Characterizing and Predicting Scientific Workloads for Heterogeneous Computing Systems

Beau Johnston

February 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 my supervisors Josh, Greg and Alistair. Josh, you've been an excellent friend and mentor. I wouldn't have a thesis if you weren't so supportive or engaged in this work – it has actually been a lot of fun working so closely and bouncing ideas off each other. Greg, you've always supported me, technically and emotionally. I feel privileged that we've been able to collaborate so closely despite being 750 kilometers apart. Alistair, thank you for bringing me to the ANU and being so patient with this perpetual student, you have shaped me – for better or worse – into thinking like a cynical scientist. You've taught me the basics and still somehow found the time – and funding – for me.

You have all had to repeat yourselves far too frequently and still somehow maintained the vigour to keep me on track – tempering my wild ideas and pushing me through the writing up process. I wouldn't have considered spending my life as a researcher or scientist if my role-models had been any less kind or brilliant. Thank you so much!

Thank you to my loving family – Sue, Paul, Eddie, Betty and Ron – I've been incredibly lucky to come from such a supportive environment, there is no way I would have had the confidence to even enrol in university much less stick through a PhD program.

My long-time suffering partner Flo... sorry! You have been on the front-lines of work complaints, frustrations and my sheer single mindedness to get this bloody thing finished for far too long. I can't imagine how you still have the determination to start your own PhD shortly – I'm very proud of you. Also, sorry about coming home at 5am with no teeth – and the PhD-esq Stockholm syndrome that caused it.

I would also like to thank all my office buddies over the years – Brian, Edwin, Jess, Shayanti, Gaurav, Anish, Luke, Kunshan and Guyver – without the support and communal suffering you all provided I would have abandoned my PhD a long time ago. At various stages, you have all fed me, helped me de-stress by running me round a badminton court, listened to my ample complaints and discuss my – from your perspectives what must be especially boring – ideas. You have all been amazingly supportive friends and family and I love you all very much.

Thank you to my colleagues at The University of Bristol's High Performance Computing Research group for the use of `The Zoo` Research cluster which was used for the experimental evaluation, it 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 in heterogeneity is explained by the nature of supercomputing workloads – certain devices offer acceleration, or a shorter time to completion, for particular application programs. Certain characteristics of these programs are fixed and impose fundamental limitations on the workloads regardless of which accelerator is used for the computation; for instance, a graph traversal program always exhibits the same high-branch and low-computation properties regardless of what device is used to execute it. To support efficient scheduling on High Performance Computing (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. On HPC systems a single node may feature a Graphics Processing Unit (GPU), a Central Processing Unit (CPU), and a Field-Programmable Gate Array (FPGA) or Many Integrated Core (MIC) device. This work presents a device independent predictor – a methodology to use device-independent characteristics of scientific codes to select the optimal accelerator device with regard to execution time or energy expenditure.

Open Compute Language (OpenCL) is a programming model designed to facilitate the development of application codes capable of running on multiple different devices. First released in late 2008, it defines a C-like language used to write kernels that can be compiled to run on the different devices. Implementations of the current release (2.2) exist for CPUs, GPUs, FPGAs and the Intel MIC systems, and as such, there is increasing interest in the use of OpenCL for developing scientific applications designed to run on next-generation supercomputer systems.

This thesis seeks to use the device-independent characteristics of an OpenCL code to select the optimal accelerator device on which to execute each OpenCL kernel. Consideration is given both to execution time and energy usage.

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 the portability and robustness of the applications, the correctness of results and the choice of problem size, and diversity of coverage through the inclusion of additional application patterns. This work manifests in performance measurements on a set 15 devices and over 12 applications.

We next present our 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 28 metrics 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 the 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 train a random forest model capable of predicting accelerator execution times. We use this model 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 using AIWC to allow performance prediction across the full range of modelled devices. The results suggest that this methodology supports the correct selection of the most appropriate device for a previously unencountered code, and is highly relevant to efficiently schedule codes to emerging heterogeneous supercomputing systems.

Contents

Declaration

Acknowledgements

Abstract

1 Introduction **1**

1.1 Context . . . 2

1.2 Thesis Contributions . . . 3

1.3 Thesis Structure . . . 4

1.4 Publications . . . 4

2 Background Information and Related Work **5**

2.1 The Dwarf Taxonomy . . . 5

2.2 Accelerator Architectures in HPC . . . 6

2.3 The Open Compute Language (OpenCL) . . . 13

2.4 Benchmark Suites . . . 15

2.4.1 Rodinia . . . 16

2.4.2 OpenDwarfs . . . 17

2.4.3 SHOC . . . 17

2.5 Hardware Performance and Scaling . . . 18

2.6 OpenCL Performance . . . 20

2.6.1 Autotuning . . . 20

2.6.2 Phase-Shifting . . . 21

2.6.3 Measurements . . . 22

2.7 Offline Ahead-of-Time Analysis . . . 22

2.8 Program Diversity Analysis and Characterization . . . 23

2.8.1 Microarchitecture-Independent Workload Characterization . . . 23

2.8.2 Architecture Independent Workload Characterization . . . 24

2.8.3 Workload Characterization for Benchmark Diversity Analysis . . . 25

2.9 Performance Prediction for Heterogeneous Architectures . . . 26

2.10 Scheduling for Heterogeneous HPC Systems . . . 28

3 Extending the OpenDwarfs Benchmark Suite **30**

3.1 Extending the OpenDwarfs Benchmark Suite . . . 31

3.2 Experimental Setup . . . 33

3.2.1 Hardware 33
3.2.2 Software 34
3.2.3 Measurements 34
3.2.4 Problem Size 35
3.2.5 kmeans 35
3.2.6 lud, fft, srad, crc, nw 36
3.2.7 csr 36
3.2.8 dwt 37
3.2.9 gem, nqueens, hmm, swat 37
3.2.10 bfs, cfd, tdm 38
3.2.11 Summary of Benchmark Settings 38
3.3 Results 39
3.3.1 Time 40
3.3.2 Energy 43
3.4 Discussion 44

4 AIWC: OpenCL based Architecture Independent Workload Characterization 51
4.1 Metrics 53
4.2 Implementation 55
4.3 Demonstration 58
4.4 Detailed Analysis of LU Decomposition Benchmark 59
4.5 Use Case: AIWC analysis of OpenDwarf bioinformatics related benchmarks . . 64
4.6 Usage and Limitations 66
4.7 Summary 70

5 Making Performance Predictions for Scheduling 71
5.1 Model Development 72
5.1.1 Experimental Setup 72
5.1.2 Constructing the Random Forest Performance Model 73
5.1.3 Parameters for the Random Forest Performance Model 76
5.1.4 Tuning the Random Forest Model 76
5.2 Evaluation 78
5.2.1 Predicting Kernel Execution Time 78
5.2.2 Choosing The Optimal Accelerator for a Kernel 80
5.3 Discussion 82

6 Conclusions and Future Directions 84
6.1 Extended OpenDwarfs – EOD 85
6.2 AIWC 86
6.3 Performance Prediction 87
6.4 Future Directions 88
6.4.1 EOD 88
6.4.2 AIWC 89

6.4.3 Performance Predictions 89
6.4.4 Finding holes in benchmarks: Evaluating the coverage and corresponding performance predictions for conventional vs synthetic benchmarking 90
6.4.5 AIWC for the Masses: Towards language-agnostic architecture-independent workload characterization 90
6.4.6 Examining the Characteristics of Scientific Codes in Supercomputing with AIWC 91
6.4.7 Guiding Device Specific Optimization using Architecture-Independent Metrics 91
6.4.8 Faster FPGA development with AIWC and the Predictive Model 91
6.5 Closing Remarks 92

Appendices **92**

A Time Results **95**

B Linear Model Fitting **103**

C Diversity Analysis **107**

D Abbreviations **111**

References **113**

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. 11
2.2 Power efficiency (GFlops/Watt) of using accelerators in the Top500 supercomputers over time. 12

3.1 Kernel execution times for the medium problem size benchmarks on different accelerator devices. 46
3.2 Kernel execution times for the single sized benchmarks on various accelerator devices. 47
3.3 Kernel execution times for the **crc** benchmark on different hardware platforms. 48
3.4 Kernel execution times for the **kmeans** benchmark on different hardware platforms. 49
3.5 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. 50

4.1 Selected AIWC metrics from each category over all kernels and 4 problem sizes. 60
4.2 A) and B) show the AIWC features of the `diagonal` and `internal` kernels of the LUD application over all problem sizes. 61
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. 62
4.4 Architecture-Independent Workload Characterization features for selected bioinformatics benchmarks. 65
4.5 Architecture-Independent Workload Characterization features for the `hmm` bioinformatics benchmark. 66
4.6 EOD runtimes for **small** problem sized bioinformatics benchmarks. 67

5.1 Full coverage of min.node.size with fixed tuning parameters: num.trees = 300 and mtry = 30. 74
5.2 Full coverage of num.trees and mtry tuning parameters with min.node.size fixed at 9. 75
5.3 Prediction error across all benchmarks for models trained with varying numbers of kernels. 79
5.4 Predicted vs. measured execution time (in log(µs)) for all kernels. 79

5.5 Error in predicted execution time for each kernel invocation over four problem sizes. 81
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. The square indicates the mean measured time, and the diamond indicates the mean predicted time. 82

A.1 Kernel execution times for the **tiny** and **small** problem sizes of the `kmeans`, `lud`, `csr` and `dwt` benchmarks on different hardware platforms. 98
A.2 Kernel execution times for the **medium** and **large** problem sizes of the `kmeans`, `lud`, `csr` and `dwt` benchmarks on different hardware platforms. 99
A.3 Kernel execution times for the **tiny** and **small** problem sizes of the `fft`, `srad` and `nw` benchmarks on different hardware platforms. 100
A.4 Kernel execution times for the **medium** and **large** problem sizes of the `fft`, `srad` and `nw` benchmarks on different hardware platforms. 101
A.5 Single problem sized benchmarks of kernel execution times on different hardware platforms. 102

B.1 Predicted vs. measured execution time (in log(μs)) for all kernels and devices with the GLM model. 104
B.2 Distribution of prediction errors with the GLM model, presented in a log scale. 104
B.3 Prediction intervals for the `kmeansPoint` kernel on the GTX1080Ti GPU over four problem sizes. 105
B.4 Error in predicted execution time with a GLM model for each kernel invocation over four problem sizes. 106

C.1 Biplot Principal Components of AIWC metrics over all application kernels and all problem sizes. 108
C.2 The proportion of explained variance of each Principal Component. 109
C.3 The t-SNE with k-means cluster results to show Principal Components. 110

List of Tables

2.1 The Berkeley Dwarfs and their limiting factors. 6

3.1 List of Extended OpenDwarfs Applications and their respective dwarfs. 32
3.2 Hardware utilized during the Extended OpenDwarfs Benchmark Suite evaluation. 33
3.3 The different problem sizes in the Extended OpenDwarfs adjusted by selecting the workload scale parameter (Φ). 38
3.4 Program Arguments for benchmarks in the Extended OpenDwarf Suite. 39

4.1 ISA-Independent Workload Characterization metrics. 53
4.2 Metrics collected by the AIWC tool ordered by type. 54
4.3 Overhead of the AIWC tool on the `fft` benchmark and the Intel i7-6700K CPU. 68

5.1 Optimal tuning parameters from the same starting location for all models omitting each individual kernel. 77

Chapter 1

Introduction

Supercomputers are used in computationally intensive tasks and are a critical tool in current scientific research. They are essential in simulations for quantum mechanics, weather forecasting, climate research, oil and gas exploration and molecular modelling. However the largest supercomputers are requiring huge amounts of electricity to operate, for example, the current world's fastest supercomputer, Summit [1], requires 8.8 MW to power, which in terms of the average Australian home (13.7 kWh per day) could power 15500 homes. To reduce this large energy footprint 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 supercomputers has been designed to incorporate a greater number of accelerators, and varying types within a node. For instance, the CAPI [2] and NVLINK [3] technologies included in the latest IBM POWER9 [4] processor offers a high-speed interconnect which allows the rapid movement of 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 future direction of supercomputing. However, this development is recent, and as such the scheduling of workloads to the most suitable accelerator becomes an issue. Without addressing this the cost of exascale computing and the corresponding energy efficiency will be prohibitive.

This thesis will argue that certain 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 and that 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 a given code – provided the feature metrics are pre-generated – on 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 enable scheduling of codes to the most appropriate device to achieve better performance and utilization on the next-generation of supercomputers.

1.1 Context

Accelerators can be programmed in a variety of different languages – CUDA [5] for Nvidia GPUs, ROCm [6] for AMD devices, OpenMP [7] on Intel CPUs and the MIC. The Open Compute Language (OpenCL) [8] 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 parts, the host and the accelerator/device side. The developer is responsible for allocating and transferring memory between the host and the 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. Specifically, a kernel does not suffer from the phase-transitions that are common when looking at larger scientific codes. The composition of all kernels forms an accelerator agnostic implementation for full scientific applications.

A benefit of the fixed/static nature of OpenCL kernels is that the collection of the characteristics is also constant – the patterns of computation and communication do not change. 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, we developed the Architecture Independent Workload Characterisation (AIWC) tool. This tool collects 28 metrics that indicate computation, memory, branching and parallelism characteristics on a per kernel basis. It simulates an OpenCL device using the Oclgrind [9] tool. The AIWC plugin analyses a kernel's execution trace, including the memory locations accessed and thread-states, to generates simple metrics that are representative of the kernel's behaviour. 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 Standard Portable Intermediate Representation (SPIR) [10] format.

There are few high-performance scientific computing benchmark suites which are capable of execution over a wide range of accelerators 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 [11] – Extended OpenDwarfs (EOD) – which was developed with a strong focus on robustness, benchmark diversity, choice of problem size, and correctness of results.

LibSciBench [12] was added to EOD to provide high precision timers along with support for the collection of hardware events and energy usage information. Kernel execution times of all EOD benchmarks were collected on 15 different accelerator devices typical of 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. Total elapsed application execution time was also collected.

This thesis develops a predictive model, capable of accurately estimating the execution times of a kernel on any accelerator based solely on its AIWC metrics.

The AIWC tool was run and the features collected from all the kernels of EOD. These AIWC metrics were used as predictor variables into a random forest model, with the executions times of kernels used as the response variables for prediction. The accelerators examined in these predictions range from CPU, GPU and MIC, although, the methodology presented is expected to perform with other accelerators such as DSPs and FPGAs.

The final random forest model performs well and is capable of accurate predictions which on average differ from the measured experimental run-times by 1.2%, which correspond to absolute execution time mispredictions between 8μs and 1s depending on 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, enables selection of the optimal accelerator device for a particular kernel.

1.2 Thesis Contributions

The OpenDwarfs [11] benchmark suite is extended to include a greater range of scientific applications and over multiple 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. The benchmark suite is run on a range of devices allowing a direct comparison to be made between these devices on a per-application basis.

Separately, an Architecture Independent Workload Characterisation (AIWC) tool is presented and shown to be capable of analysing kernels and extract a set of predefined features or characteristics. The benefits of AIWC include that it:

1) provides insights around the inclusion of an application into a benchmark suite via diversity analysis of the feature-space.

2) measures requirements in terms of FLOPs, memory movement and integer ops of any application kernel – which may indicate the 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.

These metrics are used for creating a predictive model of the performance of OpenCL kernels on different hardware devices and settings. This model is then applied 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 provides a negligible run-time overhead – codes are examined one time by the developer when instrumenting with AIWC and the AIWC features are embedded into the header of each kernel code to be evaluated by the predictive model at the time of scheduling.

1.3 Thesis Structure

Chapter 2 reviews 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 predictive model and examines the accuracy of the final predictions. Chapter 6 discusses the conclusions of this thesis and the future work required for the predictive model to be incorporated into scheduling on future supercomputing systems.

1.4 Publications

- Johnston B and Milthorpe J “AIWC: OpenCL-based Architecture-Independent Workload Characterisation”, LLVM-HPC workshop, SC18, Dallas, Texas, USA, 2018.
- Johnston B and Milthorpe J “Dwarfs on Accelerators: Enhancing OpenCL Benchmarking for Heterogeneous Computing Architectures”, ICPP, Eugene, Oregon, USA, 2018.
- Johnston B, Falzon G and Milthorpe J “OpenCL Performance Prediction using Architecture-Independent Features”, HPCS, Orleans, France, 2018.

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 provide a brief survey regarding their use in supercomputing. The hardware-agnostic programming framework OpenCL is then 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

In 2004, Phil Colella [13] 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 [14] was conceived to present the motifs commonplace in HPC. Initially performed by Asanovic et al. [15], 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. Asanovic et al. [15] present a total of 13 dwarfs, stating that whilst it was believed that more dwarfs could be added to this list in the future, all currently encountered scientific codes are 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 performance limit was unknown at the time of publication – and to the best of our knowledge, none of these has 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 the Dwarfs are discussed in the Benchmark Suites Section 2.4. The division of applications into dwarfs helps ensure diversity in the benchmarks which, in turn, provides a fair evaluation of scientific codes on HPC nodes.

2.2 Accelerator Architectures in HPC

We will use the term "accelerators" to refer to any form of hardware specialized to a particular pattern of computation; Thus, specialized hardware may accelerate a given application code according to that codes characteristics. Accelerators commonly include GPU, FPGA, DSP 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. Applications depicted by dwarfs in The Dwarf Taxonomy, Section 2.1, offer a wide range of characteristics realistic to scientific computing applications.

Lee et al. [16] evaluate the validity of reported speedup results between CPU and GPU devices over five benchmarks while considering optimizations which impact the characteristics of the codes. They find optimizations such as multithreading, cache blocking, and reorganization of memory accesses strongly affect CPU devices whilst minimizing global synchronization and using locally shared buffers benefit GPU devices. Two of their benchmarks are better suited for CPUs and suggest structural characteristics of applications impact on device performance. By extension, we expect that the performance of an application which is representative of a given dwarf will depend on the execution device and the structural characteristics which define that dwarf, in other words, certain applications are better suited to specific types of accelerator. 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, it is increasingly common to combine heterogeneous CPUs in a System-on-a-Chip (SoC). Comprised of two separate micro-architectures, SoCs have 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 is 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 [17]. 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 [18]–[21]. big.LITTLE is a 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 suits 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 heterogeneous configurations including the SW26010 and big.LITTLE devices have side cores, which, with careful work scheduling, can accelerate workloads and achieve high FLOPs. Additionally, 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.

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 [22][23] 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 [24], structured grids [25] and graph traversal [26] for GPU acceleration. Efforts are primarily algorithmic, such as reordering of operations and the padding of shared memory, and have been used with varying success on GPU architectures [27]. Avoiding bank-conflicts and non-coalesced memory accesses thus increasing the use of private and shared memory are critical to the 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 [28] is attributed to its performance [29] and energy efficiency [30] 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). [31]; 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 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 [32]. The Xeon Phi is the primary accelerator in the Trinity [33] and Cori [34] 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 exploits 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.[35] 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. [36] present a comparison between a P100 GPU and BittWare A10PL4 FPGA over an 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 [37] 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 [38].

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. [39] 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.

Reagen et al. [40] consider the energy efficiency of accelerators by exploring properties of workloads and how these properties affect the shape of the large design space of hardware accelerators. They examine implementations of five benchmarks in terms of energy and explore the architectural parameters by sweeping through four configurable directives – loop unrolling, array partitioning, pipelining and Multiplier Stages. Their work provides insight around which workloads benefit most from acceleration.

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 [41][42][43]. Isambard [44] and Astra [45] systems use the Cavium ThunderX2 CPU accelerator, where each ThunderX2 accelerator consists of 32 high-end ARM cores operating at 2.1 GHz [46]. Separately, Fujitsu proposes using ARMv8-A cores for the Post-K supercomputer [47]. 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 of tasks is higher versus conventional x86 architectures, the energy usage is much lower. Simula et al. [48] evaluate 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.

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 [49]. 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 [50] were examined, over consecutive years from 2012 to 2018. The starting year was selected as 2012 because 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 a steady increase in the use of accelerators in supercomputers depicted as the solid 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% of 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 [17]. Also shown in Figure 2.1 is the average percentage of cores in the TOP500 every year dedicated to accelerators, presented as the dashed blue 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%

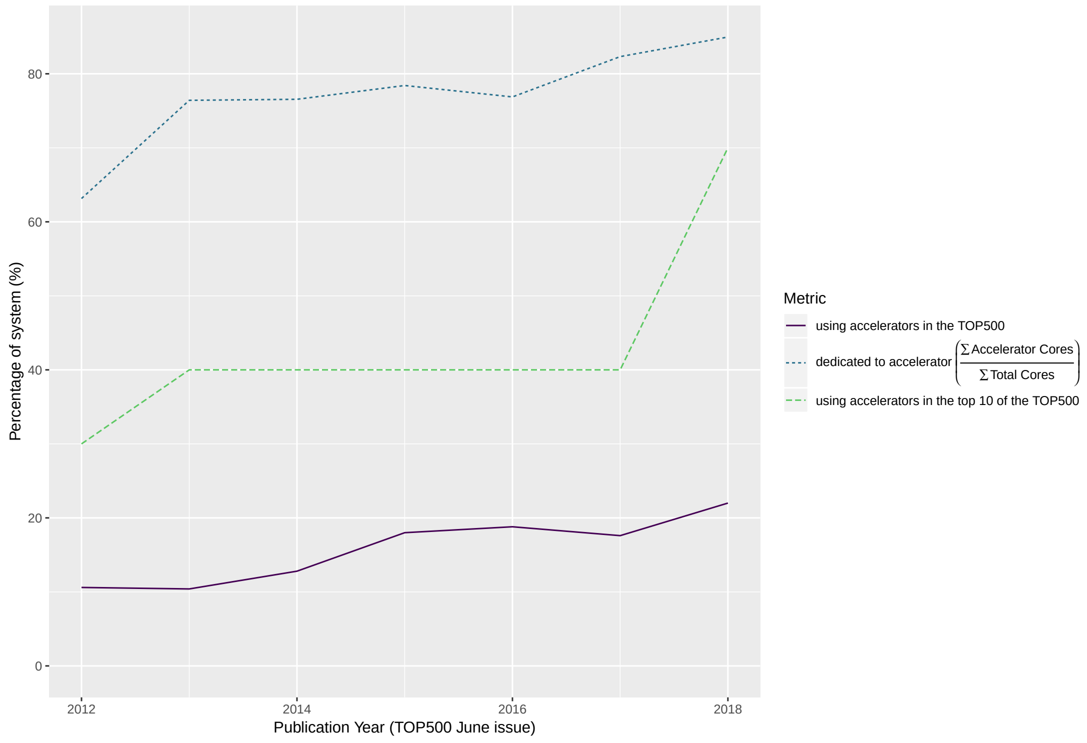


Figure 2.1: The percentage of accelerators in use and the contributions of cores found on systems with accelerators in the Top500 supercomputers over time.

