected and are used as inputs into a Random Forest Classification Model. Most of the efforts of this tool is on locating bottlenecks in applications and committing the developer to target a specific implementation and device vendor. Thus this work is orthogonal to our aim of scheduling OpenCL kernels given a variety of available devices.

Che and Skadron [112] propose a set of first-order metrics that most influence GPU performance and scalability that are separate from those bound to CPUs. Hardware counters are used to collect and generate these metrics, which are then used in a performance prediction model. Similarly, a GPU performance modeling framework is proposed by Boyer, Meng and Kumaran [113] which predicts both kernel execution time and data transfer time. The main motivation of this work is to examine a CUDA kernels potential, in terms of performance, before it is optimized. This work shows that the inclusion of transfer time is significant when improving a predictive models accuracy and is especially useful for predicting speed-up on accelerators located over slower interconnect, such as PCIe – including the data transfer time in the model improved prediction error from 255% to 9%.

In Chapter 5, we propose an alternative model which allows accurate execution time predictions of OpenCL kernels on a wide range of architecturally-diverse accelerators. This methodology uses features from AIWC – from Chapter 4 – to form a basis for a predictive model bound to run-times measured or the benchmark codes presented in Chapter 3.

114 propose the Heterogeneity-Aware Signature-Supported (HASS) scheduler – a scheduling algorithm that matching threads to the most appropriate CPU cores. The architectural properties of an application are presented as signatures – a compact summary of the applications memory-boundedness, available ILP, sensitivity to variations in clock speed. These are generated offline and can be embedded into the program binary. The scheduler then matches these signatures to the most appropriate core. HASS is targeted on heterogeneous CPU cores and is evaluated over two big.LITTLE type, asymmetric single-ISA, configurations – an Intel Xeon X5365 and AMD Opteron 8356. CPU systems were treated as heterogeneous by changing the

clock frequencies of individual cores. The evaluation examines the performance of automatic mapping of memory-bound threads to slow / smaller cores leaving threads that are capable of fully utilizing the faster cores. A caveat of this approach is that other accelerators are not considered and as such the signatures are not architecture-independe wever, this the proposed methodology is the most similar and is the predecessor to our work.

Lee and Wu [115] directly tackle the problem of scheduling OpenCL applications to the most suitable accelerator device. They propose HeteroPDP – a scalable performance degradation predictor – to dynamically balance the execution time slowdown when co-locating multiple applications in the same heterogeneous system. The device selection decision is based on individual kernel metrics such as the degree of parallelism and divergence in an application and by the amount of data movement overhead between the host system and the selected accelerator. They conclude that designing a scheduler which considers the effect of memory interference between processes provides improvements. A major focus is on schedulers and orchestrating these workloads – we believe the accuracy of our predictive framework [116] based on AIWC metrics is complimentary to this work and would only improve the accuracy of their scheduler.

Chapter 3

Extending the OpenDwarfs Benchmark Suite

This chapter presents an extended and enhanced version of the OpenDwarfs OpenCL benchmark suite (EOD) to provide a test platform of representative codes. It will be later used for workload characterization, performance prediction and ultimately scheduling, but these sophisticated studies first need simple empirical data. However, methodologies to acquire these results – in the form of execution times – must first be presented. Additionally, for reproducibility and assurances of realistic scientific applications, the codes, settings and range of heterogeneous accelerator devices must be disclosed.

The OpenDwarfs benchmark suite [58] was selected from a set of benchmark suites – discussed in Section 2.4 – but required several essential extensions to meet the needs of the broader goals of this thesis. EOD places a strong focus on the robustness of applications, curation of additional benchmarks with an increased emphasis on correctness of results and choice of problem size. Other improvements focus 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. Having a common back-end in the form of OpenCL allows a direct comparison of identical code across diverse architectures. 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. This Chapter is based off our publication in the Proceedings of the 47th International Conference on Parallel Processing Companion, ICPP 2018 [117].

3.1 Enhancing the OpenDwarfs Benchmark Suite

The OpenDwarfs benchmark suite comprises a variety of OpenCL codes, classified according to the Dwarf Taxonomy [3]. 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.

118 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.

Our philosophy for the benchmark suite is that firstly, it "must" run on all devices, and secondly, it "should" run well on them. To this end, we removed hardware specific optimizations from codes that would either diminish performance, or crash the application entirely when executed on other devices. Instead, we added autotuning support to achieve a comparable performance whilst retaining the general purpose nature which is critical to a benchmark suite. Configuration parameters for the benchmarks, such as local workgroup size, were incorporated into EOD using the OpenTuner[75] auto-tuning library.

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 [119], which worked correctly in all our tests. We have also added a 2-D discrete wavelet transform from the Rodinia suite [55] – with modifications to improve portability.

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 [78]. 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 [79] 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 chapter.

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

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 3.1. 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

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. [120] 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 §3.2.4) 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.

Table 3.2: List of Extended OpenDwarfs Applications and their respective dwarfs

Dwarf	Extended OpenDwarfs Application
Dense Linear Algebra	LU Decomposition
Sparse Linear Algebra	Compressed Sparse Row
Spectral Methods	DWT2D, FFT
N-Body Methods	Gemnoui
Structured Grid	Speckle Reducing Anisotropic Diffusion
Unstructured Grid	Computational Fluid Dynamics
Map Reduce	K-Means
Combinational Logic	Cyclic-Redundancy Check
Graph Traversal	Breadth First Search
Dynamic Programming	Smith-Waterman
Backtrack and Branch and Bound	N-Queens
Graphical Methods	Hidden Markov Models
Finite State Machines	Temporal Data Mining

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. The applications examined in this work are presented in Table3.2 alongside their representative dwarf from the Berkeley Taxonomy.

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 3.3 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 chapter 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, swat, 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)[121] 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[122] 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[123] 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[124], 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`[125] tool. Generation of the solvent excluded molecular surface used the tool `msms` [126]. 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 chapter's associated GitHub repository [127].

Smith-Waterman alignment `swat` is a variation of the Needleman-Wunsch algorithm, used for computing local sequence alignment. The original OpenDwarfs suite included a selection of data files, but no method to generate arbitrarily-sized inputs.

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 3.3.

Table 3.3: 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

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 3.4 at the selected scale of Φ. For reproducibility the entire set of Python scripts with all problem sizes is available in a GitHub repository [127]. Where Φ is substituted as the argument for each benchmark, it is taken as the respective scale from Table 3.3 and is inserted into Table 3.4.

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 eight of the benchmarks, we measured multiple problem sizes and observed distinctly different

Table 3.4: 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.

scaling patterns between devices. This underscores the importance of allowing a choice of problem size in a benchmarking suite.

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 MIC.

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][128]. 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, 3.3, 3.4 and 3.5 shows the distribution of kernel execution times for the remaining benchmarks. The **tiny** and **small** sizes for the `kmeans`, `lud`, `csr` and `dwt` benchmarks are presented in Figure 3.2 results, the **medium** and **large** problem sizes are presented in Figure 3.3. Similarly, the final three applications which support multiple problem sizes – `fft`, `srad` and `nw` – display the time results for **tiny** and **small** in Figure 3.4, and **medium** and **large** times are shown in Figure 3.5. 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

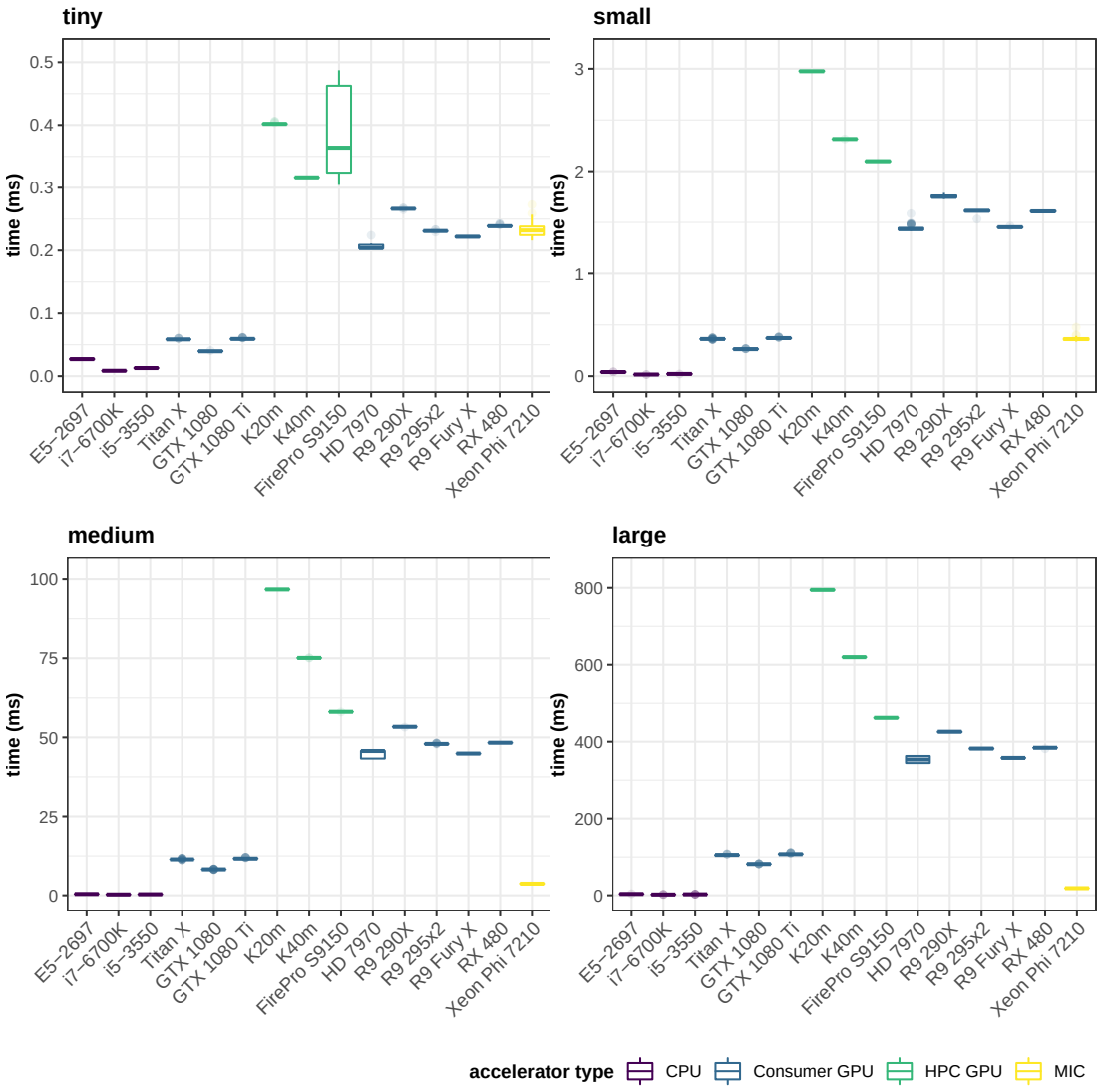


Figure 3.1: Kernel execution times for the **crc** benchmark on different hardware platforms

for all kernels. Each benchmark corresponds to a particular dwarf: From Figures 3.2 and 3.3 (a) (`kmeans`) represents the MapReduce dwarf, (b) (`lud`) represents the Dense Linear Algebra dwarf, (c) (`csr`) represents Sparse Linear Algebra, (d) (`dwt`) and from Figures 3.4 (a) and 3.5 (a) (`fft`) represent Spectral Methods, (b) (`srad`) represents the Structured Grid dwarf and (c) (`nw`) represents Dynamic Programming.

Finally, Figure 3.6 presents results for the four 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 3.6 (a), the Backtrack & Branch and Bound dwarf is represented by the (`nqueens`) application in Figure 3.6 (b), (`hmm`) results from Figure 3.6 (c) represent the Graphical Models dwarf and (`swat`) from Figure 3.6 (d) also depicts the Dynamic Programming dwarf.