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 green long dashed line in Figure 2.1 and shows a greater dependence on accelerators and a corresponding increase in heterogeneity. In 2012, three out of 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 solid purple line, and systems which do not use accelerators – shown as the dashed teal 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

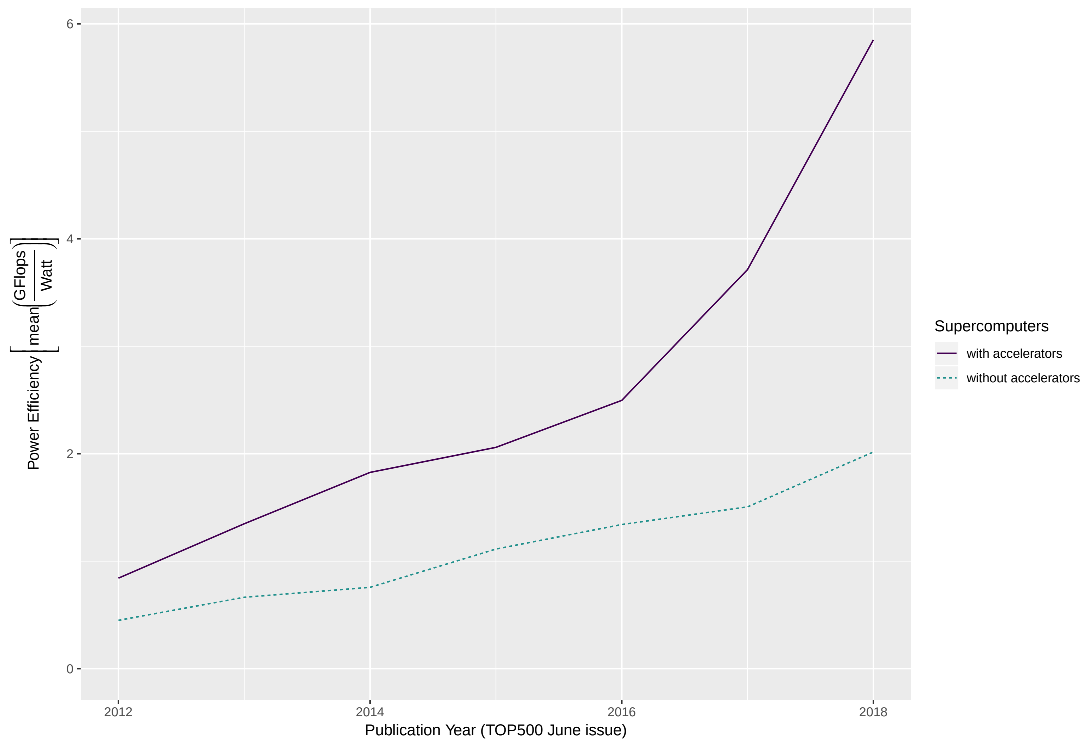


Figure 2.2: Power efficiency (GFlops/Watt) of using accelerators in the Top500 supercomputers over time.

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 have also been shown in the most energy efficient supercomputing list – the Green500 – where 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 [50]. For many systems, this was made possible by highly energy-efficient Nvidia Tesla P100 GPUs. In addition to GPUs, future HPC architectures are also likely to include nodes with FPGA, DSP, ASIC and MIC components. A single node may be heterogeneous, containing multiple different computing devices; moreover, an HPC system may offer nodes of different types. For example, the Cori system at Lawrence Berkeley National Laboratory comprises 2,388 Cray XC40 nodes with Intel Haswell CPUs, and 9,688 Intel Xeon Phi nodes [51]. The Summit supercomputer at Oak Ridge National Laboratory is based on the IBM Power9 CPU, which includes both NVLINK [52], a high bandwidth interconnect between Nvidia GPUs; and CAPI, an interconnect to support FPGAs and other accelerators [53]. Promising next-generation architectures include Fujitsu's Post-K [47], and Cray's CS-400, which forms the platform for the Isambard supercomputer [44]. 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 [54]; 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 [55].

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. It is supported on a wide range of systems including CPU, GPU, FPGA, DSP and MIC devices. Unlike device-specific languages, such as CUDA for Nvidia GPUs and Cilk for the Intel Xeon Phi, the OpenCL programming framework is well-suited to heterogeneous computing environments, as one common OpenCL code may be executed on multiple different devices. Additionally, OpenCL may also be used as a base to implement higher-level programming models such as SYCL, OpenMP and OpenACC. This technique was shown by Mitra et al., [56] where an OpenMP runtime was implemented over an OpenCL framework for Texas Instruments Keystone II DSP architecture. Similarly, Martineau et al. [57] collected a suite of benchmarks and three mini-apps to evaluate Clang OpenMP 4.5 support for Nvidia GPUs. The focus of benchmarking was as a comparison with CUDA; OpenCL was not considered. However, they provide an overview of the current clang compiler support and the success of higher level languages – the directives based abstractions of OpenMP – to be mapped to lower level frameworks; OpenCL could replace CUDA as the backend framework in a similar study.

When combined with autotuning, an OpenCL code may exhibit good performance across varied devices [58]. OpenCL has been used for DSP programming since 2012 [62]. Furthermore, Mitra et al. [39] 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 autotuners, 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. Having a common back-end in the form of OpenCL allows a direct comparison of identical code across this diverse range of architectures, making it the desirable language implementation for our benchmark suite – presented in Chapter 3.

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 with autotuning shows these parameters can be automatically optimised for any accelerator architecture as will be discussed in Section 2.6.1.

Kernel compilation flags are an additional parameter to be used by autotuners and affect 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 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 [63] specify the computational problems to be included in the benchmark suite but leave the 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.

Barnes et al. [64] 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.

Sun et al. [65] propose 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. [66], 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 Bienia et al. [67]. 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 the 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.

Reagen et al. [68] present MachSuite, a collection of 19 benchmarks to evaluate accelerators and the tools to convert C or C++ codes to FPGA devices. They aim to standardise the selection of kernels used by the HPC community by offering standardised implementations of the most commonly used algorithms in the literature to present FPGA results. 12 of the 13 dwarfs are represented by the chosen benchmarks and are presented as a C / C++ implementation; due to the wider FPGA vendor support for the high-level synthesis tools to convert these codes to register-transfer level designs to run on FPGAs accelerators. It is unclear why OpenCL was not considered.

Rodinia [69] and the original OpenDwarfs [70] 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) [71] 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. [69] 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. The Rodinia Benchmark suite originally 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 [72]. 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 analysing the original Rodinia source.

2.4.2 OpenDwarfs

As with Rodinia, Feng et al. [70] 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.

No 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. [11]. 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. [73], 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 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 support multiple nodes using MPI to assess distributed parallelism of the system – intra-node and inter-node 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. The two real-world applications presented in the level 2 applications offer 4 different problem sizes for each benchmark and represent an attempt to fully stress the hardware, but also enable the suite to run in a reasonable amount of time. These problem sizes include CPUs, Mobile/Integrated GPUs, Discrete GPUs and HPC-Focused or Large Memory GPUs. In this benchmark suite, the OpenCL versions of each application have been designed to strongly mirror the CUDA counterparts. However, both the selection of problem sizes and the selection of tuning parameters, such as workgroup sizes, is fixed according to the performance of the technology of the time – the suite was released in 2011.

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 [74] 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 [75] 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 [76] 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. [77] 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 impact on computational efficiency. This trend has been reinforced in current work by Sembrant [78] and Muller and Acar [79]; Modern 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 Sembrant [78] and Muller and Acar [79] 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 [80]. 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 [81][82][83][84]. 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 [85][86].

Taylor [87] surveys the transition of typical homogeneous cores to many accelerators on the same chip – known as dark silicon. The main reason for this transition, 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 the power use stays in proportion with area – and ensures large fractions of chips are either idle or operating at a lower clock frequency. Limitations from hitting this power-wall have meant specialized architectures are increasingly employed to "buy" energy efficiency by "spending" more on die area – thus increasing the 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 – are 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 parameters, such as local work-group, memory prefetching and blocking, and loop unrolling, which determine the allocation and partitioning of work 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. Du et al. [88] 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 autotuning 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 autotuners without requiring an exhaustive search in this search space. This has been manifested in many autotuning libraries that use machine learning. Spafford et al. [58], Chaimov et al. [59] and Nugteren and Codreanu [60] all propose open source libraries capable of autotuning dynamic execution parameters in OpenCL kernels. Filipovic et al. [89]also show OpenCL tuning strategies can take into account numerical accuracy to search for kernel implementations within specific numerical error bounds whilst optimizing for shorter execution time.

Additionally, Price and McIntosh-Smith [61] have demonstrated high performance using a general purpose autotuning library [90], for three applications across twelve devices. The OpenTuner library requires the search space to be defined as 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 OpenTuner, a modular Python library.

In the Price and McIntosh-Smith [61] 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 [91] – A general purpose molecular docking program. Presented results show the inefficiencies when autotuning for one target device and then execute this optimised program on the other systems. The usefulness of this multi-objective autotuning technique is demonstrated and shows that it is a useful tool to generate performance portable OpenCL kernels. Additionally, they show 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. Autotuning 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.

2.6.2 Phase-Shifting

A program phase is defined as a set of intervals (or slices in time) during execution that has similar behaviour [92]. Therefore, the term phase-shifting refers to the change of the execution of a program with temporal adjacency such that the program experiences time-varying effects. Sherwood et al. [93] 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 [94] profiles (a vector of per-element counts, where each element is the number of times a code block has been entered over a given interval) and off-line classification.

An assumption in the literature is that OpenCL kernels are largely unaffected by program phase-shift. Rather, the program as a whole will doubtlessly experience phase-shifts, compiling an OpenCL kernel code which is an active component of all OpenCL programs will heavily utilise the host CPU device, and when a kernel is executed and the host waits for the device to finish, CPU utilisation is low. 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 [12] for performance measurements of OpenCL kernels. It allows high precision timing events to be collected for statistical analysis. 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 Performance Application Programming Interface (PAPI) [95] counters.

2.7 Offline Ahead-of-Time Analysis

Offline Analysis does not operate on running code, for our purposes, the analysis provides a detailed examination of the structure of the code. Ahead-of-time indicates that this analysis is done before the program is executed – in the real-world usage of the code. The combination of these 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 [96]. 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 a device 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. [97] 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 [9] capable of performing simulated kernel execution. It operates on a restricted LLVM IR known as Standard Portable Intermediate Representation (SPIR) [98], 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 have been used to demonstrate program diversity [99][100]. Often this work is manually performed by those assembling the benchmark suite, indeed, much of the motivation for curating OpenCL applications in Rodinia [69], OpenDwarfs [70] and SHOC [73] 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 [101]. 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 [102] show that although conventional microarchitecture-dependent characteristics are useful in locating performance bottlenecks [103], 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 [105] 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. [106] 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 [107] 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.

Shao and Brooks [107] also present 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 [107], 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 [107] was initially proposed by Yokota [108] and uses Shannon's information entropy to determine a score of Branch History Entropy. De Pestel, Eyerman and Eeckhout [109] 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 [110] 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.

2.8.3 Workload Characterization for Benchmark Diversity Analysis

In contrast to our proposed multidimensional workload characterization, models such as Roofline [111] and Execution-Cache-Memory [112] 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 [102]. The OpenDwarfs benchmark suite has 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 presented 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 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 [113], and choosing the wrong device results in slower and more energy-intensive computation. Fowers et al. [114] show how problem size and optimizations and algorithm implementations affect energy performance on a range of accelerators. They use 1D convolutions as the area of study and found the FPGA was the most energy efficient option for large signal and small kernel sizes, GPUs achieved a comparable performance for larger kernels and the CPU implementation was fastest when signal and kernel sizes were small where accelerator devices could not amortize high bus-transfer costs. The GPU had the largest variation in energy efficiency in the time-domain tests and shows it can be an inefficient candidate if given smaller problem sizes and the characteristics of the kernel are ill-suited. The large range of inputs and the equally variable performance indicates that accurate performance prediction is critical to making optimal scheduling decisions in a heterogeneous supercomputing environment, and the associated potential energy savings are large.

Lyerly [115] 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 schedule 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 the 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. [116] 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. Che et al. [69] broaden 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.

Partial execution, as introduced by Yang et al. [117] enables low-cost performance estimates over a wide range of execution platforms. Here a short portion of 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. [118]. 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. [119] 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 principal component analysis to develop a model to show how different GPU parameters account for applications performance bottlenecks. The model predicts application behaviour with good accuracy (91%) 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 the generation of processor and is not portable between vendors.

A GPU power-estimation model was developed by Wu et al. [120] 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 the 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 model's 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 limit 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 [121] to achieve performance prediction when porting applications to accelerators. A Machine Learning based inference model is presented to predict the performance of an application on the 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 are 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 [122] 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 modelling framework is proposed by Boyer, Meng and Kumaran [123] 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%.

It is intuitive that the collection of characteristics a program collected using a simulator – such as Oclgrind discussed in Section 2.7 – offers a more general purpose, abstract, representation of the composition of the kernel and is indifferent to which accelerator it is ultimately mapped. This device abstraction offers a more accurate architecture agnostic set of metrics for an application workload, which, in turn, can be used as a basis for performance prediction on general accelerators.

2.10 Scheduling for Heterogeneous HPC Systems

Augonnet et al. [124] 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 [125], [126], [127], [128], 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. 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.

Shelepov et al. [129] 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-independent. However, this the proposed methodology is the most similar and is the predecessor to our work.

Lee and Wu [130] 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 [131] based on AIWC metrics is complementary to this work and would only improve the accuracy of their scheduler.

Chapter 3

Extending the OpenDwarfs Benchmark Suite

The purpose of this chapter is to outline the development of a testbed that can be used to evaluate approaches to workload characterization and device performance prediction that are proposed in later chapters. Key characteristics of the testbed are that it should:

1. comprise compute-intensive kernels that are representative of real scientific application codes;
2. support multiple devices;
3. support multiple problem sizes so it can be applied to embedded systems as well as top end scientific processors; and
4. span as many of the dwarfs as possible.

In the related work, Section 2.4, three potential benchmark suites were considered for the testbed: SHOC [73], Rodinia [69] and OpenDwarfs [70]. SHOC is focused on microbenchmarks rather than complete kernels as required here. Rodinia, while focused on complete kernels, is primarily developed as a benchmark suite to compare languages and does not cover all dwarfs. OpenDwarfs has the widest coverage and is the best candidate for our purposes. The problems with OpenDwarfs is that the current release:

- Includes architecture specific optimisations in many of the benchmarks, which impact the functionality when executed on other accelerators and often result in crashes.
- Has several parameters fixed, which limits the performance portability on other devices.
- Does not currently support multiple problem sizes, which impacts benchmark performance on accelerators with a memory hierarchy.

In this chapter, we show enhancements made to OpenDwarfs to remedy these issues; and present the Extended OpenDwarfs benchmark suite (EOD) to provide a testbed of representative codes required for the bulk of this thesis. The benchmarks and the associated measurements of execution time presented in this chapter will be later used for workload characterization, performance prediction and ultimately scheduling, but these sophisticated studies first need simple empirical data. First, we review the existing OpenDwarfs Benchmark

Suite, we then discuss our enhancements. The experimental setup, methodology and results are then reported concluding with a discussion of future work. Results and analysis are reported for twelve 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 on our publication in the Proceedings of the 47th International Conference on Parallel Processing Companion, ICPP 2018 [132].

3.1 Extending the OpenDwarfs Benchmark Suite

The OpenDwarfs benchmark suite comprises a variety of OpenCL codes, classified according to the Dwarf Taxonomy [15]. 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 extend these efforts to achieve a full representation of each dwarf, both by integrating other benchmark suites and adding custom kernels. It lacked coverage across all dwarfs, specifically, no representative application of the Map Reduce dwarf was identified, and the Spectral Methods Dwarf had an FFT application that either generated incorrect results or crashed.

The K-Means clustering benchmark was originally classified as the Dense Linear Algebra Dwarf, however, we believe the K-means clustering algorithm is better represented by the Map Reduce Dwarf. Dense Linear Algebra applications generally use unit-stride memory accesses to read data from rows and strided accesses to read data from columns, while Map Reduce calculations are considered embarrassingly parallel where a single function executes on independent data sets with outputs that are eventually combined to form a single or small number of results. 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 membership between iterations. Starting positions for the centroids are determined randomly. As such, we reclassified K-Means to the Map Reduce dwarf in EOD. The algorithm and its implementation are further discussed in Section 3.2.5.

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 [133], which worked correctly in all our tests. We have also added a 2-D discrete wavelet transform from the Rodinia suite [69] – with modifications to improve portability. The final coverage of all the dwarfs and their benchmarks is presented in Table 3.1.

Marjanović et al. [134] argue that the selection of problem size for HPC benchmarking critically

Table 3.1: 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

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. EOD supports four problem sizes based on the working memory footprint of each benchmark in execution. These are tiny, small, medium and large, and were selected in accordance to levels in the CPU memory hierarchy – which is the type of accelerator most affected by size. In enabling multiple problem sizes, we needed to: i) generate input sets for multiple problem sizes, ii) fix issues with code that has been developed on GPU but show memory violations on the CPU, and, iii) determine which parameters could be fixed and which need to be adjusted to have a different working memory footprint.

Where possible each benchmark now supports running with arbitrary problem sizes. The exceptions are Gemnoui, N-Queens, Hidden Markov Models and Smith-Waterman, where we only offer a fixed problem size for these applications. This is because specifying problem size according to working memory footprint does not work for these benchmarks – for instance N-Queens has a small working memory footprint and this benchmark is highly compute-bound, increasing the number of queens placed on a board is computationally intense at 20 queens, increasing the problem size to several thousand queens to satisfy the memory conditions for small sized benchmarks would take several orders of magnitude longer than the other benchmarks to solve.

We also added python scripts with these fixed parameters to allow the rapid collection of performance results on new accelerators. 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.

Table 3.2: Hardware utilized during the Extended OpenDwarfs Benchmark Suite evaluation.

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

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 [12]. 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 [95] 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)[135] and Nvidia Management Library (NVML) [136], LibSciBench also supports energy measurements, for which we report preliminary results in this chapter.

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). The selection of hardware was largely determined by the availability of these systems. CPU, Consumer GPU, HPC/Scientific GPUs – the K20m, K40m and FirePro GPUs – and MIC type accelerators are included and span 6 years of microarchitecture changes. Key characteristics of the test platforms are presented in Table 3.2. The L1 cache size should be read as having both an instruction cache and a data cache of the size displayed. For Nvidia GPUs, the L2 cache size reported is the size of the 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. [137] 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 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. 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 Problem Size

This section outlines the choice of problem size, defines the “tiny”, “small”, “medium” and “large” sizes and describes how they are influenced by cache size. A discussion around each benchmark, how it operates and how it was extended is presented. This section concludes with the arguments to reproduce our selected problem sizes for the EOD benchmarks.

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

The memory footprint was verified for each benchmark by printing the sum of the size of all memory allocated on the device. The applications examined in this work are presented in Table 3.1 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. CPUs hide memory access latency through a large and deep cache hierarchy, while GPUs solve the same problem by having a larger number of threads which can be quickly swapped out while waiting on memory operations. For this reason, GPUs are less impacted by problem size – as long as it fits on device memory – and therefore GPU cache size was not considered while setting problem sizes. 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.

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

3.2.5 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 points move between clusters 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 the 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 behaviour. 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.

3.2.6 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.

3.2.7 csr

The Compressed Sparse Row format is used to store sparse matrices by storing only non-zero values and their positions. It allows for large space savings compared to a dense matrix format, but the algorithm adds computationally intensive lookup steps to process the data.

Three different arrays are used to track the locations and values in a matrix. The benchmark implementation performs a number of matrix operations such as computing the Laplacian and performing a binary search over the irregularly spaced data. It has been extended by using the `createcsr` application to create 99.5% sparse matrices of our four selected problem sizes.

3.2.8 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 with a ratio of 3648×2736 pixels and was down-sampled to 1152×864 for medium, 200×150 for small and 72×54 for the tiny problem size.

3.2.9 gem, nqueens, hmm, swat

For four 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)[138] 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[139] 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[140] 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[141], 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`[142] tool. Generation of the solvent excluded molecular surface used the tool `msms` [143]. 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 these larger problem sizes. Although the **small** sized results have been collected, to be consistent with the other fixed sized benchmarks only the **tiny** problem size is presented. The datasets used for gem and all other benchmarks can be found in this chapter's associated GitHub repository [144].

The nqueens benchmark is a good representative of 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 the 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, uninitialized values are encountered when considering problem sizes larger than **tiny**. The **tiny** problem size has been validated – results are correct – and, as such, it is the only size presented in the evaluation.

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, as such, only the tiny problem size is considered.

3.2.10 bfs, cfd, tdm

Results for bfs are not presented due to an error in the OpenDwarfs benchmark code. Results for cfd and tdm are not presented as the provided datasets were invalid and no dataset generator was available.

3.2.11 Summary of Benchmark Settings

The problem size parameters for all benchmarks are presented in Table 3.3.

Table 3.3: The different problem sizes in the Extended OpenDwarfs adjusted by selecting the workload scale parameter (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	–	–	–
nqueens	18	–	–	–
hmm	8,1	–	–	–
swat	1k1	–	–	–

Table 3.4: Program Arguments for benchmarks in the Extended OpenDwarf Suite.

Benchmark	**Arguments**
`kmeans`	`-g -f 26 -p` Φ
`lud`	`-s` Φ
`csr`†	`-i` Ψ
	Ψ = `createcsr -n` Φ `-d 5000` $\triangle$
`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`
`swat`	`'query`Φ`' 'sampledb`Φ`'`

$^{\triangle}$ 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.

Each **Device** can be selected in a uniform way between applications using the same notation, on our sample 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. The availability and ordering of platform and device ids vary between nodes and will need to be adjusted accordingly. 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 [144]. 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. We use the benchmarks to assess and compare performance across the chosen hardware systems. 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 12 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 scaling patterns between devices. This underscores the importance of allowing a choice of problem size in a benchmarking suite. The primary analysis is for time, but energy results over two devices are also presented.

3.3.1 Time

The distribution of execution times required to execute each of the benchmarks for all available hardware is presented in Figures 3.1 and 3.2. Eight of the benchmark applications offer four different problem sizes, we only present the medium problem size in Figure 3.1 to highlight the variation in runtimes between benchmarks, while Figure 3.2 presents execution times for the four benchmarks with fixed problem size. The results are coloured according to accelerator type: purple for CPU devices, blue for consumer GPUs, green for HPC GPUs, and yellow for the KNL MIC.

The results presented in Figure 3.1 show the total kernel execution time in milliseconds for each of the benchmark applications on various accelerator devices. The reported iteration time is the sum of all compute time spent on that accelerator for all kernels.

In Figure 3.1 (a) (`kmeans`) shows applications typical of the MapReduce dwarf are best suited to GPU devices, this is unsurprising given it has embarrassingly parallel characteristics that are well suited to this accelerator type. The performance of the various GPU devices generally follows their date of manufacture. For instance, the Nvidia gaming devices such as the TitanX, GTX 1080 and GTX 1080 Ti are the newest devices and perform best; similar trends emerge in the AMD cards – the RX 480 has a higher clock frequency (1120 MHz) relative to the oldest AMD device the HD7970 (925 MHz) and performance ranges between these two extremes accordingly. Interestingly the HPC GPUs buck this trend with a lower clock frequency (706-900MHz) and older manufacture date than many of the gaming GPUs performing at a similar level to the newer AMD gaming GPUs and only 1-2ms slower per kernel run than the newest Nvidia GPU. This is attributed to the larger number of threads supported on these devices and is a good match to the high degree of parallelism in the benchmark. The i7 is the best performer of all the CPU type accelerators, taking 6ms which is 2-3ms slower than the fastest GPU, this is expected since it has the highest clock frequency. As the CPUs have fewer hardware threads they are less able to exploit the available parallelism in the benchmark than the GPUs. The Xeon Phi 7210 MIC is $\approx 10-20\times$ slower than the other accelerators, we believe this is due to the lack of vectorization of the kernel. Indeed, many of the kernels lack vectorization and explain much of the poor performance on the KNL MIC.

Figure 3.1 (b) (`lud`) from the Dense Linear Algebra dwarf shows similar trends. It is largely well suited to GPUs however is less suited to the HPC scientific cards and the MIC performs better. The oldest CPU considered (the i5-3550) performs much worse than the other CPU devices, and is because the medium problem size requires 8100KiB of cache which spills outside of the L3 cache 6133KiB on this processor thus this performance is due to the high penalties of L3 cache misses and accessing main memory.

Figure 3.1 (c) (`csr`) – Sparse Linear Algebra – shows a performance which scales with clock-frequency and the newer CPU and GPU devices with the highest frequency offer the shortest execution times. The i7 CPU and Titan X, GTX 1080 Ti GPUs are the most suitable accelerators for these type of codes. Interestingly, the difference in execution times between the GTX 1080 and the GTX 1080 Ti highlights the memory critical nature of this dwarf. Irregular memory access in sparse matrices benefits the higher memory bandwidth lower latency interconnect in the Ti, where despite being 127-151MHz slower in base clock speed, the doubling in peak memory bandwidth and the extra 125MHz memory clock speeds give the Ti better performance. The Titan X has a similarly high memory configuration and justifies why it performs as well as the GTX 1080 Ti.

Figure 3.1 (d) (`fft`) and Figure 3.1 (e) (`dwt`) represent Spectral Methods. These benchmarks are high floating point intensity applications which explain the poor suitability of the Xeon Phi 7210 MIC – which is limited to half the theoretical peak of its floating-point performance as explained in Section 3.2.2. The CPU devices also suffer from this high floating point demand which has lower raw FLOPs than the GPUs. On average, the worst performing device over both Spectral Methods applications is the i5-3550 which has the poorest FLOPs of any of the CPUs coupled with the smallest L3 cache which frequently spills over into main memory during execution of these benchmarks. It is interesting to note that both benchmarks representing the same dwarf have very similar performance results over all the accelerators – where generally the ordering of optimal device is largely the same between both applications.

Figure 3.1 (f) (`srad`) represents the Structured Grid dwarf and has similar performance results to `fft` and `dwt`. This tells us the regular grid points which are updated together are well suited to GPUs and accelerators with higher floating intensity. The high spatial locality and embarrassingly parallel nature of Structured Grids indicate that it has similar properties to Spectral Methods.

Additional experiments could be performed using the LibSciBench hardware performance counters to confirm our explanation of these poor CPU and MIC results, while Nvidia's Perftool could examine similar hardware metrics on the Nvidia accelerators. However, since the broader focus of this thesis is to use the evaluation of a range of accelerators over a broad suite of benchmarks, the interest in predicting poor performance is more interesting than a closer level inspection of the deficiencies of each platform.

Figure 3.1 (g) (`crc`) is from the Combinational Logic dwarf where the benchmark performs error-detecting code caused by network transmission or any other accidental error, work is performed in workgroups determined by polynomial division. There are no floating point operations within each workgroup and a checksum is computed for each. Integer vectorization is high and each work-item processes 8 bytes at once using the "Slice-by-8" algorithm. This requires a large number of bit shifts and conditional loop nesting which results in irregular integer comparisons that are ill-suited to GPU architectures – which are not optimized for integer operations and suffer from thread-divergence. GPU devices suffer further since they are not equipped with wide integer units – 64-bit wide ALUs are typical. By comparison, CPUs and the MIC are accelerator types which excel at this computation; the high degree of vector parallelism inherent in the algorithm suit the E5-2697 and i7-6700K CPUs with 128-bit and 256-bit wide SIMD units respectively, and is ideal for the MIC which has very wide 512-bit SIMD units. Examining the `crc` benchmark is examined in greater detail in Figure 3.3 as problem size increases.

Figures 3.1 (h) (`nw`) and 3.2 (d) (`swat`) both represent Dynamic Programming. The order of performance results is also similar, with the newest Nvidia GPUs performing best, the different generations of CPUs falling around the older GPUs and the MIC being the worst performer by a large margin (30%). The ordering of performance of devices between both benchmarks is similar. It is equally interesting to note that 3.2 (a) (`gem`) and 3.2 (a) (`nqueens`) also have a similar ordering of fastest accelerator devices.

Comparing the medium problem size between all the benchmarks, we see individual devices with significantly longer execution times than the others, these large differences in execution times hide many of the finer differences in detail between accelerators with similar good performance and identify the penalties when selecting a suboptimal accelerator device. The Xeon Phi 7210 MIC is 2.5× slower than the next worse accelerator in the `kmeans` benchmark, the rest of the accelerators all have average execution times less than 18ms. It was had the worst execution times for `csr` and `nw` benchmarks being 2-4× slower than the other accelerators. The i5-3550 performed 6× worse than the MIC on the `lud` benchmark which on average take ≈ 1ms. Similarly, the i5-3550 CPU was the poorest choice of accelerator for `dwt` and `srad` benchmarks, being on average 6× and 5× slower, respectively, than the other accelerators. The Xeon Phi and i5-3550 were equally poor on the `fft` benchmark, taking 12ms per run, despite the other non-CPU accelerators taking less than 2ms. Finally, the GPUs performed worse on the `crc` benchmark, with the K20m taking 100ms, compared to the CPU and MIC taking <5ms. These large differences in execution times show the importance in selecting an optimal accelerator by highlighting the large difference between the good performance of accelerators on average and the poorest device for a benchmark – it results in a 2-100× longer execution time.

Figure 3.2 presents results for the four applications with restricted problem sizes. The per kernel invocation is relatively low regardless of device selected for the (a) `gem` or (b) `nqueens` benchmark. The newer Nvidia GPUs collectively tended to be the best-performed accelerator on `gem` taking ≈ 110μs while the MIC saw the worst performance at 0.85 ms. The `nqueens` benchmark saw the i7-6700K and i5-3550 CPUs finish the kernel in ≈ 80μs to ≈ 100μs per invocation, respectively, again the MIC had the worst performance at 900μs on average. Figures (c) `hmm` and (d) `swat` are more computationally intensive and took longer to complete. The `hmm` benchmark shows the CPU and modern Nvidia GPUs performing equally well < 1ms, the older AMD and HPC GPUs ranged from 1-3ms, and the MIC averaged 7.5ms per run. Finally, `swat` had the modern Nvidia GPUs as the fastest devices at ≈ 5ms and ranged up to 40ms on the MIC which was the slowest device for this benchmark.

We selected the `crc` and `kmeans` benchmarks for the detailed analysis to show how the amount of work increases over each of the four problem sizes, since the former experiences exceptionally good performance on the KNL MIC, while the latter typifies the benchmarks suitable to GPU architectures as problem size increases.

The `crc` benchmark is a standout in benchmarks for the MIC; it the only benchmark where the MIC is competitive with the other accelerators, probably due to the low floating-point intensity of the crc computation[145]. The effect of problem size on this application is presented as Figure 3.3. Starting with the tiny size, it experiences comparable performance to all of the older GPUs, for the small size it offers similar performance to the latest Nvidia GPUs, and for the medium and large problem sizes its performance rivals the CPU accelerators. This is due to the larger problem sizes generating enough work to fully utilize the 512-bit wide SIMD units over the 256-threads on the MIC.

We have omitted the KNL MIC platform from the `kmeans` results in Figure 3.4 because they are typically an order of magnitude worse than the other devices. The full results with the MIC device are presented in Appendix A. The devices are grouped in this analysis: CPU devices (1-3) are presented in purple; the high-performance GPUs designed for scientific workloads (devices 7-9) are presented in green; the modern Nvidia GPUs are in blue to the left of HPC GPUs (devices 4-6); and the last group consists of the older AMD GPUs (devices 10-14) and are also in blue to the right of the green results. Both groups of Nvidia GPUs and AMD GPUs are both presenting in blue since they are both consumer GPUs. As problem size increases, the order of devices in a group rarely changes and only the magnitude of differences increase between groups. The CPU accelerator group performs worse as the problem size increases, this is because performance is tightly bound to the level of the cache hierarchy used. The modern Nvidia GPU was the 2nd fastest set of devices in the tiny problem size, and this performance improved under the demand of larger workloads and culminates in being 2-5× faster than the CPU devices. HPC GPUs had average performance over the increasing problem sizes, while the HD7970 GPU suffered worse runtimes relative to the rest of the AMD gaming GPUs, this confirms that the performance of this benchmark corresponds to clock frequency and FLOPs achievable for the devices.

The entire set of results and a detailed discussion is presented in Appendix A.

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 the collection of PAPI energy measurements (with LibSciBench) requires root access, and these devices were the only accelerators available with this permission. The distributions were collected by measuring solely the kernel execution over 50 runs. RAPL CPU energy measurements were collected over all cores in package 0 `rapl:::PP0_ENERGY:PACKAGE0`. NVML GPU energy was collected using the power usage readings `nvml:::GeForce_GTX_1080:power` for the device and presents the total power draw (+/-5 watts) for the entire card – memory and chip. Measurements results converted to energy J from their original resolution nJ and mW on the CPU and GPU respectively.

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

Figures 3.5 (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 blue for the Nvidia GTX1080 GPU. The logarithmic transformation has been applied to Figure 3.5 (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. The execution times and corresponding energy usage is tightly coupled for all the benchmarks presented. While not initially apparent in Figures 3.5 (a) and (b) the variability of energy usage is slightly larger on the CPU, which is consistent with the larger variation in execution time results.

3.4 Discussion

In this chapter, we presented EOD which places a strong focus on the robustness of benchmarks, 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 to allow 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. We improved coverage of spectral methods by adding a new Discrete Wavelet Transform benchmark, and replacing the previous inadequate fft benchmark.

Older hardware was used in this evaluation, but having a greater diversity between generations of microarchitecture could be useful when examining the general purpose nature of the predictive model in Chapter 5. Energy results were not able to be collected on many of these systems since we lacked root access, however, we have proposed a methodology that can easily be applied to collect additional energy results on the next generation of hardware.

The work presented in this chapter presents the ground-work required to evaluate the performance of heterogeneous devices from a shared language – OpenCL.[1] The introduced benchmarking suite – EOD – and the corresponding execution times on the full range of

[1]No claim is made regarding the optimality of OpenCL for accelerator programming.

accelerators are used in the remainder of this thesis. While performance could be individually analysed for each kernel in EOD, we will instead propose to collect results on an abstract OpenCL device to enable 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 testbed for the predictive model presented in Chapter 5. It serves as a platform which is essential to measure the performance of scientific workloads on accelerators. The goal of this thesis is scheduling of scientific workloads to accelerator devices which will be a standard feature of the next-generation of 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, each dwarf is ill-suited to at least 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"? Our proposed workload characterization tool – AIWC – is introduced in the next Chapter whilst the above question is addressed in Chapter 5.

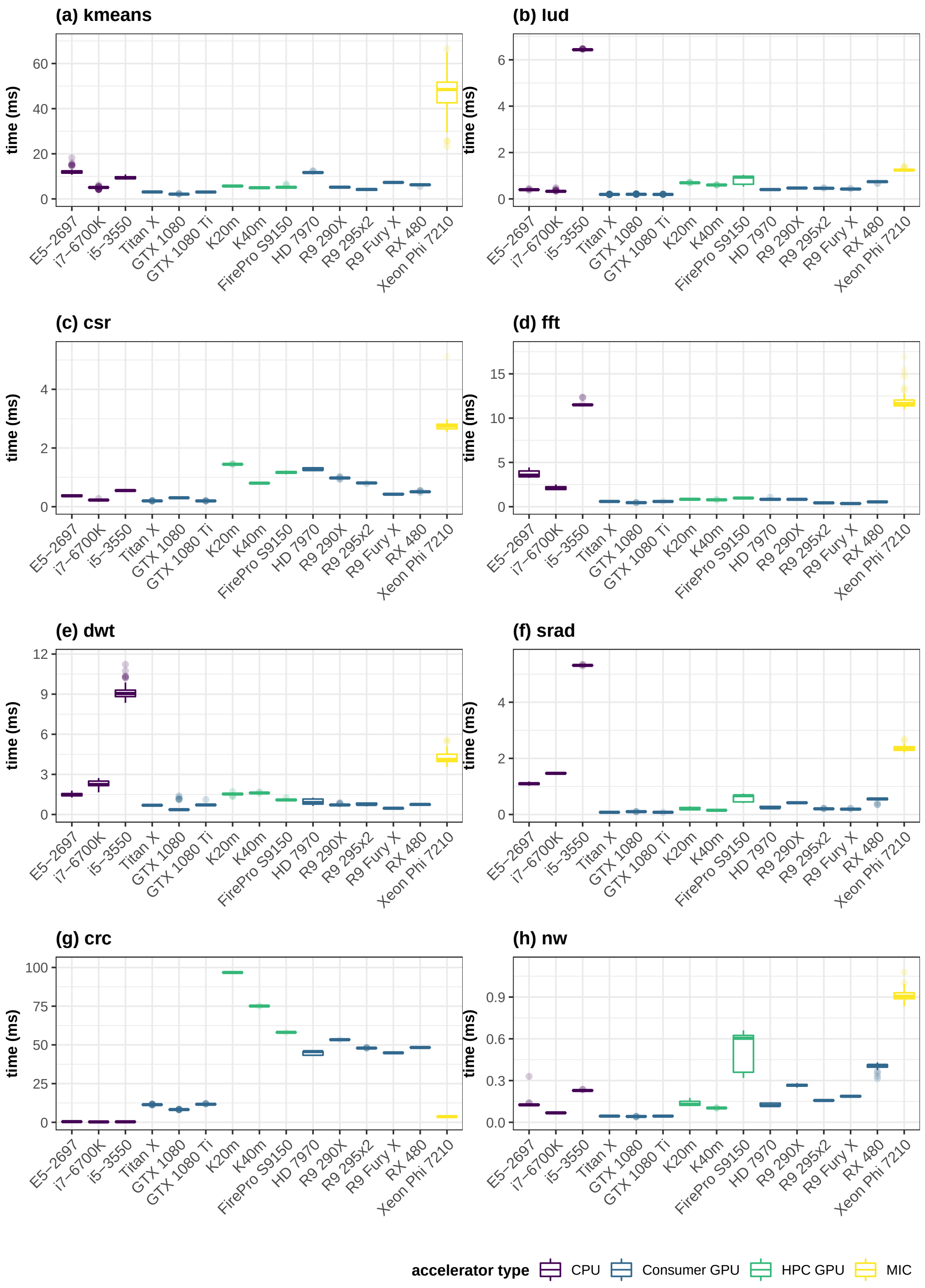


Figure 3.1: Kernel execution times for the medium problem size benchmarks on different accelerator devices.

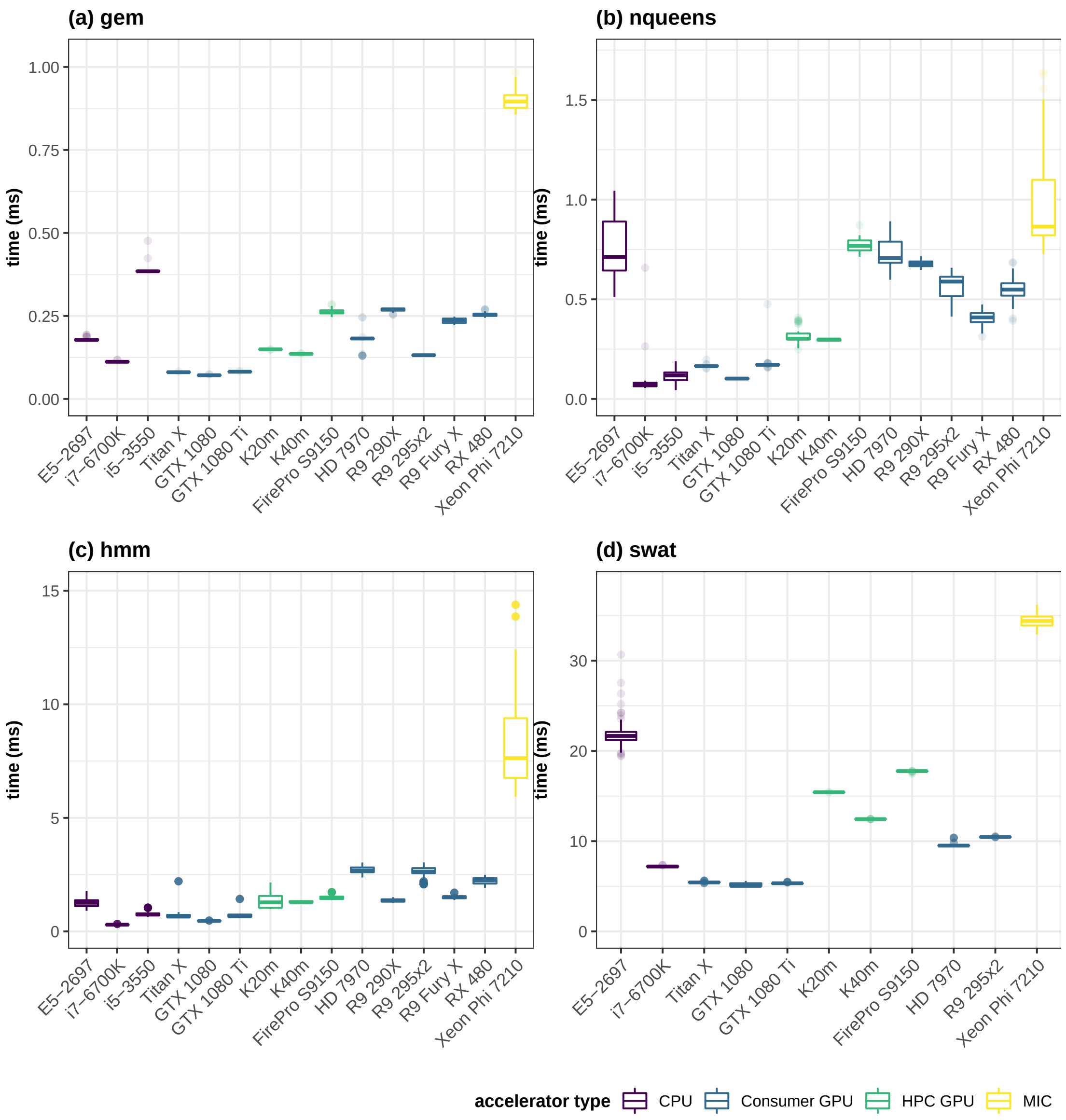


Figure 3.2: Kernel execution times for the single sized benchmarks on various accelerator devices.

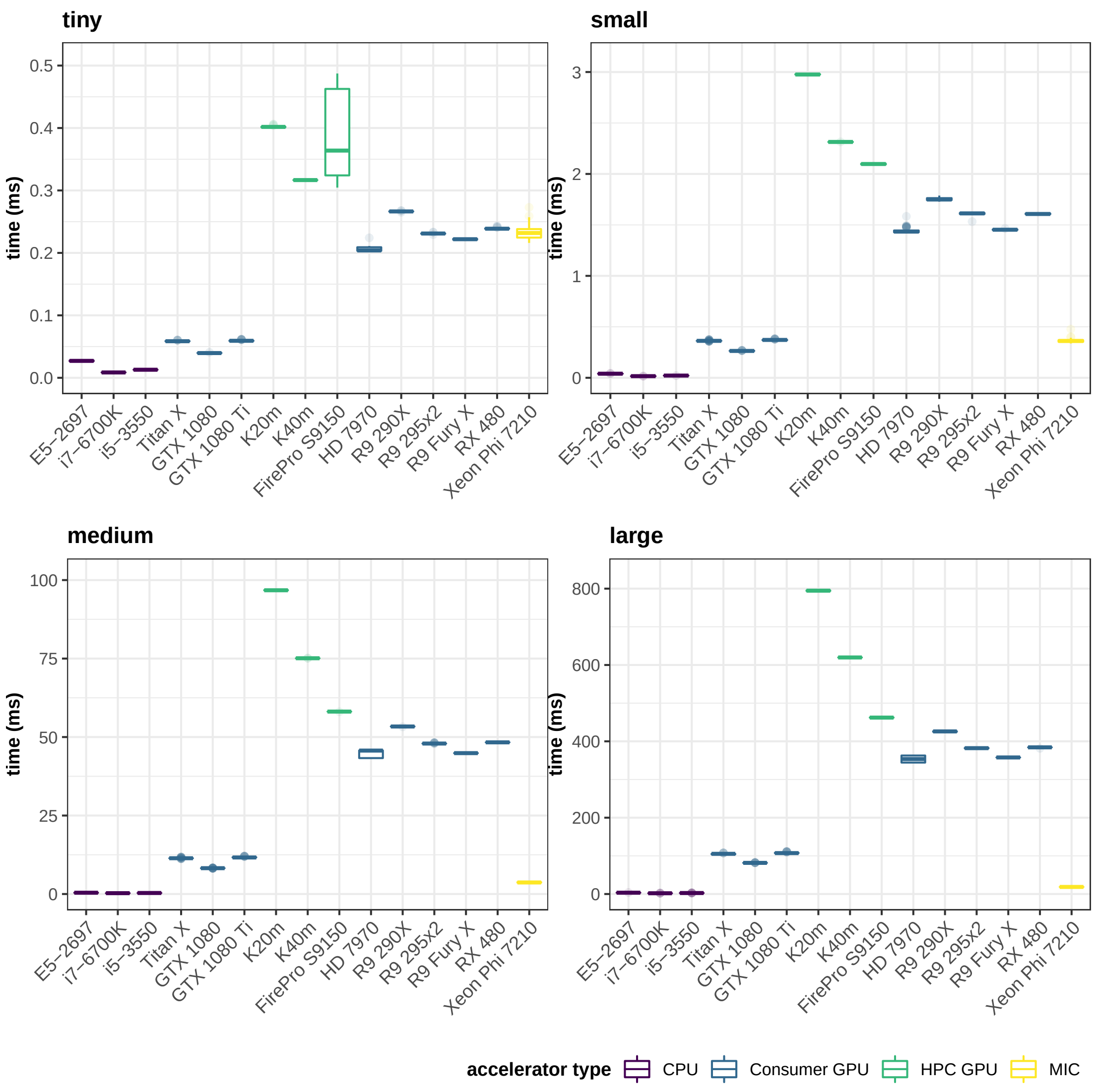


Figure 3.3: Kernel execution times for the **crc** benchmark on different hardware platforms.

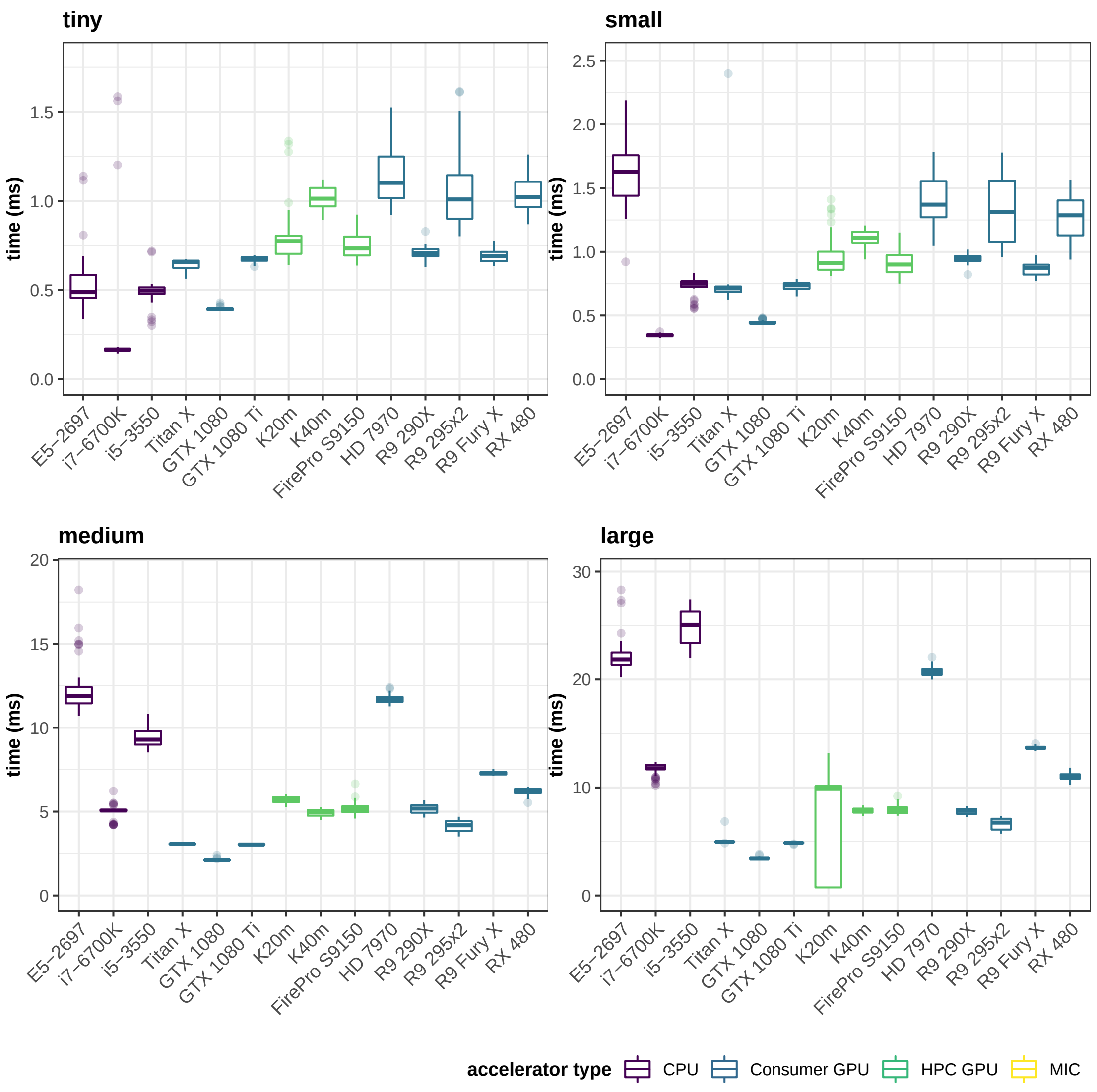


Figure 3.4: Kernel execution times for the **kmeans** benchmark on different hardware platforms.