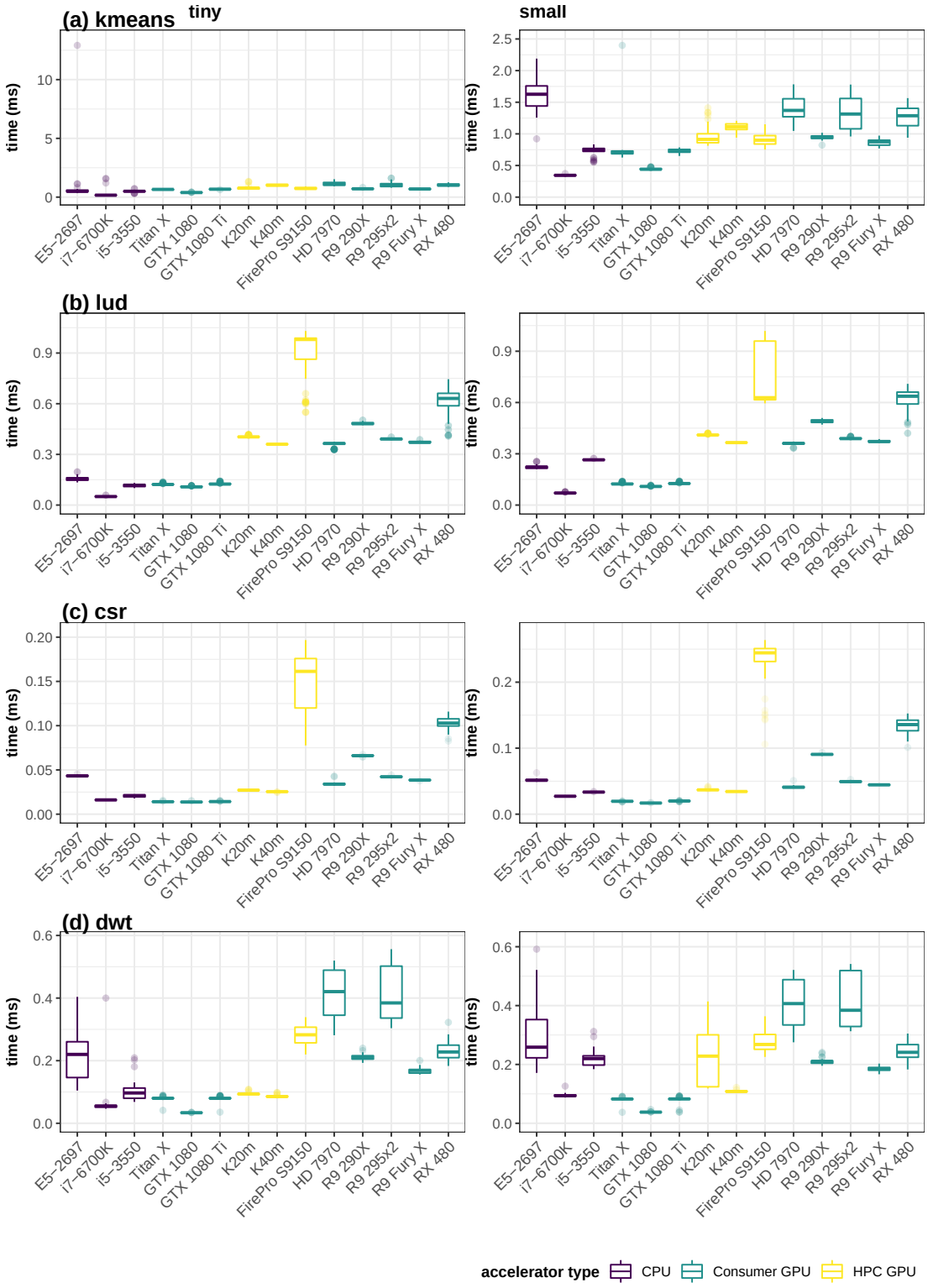


Figure 3.2: Kernel execution times for the **tiny** and **small** problem sizes of the `kmeans`, `lud`, `csr` and `dwt` benchmarks on different hardware platforms

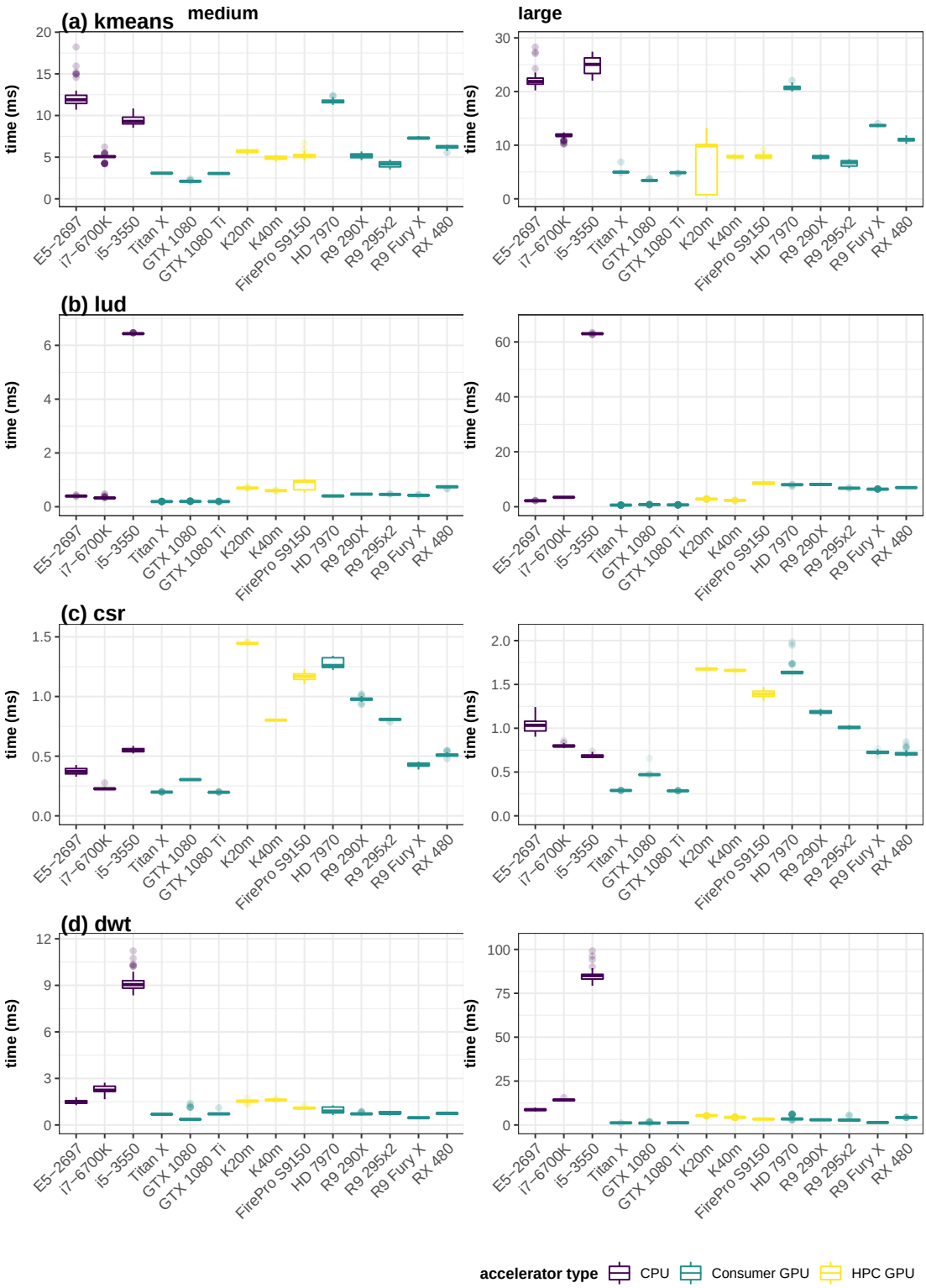


Figure 3.3: Kernel execution times for the **medium** and **large** problem sizes of the `kmeans`, `lud`, `csr` and `dwt` benchmarks on different hardware platforms

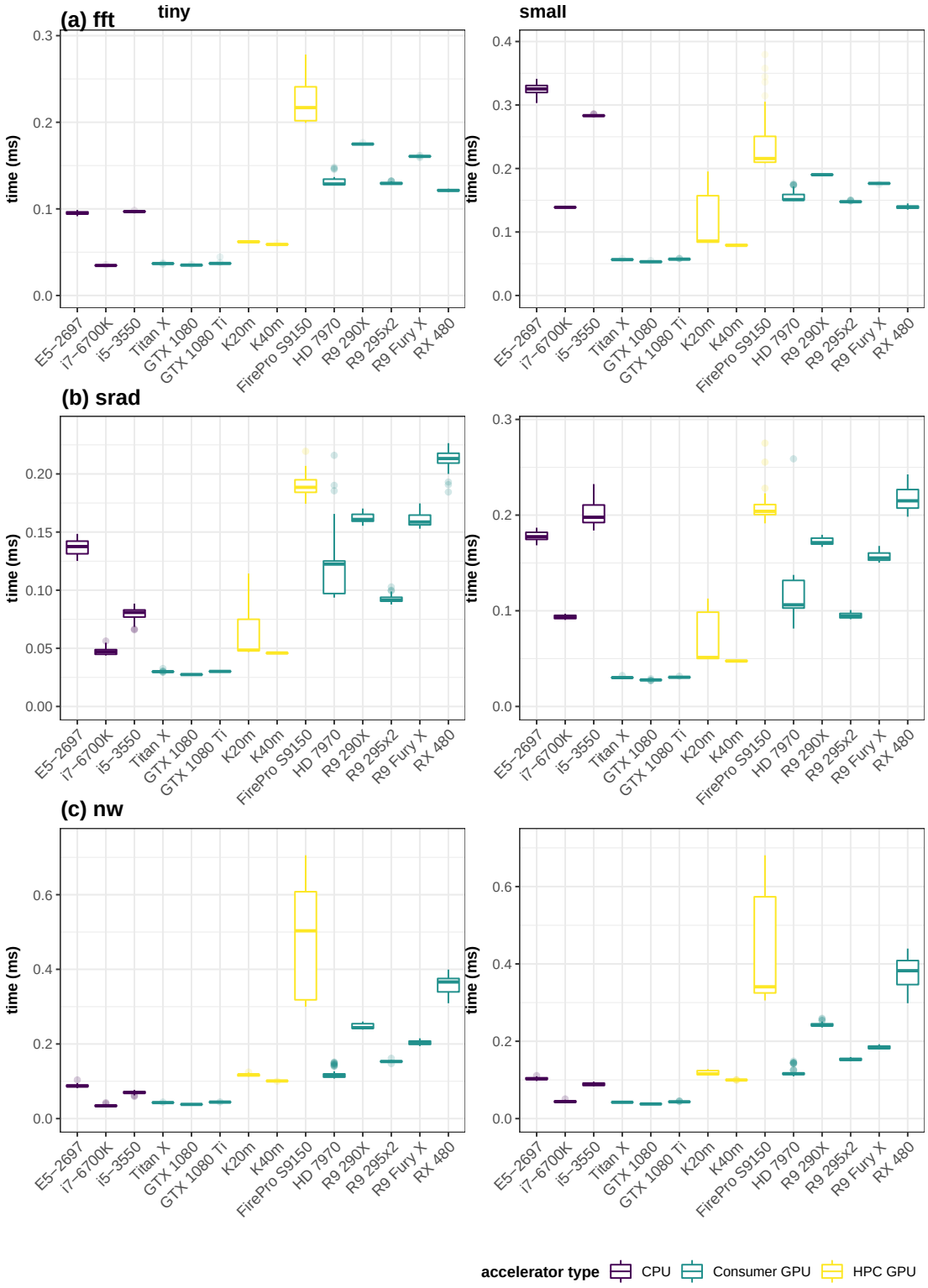


Figure 3.4: Kernel execution times for the **tiny** and **small** problem sizes of the `fft`, `srad` and `nw` benchmarks on different hardware platforms

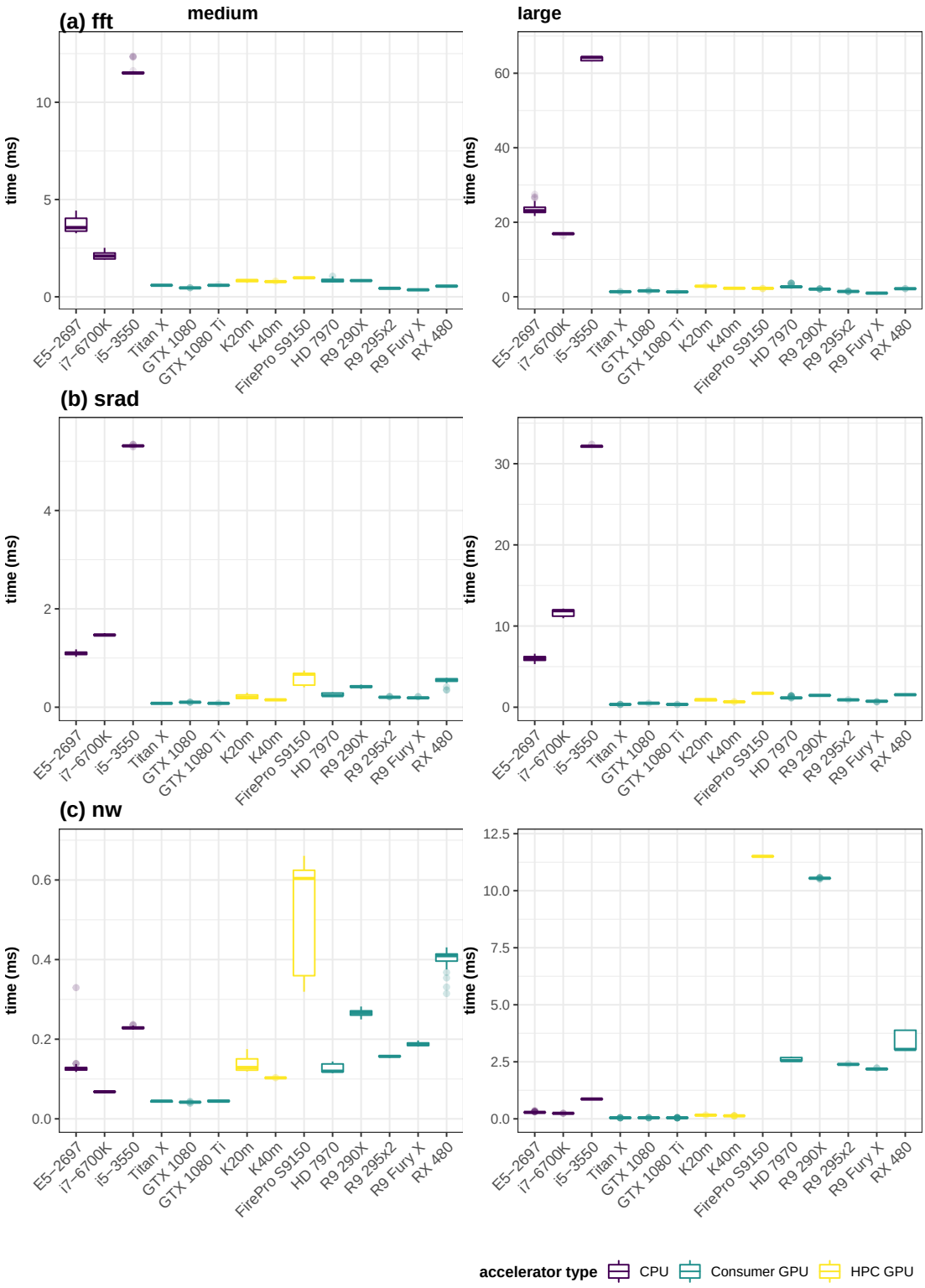


Figure 3.5: Kernel execution times for the **medium** and **large** problem sizes of the `fft`, `srad` and `nw` benchmarks on different hardware platforms