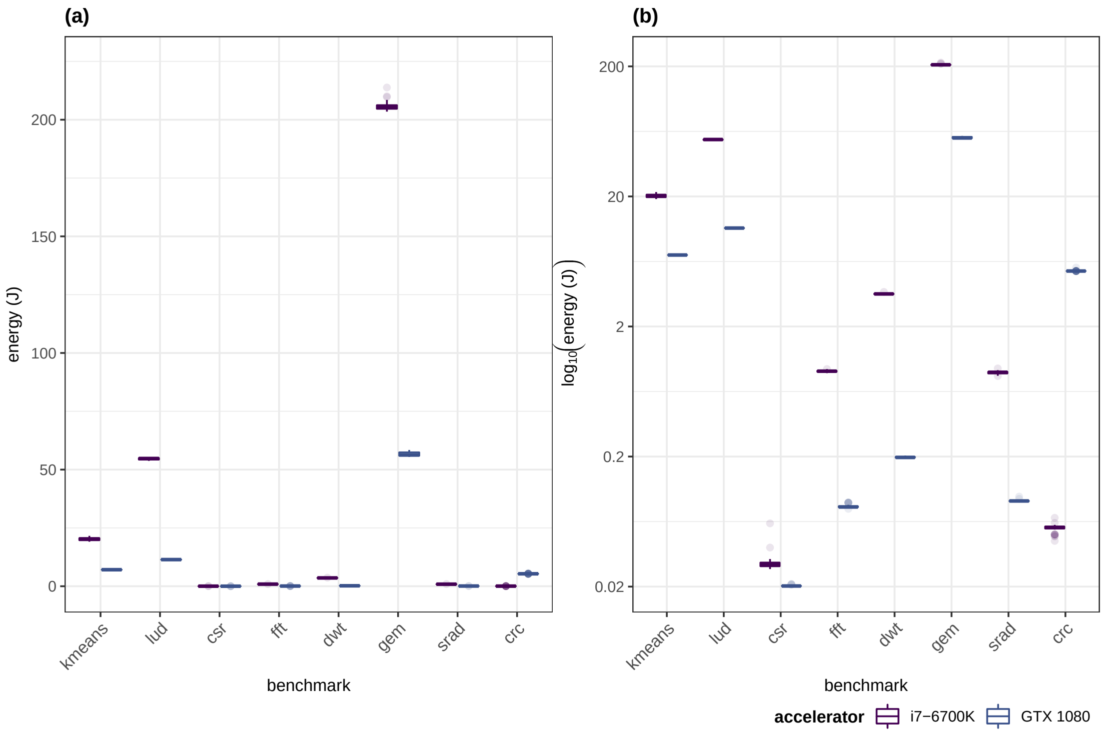


Figure 3.5: Execution energy required to perform EOD benchmarks, presented on a linear (a) and logarithmic scale (b) from left to right respectively, on the (**large** problem size) on the Intel i7-6700K and Nvidia GTX1080.

Chapter 4

AIWC: OpenCL based Architecture Independent Workload Characterization

In this chapter, 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 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 (IR) 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.

We migrate the metrics presented in the ISA-independent workload characterization paper [107] to the Oclgrind tool offers an accessible, high-accuracy and reproducible method to acquire these AIWC features. We also add additional metrics to be more general architecture-independent instead of ISA-independent workload characterization. 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, it is deterministic: the order the work-items execute is identical between runs with the AIWC simulator.

A 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.

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; for the results presented in this thesis, we chose the latter. Additionally, given the number of kernels we present in this chapter, we believe AIWC will generalize to full codes in other domains.

Application codes differ in resource requirements, control structure and available parallelism. Similarly, accelerator 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 accelerator 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.

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. We demonstrate the use of AIWC to characterize a variety of codes in the Extended OpenDwarfs Benchmark Suite [132] – presented in chapter 3. We begin with an introduction of the metrics collected by AIWC, we then discuss how AIWC was implemented and demonstrate its usage on the `lud`, `nw`, `swat`, `gem`, `kmeans` and `hmm` benchmarks. Finally, we conclude with a summary of what AIWC and the associated metrics provide to prediction. 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 [146].

Table 4.1: ISA-Independent Workload Characterization metrics.

Type	Metric	Description
Compute	Opcode	Unique Opcodes required to cover 90% of dynamic instructions
Memory	Total Memory Footprint	Total number of unique memory addresses accessed
	90% Memory Footprint	Number of unique memory addresses that cover 90% of memory accesses
	Global Memory Address Entropy	Measure of the randomness of memory addresses
	Local Memory Address Entropy	Measure of the spatial locality of memory addresses
Control	Unique Branch Instructions	Total number of unique branch instructions
	Branch Entropy	Measure of the randomness of branch behavior, representing branch predictability

4.1 Metrics

Shao and Brooks [107] proposed metrics to represent the characteristics of workloads independent of Instruction Set Architectures (ISA) – these are provided in Table 4.1. The selection of these metrics was considered to characterize the workload, however, they were focused on micro-architecture independence and were evaluated on x86 CPUs. It is the focus of our work to collect metrics that are a higher level of abstraction and to this end, we present a larger set of metrics which are more comprehensive in order to characterize the workload in an architecture independent way. The full list of our AIWC metrics is presented in Table 4.2. The original choice of metrics does not consider parallel codes, but since this is critical for modern accelerators and the OpenCL framework it was the largely the focus of our work. In this section, we discuss each metric and what they represent for workload characterization. In particular, we added a new category of metric, called *parallelism*, where we present metrics for Granularity, Barriers Per Instruction, Instructions per Operand and Load Imbalance.

The Oclgrind simulator is a free and open source tool to debug OpenCL codes and is achieved by simulating on LLVM SPIR instructions. To this end, the OpenCL device is architecture-independent – no final register allocation or ISA has been selected at this level of execution (directly on the IR). A secondary contribution of our work is that we facilitate the collection of our metrics directly in Oclgrind. Whereas Shao collected metrics from static traces using a JIT compiler which emitted ISA-independent instructions, we collect metrics on-the-fly during OpenCL device simulator runs. These live traces are evaluated before each kernel terminates, the statistics computed and metrics presented at the end of each kernels execution. None of these metrics was previously available in Oclgrind and they are a direct contribution of the AIWC tool. Each of our metrics is described in greater detail in the remainder of this Section while the methodology around their collection is the focus for the next Section (4.2).

For each OpenCL kernel invocation, the Oclgrind simulator AIWC tool collects a set of 28 metrics, which are listed in Table 4.2. 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 kernels 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 an 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 an increasing

Table 4.2: 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
	Total Instruction Count	total # of instructions executed
Parallelism	Work-items	total # of work-items or threads executed
	Total Barriers Hit	total # of barrier instructions
	Min ITB	minimum # of instructions executed until a barrier
	Max ITB	maximum # of instructions executed until a barrier
	Median ITB	median # of instructions executed until a barrier
	Min IPT	minimum # of instructions executed per thread
	Max IPT	maximum # of instructions executed per thread
	Median IPT	median # of instructions executed per thread
	Max SIMD Width	maximum # of data items operated on during an instruction
	Mean SIMD Width	mean # of data items operated on during an instruction
	SD SIMD Width	standard deviation across # of data items affected
Memory	Total Memory Footprint	total # of unique memory addresses accessed
	90% Memory Footprint	# of unique memory addresses that cover 90% of memory accesses
	Unique Reads	total # of unique memory addresses read
	Unique Writes	total # of unique memory addresses written
	Unique Read/Write Ratio	indication of workload being (unique reads / unique writes)
	Total Reads	total # of memory addresses read
	Total Writes	total # of memory addresses written
	Reread Ratio	indication of memory reuse for reads (unique reads/total reads)
	Rewrite Ratio	indication of memory reuse for writes (unique writes/total writes)
	Global Memory Address Entropy	measure of the randomness of memory addresses
	Local Memory Address Entropy	measure of the spatial locality of memory addresses
Control	Total Unique Branch Instructions	total # of unique branch instructions
	90% Branch Instructions	# of unique branch instructions that cover 90% of branch instructions
	Yokota Branch Entropy	branch history entropy using Shannon's information entropy
	Average Linear Branch Entropy	branch history entropy score using the average linear branch entropy

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. [108] 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 Implementation

AIWC is implemented as a plugin for Oclgrind, which simulates kernel execution on an abstract 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 entire AIWC plugin is $\approx$ 1000 lines of `C++` code and collects the metrics based on the relative callbacks triggered during Oclgrind kernel execution. The remainder of this section outlines some of the logic required to collect these metrics.

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 each `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 the `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 the combined branches 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 [107], and **Average Linear Branch Entropy**, from De Pestel [109] and have been computed and are also presented based on this implementation. `workGroupComplete` events trigger the collection of the intermediate work-item and work-group counter variables to be added to the global suite, while `workGroupBegin` events reset all the local/intermediate counters.

Finally, `kernelBegin` initializes the global counters and `kernelEnd` triggers the generation and presentation of all the statistics listed in Table 4.2. The AIWC source code is available at the GitHub Repository [147].

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

```
void WorkloadCharacterisation::instructionExecuted(..., const llvm::Instruction *instruction, ...){
    unsigned opcode = instruction->getOpcode();
    std::string opcode_name = llvm::Instruction::getOpcodeName(opcode);
    //update key-value pair of instruction name and its occurrence in the kernel
    (*m_state.computeOps)[opcode_name]++;
    std::string Str = "";
    //if a conditional branch which has labels, store the labels to track
    //in the next instruction which of the two lines we end up in
    if (opcode == llvm::Instruction::Br && instruction->getNumOperands() == 3){
        if(instruction->getOperand(1)->getType()->isLabelTy() &&
                instruction->getOperand(2)->getType()->isLabelTy()){
            m_state.previous_instruction_is_branch = true;
            llvm::raw_string_ostream OS(Str);
            instruction->getOperand(1)->printAsOperand(OS, false);
            m_state.target1 = Str;
            Str = "";
            instruction->getOperand(2)->printAsOperand(OS, false);
            m_state.target2 = Str;
            llvm::DebugLoc loc = instruction->getDebugLoc();
            m_state.branch_loc = loc.getLine();
         }
    }
    //if the last instruction was a branch, log which of the two targets were taken
    else if (m_state.previous_instruction_is_branch == true){
        llvm::raw_string_ostream OS(Str);
        instruction->getParent()->printAsOperand(OS, false);
        if(Str == m_state.target1)
            (*m_state.branchOps)[m_state.branch_loc].push_back(true);//taken
        else if(Str == m_state.target2){
            (*m_state.branchOps)[m_state.branch_loc].push_back(false);//not taken
        }
        m_state.previous_instruction_is_branch = false;
    }
    //counter for instructions to barrier and other parallelism metrics
    m_state.instruction_count++;
    m_state.workitem_instruction_count++;
    //SIMD instruction width metrics use the following
    m_state.instructionWidth->push_back(result.num);
```

4.3 Demonstration

We now demonstrate the use of AIWC on several scientific application kernels selected from the Extended OpenDwarfs Benchmark Suite [132]. The details of the suite are described in Chapter 3. Our selection of benchmarks run with AIWC 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.

We present metrics for the four different problem sizes, and all 11 different application codes (37 kernels) from EOD, as described in Chapter 3. As simulation within Oclgrind is deterministic, all results presented are for a single run for each combination of code and problem size.

To briefly examine AIWC metrics over the entire EOD suite, 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. The x-axis shows different kernels but due to the unavailability of kernels on larger problem sizes the bottom half (4 metric) is not aligned to the top half.

The top-left quadrant displays each of the four chosen metrics on the tiny problem size. The linear average branch entropy (purple) between tiny kernels the `bfs_kernel2` has the largest value, and thus least predictable branching behaviour, which we expect for sorting algorithms where the behaviour around swapping values depends on the ordering of the data. Some kernels display irregular branching while a majority of the tiny kernels are predictable. The diversity in opcodes ranges from 7-15 unique instructions on the tiny problem sized kernels, with the `srad` kernels using the most unique opcodes and some initialization kernels using the least. Barriers per instruction (shown in green) on the tiny problem size shows that most of the kernels in EOD have no internal barriers, except for `lud_diagonal`, `cl_fdwt53Kernel`, and the `nw` (`needle_opencl_shared_*`) kernels. 18% of instructions in the `lud_diagonal` kernel will hit a barrier – and is slightly less balanced given the unequal distribution of work for different starting locations of the decomposition. The `nw` kernels frequently block, with many barriers comprising 40% of the instructions encountered, this dependency between other work-items indicate these benchmarks are less able to benefit from Single-Instruction Multiple-Thread (SIMT) parallelism. The top-right quadrant displays the four chosen metrics on the small problem size, while the bottom-left quadrant present the same metrics on the medium problem size and the bottom-right quadrant shows the large sized EOD problems.

Almost all benchmarks show the global memory address entropy metric increase with problem size, whereas the other metrics do not. 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. We do not present all problem sizes for the kernels corresponding to `gem`, `nqueens`, `hmm` and `swat` benchmarks, since these only operate on a fixed problem size – as discussed in Chapter 3.

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 are 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. Barriers per instruction are unchanged regardless of problem size and show that the patterns of computation are fixed and independent from the data in many of these benchmarks. The performance of this kernel on a particular architecture could be expected to be highly dependent on the cost of synchronization.

4.4 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. The AIWC metrics for a kernel are presented as a Kiviat or radar diagram, for each of the problem sizes. 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.

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 exploitable parallelism; smaller barriers per instruction indicates less synchronization; and so on.

The only metrics not normalized relative to the maximum measured values are **granularity**, **barriers per instruction**, **instructions per operand** and **load imbalance**. Parallelism

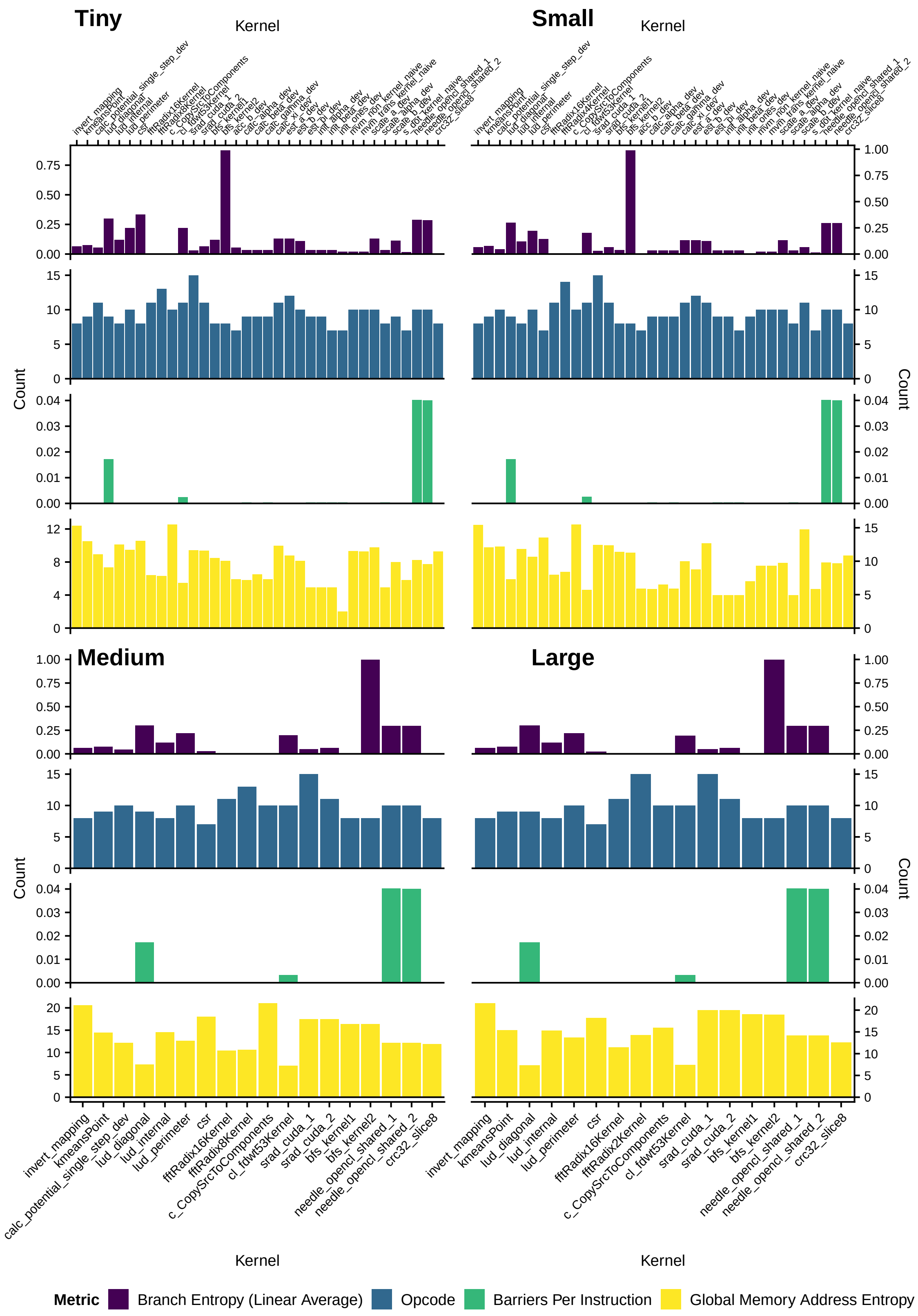


Figure 4.1: Selected AIWC metrics from each category over all kernels and 4 problem sizes.

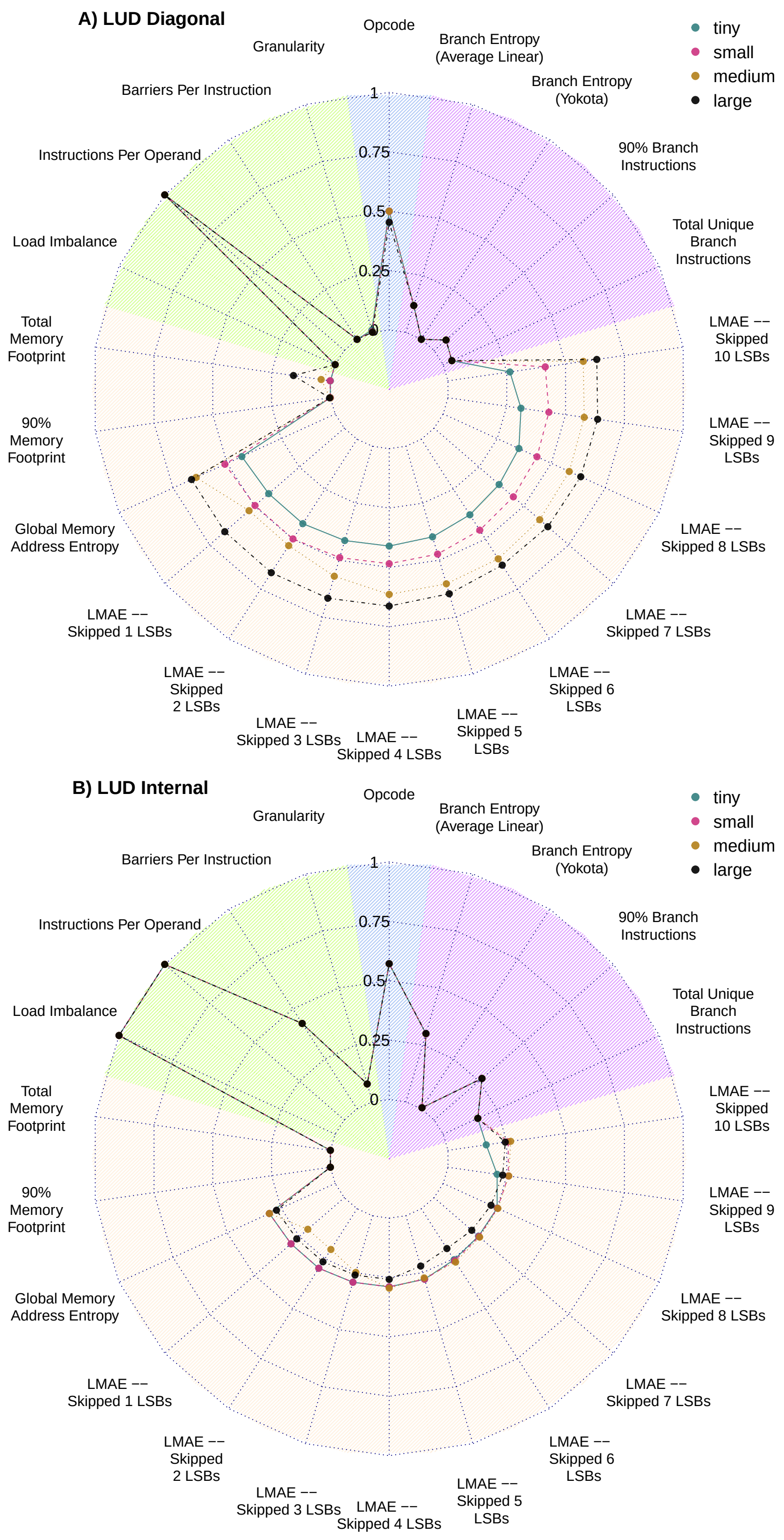


Figure 4.2: A) and B) show the AIWC features of the `diagonal` and `internal` kernels of the LUD application over all problem sizes.

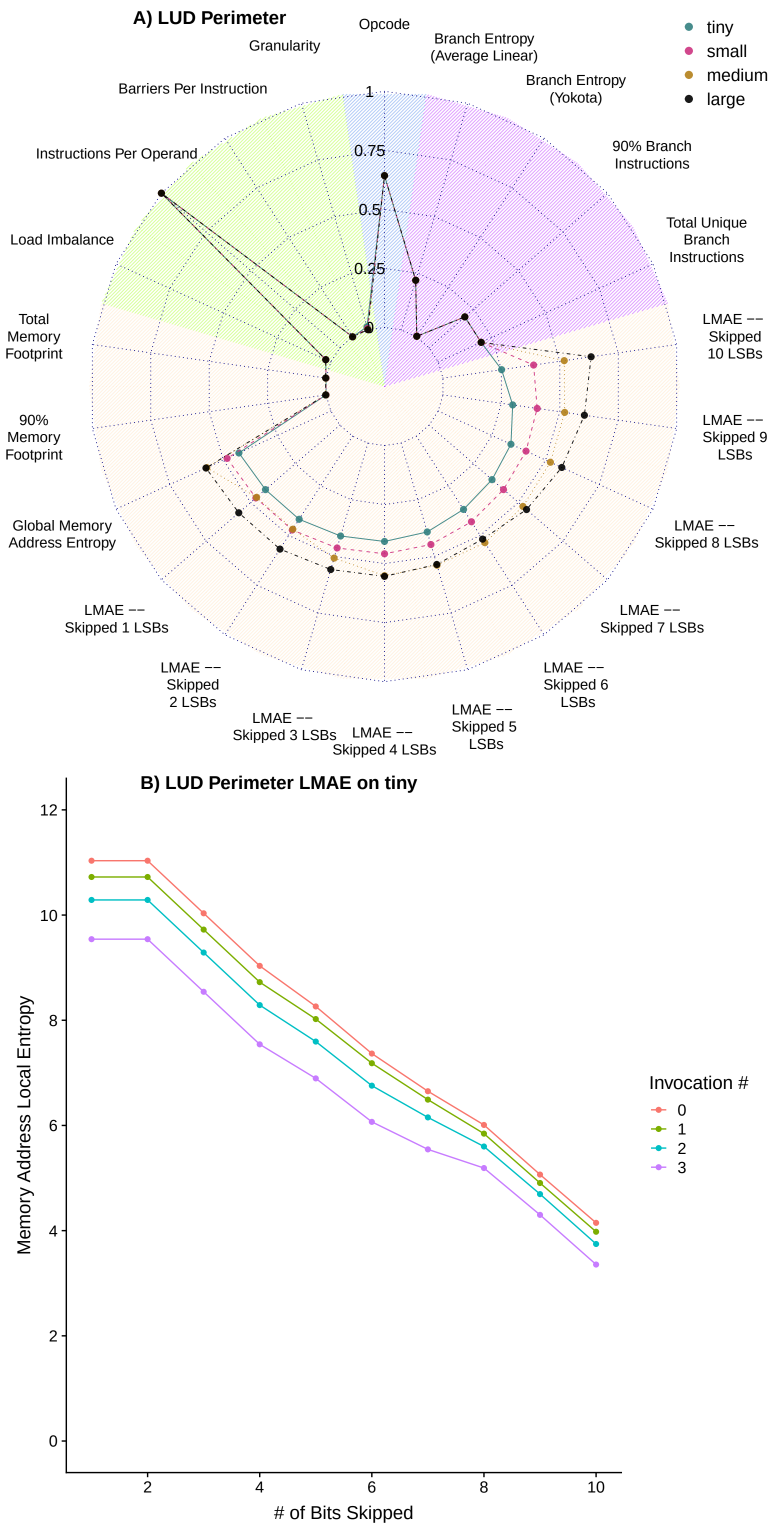


Figure 4.3: A) shows the AIWC features of the `perimeter` kernel of the LUD application over all problem sizes. B) shows the corresponding Local Memory Address Entropy for the `perimeter` kernel over the tiny problem size.

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**.

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, 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. It allows developers to be able to quickly evaluate AIWC features. Which may allow the effectiveness of vectorization to be evaluated or to gauge the baseline predictability of memory access and branch behaviour. For instance, reordering a for-loop would change both the branching and memory access entropy scores. Additionally, the bottlenecks when vectorizing codes can be evaluated by examining the mean vectorization in the kiviat diagrams. This change would allow the suitability of a code on a range of expected data sets to be tested between AIWC runs.

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. This corresponds to repeatedly performing elimination in order to solve the lower and upper matrices, this iterative method results in solving intermediate and incrementally smaller matrices. Computing the memory access entropy with AIWC for each of these intermediate results shows increasingly smaller memory accesses, this is intuitive since smaller matrices will require more localised memory accesses. 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.

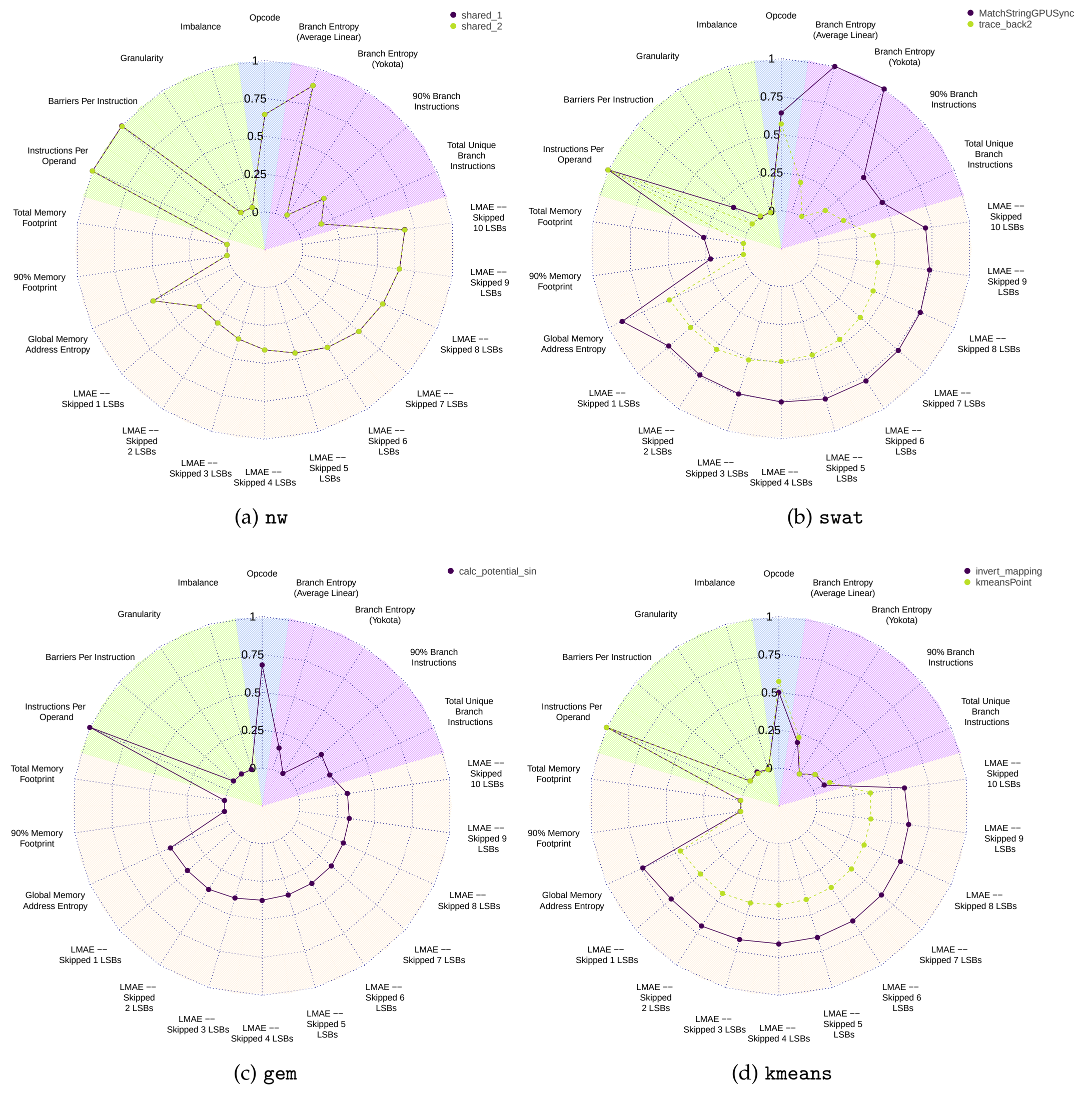
4.5 Use Case: AIWC analysis of OpenDwarf bioinformatics related benchmarks

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. The **small** size was selected since `hmm`, `gem` and `swat` benchmarks are from the fixed benchmarks – they only offer one size – however, the execution times typify those seen in the **small** sized `nw` and `kmeans` applications.

The corresponding execution times of these applications is presented in Figure 4.6. 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 – shown in Figure 4.6(a). The Nvidia 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 when examining execution times in Figure 4.6(b) – 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 Figure 4.6 (c).



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

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

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

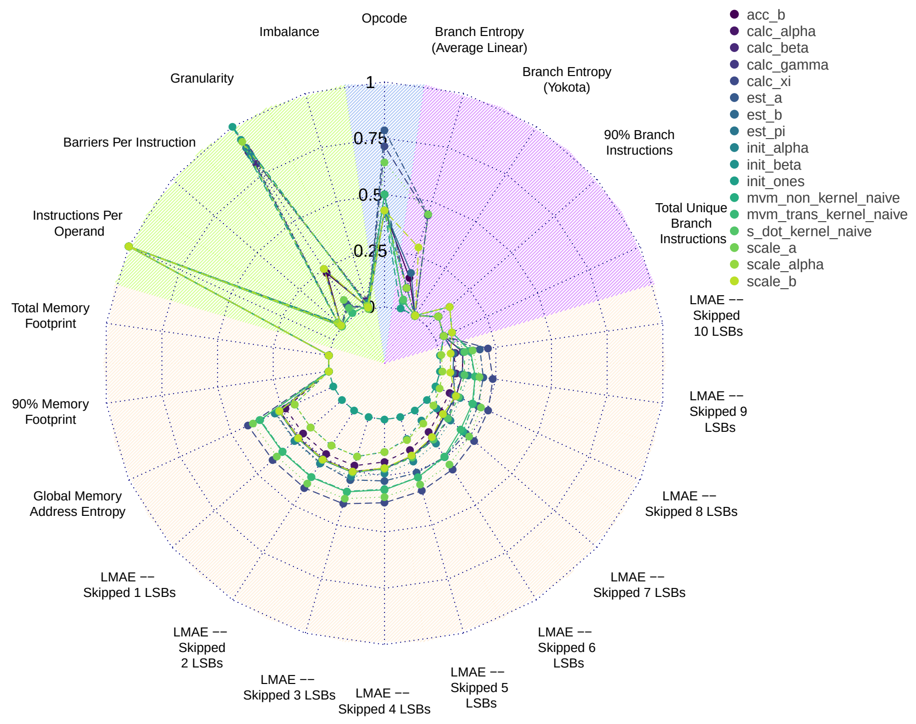


Figure 4.5: Architecture-Independent Workload Characterization features for the `hmm` bioinformatics benchmark.

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 Figure 4.6(d) show that both modern GPUs and older HPC GPUs perform equally well as CPUs for this benchmark, for larger problem sizes the GPUs outperform the CPU since a sufficient amount of work has been given and is presented in the Appendix (Figure A.2).

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 4.6 (e) 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.

4.6 Usage and Limitations

We believe that AIWC will be useful to diversity analysis, to this end, this Section presents information about using the tool. The AIWC plugin is only ≈ 1000 lines of code and it is

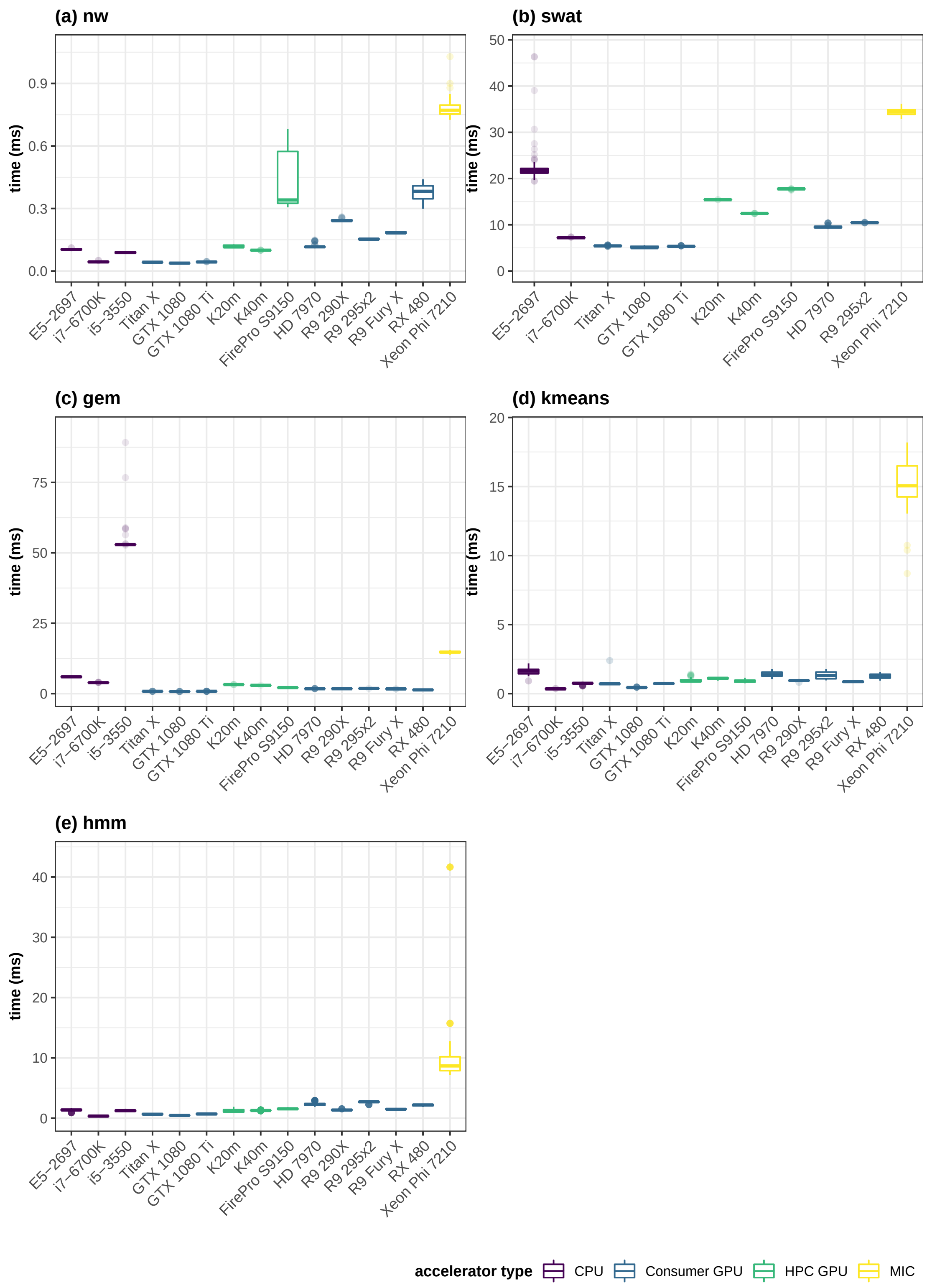


Figure 4.6: EOD runtimes for **small** problem sized bioinformatics benchmarks.

Table 4.3: Overhead of the AIWC tool on the `fft` benchmark and the Intel i7-6700K CPU.

	time			memory		
	usage (ms)		increase	usage (MB)		increase
	without AIWC	with AIWC		without AIWC	with AIWC	
tiny	0.04	73.4	≈1830×	80.0	85.9	1.07×
small	0.2	427.8	≈2800×	75.9	149.0	1.96×
medium	2.9	12420	≈4300×	101.4	636.8	6.28×
large	19.6	69300	≈3540×	203.8	2213.2	10.86×

available as a fork of Oclgrind and can be publicly found on GitHub[1]. To use AIWC over the command line it is passed the appropriate `--aiwc` argument immediately after calling the oclgrind program. An example of its usage on the kmeans application is shown below:

```
oclgrind --aiwc ./kmeans <args>
```

The collected metrics are logged as text in the command line interface during execution and also in a csv file, stored separately for each kernel and invocation. These files can be found in the working directory with the naming convention `aiwc_`α`_`β`.csv`, where α is the kernel name and β is the invocation count.

AIWC imposes significant overheads compared to normal execution, for illustration, we examined the overheads of using AIWC on the `fft` benchmark – from Chapter 3. The `fft` benchmark was selected as it has average runtime results – it falls roughly in the middle of the other benchmarks. Table 4.3 shows the relative overhead in terms of the elapsed time per kernel invocation and the maximum resident set size (peak virtual memory usage) during the benchmark execution, the results report with and without AIWC on four sizes of the `fft` benchmark on the Intel i7-6700K CPU. These results were collected with LibSciBench, for the kernel execution time, and Unix GNU time tool for the maximum resident set size. The execution times are the mean time from collecting a two second sample – the `fft` benchmark invokes the top level kernel many times during a two second run depending on problem size and the choice of OpenCL device.

We see that executing the same application on a simulator instead of directly on the Intel OpenCL runtime has significant performance costs, both in terms of execution time and memory usage. AIWC takes 1800-4300× longer to execute depending on the problem size and uses 1.07×, 1.96×, 6.2×, and 10.8× more memory as the problem size increases from tiny, small, medium and large respectively. The large memory footprint was limiting for us on one of the benchmarks; we encountered an issue with running the largest `lud` application where we exhausted the available RAM on our test system (16 GB), this was overcome by running the same experiment on a system with more RAM. The memory usage of AIWC is due to storing the index of every memory location accessed during the simulated kernel run, it is needed for the local and global memory accesses entropy metrics which are calculated over different striding distances once the kernel has finished. Instead of these addresses being stored in memory they instead could be written to disk. Until this improvement is

[1]https://github.com/BeauJoh/Oclgrind

performed, alternatives exist if the user is running out of memory, instead of executing the full range of kernel invocations to completion – since some applications will repeat kernel execution hundreds or thousands of times to completion – the developer could use AIWC for performance analysis on just a few iterations or a subset of the larger problem. The performance of the tool was not a limiting factor on a majority of the codes examined with AIWC taking just a few minutes to be generated on large problem sizes.

The use of a simulator incurs performance penalties when comparing to using hardware directly. For comparison, on the Intel Xeon Gold 6134 CPU running at 3.20 GHz and on the tiny size of `kmeans` benchmark, the mean kernel execution time was ≈ 44.4 µs, however when running Oclgrind on the same CPU hardware the mean kernel execution time increased to ≈ 8.7 ms, two orders of magnitude slower. The overhead of adding the AIWC plugin on the simulator results in a mean kernel execution time of ≈ 116 ms. In other words, using the simulator is almost 200× slower than directly using the same hardware, and using AIWC takes 10× longer than solely using Oclgrind, which is ≈ 2600× longer than immediately the CPU. When we consider the same `kmeans` application, on the large problem size, the mean kernel execution elapsed time directly on the Xeon Gold CPU was ≈ 395 µs, with the Oclgrind simulator included in the mix mean kernel execution times increased to ≈ 1 second, and with AIWC also added this further increased the mean time taken to ≈ 50 seconds. Simply put, on the large problem size, using the simulator takes ≈ 2500× longer than directly using the same hardware, and using AIWC takes 50 × longer again than solely using Oclgrind.

The computation times grow substantially when using the Oclgrind simulator and only worsen when AIWC is added into the mix. If we decide to use the Oclgrind simulator, say for profiling, and it runs 2.5k× slower, that is a significant hit on performance, the further slow down of using AIWC is only an inconvenience. However, this is not seen as restrictive once we consider the one-shot nature of AIWC for testing and profiling instead of it being needed at run-time for scheduling.

The envisaged use of AIWC is that it is only run once, for instance, a developer wished to examine the characteristics of the kernel in order to identify suitability for accelerators or verify that a high degree of SIMD vectorization had been achieved. In the predictive scheduling setting, AIWC would be run on the codes prior to them being shipped/delivered; since these characteristics are collected by the developer on a realistic problem size, the metrics can be included as a comment in each kernel's SPIR code, and the scheduler can use them by evaluating the shipped metrics on the model. Given the proposed workflow, the overhead added by AIWC is not significant or prohibitive to the prediction and scheduler pipeline. The predictive model is presented in detail in Chapter 5.

Examples of how AIWC metrics can be used for diversity analysis and device predictions are presented as Jupyter artefacts[2] [3].

[2]https://github.com/BeauJoh/aiwc-opencl-based-architecture-independent-workload-characterization-artefact
[3]https://github.com/BeauJoh/opencl-predictions-with-aiwc

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 parallel workloads. The collected 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 accelerator 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 bottlenecks 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 a few kernels with higher branch and memory access entropies, which suggests that CPUs are critical to achieving good performance on these applications.

Chapter 5

Making Performance Predictions for Scheduling

Predicting the performance of a particular OpenCL application on a selected accelerator 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 [113], and choosing the wrong device results in slower and more energy-intensive computation [114]. The penalties involved in selecting the wrong accelerator for a given code is shown in the ranges between the best and worst execution times of the Figures 3.1 and 3.2 in Chapter 3. Thus, accelerator selection is critical to making optimal scheduling decisions to achieve good performance in a heterogeneous supercomputing environment. The ability to predict which device is optimal without having to first run a new code on all devices first is desirable. AIWC metrics – from Chapter 4 – provide a good representation of the characteristics of codes; we propose that these metrics can be used directly for the prediction of execution times over various accelerators. In this chapter, we develop a model that employs the AIWC features to make accurate predictions over a range of current accelerators. The execution times from Chapter 3 are used as response variables and the AIWC metrics are used as input variables to train this model. This chapter discusses how the model is developed and optimized for our data, an evaluation is presented along with a discussion of its use case on predictions for scheduling.

There are many current projects which attempt task scheduling on heterogeneous multicore architectures, these include StarPU [148], Ompss [149] and CoreTSAR [150] Many of these schedulers track dependencies within tasks and target either compute, bandwidth or latency by scheduling work to the most appropriate accelerator at the granularity of function call level or the work inside a single parallel region. The history of the performance of a task is used to determine the optimal device to use in the future. However, the nature of this approach means these schedulers must execute a new kernel code on all available accelerators before any scheduler smart strategies can be used. Our model can predict the expected execution time of a kernel before it is run; if this prediction is incorrect, these schedulers can default back to their old strategy of measuring the performance on all available accelerators. This work was published in the 16th International Conference on High Performance Computing & Simulation, HPCS 2018 [131].

5.1 Model Development

This section outlines how AIWC features are used to build a model which accurately predicts the execution times of a previously unencountered OpenCL code over the range of available devices. The AIWC metrics are generated over the EOD benchmark suite and serve as input variables, while all the execution times presented in Chapter 3 serve as response variables for model training. The generation of a random forest model was used to learn each machine profile. This model should be able to offer accurate predictions of execution times based only on the AIWC metrics – this would be used in the real world by having the trained model available to the scheduler, the AIWC metrics shipped with kernel codes, and the scheduler making accelerator selections entirely by querying the model with these metrics.

We initially performed an evaluation with general linear mixed (GLM) models but found the random forest model to offer a higher accuracy of predictions. The evaluation and associated prediction results of the GLM are presented in Appendix B.

The methodology to develop the model is outlined in this section. All tools used are open source, and all code is available in the respective repositories: [144] and [147]. In the remainder of this section, we outline the experimental setup, describe how the initial predictive model was constructed, examine various optimizations to improve the accuracy of the model and conclude with a study on how the model performs with unencountered codes.

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 to fit a random forest regression model. The metrics were generated over 4 problem sizes for each of the 12 applications / 37 kernels. Response variables were collected following the same methodology outlined in Chapter 3 – 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 each kernel collected at four different problem sizes. Our data comprises of 2200+ unique mean runtime entries but when coupled with the AIWC metrics for each observation our data comprises up to 64k entries in total; we train our model with 20% of the data entries (randomly selected) and use the remaining 80% for evaluation.

5.1.2 Constructing the Random Forest Performance Model

The random forest model is used to estimate the execution times based on the 28 AIWC metrics for all 64k observations. This regression model uses the measured execution times from EOD as the response and AIWC metrics as predictor variables. Other predictive models such as linear regression, Principal component regression, generalised linear models, vectorized generalised additive models, however, were discarded due to their multivariate outcomes. K-nearest neighbours were also considered but the dimensionality of the search-space was too high. Feed-forward general networks with multiple hidden layers were considered but the sample size was insufficient to ensure valid convergence of the learning function and also the network structure was too simple for the complicated manifold induced by the data. Random forests were selected since they are a well known robust performer, quick to compute and easy to store the computed object model. Random forests do not assume any underlying structure in the data but rather finds these automatically using tree pruning methods – and they are good at segmenting the data for individual regression problems and thus are well suited to our goals; building a performance prediction model that can select between various devices based solely on that kernel's AIWC feature-space.