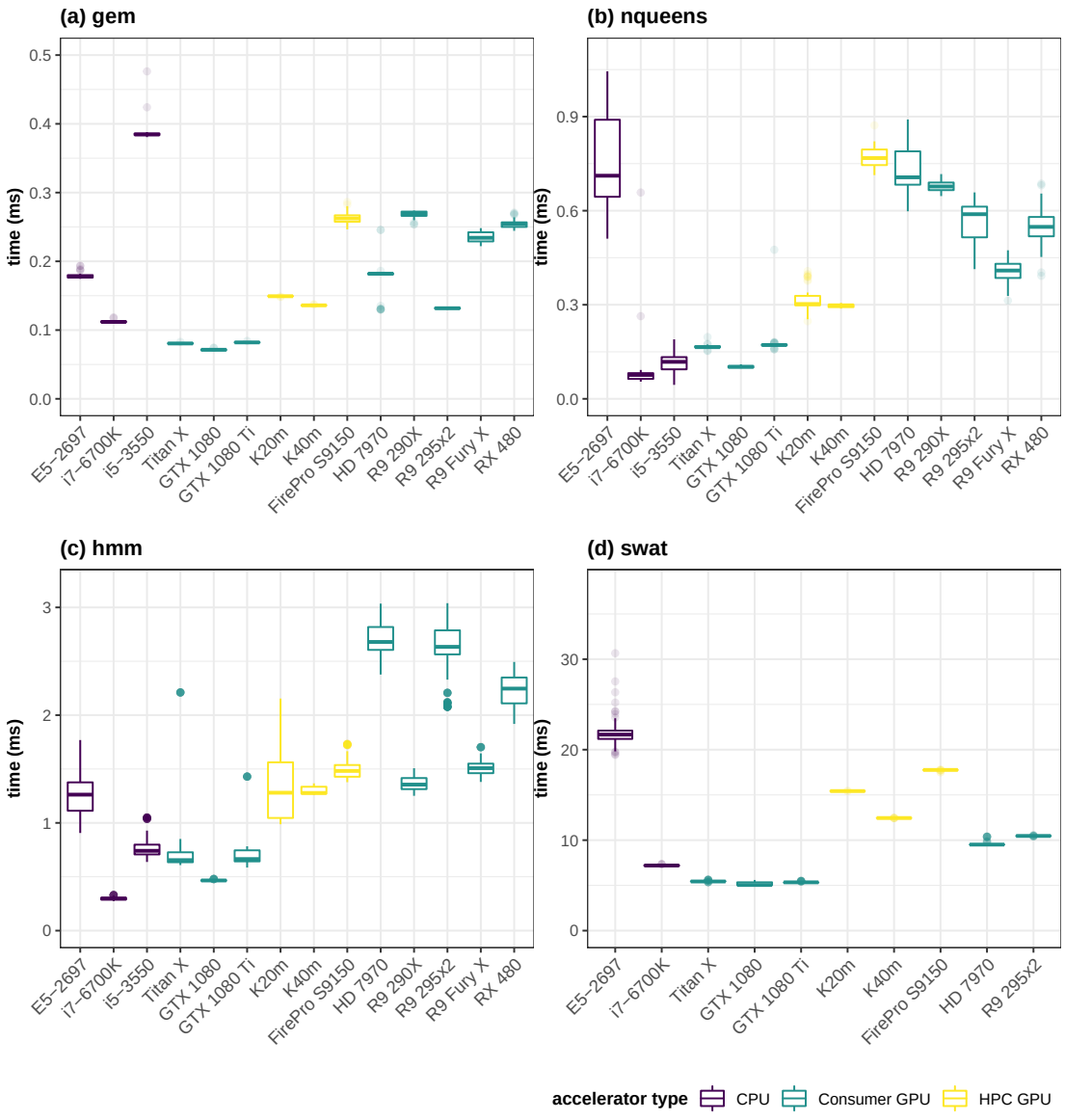


Figure 3.6: Single problem sized benchmarks of kernel execution times on different hardware platforms

Examining the transition from tiny to large problem sizes in Figures 3.4 (b) and 3.5 (b) 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, `nw` – (b) from Figures 3.4 and 3.5 – shows that the Intel CPUs and NVIDIA GPUs perform comparably for all problem sizes, whereas all AMD GPUs exhibit worse performance as size increases. This suggests that performance for this Dynamic Programming problem cannot be explained solely by considering accelerator type and may be tied to micro-architecture or OpenCL runtime support.

For most benchmarks, the variability in execution times is 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 `kmeans` 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 3.1.) 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 (b),(d) and (e) in Figures 3.2 and 3.3, and in (a) from Figures 3.4 and 3.5.

Increasing problem size also hinders the performance in certain circumstances for GPU devices. For example, (b) from Figures 3.4 and 3.5 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. [3] 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 3.3 (d) and Figure 3.5 (a) 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. [3] state that the Structured Grid dwarf is memory bandwidth limited. The Structured Grid dwarf is represented by the `srad` benchmark shown in Figure 3.5 (b). 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 3.7 (a) and (b) 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: purple for the Intel Skylake i7-6700k CPU and yellow for the Nvidia GTX1080 GPU. The logarithmic transformation has been applied to Figure 3.7 (b) 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.

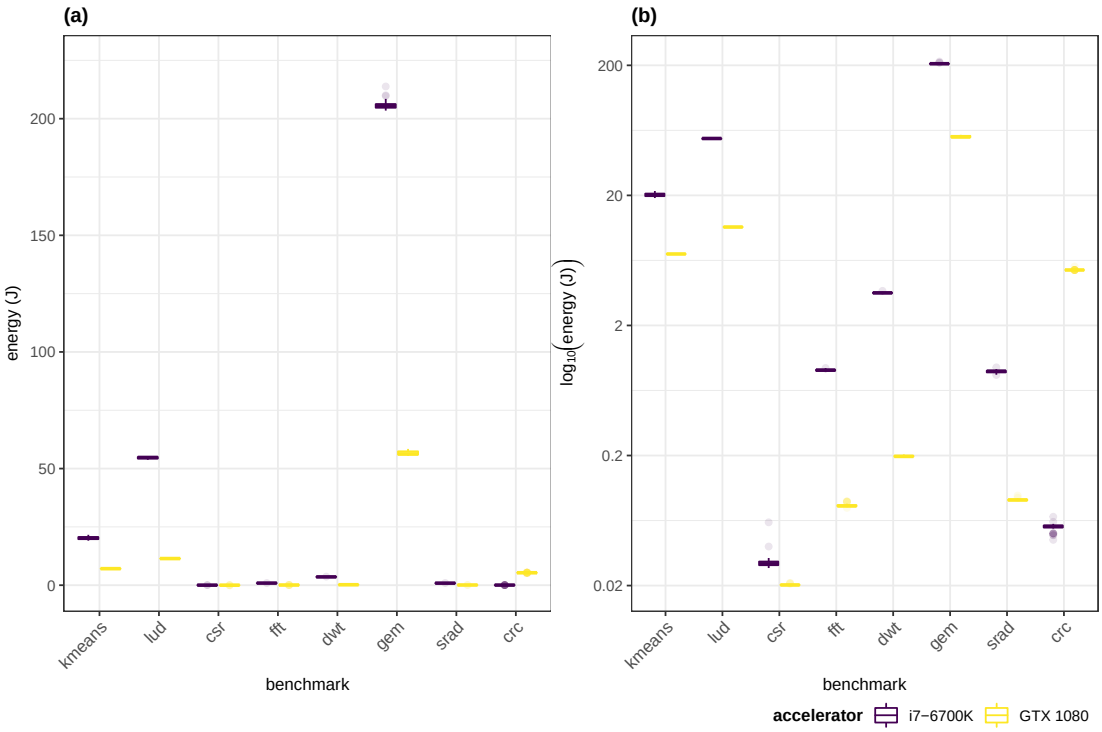


Figure 3.7: Execution energy required to perform EOD benchmarks, presented on a linear (a) and logarithmic scale (b) from left to right respectively, on the (**large** problem size) on the Intel i7-6700K and Nvidia GTX1080

3.4 Discussion

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 heterogeneous devices from a shared language – OpenCL. The introduced benchmarking suite – EOD – and the corresponding execution times on the full range of accelerators are used in the remainder of this thesis. Working on EOD led to the consideration of developing a tool to examine the limitations and characteristics of these codes in a device-independent, and more generally, architecture-independent, way. Comparing the runtimes of all kernels and examining the architectural effects on so 15 systems encouraged the development of AIWC – presented in the next Chapter. While performance on a kernel-by-kernel basis could be manually performed over EOD – collecting results on an abstract OpenCL device via simulation of these kernel codes has enabled a largely automated approach to compare feature-spaces and their suitability / mapping to accelerators.

Additionally, the recorded EOD runtimes from this chapter are used as a test-bed for the predictive model presented in Chapter 5. 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.

In general, the results of this chapter identify a few major points. Firstly, 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 characterization of a kernel allow the efficient scheduling of work to the most appropriate accelerator", the workload characterization tool – AIWC – is introduced in the next Chapter whilst the broader question is addressed in Chapter 5.

Chapter 4

AIWC: OpenCL based Architecture Independent Workload Characterization

Application codes differ in resource requirements, control structure and available parallelism. Similarly, compute devices differ in number and capabilities of execution units, processing model, and available resources. Given performance measurements for particular combinations of codes and devices, it is difficult to generalize to novel combinations. Hardware designers and HPC integrators would benefit from accurate and systematic performance prediction, for example, in designing an HPC system, to choose a mix of accelerators that are well-suited to the expected workload.

Measuring performance-critical characteristics of application workloads is important both for developers, who must understand and optimize the performance of codes, as well as designers and integrators of HPC systems, who must ensure that compute architectures are suitable for the intended workloads. However, if these workload characteristics are tied to architectural features that are specific to a particular system, they may not generalize well to alternative or future systems.

An architecture-independent method ensures an accurate characterization of inherent program behaviour, without bias due to architecture-dependent features that vary widely between different types of accelerators.

To this end, we present the Architecture Independent Workload Characterization (AIWC) tool. AIWC simulates the execution of OpenCL kernels to collect architecture-independent features that characterize each code, which may also be used in performance prediction.

AIWC is the first workload characterization tool to support multi-threaded or parallel workloads, which it achieves by collecting metrics that indicate both instruction and thread-level parallelism. Exploitable coarse-grained parallelism is measured by counting the number of work-items and barriers encountered. Instructions To Barrier (ITB) and Instructions per Thread (IPT) can be used to indicate workload irregularity or imbalance.

Table 4.1: Metrics collected by the **AIWC** tool ordered by type.

Type	Metric	Description
Compute	Opcode	total # of unique opcodes required to cover 90% of dynamic instructions
Compute	Total Instruction Count	total # of instructions executed
Parallelism	Work-items	total # of work-items or threads executed
Parallelism	Total Barriers Hit	total # of barrier instructions
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	Min IPT	minimum # of instructions executed per thread
Parallelism	Max IPT	maximum # of instructions executed per thread
Parallelism	Median IPT	median # of instructions executed per thread
Parallelism	Max SIMD Width	maximum # of data items operated on during an instruction
Parallelism	Mean SIMD Width	mean # of data items operated on during an instruction
Parallelism	SD SIMD Width	standard deviation across # of data items affected
Memory	Total Memory Footprint	total # of unique memory addresses accessed
Memory	90% Memory Footprint	# of unique memory addresses that cover 90% of memory accesses
Memory	Unique Reads	total # of unique memory addresses read
Memory	Unique Writes	total # of unique memory addresses written
Memory	Unique Read/Write Ratio	indication of workload being (unique reads / unique writes)
Memory	Total Reads	total # of memory addresses read
Memory	Total Writes	total # of memory addresses written
Memory	Reread Ratio	indication of memory reuse for reads (unique reads/total reads)
Memory	Rewrite Ratio	indication of memory reuse for writes (unique writes/total writes)
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	total # of unique branch instructions
Control	90% Branch Instructions	# of 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

We demonstrate the use of AIWC to characterize a variety of codes in the Extended OpenDwarfs Benchmark Suite [117] – presented in chapter 3. A majority of this Chapter was published in the LLVM-HPC workshop proceedings as part of the 30th International Conference for High Performance Computing, Networking, Storage, and Analysis (SC18) 2018 [129]. Additionally, work from Section 4.6 has been submitted as as Special Issue paper in the International Journal of High Performance Computing Applications (IJHPCA) and is currently under review.