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 [151] was used for the development of the regression model. The *ranger* package provides computationally efficient implementations of the Random Forest model [152] 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.

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. Autotuning to determine the suitability of these parameters has been performed by Ließ et al. [153] 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 [154]. 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

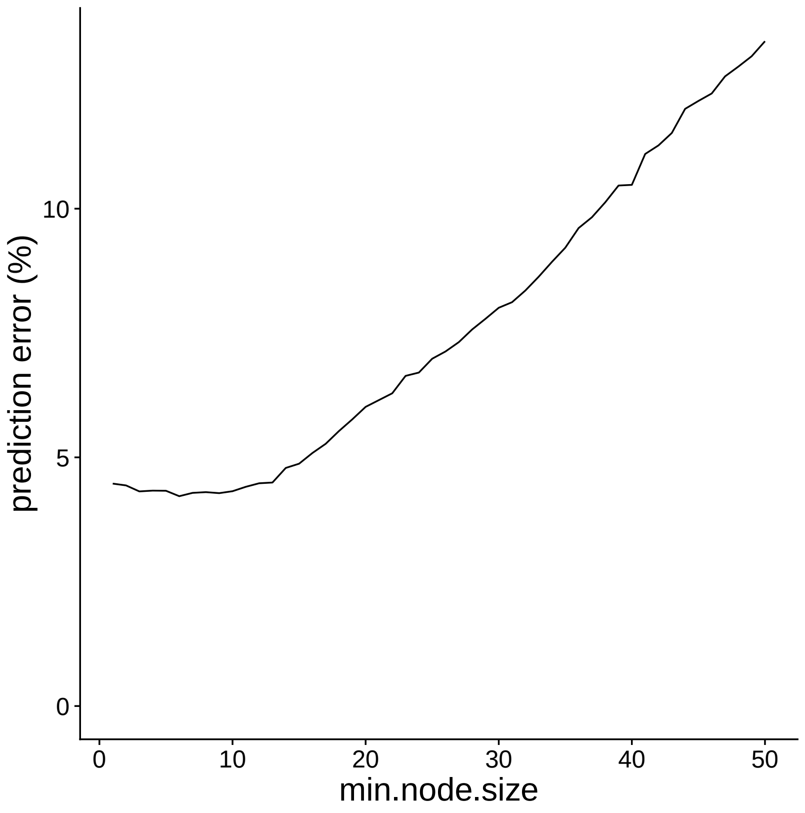


Figure 5.1: Full coverage of min.node.size with fixed tuning parameters: num.trees = 300 and mtry = 30.

the resultant fitted model, by simultaneously changing the parameters (num.trees, mtry and min.node.size). The *optim_sa* function allows defining the search space of interest, a starting position, the magnitude of the steps according to the relative change in temperature and the wrapper around the ranger function (which parses the 3 parameters and returns a cost function — the predicted error). It allows for an approximate global minimum to be detected with significantly fewer iterations than an exhaustive grid search.

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

Figure 5.2 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

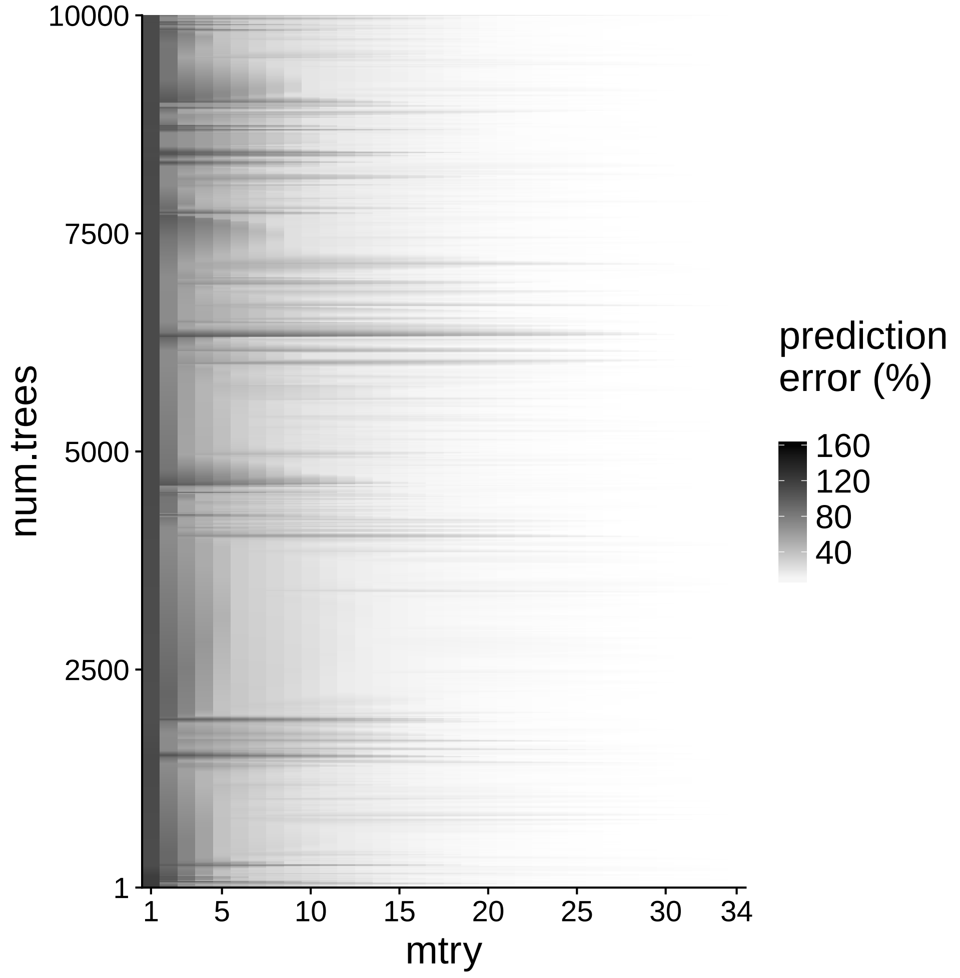


Figure 5.2: Full coverage of num.trees and mtry tuning parameters with min.node.size fixed at 9.

allowed to execute until a global minimum was found. At each step of optimization a full trace was collected, where all parameters and the corresponding out-of-bag prediction error value were logged to a file. This file was finally loaded, the points interpolated using the R package *akima*, without extrapolation between points, using the mean values for duplication between points. The generated heatmap is shown in Figure 5.2.

A lower out-of-bag prediction error is better. For values of mtry above 25, there is a 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.

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
```

5.1.3 Parameters for the Random Forest Performance Model

The selected model should be able to accurately predict execution times for a previously unencountered 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 64k entries are stored in an R data frame – a table or a two-dimensional array-like structure.

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 Tuning the Random Forest Model

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

We measured how model fit improves with the number of kernels used in training, following the method presented in Algorithm 2. The set of unique kernels available during model development is denoted by k (37 kernels in this study), s is the maximum number of sample models (including different combinations of kernels) to evaluate for each number of kernels $1..|k|$, ϕ is a data frame of the combined AIWC feature-space with measured runtime results. The parameters to the random forest model were fixed at num.trees = 505, mtry = 30 and min.node.size = 9, according to the methodology in Section 5.1.3.

The results presented in Figure 5.3 show the mean absolute error of models trained on varying

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.

```
$s \leftarrow 500$
$k \leftarrow$unique(kernel)
for $i \leftarrow 1$ to $length(k)$ do
    $v_p \leftarrow []$
    $v_m \leftarrow []$
    for $j \leftarrow 1$ to $s$ do
        $x \leftarrow$shuffle($k$)
        $y \leftarrow x[1 .. i]$
        training data $\leftarrow$ subset($\phi$, kernel == $y$)
        test data $\leftarrow$ subset($\phi$, 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($e$)
```

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 (in log(µs)) against the predicted execution times from the trained model. Each point represents a single combination of kernel and problem size – there are 64k points in total. The plot shows a strong linear correlation indicating a good model fit. Under-predictions typically occur on four kernels over the medium and large problem sizes, while over-predictions occur on the tiny and small problem sizes. However, these outliers are visually over-represented in this figure as the final mean absolute error is low, at ~0.1.

5.2.1 Predicting Kernel Execution Time

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

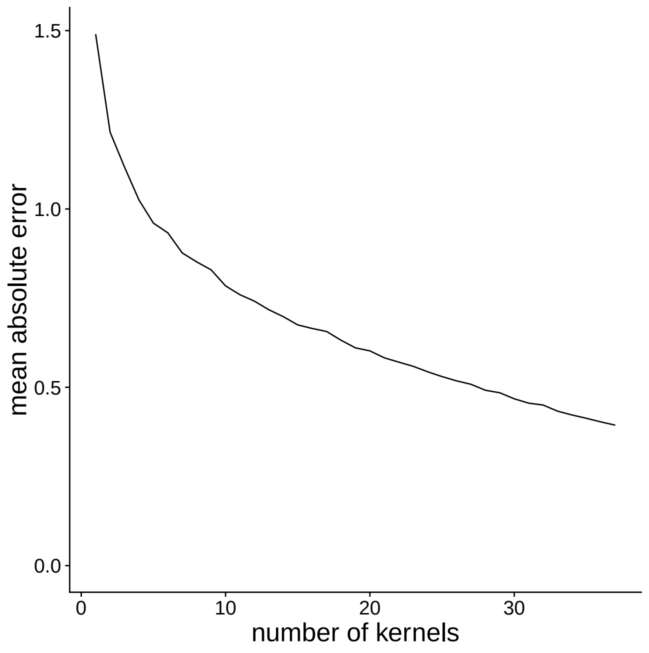


Figure 5.3: Prediction error across all benchmarks for models trained with varying numbers of kernels.

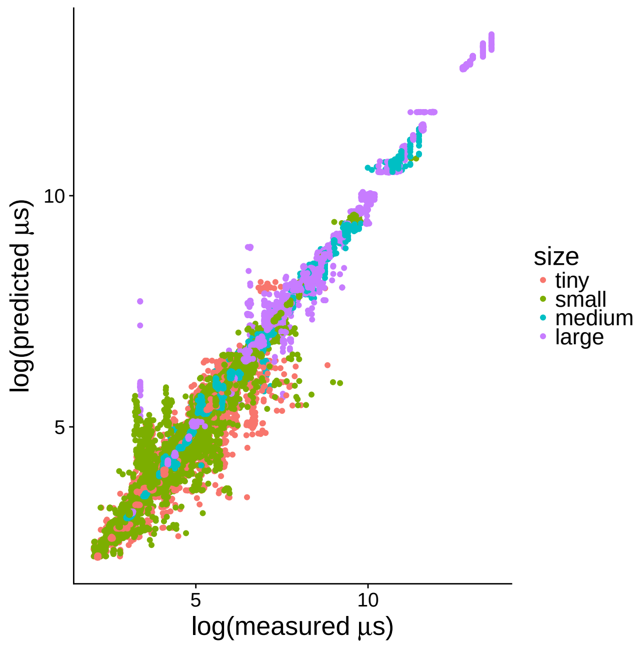


Figure 5.4: Predicted vs. measured execution time (in log(µs)) for all kernels.

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. The kernels (y-axis) between each of problem size do not align due to the number of supported applications, and kernels, in each problem size – this is discussed in Chapter 3.

In general, we see highly accurate predictions which on average differ from the measured experimental run-times by 1.2%, which correspond to actual execution time mispredictions of 8 µs to 1s 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 samples available to influence the model, this may lead its poorer predictions.

5.2.2 Choosing The Optimal Accelerator for a Kernel

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 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 predicted time. Thus, a perfect prediction occurs where the measured time – square – fits perfectly within the predicted – diamond – as shown in the legend.

The purpose of showing these results is to highlight the setting in which they could be used – on the supercomputing node. In this instance, it is expected a node to be composed of any combination of the 15 devices presented in the Figure 5.6. Thus, to be able to advise a scheduler which device to use to execute a kernel, the model must be able to correctly predict on which of a given pair of devices the kernel will run fastest. For any selected pair of devices, if the relative ordering of the measured and predicted execution times is different, the scheduler would choose the wrong device. In almost all cases, the relative order is preserved using our model. In other words, our model will correctly predict the fastest device in all cases – with one exception, the `kmeansPoint` kernel. For this kernel, the predicted time of the fiji-furyx is lower than the hawaii-r9-290x, however the measured times between the two shows the furyx completing the task in a shorter time. For all other device pairs, the relative order for the `kmeansPoint` kernel is correct. Additionally, the `lud_diagonal` kernel suffers from systematic under-prediction of execution times on AMD GPU devices, however, the relative ordering is still correct. As such, the proposed model provides sufficiently accurate execution time predictions to be useful for scheduling to heterogeneous compute devices on supercomputers.

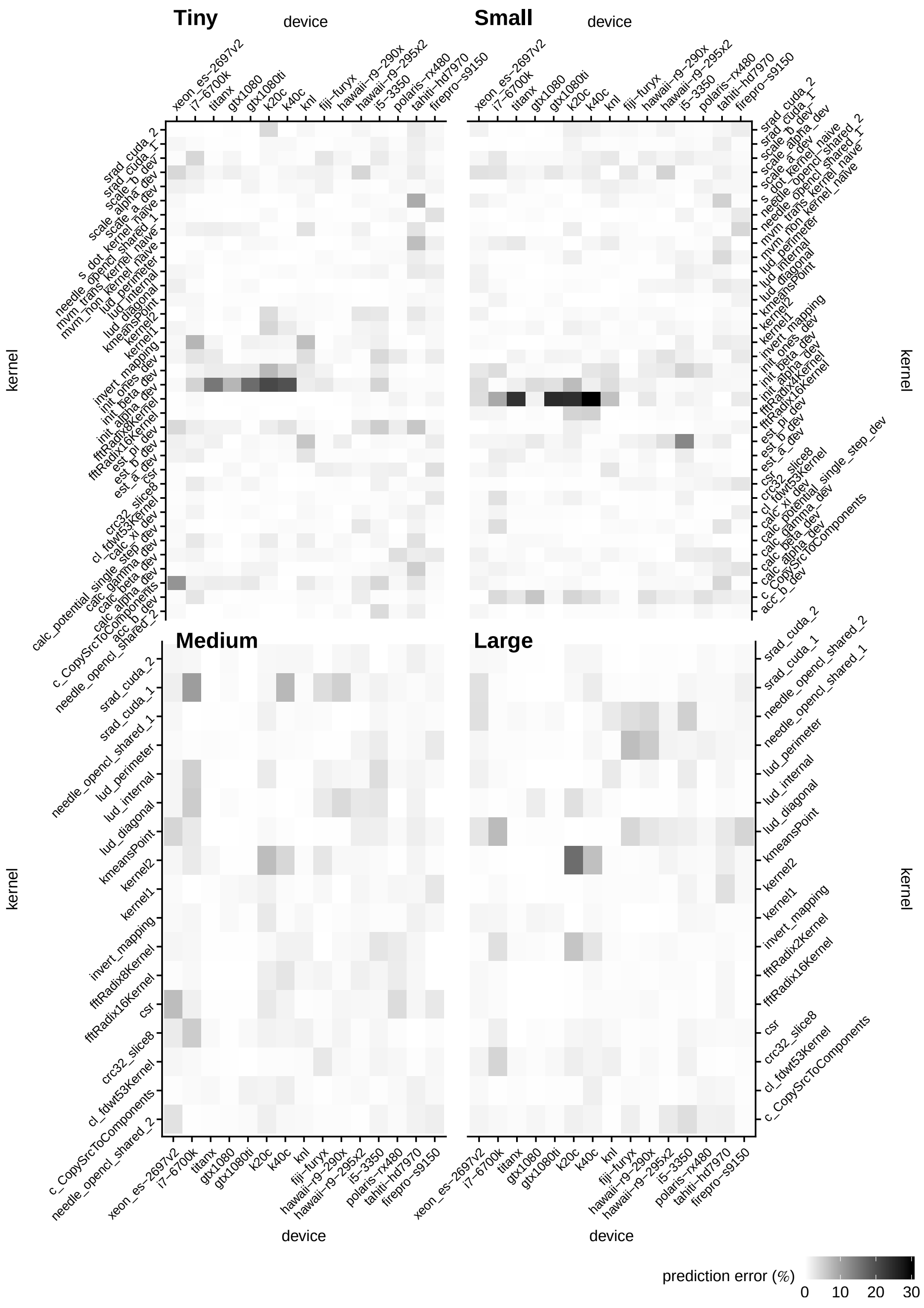


Figure 5.5: Error in predicted execution time for each kernel invocation over four problem sizes.

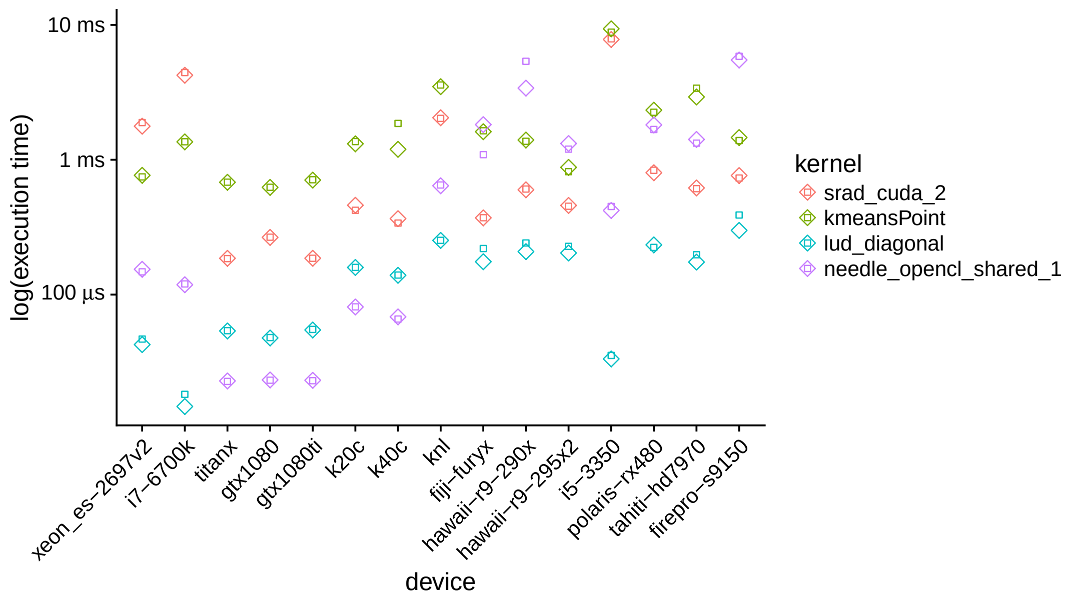


Figure 5.6: Mean measured kernel execution times compared against mean predicted kernel execution times to perform a selection of kernels on large problem sizes across 15 accelerator devices. The square indicates the mean measured time, and the diamond indicates the mean predicted time.

5.3 Discussion

The AIWC metrics generated from the full set of Extended OpenDwarfs kernels are used as input variables in a regression model to predict kernel execution time on each device [131]. 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. The model predictions differed from the measured experimental results by an average of 1.2%, had a mean predicted accuracy of 98.8%, which corresponds to the actual execution time mispredictions of ≈ 8 µs to ≈ 1 s according to problem size.

A major motivation of our prediction work is on reducing energy consumption of modern supercomputers. Typically, minimizing execution time also reduces energy if a race-to-halt strategy is used thus predicting the execution time also can be used to directly predict energy consumption. We acknowledge that this is not always the case, for instance, control parameters might slow down the processor slightly for a much larger reduction in energy use. However, we believe the same predictive modelling strategy outlined in this Chapter can be employed to predict energy consumption directly – by using the joules collected by RAPL and NVML in LibSciBench instead of the execution times as the response variable – but this is left as future work.

There are limitations of the random forest model for extrapolation of data. Namely, if you have different kernels then you are going to need to collect lots of data concerning the performance of these kernels and then re-fit the random forest model again. This is not very efficient to have to re-train but if you don't then there is a strong risk of poor prediction. Other approaches are more robust in this situation.

Other potential critiques of using the random forest for this problem include the potential for comparatively large model storage as dimensionality increases, and that there is no feedback from the model as to why a particular device choice is optimal. The metrics used in the assessment are quite limited and more detailed error investigation analysis could include confidence scores or uncertainty on the predictions based on a more comprehensive error analysis which explores the levels of prediction uncertainty associated with each kernel.

If the predictive model were used in a real-world setting – say on an HPC system – 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 with the program settings that will be used at runtime. Our proposed solution uses AIWC as a plugin to the Oclgrind tool, which is already widely used by OpenCL developers. These metrics are included as a comment into the kernel – either in source or SPIR form. The scheduler extracts these metrics at runtime and evaluates them with the model to make performance predictions on the available devices (if the runtime settings lead to substantially different AIWC features to the ones collected than the runtimes predictions may be inaccurate). 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. 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.

The predictive model can choose the most appropriate accelerator for a given kernel. Given a workload of varied applications, execution time predictions can be used to choose which nodes to allocate for each application. The execution time predictions can be used to determine whether to migrate applications between nodes e.g. when new nodes become available.

AIWC and the prediction methodology could also be used to guide system designers on the optimal mix of accelerators for future supercomputers. For instance, the range of codes expected to run on the machine can be examined with AIWC before any hardware is purchased. The predictive model can be trained by the hardware vendor using EOD (or other benchmark suites) and the trained model can be used by an HPC facility owner to predict the performance of their own suite of codes, without the need to provide the characteristics of these codes to the vendor.

Chapter 6

Conclusions and Future Directions

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. We 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.

This thesis demonstrates that architecture-independent features are sufficient to characterize codes and to predict performance so as to schedule the optimal device. We proposed an extended benchmark suite which supports diverse accelerators and demonstrated the performance of these devices over a large number of codes/kernels. We developed the Architecture-Independent Workload Characterization Tool (AIWC) to examine the structural characteristics and implementation constraints of kernels to offer an understanding of the algorithm without having to consider the hardware. Finally, we used these AIWC metrics to identify the most suitable device for each kernel using a predictive regression model.

The extended OpenDwarf suite provides a reliable benchmark suite with multiple problem sizes and high precision measurements. Reproducible results can be generated quickly and over a range of heterogeneous accelerator devices. A full set of execution times and other performance metrics were generated using 15 devices over 12 benchmarks and 42 kernels. The energy and hardware events metrics allowed direct performance evaluations to be made between devices.

The exploration of the differences in the characteristics of codes is used to examine this variation in performance between heterogeneous devices. To this end, AIWC was developed and is capable of identifying the fundamental characteristics of programs free from any specific device. AIWC allowed extraction of a set of pre-defined features or characteristics for analysis of kernels. The tool was used in diversity analysis – see Appendix C – which is essential when assembling benchmark suites and justifying the inclusion of a benchmark. These AIWC metrics were used for creating a predictive model of the performance of OpenCL kernels on different hardware devices and settings. This model can be incorporated into existing HPC schedulers and has no run-time overhead – a code is instrumented once only using AIWC and the resulting features are embedded into the header of each kernel code to be used by the scheduler at runtime.

The use of accelerators is pervasive in HPC and will become more so in the future. We showed that AIWC and the predictive model support 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, which we expect will lead to more efficient scheduling of HPC workloads.

The contributions of each chapter are now discussed in greater detail, concluding 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 in Chapter 3. We selected OpenDwarfs as the basis for 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) this work was current with the latest use as an evaluation of OpenCL for FPGA devices [11].

We removed hardware specific optimizations from codes that would either diminish performance or crash the application on other devices, these optimizations adversely affect 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 reported results for four different problem sizes, selected according to the memory hierarchy of CPU systems as motivated by Marjanović's findings [134]. These can now be easily adjusted for next-generation accelerator systems using the methodology outlined in Section 3.2.4.

All of the benchmarks presented in the most recent (2016) OpenDwarfs [11] paper were rerun on current hardware. This was done for two reasons, firstly to attempt to replicate the original findings to the modern systems and secondly to extend the usefulness of the benchmark suite. Re-examining the original codes on a 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 favoured 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. We fixed these issues in the Extended OpenDwarfs benchmark suite to support multiple devices, over multiple problem sizes – so it can be applied to embedded systems as well as top end scientific processors – and added the DWT and a stable FFT implementation to allow the benchmarks to span as many of the Dwarfs as possible.

Finally, a major contribution of this work was to integrate LibSciBench into the benchmark suite, which adds high precision timing 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 benchmark. The use of LibSciBench has also increased the reproducibility of timing data for both the current study and on new architectures in the future.

The Extended OpenDwarfs Benchmark Suite can be found on Github[1] and a Jupyter artefact demonstrating its usage is also available. [2]

6.2 AIWC

In Chapter 4, we 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.

Each OpenCL kernel presented in Chapter 3 of EOD was inspected using AIWC. Analysis using AIWC helps to 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 when codes will need to be run on a wider range of accelerators.

AIWC is an additional tool to be used by developers and does not attempt to replace classical device-specific instrumentation and profiling. It is intended to integrate with existing development workflows, 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 to examine its architecture-independent workload characteristics. Optimization could be guided by AIWC metrics but does not exclude

[1]https://github.com/BeauJoh/OpenDwarfs

[2]https://github.com/BeauJoh/Benchmarking-bioinformatics-workloads-and-exploring-suitability-for-heterogeneous-HPC-artefact/blob/master/codes/AIWC spaces of bioinformatics workloads.ipynb

the ability to use hardware performance counters, PIN events or vendor-specific profiler tools. AIWC is available on Github[3], there is a Jupyter artefact to demonstrate how metrics are collected on the EOD benchmark suite and how the figures are produced[4], there is also a Binder version of the artefact also available.[5]

6.3 Performance Prediction

A highly accurate model, capable of predicting execution times of OpenCL kernels on specific devices based on the computational characteristics captured by the AIWC tool was presented in Chapter 5. 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 – Section 6.4.1 outlines future work to address this – 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.

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 ready for a code 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 and is based on the measured bias from the recorded runtimes – if the node were updated the EOD suite would need to be rerun over updated devices and the performance runtimes incorporated 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.

Using the same predictive model for run-times generated over compute devices spanning 6 years and four processor generations shows both that OpenCL has reached a position of maturity and stability, and 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 we expect this same methodology to continue to be equally accurate on future systems. The model is

[3]https://github.com/BeauJoh/Oclgrind

[4]https://github.com/BeauJoh/aiwc-opencl-based-architecture-independent-workload-characterization-artefact/blob/master/AIWC-figures.ipynb

[5]https://mybinder.org/v2/gh/BeauJoh/aiwc-opencl-based-architecture-independent-workload-characterization-artefact/master

available as a Jupyter workbook[6] which allows users to run new predictions, automatically compare them to the measured runtimes and provides transparency around how each figure in Chapter 5 was generated.

6.4 Future Directions

Each of the lines of investigation described in this thesis has a future – from examining benchmark diversity to improving the characterization of codes. The following sections, however, focus on directions that may have the most significant impact in shifting us toward understanding the characteristics of codes and how to best improve the performance on the accelerator rich systems of the future.

6.4.1 EOD

EOD and the work presented in Chapter 3 resulted in a flexible benchmark suite that can be run quickly and reliably on a range of accelerators and forms a foundation for testing AIWC and the predictive model. We started to use the OpenTuner[90] autotuning library to achieve the optimal performance of each device on all the benchmarks in EOD but realised that it is beyond the scope of this thesis. Others [88], [58], [59], [60], [89], [61] have shown that autotuners offer good performance for configuring OpenCL kernel parameters – such as local workgroup size – for the different accelerators and could be readily incorporated into EOD in a consistent manner. However, the presented execution times do not change the presented methodologies around workload characterization and prediction, individual features and the predictions may change with different tuning arguments but the use case is the same. The developer needs to instrument a kernel before it is shipped and the most accurate predictions will come from instrumenting under a realistic setting – tuning arguments included. Schedulers will need to take autotuning and optimization into account but our prediction methodology offers a good initial performance estimate without having to perform the historic approach of running the same kernel on all the devices.

The `cfd`, `bfs` and `tdm` benchmarks due to the unavailability of external software to generate the datasets lack multiple problem sizes. It would be nice to have this for completeness of the extensions presented in the EOD benchmark suite, but ultimately, is not the focus of this thesis.

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. This upper-bound for performance could arise from the AIWC analysis on each benchmark. Additional architectures such as FPGA, DSP and Radeon Open Compute based APUs will be considered.

[6]https://nbviewer.jupyter.org/github/BeauJoh/opencl-predictions-with-aiwc/blob/master/ OpenCL Performance Prediction using Architecture-Independent Features.ipynb

6.4.2 AIWC

Caparrós Cabezas and Stanley-Marbell [110] examine the Berkeley dwarf taxonomy by measuring instruction-level parallelism (ILP), 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 (TLP) was measured by analysing the block dependency graph. Whilst we propose alternative metrics to evaluate ILP (SIMD width) and the TLP (Total Barriers Hit and Instructions To Barrier) – a quantitative evaluation of the dwarf taxonomy using these metrics is left as future work. We expect that the additional AIWC metrics mirroring Caparrós Cabezas and Stanley-Marbells measurements will generate a comprehensive feature-space representation. This comprehensive feature-space will permit cluster analysis and comparison with the dwarf taxonomy.

A major limitation of running large applications under AIWC is the high memory footprint – as discussed in Chapter 4.6. Memory access entropy scores require a full recorded trace of every memory access during a kernel's execution. However, 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.

6.4.3 Performance Predictions