4.1 Metrics

For each OpenCL kernel invocation, the Oclgrind simulator **AIWC** tool collects a set of metrics, which are listed in Table 4.1.

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 a 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 the 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. [93] 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** (ITB) 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 a large difference between min and max ITB may indicate an irregular workload.

Instructions per thread (IPT) based metrics are generated by performing a running tally of instructions executed per work-item until completion. The count is saved and resets. **Min**, **max** and **median IPT** are reported to understand load imbalance.

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 standard deviation – **SD SIMD width**. Further characterisation of parallelism is presented in the **work-items** and **total barriers hit** metrics.

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.

Finally, unique verses absolute reads and writes can indicate shared and local memory reuse between work-items within a work-group, and globally, which shows the predictability of a workload. To present these characteristics the **unique reads**, **unique writes**, **unique read/write ratio**, **total reads**, **total writes**, **reread ratio**, **rewrite ratio** metrics are proposed. The **unique read/write ratio** shows that the workload is balanced, read intensive or write intensive. They are computed by storing read and write memory accesses separately and are

later combined, to compute the **global memory address entropy** and **local memory address entropy** scores.

4.2 Methodology – Workload Characterization by tooling Oclgrind

AIWC verifies the architecture independent metrics since they are collected on a toolchain 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 – 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 [92] to the Oclgrind tool offers an 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 an 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.

4.3 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 compared against which of the two targets were taken, the result is stored in the branch history trace. The implementation of this is shown in Listing 4.1. Note the `instructionExecuted` callback is propagated from Oclgrind during every OpenCL kernel instruction – emulated in LLVM IR. This function also updates variables to track instruction diversity by counting the occurrences of each instruction, instructions to barrier and other parallelism metrics by running a counter until a barrier is hit, finally, the vectorization – as part of the parallelism metrics – are updated by recording the width of executed instructions. The `m_state` variable is shared between all work-items in a work-group and these are stored into a global set of variables using a mutex lock once the work-group has completed execution.

The branch metrics are then computed by evaluating the full history of combined branch's taken and not-taken.

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 [92], and **Average Linear Branch Entropy**, from De Pestel [94] 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.

Listing 4.1: The Instruction Executed callback function collects specific program metrics and adds them to a history trace for later analysis.

```
1  void WorkloadCharacterisation::instructionExecuted(..., const llvm
       ::Instruction *instruction, ...){
2      unsigned opcode = instruction->getOpcode();
3      std::string opcode_name = llvm::Instruction::getOpcodeName(
           opcode);
4      //update key-value pair of instruction name and its occurrence
           in the kernel
5      (*m_state.computeOps)[opcode_name]++;
6      std::string Str = "";
7      //if a conditional branch which has labels, store the labels to
           track
8      //in the next instruction which of the two lines we end up in
9      if (opcode == llvm::Instruction::Br && instruction->
           getNumOperands() == 3){
10         if(instruction->getOperand(1)->getType()->isLabelTy() &&
11                 instruction->getOperand(2)->getType()->isLabelTy()){
12             m_state.previous_instruction_is_branch = true;
13             llvm::raw_string_ostream OS(Str);
14             instruction->getOperand(1)->printAsOperand(OS,false);
15             m_state.target1 = Str;
16             Str = "";
17             instruction->getOperand(2)->printAsOperand(OS,false);
18             m_state.target2 = Str;
19             llvm::DebugLoc loc = instruction->getDebugLoc();
20             m_state.branch_loc = loc.getLine();
21          }
22     }
23     //if the last instruction was a branch, log which of the two
           targets were taken
24     else if (m_state.previous_instruction_is_branch == true){
25         llvm::raw_string_ostream OS(Str);
26         instruction->getParent()->printAsOperand(OS,false);
27         if(Str == m_state.target1)
28             (*m_state.branchOps)[m_state.branch_loc].push_back(true)
                   ;//taken
29         else if(Str == m_state.target2){
30             (*m_state.branchOps)[m_state.branch_loc].push_back(false
                   );//not taken
31         }
32         m_state.previous_instruction_is_branch = false;
33     }
34     //counter for instructions to barrier and other parallelism
           metrics
35     m_state.instruction_count++;
36     m_state.workitem_instruction_count++;
37     //SIMD instruction width metrics use the following
38     m_state.instructionWidth->push_back(result.num);
```

Finally, `kernelBegin` initializes the global counters and `kernelEnd` triggers the generation and presentation of all the statistics listed in Table 4.1. The source code is available at the GitHub Repository [130].

4.4 Demonstration

We now demonstrate the use of **AIWC** on several scientific application kernels selected from the Extended OpenDwarfs Benchmark Suite [117]. 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, 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 generalize 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, four selected metrics are presented in Figure 4.1. One metric was chosen from 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*, yellow 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 4 different problem sizes.

For almost all benchmarks the global memory address entropy increases with problem size, whereas the other metrics do not increase. Notably, memory entropy is low for `lud_diagonal`, reflecting memory access with constant strides of diagonal matrix elements, and `cl_fdt53Kernel`, again reflecting regular strides generated by downsampling in the discrete wavelet transform. Note, we do not present **medium** and **large** problem sizes for some kernels due to various issues including: a lack of input datasets, failure of AIWC in

tracing large numbers of memory and branch operations for entropy calculations. These issues will be addressed in future work.

Looking at branch entropy, `bfs_kernel2` stands out as having by far the greatest entropy. This kernel is dominated by a single branch instruction based on a flag value which is entirely unpredictable, and could be expected to perform poorly on a SIMT architecture such as a GPU.

Barriers per instruction is quite low for most kernels, with the exception of `needle_opencl_shared_1` and `needle_opencl_shared_2` from the Needleman-Wunsch DNA sequence alignment dynamic programming benchmark. These kernels each have 0.04 barriers per instruction (i.e. one barrier per 25 instructions), as they follow a highly-synchronized wavefront pattern through the matrix representing matching pairs. The performance of this kernel on a particular architecture could be expected to be highly dependent on the cost of synchronization.

4.5 Detailed Analysis of LU Decomposition Benchmark

We now proceed with a more detailed investigation of one of the benchmarks, **lud**, which performs decomposition of a matrix into upper and lower triangular matrices. Following Shao and Brooks [92], 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 relationship 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**.

Additionally, a common problem in parallel applications is load imbalance – or the overhead introduced by unequal work distribution among threads. A simple measure to quantify imbalance can be achieved using a subset of the existing AIWC metrics and is included as a further derived parallelism metric by computing **load imbalance** = **max IPT** − **min IPT**.

All other values are normalized according to 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 centre (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

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

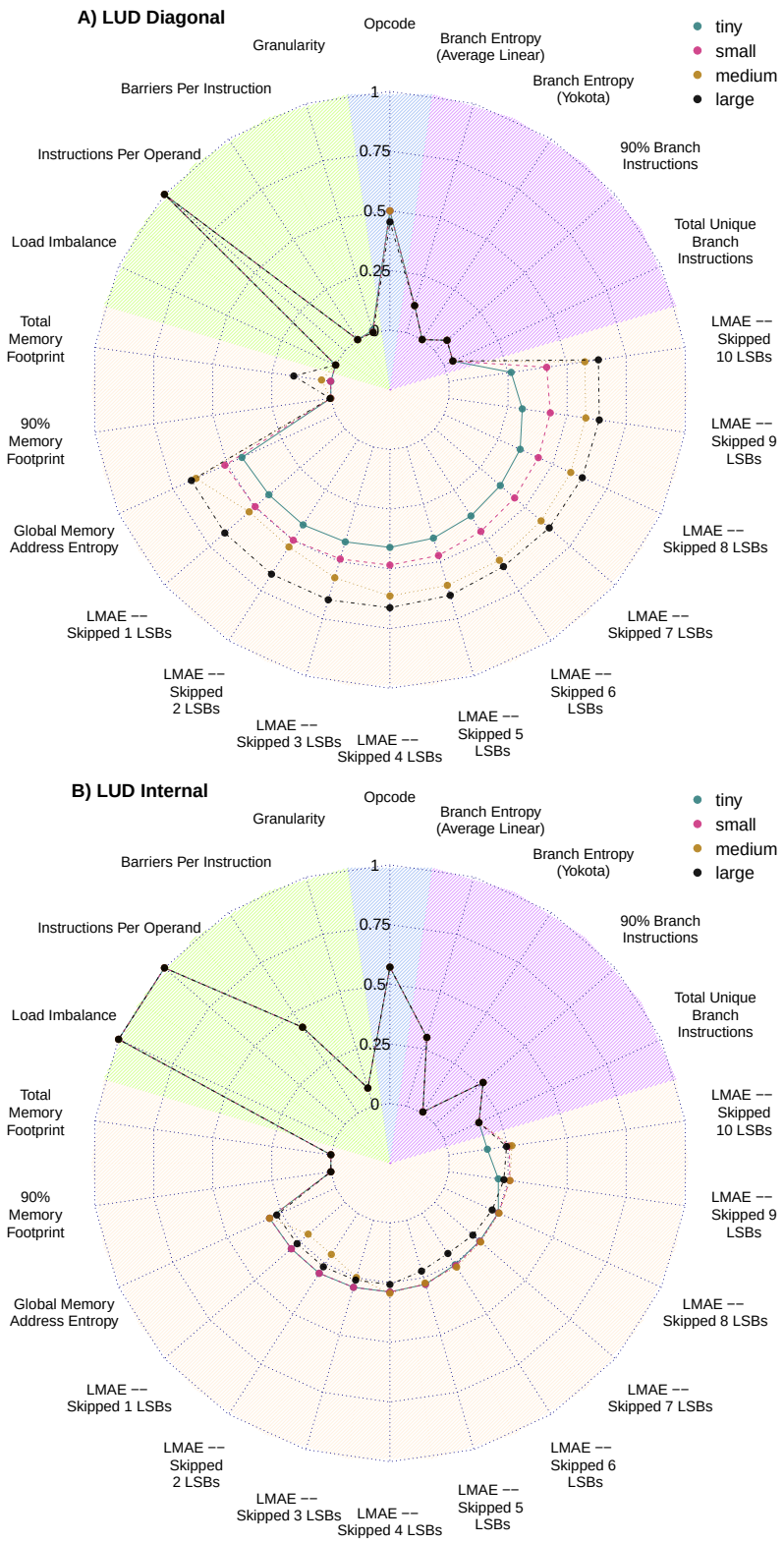


Figure 4.2: A) and B) show the AIWC features of the `diagonal` and `internal` kernels of the LUD application over all problem sizes.

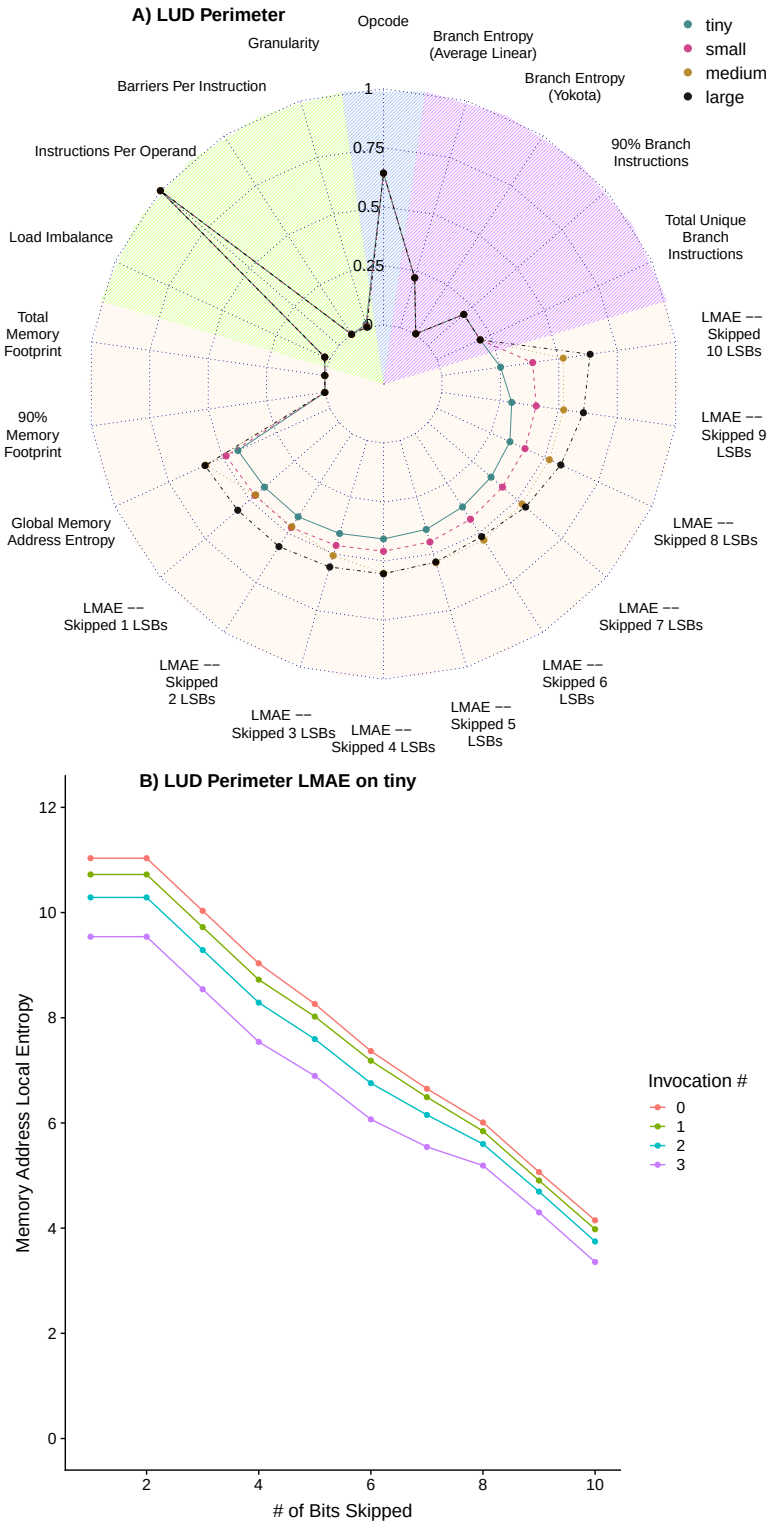


Figure 4.3: A) shows the AIWC features of the `perimeter` kernel of the LUD application over all problem sizes. B) shows the corresponding Local Memory Address Entropy for the `perimeter` kernel over the tiny problem size.

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 Figure 4.3 A) 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.

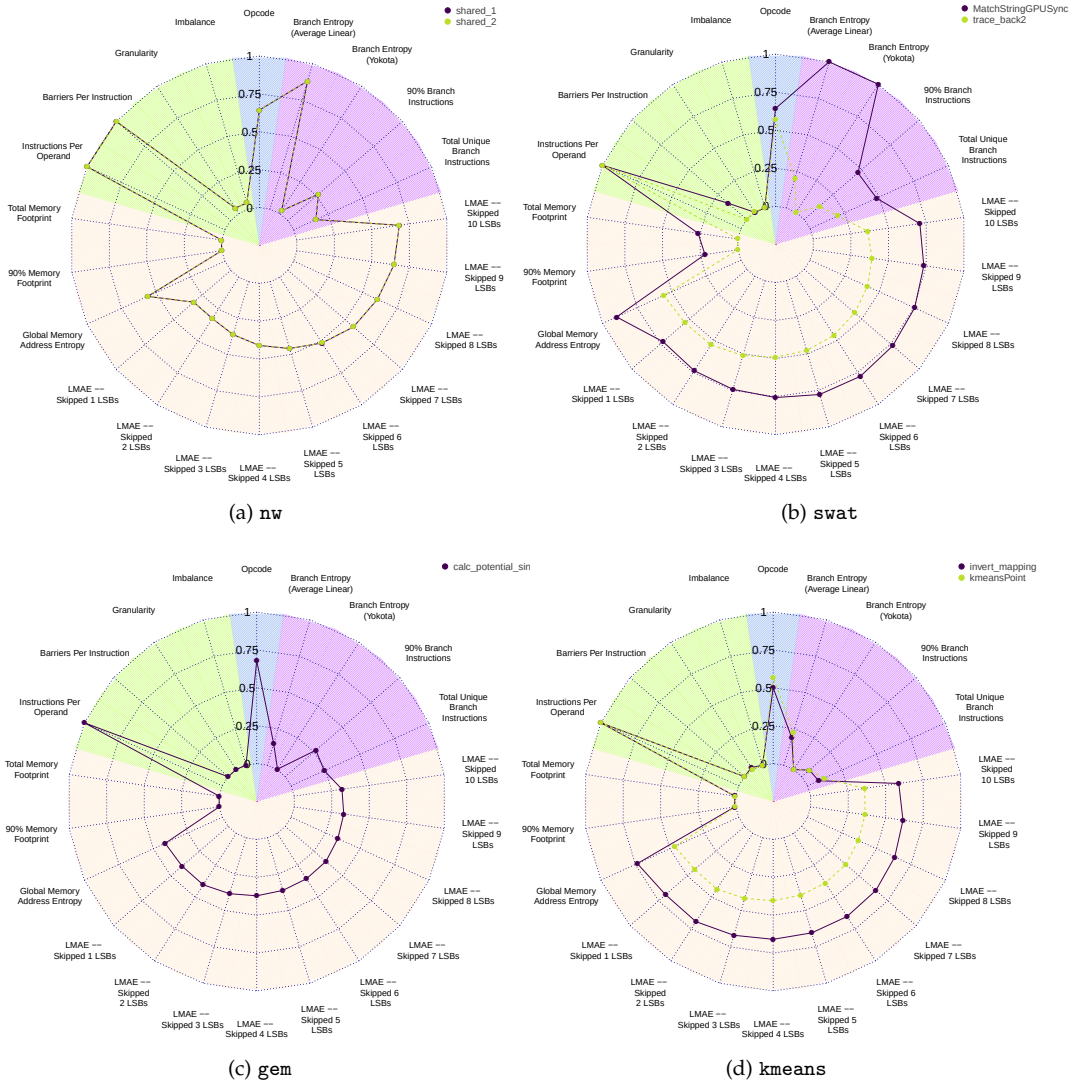
The visual representation provided from the Kiviat diagrams allows the characteristics of OpenCL kernels to be readily assessed and compared.

Finally, we examine the local memory access entropy (LMAE) presented in the Kiviat diagrams in greater detail. Figure 4.3 B) presents a sample of the local 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 a 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. In general, for cache-sensitive workloads – such as LU-Decomposition – a steeper descent between increasing LMAE distances indicates more localized memory accesses, and this corresponds to better cache utilisation when these applications are run on physical OpenCL devices. It is unsurprising that applications with a smaller working memory footprint would exhibit more cache reuse with highly predictable memory access patterns.

4.6 Use Case: AIWC analysis on bioinformatics

A further study of the AIWC feature-space is now performed on bioinformatics type computations to show the benefits of performing AIWC analysis and a sample methodology to examine the change in AIWC metrics over a range of kernels. The bioinformatics subset of applications from the extended OpenDwarfs benchmark suite includes computations used in sequence analysis, biophysics, gene expression/similarity and pattern identification. `nw` and `swat` applications from the Dynamic-Programming dwarf are both directly used in sequence analysis, `gem` from the N-Body-Methods dwarf to cover biophysics computations, `hmm` from the

Graphical-Models dwarf considers both sequence analysis and gene expression. Finally, the MapReduce dwarf features the `kmeans` benchmark, which can be used directly in both pattern identification and gene similarity comparisons. Figures 4.4 and 4.5 present radar/Kiviat diagrams of architecture-independent characteristics collected for each of the bioinformatics benchmarks. All results are presented over a single **small** problem size, and show the multiple kernels required to compute each benchmark application as superimposed plots in the same diagram.



(a) `nw` (b) `swat`

(c) `gem` (d) `kmeans`

Figure 4.4: Architecture-Independent Workload Characterization features for selected bioinformatics benchmarks

Figure 4.4a shows that the `nw` benchmark is characterized by high available thread parallelism (low values for granularity and imbalance) and a very high level of barrier synchronization. This explains its superior performance on Nvidia GPUs compared to CPUs. The Nvidia

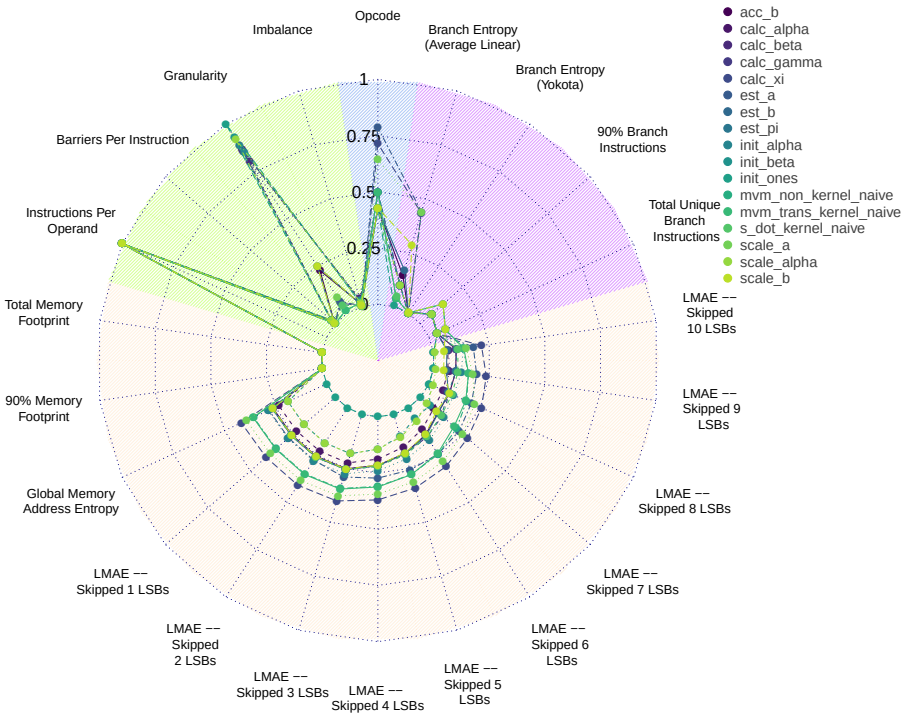


Figure 4.5: Architecture-Independent Workload Characterization features for the `hmm` bioinformatics benchmark

devices examined are roughly two years newer than the AMD GPUs, we expect modern AMD GPUs to form a better comparison.

Figure 4.4b shows that the `swat` benchmark also has a high level of available thread parallelism, however, it has many fewer barriers and a much higher branch entropy. Given this, we expect to see relatively better performance on CPU architectures – the i7600k CPU is two years older than the optimal Nvidia GPUs presented – it would be interesting to repeat this evaluation on a CPU of comparable vintage.

Figure 4.4c shows that the `gem` benchmark is characterized by very high available thread parallelism, and low branch and memory entropies. This makes it ideal for GPU architectures, which is reflected in the superior performance for the modern NVidia GPUs in Figures 3.6 (a).

The `kmeans` benchmark (Figure 4.4d) also has a high level of available parallelism and low branch and memory entropies; as expected, the measurements in (a) – from Figures 3.2 and 3.3 – and show that both modern GPUs and older HPC GPUs perform significantly better than CPUs for this benchmark.

The `hmm` benchmark (Figure 4.5) is composed of a large number of kernels, which differ significantly in granularity. Most kernels have very little available parallelism, suggesting that this benchmark would perform best on CPU architectures with a small number of powerful cores; this is borne out by the measurements in Figure 3.6 (c) which show the smallest

benchmark time was recorded on the powerful i7-6700k CPU.

None of the bioinformatics benchmarks is vectorized (instructions per operand = 1), and therefore fail to take advantage of the floating point capabilities available on CPU and MIC architectures.

4.7 Summary

We have presented the Architecture-Independent Workload Characterization tool (AIWC), which supports the collection of architecture-independent features of OpenCL application kernels. It is the first workload characterization tool to support multi-threaded or parallel workloads. 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. In addition, the architecture independent characteristics of a scientific workload will inform designers and integrators of HPC systems, who must ensure that compute architectures are suitable for the intended workloads.

To identify which AIWC characteristics are the best indicators of opportunities for optimization, we are currently looking at how individual characteristics change for a particular code through the application of best-practice optimizations for CPUs and GPUs (as recommended in vendor optimization guides).

AIWC was also used to evaluate the performance bottle-necks of bioinformatics codes from the EOD suite. When also coupled with the runtime performance results of Chapter 3, it is interesting to note that optimal accelerators are typically GPU based, given the high available thread parallelism and high barrier synchronization counts of many sequencing analysis applications. However, the bioinformatics applications examined contain few kernels with higher branch and memory access entropies, interspersed with the GPU suited workloads, which suggests that CPUs are critical to achieving good performance on these systems. Indeed, partitioning applications by scheduling kernel to their optimal accelerator may generally provide better performance for HPC bioinformatics applications.

Recently, AIWC has been used for predictive modelling on a set of 15 compute devices including CPUs, GPUs and MIC. The AIWC metrics generated from the full set of Extended OpenDwarfs kernels were used as input variables in a regression model to predict kernel execution time on each device [116]. The model predictions differed from the measured experimental results by an average of 1.1%, which corresponds to actual execution time mispredictions of 8 μs to 1 second according to problem size. From the accuracy of these predictions, we can conclude that while our choice of AIWC metrics is not necessarily optimal, they are sufficient to characterize the behaviour of OpenCL kernel codes and identify the optimal execution device for a particular kernel. This is discussed in detail in the next chapter.

Chapter 5

Making Performance Predictions for Scheduling

The OpenCL programming framework is well-suited 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 [98], and choosing the wrong device results in slower and more energy-intensive computation [99]. Thus accurate performance prediction is critical to making optimal scheduling decisions in a heterogeneous supercomputing environment. This work was published in 16th International Conference on High Performance Computing & Simulation, HPCS 2018 [116].

5.1 Methodology

The Architecture-Independent Workload Characterization (AIWC) tool – Chapter 4 – 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 chapter, 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. Our proposed solution uses AIWC as a plugin to the Oclgrind tool, which is already widely used by OpenCL developers. AIWC is used to generate each application signature and the generation of a random forest model to learn each machine profile. The methodology to develop the model is outlined in the remainder of this section. All tools used are open source, and all code is available in the respective repositories: [127] and [130].

5.1.1 Experimental Setup

AIWC – from Chapter 4 – was used to characterize a variety of codes in the OpenDwarfs Extended (EOD) Benchmark Suite – from Chapter 3 – 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 [131] – 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 3.1. 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`.

5.1.2 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 [132] was used for the development of the regression model. The ranger package provides computationally efficient implementations of the Random Forest model [133] 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,
- min.node.size, the minimal node size per tree: ranges from 1 – 50, where 50 is the number of observations per sample.

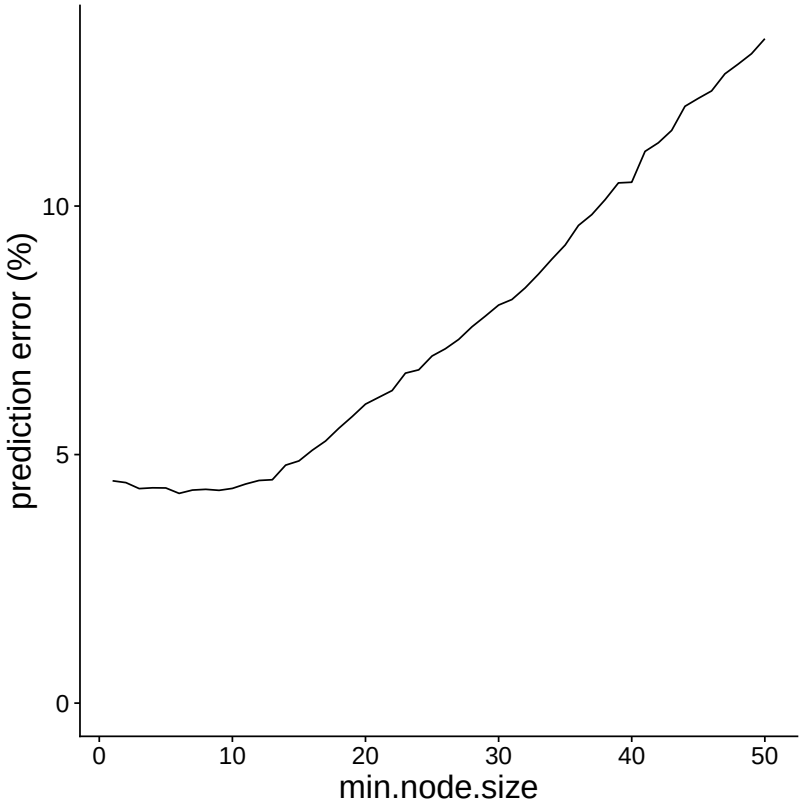


Figure 5.1: Full coverage of min.node.size with fixed tuning parameters: num.trees = 300 and mtry = 30.

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. [134] 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 [135]. 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

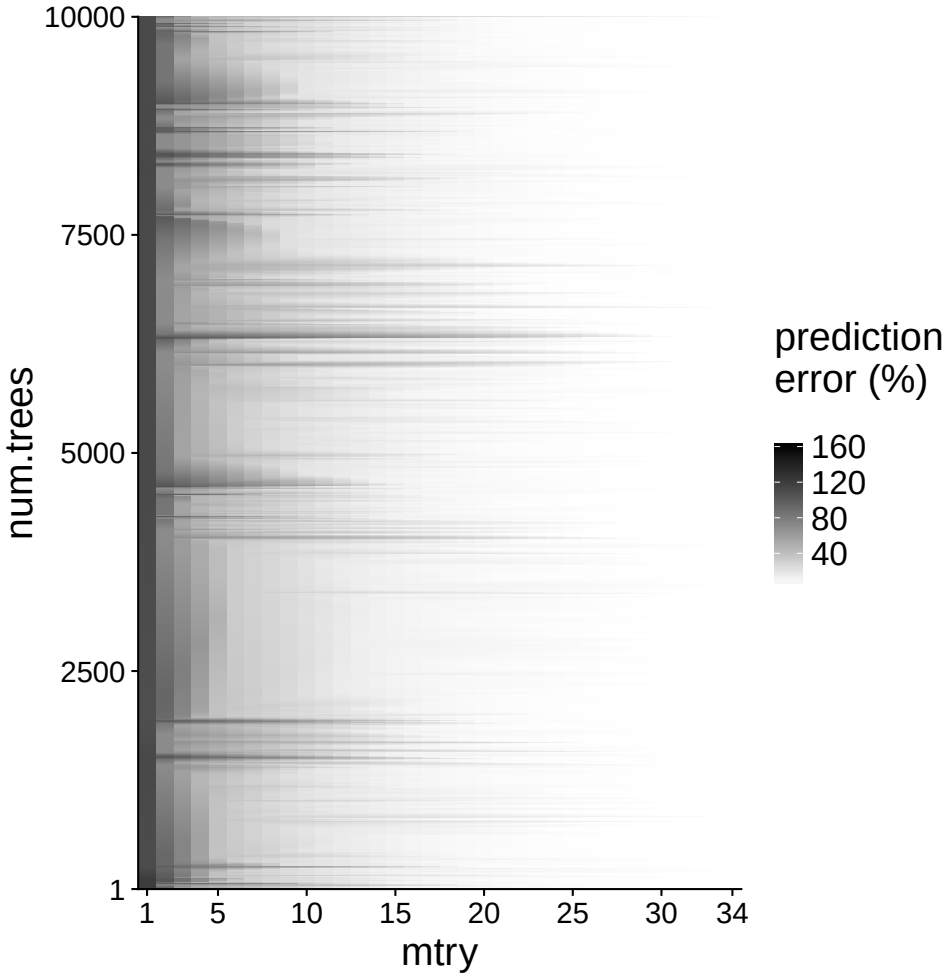


Figure 5.2: Full coverage of num.trees and mtry tuning parameters with min.node.size fixed at 9.

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 5.2 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

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

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.

5.1.3 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 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 5.1. 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.

5.1.4 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.

Table 5.1: 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

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)
```

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 5.1.3.

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.

5.2 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

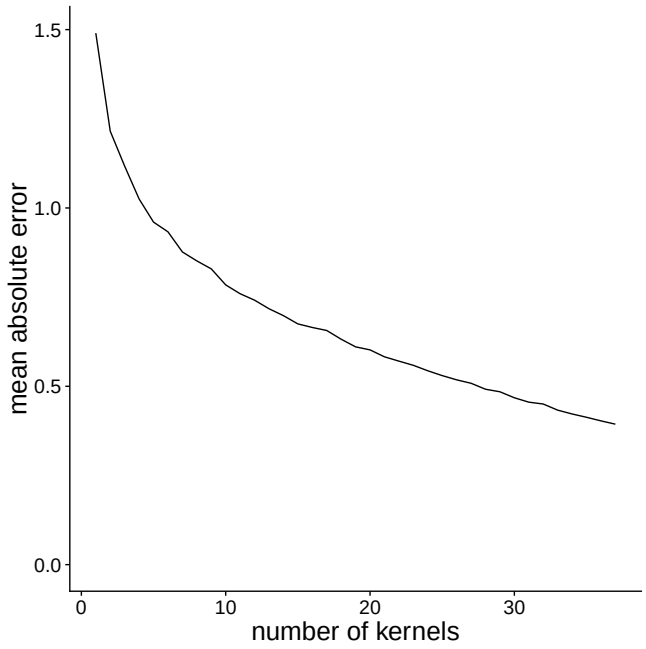


Figure 5.3: Prediction error across all benchmarks for models trained with varying numbers of kernels.

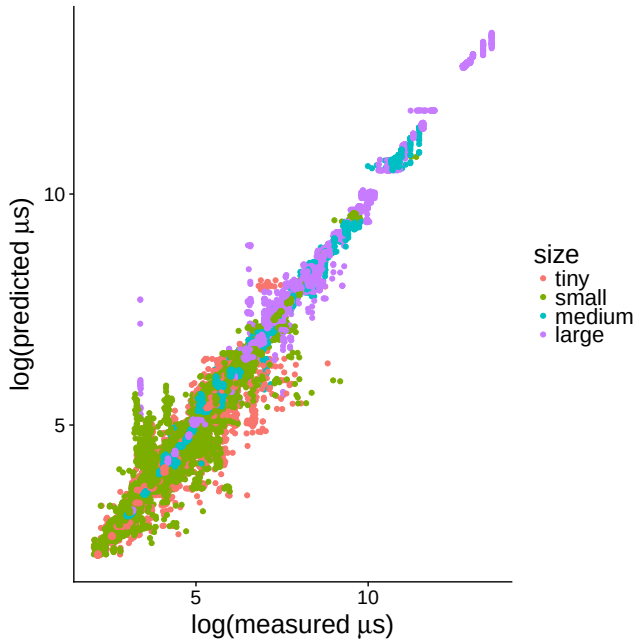


Figure 5.4: Predicted vs. measured execution time for all kernels

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.

5.2.1 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 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.

5.2.2 The benefits of this approach

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

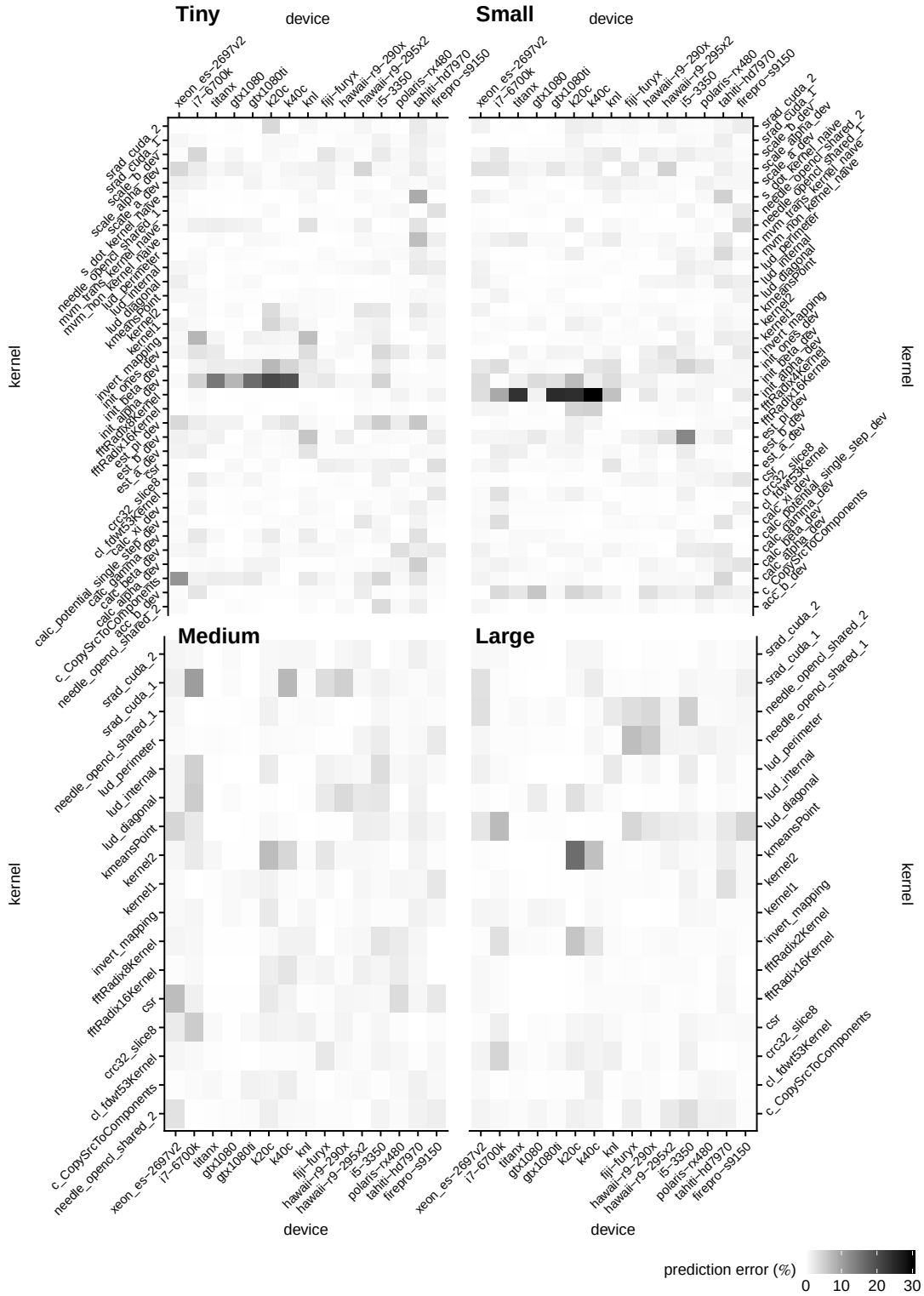


Figure 5.5: Error in predicted execution time for each kernel invocation over four problem sizes

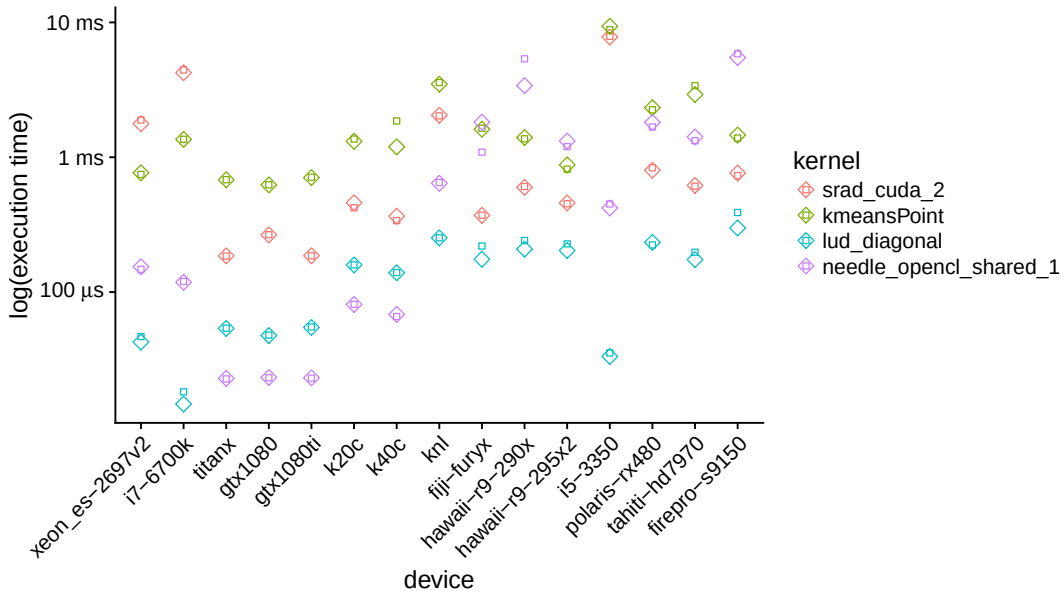


Figure 5.6: Mean measured kernel execution times compared against mean predicted kernel execution times to perform a selection of kernels on large problem sizes across 15 accelerator devices.

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.

5.3 Discussion

If the predictive model were used in a real-world setting – say on a HPC node – the final metrics collected by AIWC could be embedded as a comment at the beginning of each kernel code. This would follow the use-case for AIWC as a plugin to the OpenCL debugger Oclgrind. The developer would first use Oclgrind to debug, optimize and confirm functionality of a kernel, then, enable the AIWC plugin to generate the metrics for the final kernel code. This approach would allow the high accuracy of the predictive model without any significant overhead – metrics are only generated and embedded once per kernel and is done largely automatically, with the guidance of the developer. Separately, the training of the model would only need to occur when the HPC system is updated, such that, a new accelerator device is added, or the drivers, or compiler updated. The extent of model training is also largely automatic following the methodology presented in this thesis. EOD is run over updated devices and the performance runtimes provided into a newly trained regression model – by

following the approach outlined in this Chapter.

Chapter 6

Conclusions and Future Directions

This thesis aims to show that a diverse mix of accelerators offer equally diverse performance, and the collection of architecture-independent features is sufficient to characterize codes, which in turn, can be used to predict performance which can be used to schedule the optimal device. It proposes an extended benchmark suite, which supports many heterogeneous accelerators, and demonstrates the performance of these devices over a large number of codes / kernels. Next, the Architecture-Independent Workload Characterization Tool (AIWC) was proposed to examine the structural characteristics of these kernels can be reduced to a small set of fundamental statistics. These statistics can explain the degree of optimization, structural constraints of the implementation, and be examined to offer an understanding of the algorithm without having to consider the hardware. Finally, we use these metrics to identify the suitability of device in a predictive regression model.

The efforts in extending the OpenDwarfs provides 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 energy and hardware events metrics allow direct performance evaluations to be made between 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. The exploration of the differences in codes was an increasingly compelling justification to explain the performance differences on heterogeneous devices. To this end, the Architecture Independent Workload Characterisation (AIWC) was developed and is capable of identifying the fundamental characteristics of programs free from any specific device. AIWC 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 were then used for creating the prediction model to evaluate the performance of OpenCL kernels on different hardware devices and settings. Such a model was 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.

The use of accelerators is pervasive in HPC and it is in our view that will become more so in the future. We showed that AIWC and the predictive model outline a methodology to achieve better performance on HPC systems composed of heterogeneous accelerators. Fine grained scheduling decisions could be supported with the high accuracy of predictions. This is the obvious next step for the work presented in this thesis and will hopefully become a significant addition to the emerging problem of scheduler selection in HPC workload scheduling. The contents of this thesis fall into the areas of benchmarking, workload characterization, high-performance computing, predictive modelling, software engineering and performance evaluation. Its main goal, is however, improving the performance of large HPC systems by providing useful scheduling information of scientific applications to the most appropriate accelerator. I hope this work will modestly contribute to the increasing interaction between domain sciences and high-performance computing. As tool builders for domain sciences, computer scientists face a challenging task imposed by increasingly complex computer architectures.

The contributions of each Chapter are now discussed in greater detail and concludes with a summary of the future directions currently being pursued as a result of this thesis.

6.1 Extended OpenDwarfs – EOD

We have performed essential curation of the OpenDwarfs benchmark suite. OpenDwarfs was the selected benchmark suite on which to apply our extensions as it: 1) solely focused on an OpenCL implementation, which avoids the fragmentation and different optimizations between language codes common to the SHOC and Rodinia Suites, 2) existing benchmarks had already been classified according to the Dwarf Taxonomy to justify each addition, and, 3) active, this work was the most updated, with the latest use as an evaluation of OpenCL for FPGA devices [58].

We removed hardware specific optimizations from codes that would either diminish performance, or crash the application on other devices. Instead, we added autotuning support to achieve a comparable performance whilst retaining the general purpose nature which is critical to a 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 Chapter 3 we report results for four different problem sizes, selected according to the memory hierarchy of CPU systems as motivated by

Marjanović's findings [118]. These can now be easily adjusted for next generation accelerator systems using the methodology outlined in Section~3.2.4.

We ran many of the benchmarks presented in the original OpenDwarfs [58] 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.

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.

Certain configuration parameters for the benchmarks, e.g. local workgroup size, are amenable to auto-tuning. Many portions of the suite contain auto-tuning for workgroup sizes, however, we plan to fully integrate auto-tuning into the benchmarking framework to provide confidence that the optimal parameters are used for each combination of code and accelerator. Adding auto-tuning support for kernel compiler level optimizations, such as level of loop nesting and unrolling, will be performed in the future.

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. This work resulted in a flexible benchmark suite with results that could can be generated quickly and reliably on a range of accelerators, and formed foundation for testing AIWC and the predictive model.

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.

Other metrics could be added to AIWC. Caparrós Cabezas and Stanley-Marbell [95] 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.

Each OpenCL kernel presented in the Chapter 3 in EOD was been inspected using AIWC. Analysis using AIWC helps understand how the structure of kernels contributes to the varying runtime characteristics between devices, it is envisaged that this will be of greater importance in the future.

A major limitation of running large applications under AIWC is the high memory footprint. Memory access entropy scores require a full recorded trace of every memory access during a kernel's execution. However, a graceful degradation in performance is preferable to an abrupt crash in AIWC if virtual memory is exhausted. For this reason, work is currently being undertaken for an optional build of AIWC with low memory usage by writing these traces to disk. The coverage of characteristics and the suitability AIWC metrics can now be assessed. This was performed in 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.

AIWC is an additional tool to be used by developers and does not attempt to replace classical

device-specific instrumentation and profiling. It can be used with the existing workflow. Indeed, since AIWC is a plugin into Oclgrind which is an OpenCL device simulator, and is mostly used for debugging, the developer may check for memory leaks and race conditions in their code and use the same tool – but with the AIWC argument – to examine its architecture-independent workload characteristics. Optimization could happen based on AIWC metrics, but does not exclude the ability to use hardware performance counters, PIN events or vendor specific profiler tools.

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 Chapter 5.

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.

The suitability of device, and more generally, the optimal type of accelerator, could also be examined using the same predictive modelling methodology.

To use this predictive model in a real-world setting, the final metrics collected by AIWC could be embedded as a comment at the beginning of each kernel code. This approach would allow the high accuracy of the predictive model without any significant overhead – metrics are only generated and embedded once each kernel was written and could be done automatically with AIWC once a developer was happy that a code was ready to be shipped. Separately, the training of the model would only need to occur when the HPC system is updated, such that, a new accelerator device is added, or the drivers, or compiler updated. The extent of model training is also largely automatic. EOD is run over updated devices and the performance runtimes provided into a newly trained regression model. The runtime results from EOD could also be saved in an online corpus / database with the corresponding devices name allowing the automatic training of one large shared model.

7 years of hardware is assuming I rerun results with P100 and xeon gold

Using the same predictive model over run-times generated over 7 years of different hardware

and four processor generations shows both, that OpenCL has reached a position of maturity and stability, and also that the methodology of prediction is sound. Specifically, performing predictions with a single model generated over a large window of time shows that with each generation the individual device prediction accuracy is good and therefore we expect this same methodology to continue to be equally accurate on future systems.

6.4 Future Directions

Following the work presented in this thesis, five additional research topics are actively being pursued.

6.4.1 Finding holes in benchmarks: Evaluating the coverage and corresponding performance predictions for conventional vs synthetic benchmarking

This work is currently focused on augmenting EOD with synthetic benchmarks. The predictive model is used to make predictions on concealed codes against the trained set of EOD runtime results. These unknown codes are randomly generated using the Machine-Learning OpenCL kernel generation framework – CLgen by 136 and in this sense are treated as synthetic benchmarks. The extensive set of kernels are automatically produced using a training corpus of all OpenCL applications available on GitHub. The previous success of this model to predict execution times simultaneously across many devices with high accuracy has lead us to believe that the Extended OpenDwarfs Benchmark Suite is a good platform for training – it adequately covers the feature space for many scientific problems typical of the HPC setting. However, it is believed that testing the model with synthetic benchmarking may identify gaps in the selected applications, since they are poorly predicted in the model. These poorly predicted kernels could be added back into the EOD benchmark suite – thus better encompassing work expected to be run on these accelerator devices.

6.4.2 AIWC for the Masses: Towards language-agnostic architecture-independent workload characterization

CUDA is a widely used competitor to OpenCL additionally, other languages more commonly used in HPC is OpenMP and OpenACC. The latter offers an accelerator directives approach to offload work to accelerators. It would be useful to perform the same architecture-independent workload characterization on all these languages. Thankfully, there exist source-to-source translation tools such as Coriander [137] which allows a largely automatic conversion from CUDA to OpenCL codes. Also, LLVM is the common intermediate-representation or backend between OpenMP, OpenACC and OpenCL. An active area of work is currently placed on

writing an LLVM pass to generating OpenCL device payloads for AIWC from OpenMP and OpenACC.

6.4.3 Examining the Characteristics of Scientific Codes in Supercomputing with AIWC

We are in the early stages of collaborating with colleagues at Shanghai Jiao Tong University's HPC Center, in examining the performance properties of scientific codes on the Sunway TaihuLight, Tianhe-2A and future Tianhe-3 supercomputer systems. Porting large HPC codes, such as those seen in weather forecasting and bioinfomatics, from conventional CPU architectures to accelerators is intensive on the developer. However, this work must be done to accommodate the transition of supercomputers from the conventional historical many homogeneous CPU cores to multiple accelerators per node. Thankfully, many of these codes were written in OpenMP – and more recently with OpenACC – these directives based approaches support offloading to accelerator devices. Once the AIWC for the Masses work has been completed, from the previous section, we can perform language-agnostic architecture-independent workload characterization on many of these applications. Aside from assisting in scheduling the work presented in this thesis, AIWC and the predictive model could be used to identify primary characteristics of codes run on supercomputers. For instance, if the research institutes which provide access to a supercomputer know that 80% of an application is most suited to a GPU the best theoretical machine could contain 4 GPUs to 1 CPU on a node. This can be used to suggest optimal configurations of accelerators for the next-generation of supercomputers. Additionally, the AIWC metrics can be used to inform a developer around general suitability of a kernel to accelerator before devoting a large amount of time optimizing kernels that already perform sufficiently on the current architecture. We are also pursuing collaboration with some of the US National Laboratories who have expressed interest in using AIWC to examine the characteristics of the codes run on some of the largest supercomputers in the world. Finally, optimization strategies could be hinted to by AIWC but this is discussed as the next area of future work.

6.4.4 Guiding Device Specific Optimization using Architecture-Independent Metrics

We believe AIWC will also be useful in guiding device-specific optimization by providing feedback on how particular optimizations change performance-critical characteristics. To identify which AIWC characteristics are the best indicators of opportunities for optimization, we are currently looking at how individual characteristics change for a particular code through the application of best-practice optimizations for CPUs and GPUs (as recommended in vendor optimization guides).

The methodlogy is evaluated by comparing the suggested programming practices of CPU and

GPU specific algorithmic optimisations, on portable OpenCL codes, and how architectural independent analysis can identify poor adherence to these practices. A selection of each of these practices was taken from their respective source code and ported to OpenCL. The practices for CPU devices was taken from the "Intel 64 and IA-32 Architectures Optimization Reference Manual", while the GPU practices were based on the "CUDA C Best Practices Guide, Design Guide"

Cluster analysis of this feature-space may be performed and could lay the way towards the automatic classification of an application, in terms of which Dwarf it best represents or perhaps a more descriptive Taxonomy than the Berkeley Dwarfs.

6.4.5 Faster FPGA development with AIWC and the Predictive Model

Finally, complicated OpenCL codes can take many hours – if not days – to compile for FPGA devices. This makes the trial-and-error approach commonly taken when optimizing a code for a device untenable. Given the accuracy of the predictive model, there is a use-case to use AIWC to augment this workflow. Speculative optimization changes could be made to an OpenCL code, the AIWC metrics generate and query the predictive model the result predicted device execution time would be the indicator of success of the optimization tactic. This would potentially take seconds instead of the long compile times when evaluating FPGA performance. A limitation of this approach is the initial generation of the response times required to train the model – it would take weeks to collect enough data around run-times to compile the full EOD benchmark suite.

References

1. P. Colella, "Defining software requirements for scientific computing, 2004," *DARPA HPCS presentation*.
2. D. Patterson, K. Keutzer, K. Asanovic, K. Yelick, and R. Bodik, "Dwarf Mine," *Berkeley Wiki*. http://view.eecs.berkeley.edu/wiki/Dwarf_Mine, Dec-2006.
3. 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.
4. J. Dongarra, "Report on the sunway taihulight system," *PDF). www. netlib. org. Retrieved June*, vol. 20, 2016.
5. E. L. Padoin, L. L. Pilla, M. Castro, F. Z. Boito, P. O. A. Navaux, and J.-F. Méhaut, "Performance/energy trade-off in scientific computing: The case of arm big. little and intel sandy bridge," *IET Computers & Digital Techniques*, vol. 9, no. 1, pp. 27–35, 2014.
6. R. V. Aroca and L. M. G. Gonçalves, "Towards green data centers: A comparison of x86 and arm architectures power efficiency," *Journal of Parallel and Distributed Computing*, vol. 72, no. 12, pp. 1770–1780, 2012.
7. N. Rajovic, L. Vilanova, C. Villavieja, N. Puzovic, and A. Ramirez, "The low power architecture approach towards exascale computing," *Journal of Computational Science*, vol. 4, no. 6, pp. 439–443, 2013.
8. K. Keipert *et al.*, "Energy-efficient computational chemistry: Comparison of x86 and arm systems," *Journal of chemical theory and computation*, vol. 11, no. 11, pp. 5055–5061, 2015.
9. V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *High performance computing, networking, storage and analysis, 2008. sc 2008. international conference for*, 2008, pp. 1–11.
10. S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with gpu accelerators," in *Parallel & distributed processing, workshops and phd forum (ipdpsw), 2010 ieee international symposium on*, 2010, pp. 1–8.
11. D. Komatitsch, G. Erlebacher, D. Göddeke, and D. Michéa, "High-order finite-element seismic wave propagation modeling with mpi on a large gpu cluster," *Journal of computational physics*, vol. 229, no. 20, pp. 7692–7714, 2010.
12. R. Nicolescu, "Structured grid algorithms modelled with complex objects," in *International conference on membrane computing*, 2015, pp. 321–337.

13. D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *ACM sigplan notices*, 2012, vol. 47, pp. 117–128.

14. P. Springer, "Berkeley's dwarfs on cuda," *RWTH Aachen University, Tech. Rep*, 2011.

15. S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA tensor core programmability, performance & precision," *arXiv preprint arXiv:1803.04014*, 2018.

16. S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid gpu accelerated manycore systems," *Parallel Computing*, vol. 36, nos. 5-6, pp. 232–240, 2010.

17. A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Analysis and design techniques towards high-performance and energy-efficient dense linear solvers on gpus," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

18. A. Sodani *et al.*, "Knights landing: Second-generation intel xeon phi product," *Ieee micro*, vol. 36, no. 2, pp. 34–46, 2016.

19. J. Dongarra *et al.*, "Hpc programming on intel many-integrated-core hardware with magma port to xeon phi," *Scientific Programming*, vol. 2015, p. 9, 2015.

20. M. Rajan, D. Doerfler, and S. Hammond, "Trinity benchmarks on intel xeon phi (knights corner)."

21. K. Antypas, N. Wright, N. P. Cardo, A. Andrews, and M. Cordery, "Cori: A cray xc pre-exascale system for nersc," *Cray User Group Proceedings. Cray*, 2014.

22. W. Akram, T. Hussain, and E. Ayguade, "FPGA and arm processor based supercomputing," in *Computing, mathematics and engineering technologies (iCoMET), 2018 international conference on*, 2018, pp. 1–5.

23. N. Fujita *et al.*, "Accelerating space radiative transfer on fpga using opencl," in *Proceedings of the 9th international symposium on highly-efficient accelerators and reconfigurable technologies*, 2018, p. 6.

24. M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning." in *OSDI*, 2016, vol. 16, pp. 265–283.

25. E. Gallopoulos, B. Philippe, and A. H. Sameh, *Parallelism in matrix computations*. Springer, 2016.

26. G. Mitra, J. Bohmann, I. Lintault, and A. P. Rendell, "Development and application of a hybrid programming environment on an arm/dsp system for high performance computing," in *2018 ieee international parallel and distributed processing symposium (ipdps)*, 2018, pp. 286–295.

27. J. Maqbool, S. Oh, and G. C. Fox, "Evaluating arm hpc clusters for scientific workloads," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 5390–5410, 2015.

28. N. Rajovic, A. Rico, N. Puzovic, C. Adeniyi-Jones, and A. Ramirez, "Tibidabo1: Making the case for an arm-based hpc system," *Future Generation Computer Systems*, vol. 36, pp.

322–334, 2014.

29. M. Jarus, S. Varrette, A. Oleksiak, and P. Bouvry, "Performance evaluation and energy efficiency of high-density hpc platforms based on intel, amd and arm processors," in *European conference on energy efficiency in large scale distributed systems*, 2013, pp. 182–200.

30. M. Feldman, "Cray to deliver ARM-powered supercomputer to UK consortium," *TOP500 Supercomputer Sites*, Jan. 2017.

31. S. W. Lacy, J. P. Noe, J. B. Ogden, and S. D. Hammond, "Building 725 astra and vanguard." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2018.

32. S. McIntosh-Smith, J. Price, T. Deakin, and A. Poenaru, "Comparative benchmarking of the first generation of hpc-optimised arm processors on isambard."

33. T. Morgan, "Inside Japan's future exascale ARM supercomputer," *The Next Platform*. https://www.nextplatform.com/2016/06/23/inside-japans-future-exaflops-arm-supercomputer/; Stackhouse Publishing Inc., Jun-2016.

34. F. Simula *et al.*, "Real-time cortical simulations-energy and interconnect scaling on distributed systems," *arXiv preprint arXiv:1812.04974*, 2018.

35. O. Villa *et al.*, "Scaling the power wall: A path to exascale," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2014, pp. 830–841.

36. M. Feldman, "TOP500 meanderings: Supercomputers take big green leap in 2017," *TOP500 Supercomputer Sites*. https://www.top500.org/news/top500-meanderings-supercomputers-take-big-green-leap-in-2017/; Top500.org, Sep-2017.

37. T. Declerck *et al.*, "Cori - a system to support data-intensive computing," *Proceedings of the Cray User Group*, p. 8, 2016.

38. T. Morgan, "NVLink takes GPU acceleration to the next level," *The Next Platform*, May 2016.

39. T. Morgan, "The Power9 rollout begins with Summit and Sierra supercomputers," *The Next Platform*. https://www.nextplatform.com/2017/09/19/power9-rollout-begins-summit-sierra/; Stackhouse Publishing Inc., Sep-2017.

40. T. Morgan, "China arms upgraded tianhe-2A hybrid supercomputer," *TOP500 Supercomputer Sites*. https://www.nextplatform.com/2017/09/20/china-arms-upgraded-tianhe-2a-hybrid-supercomputer/; Top500.org, Sep-2017.

41. M. Feldman, "Prototypes of china's exascale supercomputers point to some new realities," *TOP500 Supercomputer Sites*. https://www.top500.org/news/prototypes-of-chinas-exascale-supercomputers-point-to-some-new-realities/; Top500.org, Aug-2018.

42. 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.

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

44. 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.

45. 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.

46. 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.

47. J.-J. Li, C.-B. Kuan, T.-Y. Wu, and J. K. Lee, "Enabling an opencl compiler for embedded multicore dsp systems," in *Parallel processing workshops (icppw), 2012 41st international conference on*, 2012, pp. 545–552.

48. D. H. Bailey *et al.*, "The NAS parallel benchmarks," *International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

49. 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.

50. 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.

51. Y. Sun *et al.*, "Hetero-mark, a benchmark suite for cpu-gpu collaborative computing," in *IEEE international symposium on workload characterization (iiswc)*, 2016.

52. 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.

53. C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on parallel architectures and compilation techniques*, 2008, pp. 72–81.

54. 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.

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

56. S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Workload characterization (iiswc), 2010 ieee international symposium on*, 2010, pp. 1–11.

57. 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.

58. 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.

59. 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.

60. 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.

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

62. 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.

63. 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.

64. A. Sembrant, "Hiding and reducing memory latency: Energy-efficient pipeline and memory system techniques," PhD thesis, Acta Universitatis Upsaliensis, 2016.

65. S. K. Muller and U. A. Acar, "Latency-hiding work stealing: Scheduling interacting parallel computations with work stealing," in *Proceedings of the 28th acm symposium on parallelism in algorithms and architectures*, 2016, pp. 71–82.

66. C. Lively, X. Wu, V. Taylor, S. Moore, H.-C. Chang, and K. Cameron, "Energy and performance characteristics of different parallel implementations of scientific applications on multicore systems," *The International Journal of High Performance Computing Applications*, vol. 25, no. 3, pp. 342–350, 2011.

67. B. Johnston, B. Lee, L. Angove, and A. Rendell, "Embedded accelerators for scientific high-performance computing: An energy study of OpenCL Gaussian elimination workloads," in *International conference on parallel processing workshops (icppw)*, 2017, pp. 59–68.

68. B. Johnston and E. C. McCreath, "Parallel huffman decoding: Presenting fast and scalable algorithm for increasingly multicore devices," in *International symposium on parallel and*

distributed processing with applications (ispa), 2017.

69. A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "Scaling hypre's multigrid solvers to 100,000 cores," in *High-performance scientific computing*, Springer, 2012, pp. 261–279.

70. M. J. Abraham *et al.*, "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers," *SoftwareX*, vol. 1, pp. 19–25, 2015.

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

72. H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Computer architecture (isca), 2011 38th annual international symposium on*, 2011, pp. 365–376.

73. 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.

74. P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming," *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2012.

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

76. 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.

77. 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.

78. 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.

79. 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.

80. 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.

81. 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.

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

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

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

85. 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.

86. 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.

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

88. 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.

89. 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.

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

91. J. H. Lee, N. Nigania, H. Kim, K. Patel, and H. Kim, "OpenCL performance evaluation on modern multicore cpus," *Scientific Programming*, vol. 2015, p. 4, 2015.

92. 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.

93. 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.

94. 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.

95. 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.

96. S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," *Communications of the Association for Computing Machinery*, 2009.

97. G. Hager, J. Treibig, J. Habich, and G. Wellein, "Exploring performance and power properties of modern multi-core chips via simple machine models," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 2, pp. 189–210, 2013.

98. 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.

99. 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.

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

101. 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.

102. 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.

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

104. 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.

105. 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.

106. 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.

107. 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.

108. 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.

109. A. Karami, S. A. Mirsoleimani, and F. Khunjush, "A statistical performance prediction model for opencl kernels on nvidia gpus," in *Computer architecture and digital systems (cads), 2013 17th csi international symposium on*, 2013, pp. 15–22.

110. G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "GPGPU performance and power estimation using machine learning," in *High performance computer architecture (hpca), 2015 ieee 21st international symposium on*, 2015, pp. 564–576.

111. A. Shetty, "X-map a performance prediction tool for porting algorithms and applications to accelerators," 2017.

112. S. Che and K. Skadron, "BenchFriend: Correlating the performance of gpu benchmarks," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 238–250, 2014.

113. M. Boyer, J. Meng, and K. Kumaran, "Improving gpu performance prediction with data transfer modeling," in *Parallel and distributed processing symposium workshops & phd forum (ipdpsw), 2013 ieee 27th international*, 2013, pp. 1097–1106.

114. D. Shelepov *et al.*, "HASS: A scheduler for heterogeneous multicore systems," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 66–75, Apr. 2009.

115. S.-Y. Lee and C.-J. Wu, "Performance characterization, prediction, and optimization for heterogeneous systems with multi-level memory interference," in *Workload characterization (iiswc), 2017 ieee international symposium on*, 2017, pp. 43–53.

116. B. Johnston, G. Falzon, and J. Milthorpe, "OpenCL performance prediction using architecture-independent features," in *2018 international conference on high performance computing & simulation (hpcs)*, 2018, pp. 561–569.

117. B. Johnston and J. Milthorpe, "Dwarfs on accelerators: Enhancing OpenCL benchmarking for heterogeneous computing architectures," in *Proceedings of the* 47^{th} *international conference on parallel processing companion*, 2018, pp. 4:1–4:10.

118. 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.

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

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

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

122. 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.

123. 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.

124. 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.

125. 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.

126. 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.

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

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

129. B. Johnston and J. Milthorpe, "AIWC: OpenCL-based Architecture-Independent Workload Characterisation," *LLVM-HPC2018 Workshop, held in conjunction with the 30th International Conference for High Performance Computing, Networking, Storage, and Analysis (SC18)*, May 2018.

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

131. B. Johnston and J. Milthorpe, "Dwarfs on accelerators: Extending OpenCL benchmarking for heterogeneous computing architectures," *ICPP '18 Proceedings of the 47th International Conference on Parallel Processing Companion*, 2018.

132. 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.

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

134. 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.

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

136. C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "Synthesizing benchmarks for predictive modeling," in *CGO*, 2017.

137. H. Perkins, "CUDA-on-cl: A compiler and runtime for running nvidia&Reg; cuda&Trade; c++11 applications on opencl&Trade; 1.2 devices," in *Proceedings of the 5th international workshop on opencl*, 2017, pp. 6:1–6:4.