Our model currently predicts execution time, however, we expect that a similar model could be constructed to predict energy or power consumption. We have not yet collected the energy measurements over the wide range of devices required to construct such a model.

We have not examined which AIWC features are most important in the predictive model. Presenting a subset of the metrics may reduce complexity showing only the most important data may be more informative for the developer when making these considerations. Principal component analysis of these features was considered when evaluating potential modelling approaches and is included in Appendix C.

Kumar et al. [155] provide an interesting and different use of Shao's [107] ISA-independent features. They present Peruse, a tool to characterize the features of loops at an IR level to guide a programmer's efforts in locating loops suitable for parallel execution. In an approach similar to ours, they use machine-learning algorithms directly on ISA metrics to predict the accelerability of loops. The model they present predicts the speedup of loops with an accuracy of 79%. It is promising that a similar methodology has been developed based on the same intuition and common set of tools, and is exciting to see if both works could be combined in the form of scheduling abstract for-loops instead of OpenCL kernels. This would allow a language-agnostic approach to accelerator scheduling – say on C codes instead of depending on OpenCL specifically.

Following the work presented in this thesis, five additional research topics have become apparent and will be pursued. They fall outside of the original scope of this thesis but are nonetheless important.

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

Our prediction methodology can be used to evaluate the coverage/diversity of the benchmarks included in the EOD benchmark suite. This work is currently focused on augmenting EOD with synthetic benchmarks. The predictive model is used to make predictions on previously unseen codes against the trained set of EOD runtime results. These unseen codes are randomly generated using the OpenCL kernel generation framework (CLgen) by Cummins et al. [156] with a training corpus of all OpenCL applications available on GitHub. The previous success of our model to predict execution times across many devices with high accuracy has led 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, we expect that testing the model with synthetic benchmarking may identify gaps in the coverage provided by the existing suite of benchmarks, which would manifest as poor predictions on particular synthetic kernels. These poorly predicted kernels could be added back into the EOD benchmark suite – thus better encompassing the work expected to be run on these accelerator devices.

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

OpenCL was the optimal language for the evaluation of codes on the broadest range of accelerators required for this thesis, however, several other programming systems are commonly used for accelerators in HPC, including CUDA, OpenMP and OpenACC. The last two offer 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 [157] 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. We are currently writing an LLVM pass to generate OpenCL device payloads for AIWC from OpenMP and OpenACC.

6.4.6 Examining the Characteristics of Scientific Codes in Supercomputing with AIWC

Porting large HPC codes from conventional CPU architectures to accelerators is intensive on the developer. However, many codes currently run on supercomputer systems are legacy and as these systems increasingly utilize accelerators, more of this porting work will be required. Many of these codes were written in OpenMP making them a suitable target for the language-agnostic architecture-independent workload characterization. In addition to supporting scheduling as presented in this thesis, AIWC and the predictive model could be used to identify the primary characteristics of codes run on supercomputers. For instance, if the supercomputing centre knows which codes are likely to be frequently executed, by identifying the characteristics of these codes and the most suitable accelerators they can design nodes with the optimal accelerator configurations.

6.4.7 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).

Metrics from AIWC could be compared after applying device-specific optimizations to see how these features change and could identify performance-critical characteristics. The selection of best practices such as, "Intel 64 and IA-32 Architectures Optimization Reference Manual" for CPU and "CUDA C Best Practices Guide, Design Guide" for GPU could be taken from their respective source code and ported to OpenCL. The examination of the change of AIWC feature-spaces after each device specific optimization may suggest accelerator-agnostic optimization strategies.

6.4.8 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 code for a device untenable. Given the accuracy of the predictive model, there is a use-case for AIWC to augment this workflow. Speculative optimization changes could be made to an OpenCL code, the AIWC metric regenerated and predictive model queried, if the predicted device execution result is better it could indicate a suitable optimization. This would potentially take seconds instead of the long compile times when evaluating FPGA performance. The inherent difficulty of predicting application performance on a reconfigurable architecture is the ultimate test for the predictive model outlined in this thesis.

6.5 Closing Remarks

We hope the work presented in this thesis will serve as the basis for the scheduling of HPC workloads as accelerator usage becomes more prevalent in this space. Our next goal is to incorporate our methodology with AIWC and random forest models into the StarPU accelerator scheduler, this will serve as a prototype scheduler, which we hope will become the norm on the next generation of accelerator-based supercomputers. We hope our work on benchmarking, workload characterization and prediction has made a modest contribution and that the proposed techniques may help HPC developers make sense of an increasingly complex hardware and software environment.

Appendices

Appendix A

Time Results

The primary purpose of including these time results is to demonstrate the benefits of the extensions made to the OpenDwarfs Benchmark suite. In this appendix, we present all runtime results collected from EOD and use the benchmarks to assess and compare performance across the chosen hardware systems. 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 12 benchmarks running on 15 different devices representing four distinct classes of an accelerator. For eight of the benchmarks, we measured multiple problem sizes and observed distinctly different scaling patterns between devices. This underscores the importance of allowing a choice of problem size in a benchmarking suite. The remaining four benchmarks only support one fixed problem size and are included in Figure A.5.

Figures A.1, A.2, A.3 and A.4 shows the distribution of kernel execution times for the benchmarks with multiple problem sizes. The **tiny** and **small** sizes for the `kmeans`, `lud`, `csr` and `dwt` benchmarks are presented in Figure A.1 results, the **medium** and **large** problem sizes are presented in Figure A.2. Similarly, the remaining four applications which support multiple problem sizes – `fft`, `srad`, `crc` and `nw` – display the time results for **tiny** and **small** in Figure A.3, and **medium** and **large** times are shown in Figure A.4.

Some benchmarks execute more than one kernel on the accelerator device; the reported iteration time is the sum of all compute time spent on the accelerator for all kernels. Each benchmark corresponds to a particular dwarf: From Figures A.1 and A.2 (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 A.3 (a) and A.4 (a) (`fft`) represents Spectral Methods, (b) (`srad`) represents the Structured Grid dwarf, (c) (`crc`) represents Combinational Logic and (d) (`nw`) represents Dynamic Programming.

Finally, Figure A.5 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 A.5 (a), the Backtrack & Branch and Bound dwarf is represented by the (`nqueens`) application in Figure A.5 (b), (`hmm`) results from Figure A.5 (c) represent the Graphical Models dwarf and (`swat`) from Figure A.5 (d) also depicts the Dynamic

Programming dwarf.

The results are coloured according to the accelerator type: purple for CPU devices, blue for consumer GPUs, green for HPC GPUs, and yellow for the KNL MIC. Examining the transition from tiny to large problem sizes in Figures A.3 (b) and A.4 (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 A.3 and A.4 – 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 3.2.) Despite their larger memory sizes, the clock speed of all HPC GPUs is slower than all evaluated consumer GPUs. While the HPC GPUs (devices 7-9, in green) outperformed consumer GPUs of the same generation (devices 4-6 and 10-13, in blue) 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 A.1 and A.2, and in (a) from Figures A.3 and A.4.

Increasing problem size also hinders the performance in certain circumstances for GPU devices. For example, (b) from Figures A.3 and A.4 shows a widening performance gap over each increase in problem size between AMD GPUs and the other devices.

Execution times for `crc` are lowest on CPU-type architectures, probably due to the low floating-point intensity of the CRC computation [145]. In general, the performance on the Xeon Phi 7210 MIC is poor due to the lack of support for wide vector registers in Intel's OpenCL SDK. The low clock frequency and inability to exploit sufficient levels of parallelism on tiny and small problem sizes usually means it is the worst performer on these sized benchmarks. As we move onto larger problem sizes the MIC outperforms the AMD GPUs on the largest `lud` and `nw` benchmarks. Similarly, for the large `srad` benchmark the MIC bests most of the CPUs. The `crc` benchmark is a standout in benchmarks for the MIC; It is one of the only applications where the MIC is competitive with the performance on the whole mix of accelerators. Starting with the tiny size, it experiences comparable performance to all of the older GPUs, for the small size it offers similar performance to the latest Nvidia GPUs, and for the medium and large problem sizes it is almost the best performing device rivalling the CPU accelerators.

For the fixed problem sized benchmarks, presented in Figure A.5, the per kernel invocation is relatively low regardless of device selected for the (a) `gem` or (b) `nqueens` benchmark. The newer Nvidia GPUs collectively tended to be the best-performed accelerator on `gem` taking $\approx 110\mu$s while the MIC saw the worst performance at 0.85 ms. The `nqueens` benchmark saw the i7-6700K and i5-3550 CPUs finish the kernel in $\approx 80\mu$s to $\approx 100\mu$s per invocation, respectively, again the MIC had the worst performance at 900μs on average. Figures (c) `hmm` and (d) `swat` are more computationally intensive and took longer to complete. The `hmm` benchmark shows the CPU and modern Nvidia GPUs performing equally well < 1ms, the older AMD and HPC GPUs ranged from 1-3ms, and the MIC averaged 7.5ms per run. Finally, `swat` had the modern Nvidia GPUs as the fastest devices at $\approx$ 5ms and ranged up to 40ms on the MIC which was the slowest device for this benchmark.

Predicted application properties for the various Berkeley Dwarfs are evident in the measured runtime results. For example, Asanović et al. [15] state that applications from the Spectral Methods dwarf are memory latency limited. If we examine `dwt` and `fft` – the applications which represent Spectral Methods – in Figure A.2 (d) and Figure A.4 (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 memory access to main memory from the CPU systems. As a further example, Asanović et al. [15] state that the Structured Grid dwarf is memory bandwidth limited. The Structured Grid dwarf is represented by the `srad` benchmark shown in Figure A.4 (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.

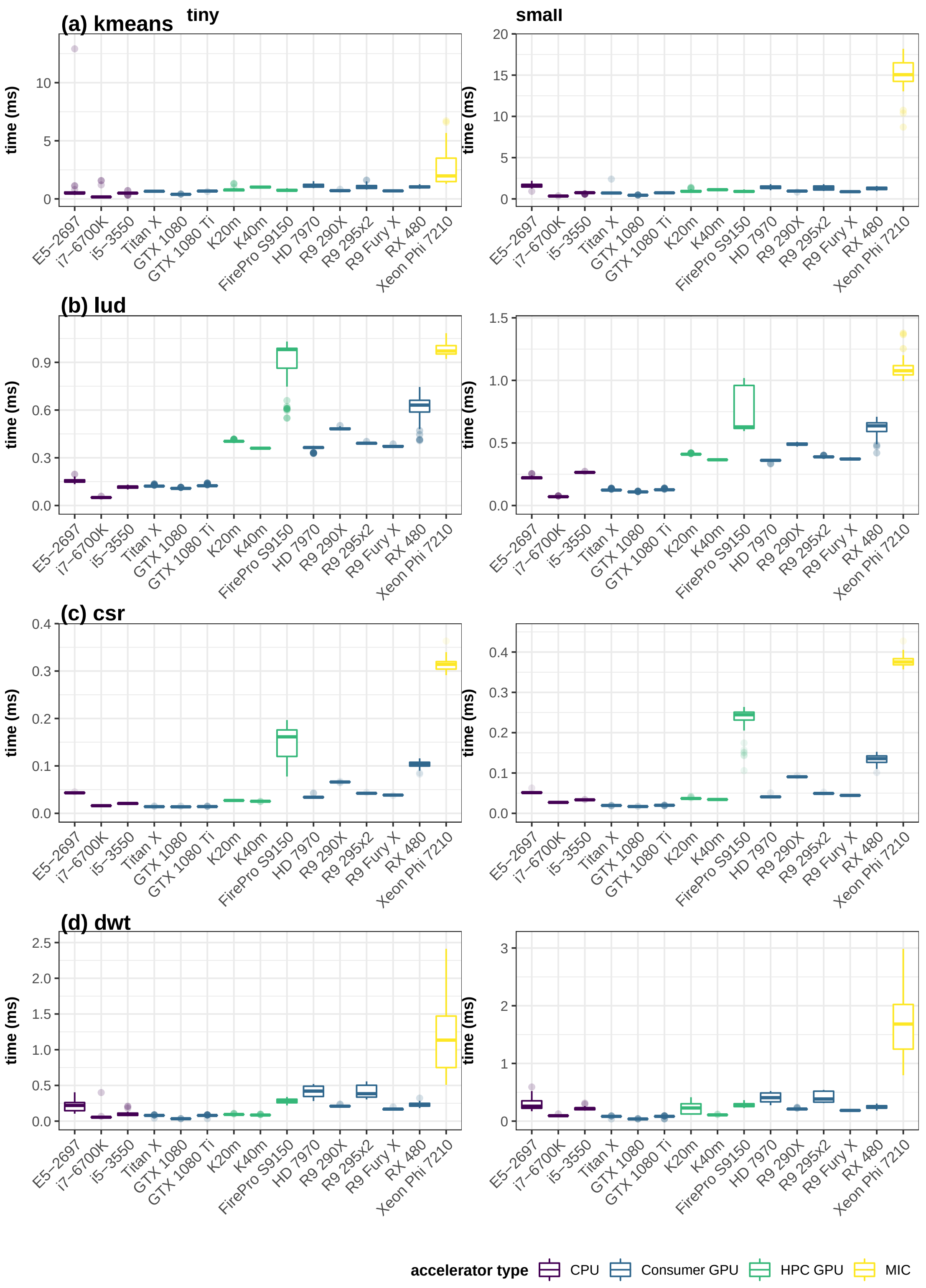


Figure A.1: Kernel execution times for the **tiny** and **small** problem sizes of the `kmeans`, `lud`, `csr` and `dwt` benchmarks on different hardware platforms.

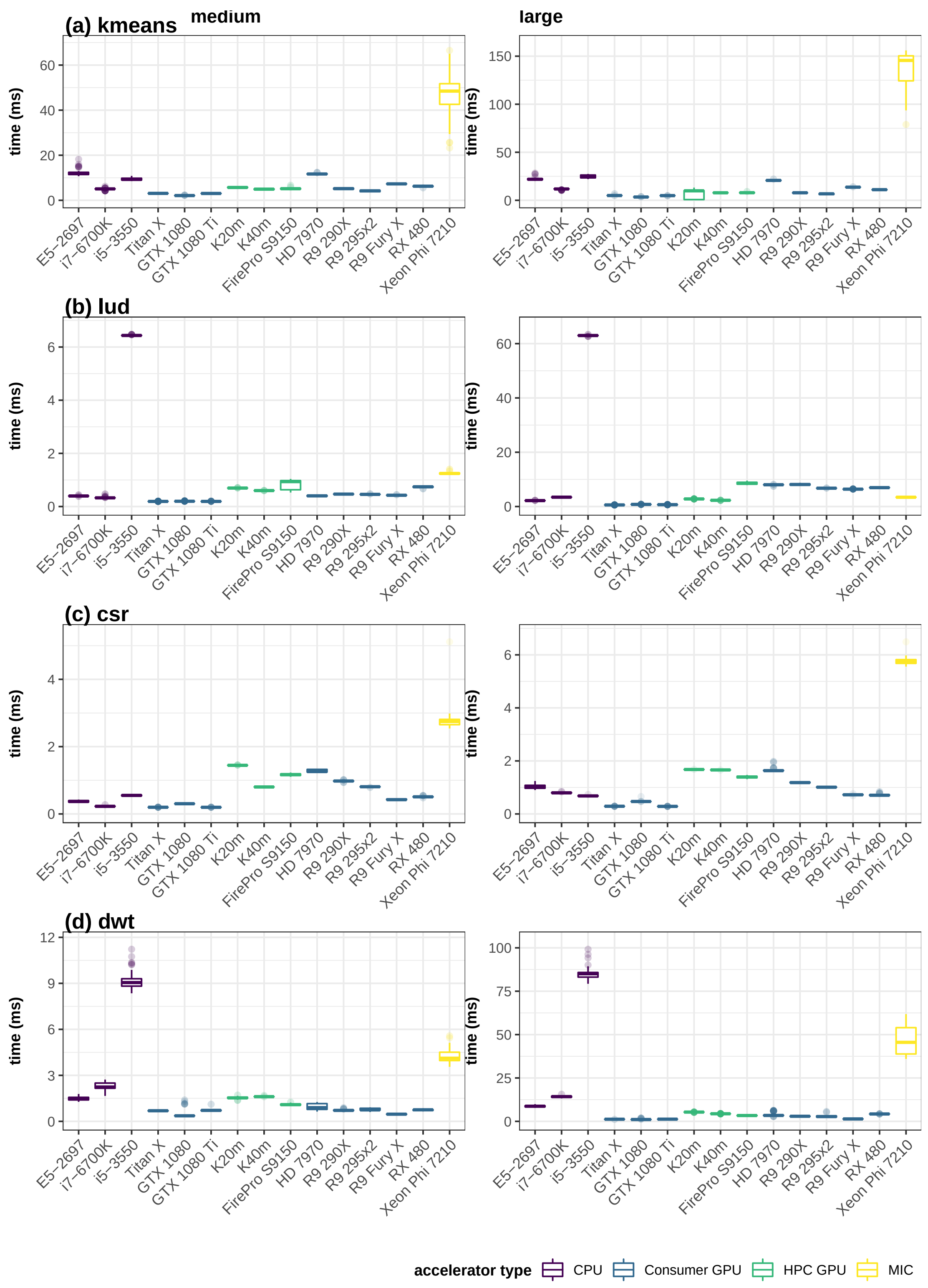


Figure A.2: Kernel execution times for the **medium** and **large** problem sizes of the `kmeans`, `lud`, `csr` and `dwt` benchmarks on different hardware platforms.

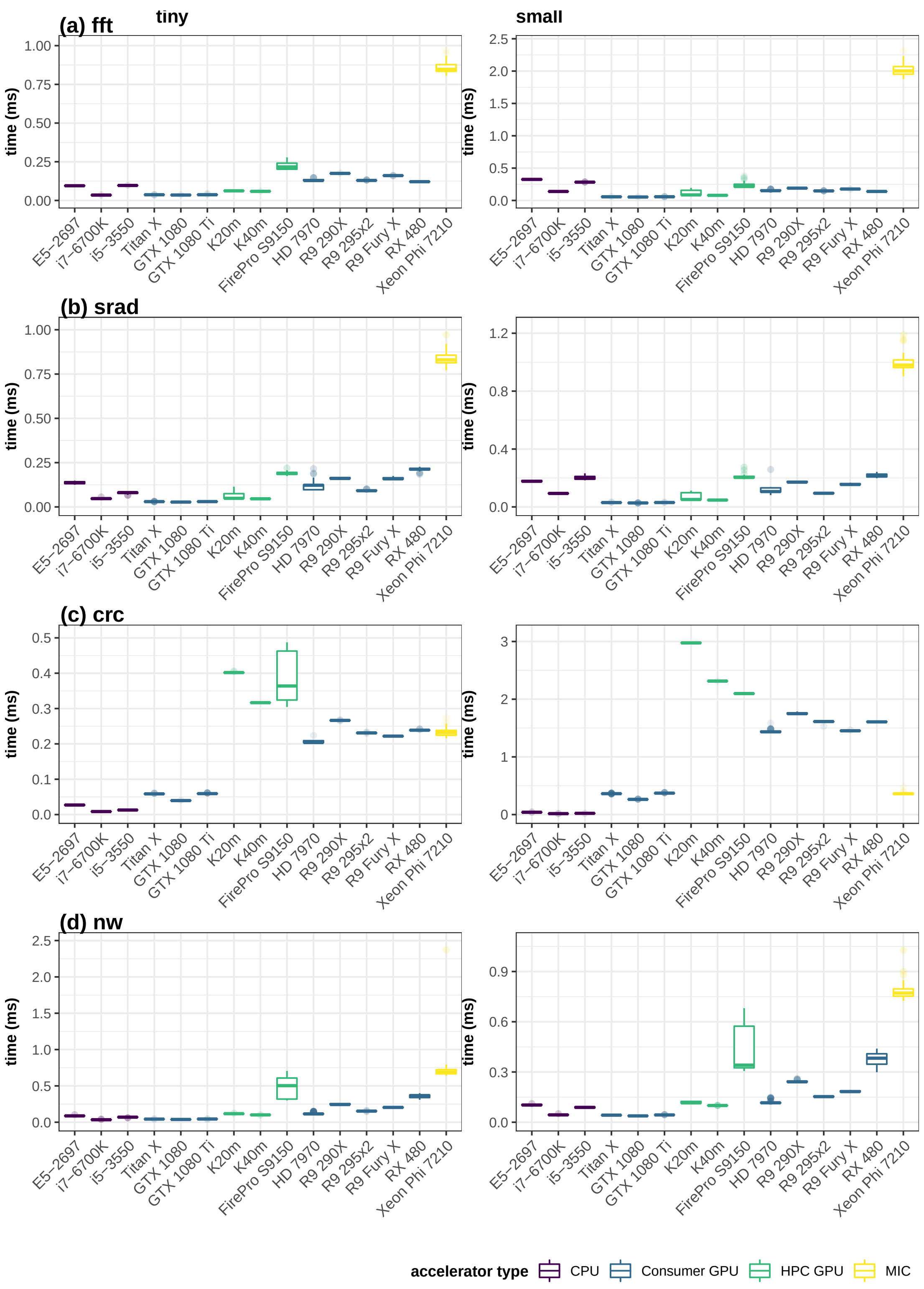


Figure A.3: Kernel execution times for the **tiny** and **small** problem sizes of the `fft`, `srad` and `nw` benchmarks on different hardware platforms.

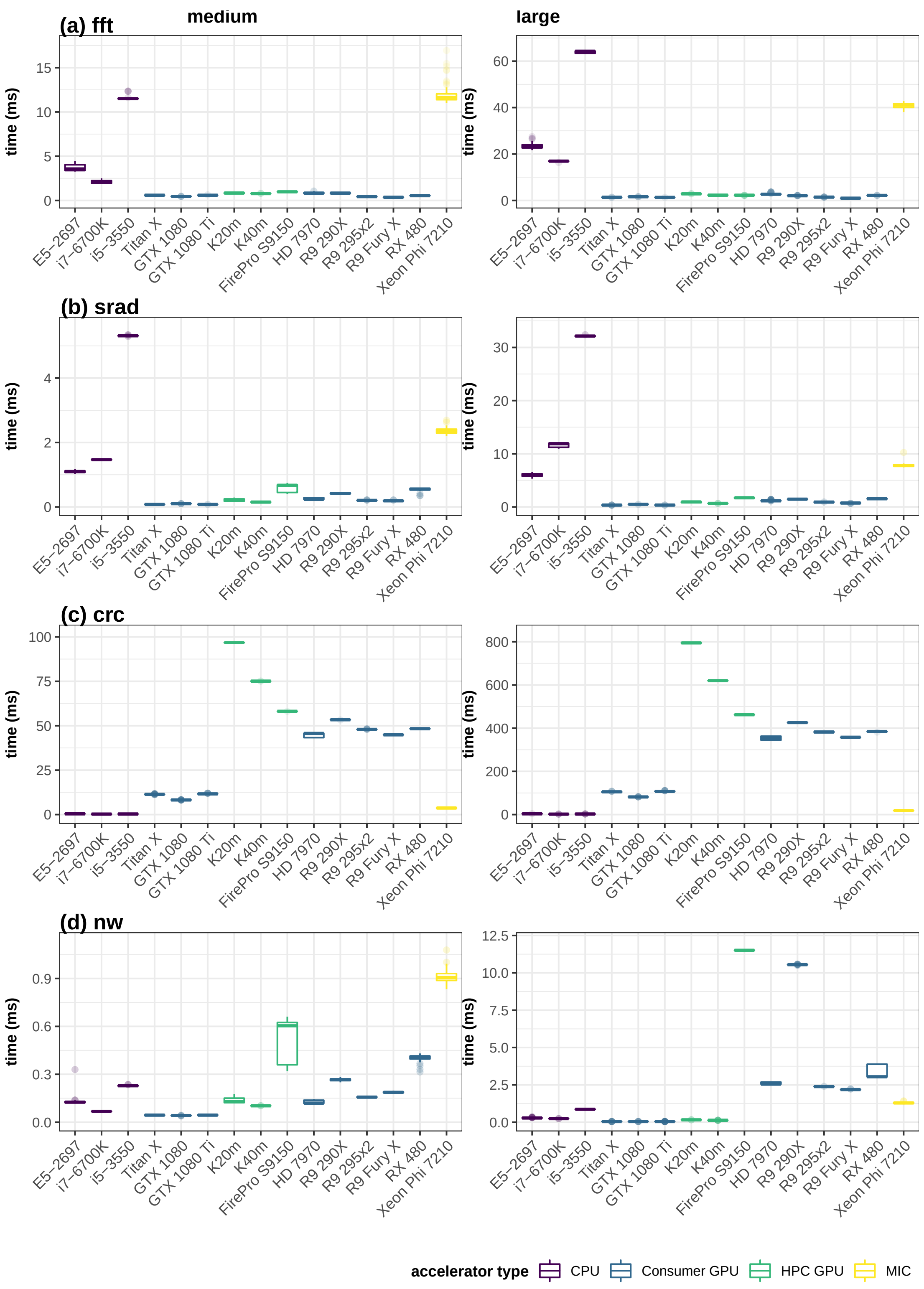


Figure A.4: Kernel execution times for the **medium** and **large** problem sizes of the `fft`, `srad` and `nw` benchmarks on different hardware platforms.

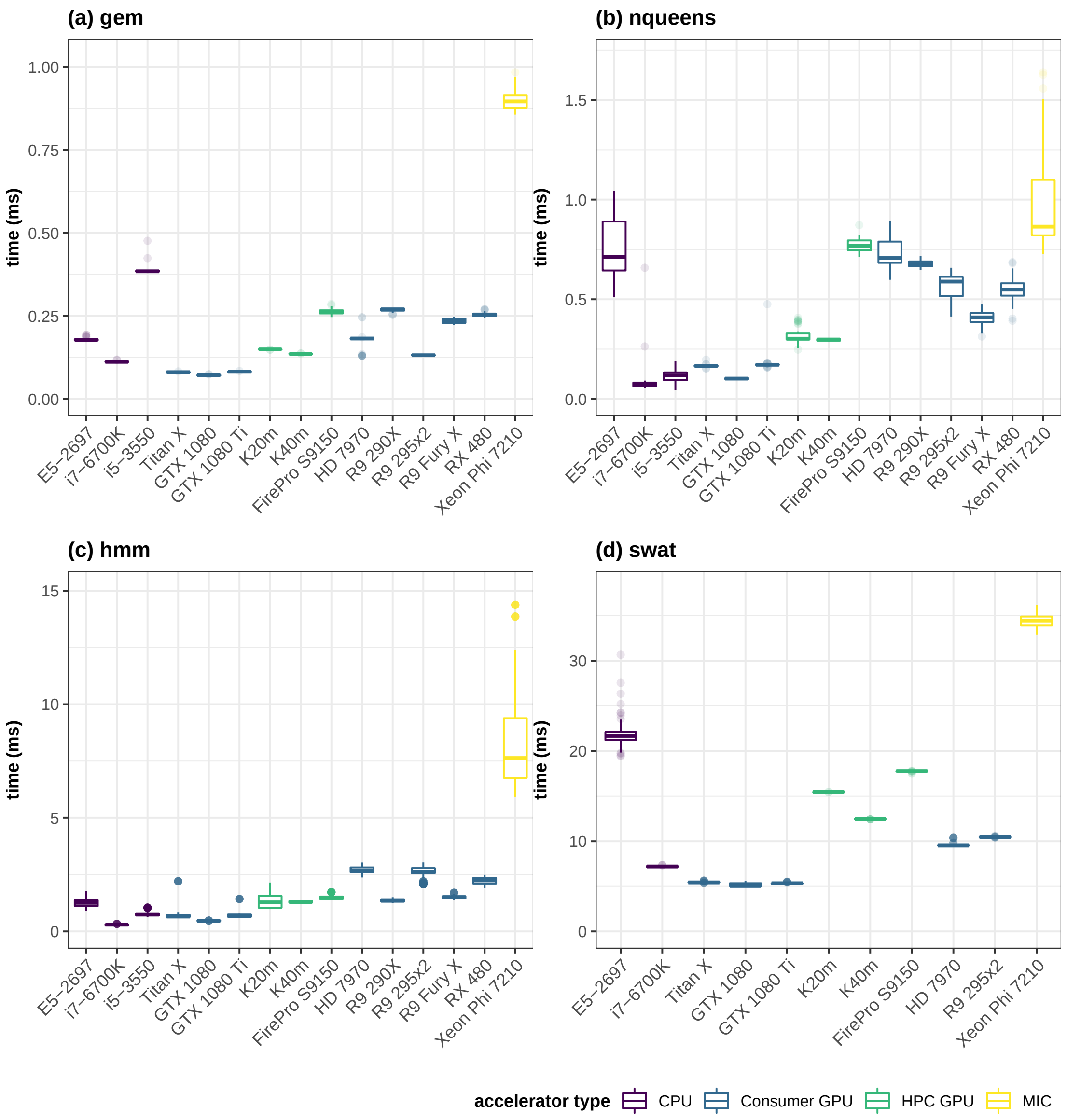


Figure A.5: Single problem sized benchmarks of kernel execution times on different hardware platforms.

Appendix B

Linear Model Fitting

A linear model was evaluated during the initial attempt of the device prediction work presented in this thesis. A Generalized Linear Model (GLM) was fitted to the AIWC features and runtime data. For the comparison, the GLM used a gamma family variance and log link function. In this appendix, we show the predictive relative accuracy of this model and emphasize the high magnitude of error, and wide prediction intervals. To reproduce this analysis, see the associated Jupyter artefact[1].

The Gamma family is justified by the presence of skewed strictly positive responses, the log-link was previously found by empirical comparisons of link functions options. Selection of the log-link over the canonical (inverse) link also avoids limitations on the estimation of the nu-parameter and the corresponding Beta model coefficients. Figure B.1 presents the predicted vs measured execution times of the fitted GLM model and is coloured according to problem size, we see poor prediction results – for instance we see a much wider range of ground-truth values in the sample of measured execution times relative to the main linear trend.

Figure B.2 presents the distribution of errors and their associated magnitude. The model has a predictive relative accuracy[2] of 84% and is skewed to underestimate execution times of devices. The mean error of prediction times was ≈ 3 µs but this error could be much greater and up to 450 µs. This level of error was unacceptable and it is suggested that the GLM was unable to capture the complicated statistical relationships between AIWC features and execution time. This motivated the use of non-parametric models — such as the random forest — which we have demonstrated is more suitable for the data.

The prediction intervals for the GLM are presented in Figure B.3. These intervals show the lower and upper error prediction bounds and were achieved by fixing all but one variable – we selected the kmeansPoint kernel on the GTX1080Ti GPU – and vary the problem size. In practice, any other variable could be fixed and varied, but this is only done to show the huge variation in predictions and that the GLM is a poor choice of model, which is shown by the significant errors – larger spread in mispredictions – for larger problem sizes. The

[1]https://github.com/BeauJoh/opencl-predictions-with-aiwc

[2]Relative accuracy (%): $\left(1 - \frac{|\text{predicted} - \text{measured}|}{\text{measured}}\right) \times 100$ summed over all kernels and devices.

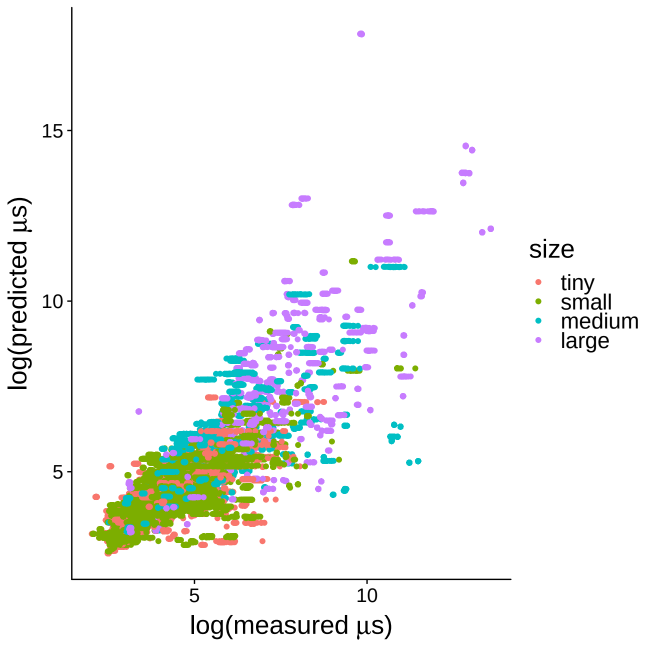


Figure B.1: Predicted vs. measured execution time (in log(µs)) for all kernels and devices with the GLM model.

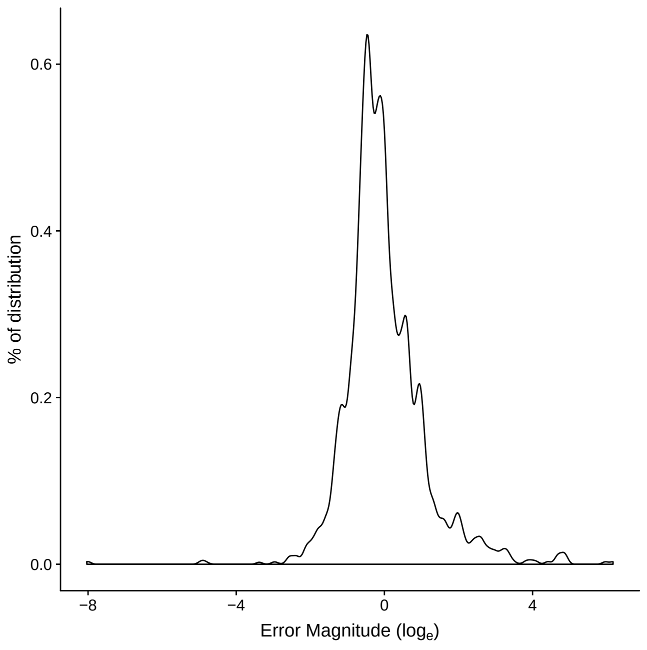


Figure B.2: Distribution of prediction errors with the GLM model, presented in a log scale.

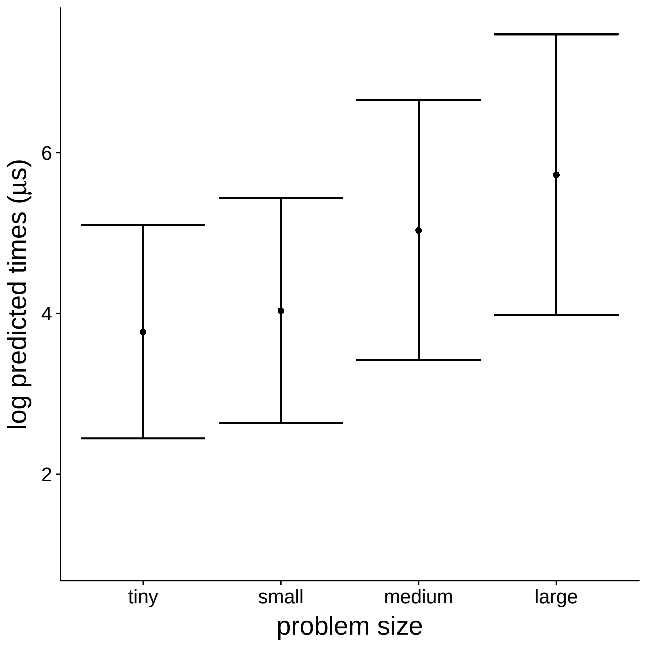


Figure B.3: Prediction intervals for the `kmeansPoint` kernel on the GTX1080Ti GPU over four problem sizes.

intervals shown in Figure B.3 are the predictions and ranges for each problem size, the dots are individual predictions, while the error-bars show the range of where expected future values will fall. We see the predictions have a similarly wide spread for all problem sizes. Given the resolution of the times, where on the tiny problem size, an accurate prediction is 40 µs, having a prediction interval of 30 µs to 148 µs demonstrates the unsuitability of the GLM model; larger problem sizes will experience more severe mispredictions associated with these wide prediction intervals.

The four heat maps presented in Figure B.4 show the difference between mean predicted and measured kernel execution times as a percentage of the measured time when using the GLM based model. 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. The kernels (y-axis) between each of problem size do not align due to the number of supported applications, and kernels, in each problem size – this is discussed in Chapter 3. There are many kernels which show bad prediction performance, ranging from 25-100% error – for instance, the `crc32_slice8` kernel suffers near 100% error when compared to the measured times on most devices. The model predictions differed from the measured experimental results by an average of 14.5%, had a mean predicted relative accuracy of 85.5%, which corresponds to the actual execution time mispredictions of ≈ 109 µs to ≈ 12 s according to problem size.

The unsuitability of the GLM motivates the use of non-parametric models – we have shown the random forest in Chapter 5 to be an effective model for the data.

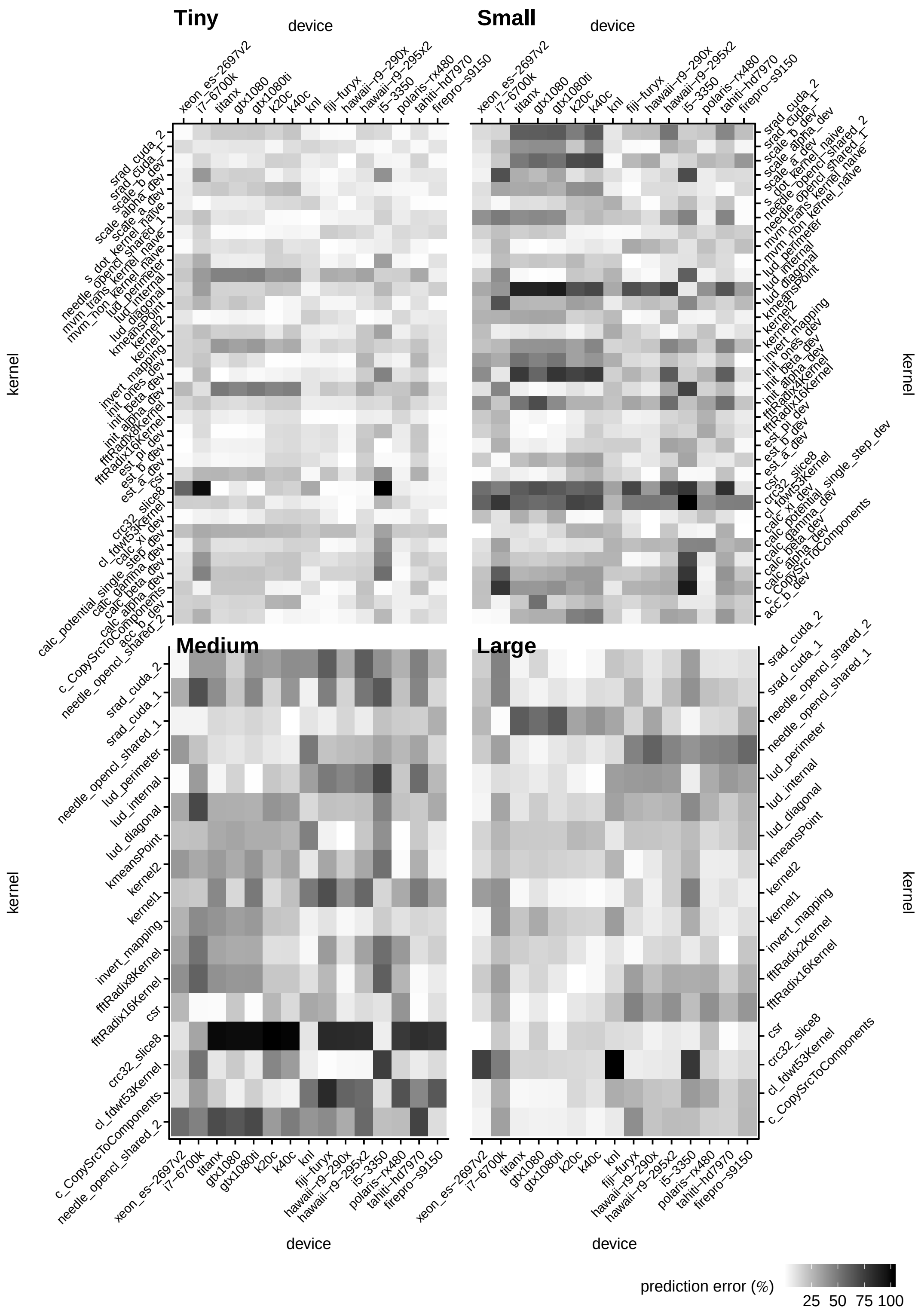


Figure B.4: Error in predicted execution time with a GLM model for each kernel invocation over four problem sizes.

Appendix C

Diversity Analysis

A brief overview of the diversity analysis conducted is presented in this appendix. Features from AIWC are used as the predictor variables in the random forest model – presented in Chapter 5. This model was trained from the combined results of all application kernels and all problem sizes. In this section, the predictor variables are examined independently to evaluate the variances between kernels and problem sizes in the AIWC feature-space.

Evaluation uses dimensionality reduction techniques, from Principal Component Analysis (PCA) and t-Distributed Stochastic Neighbor Embedding (t-SNE). The feature-space reduction methods allow the determination of the loading, or relative contributions, of each component metric. t-SNE is a machine learning visualization algorithm used to find the optimal projection of high dimensional data into two-dimensional point by a way that similar objects are modelled by nearby points and dissimilar objects are modelled by distant points with high probability. On the t-SNE visualization use k-means clustering to present the grouping between features.

From the PCA biplot in Figure C.1, we can determine that Total Memory Footprint, any of the branch entropy metrics and one of the memory address entropy variables are the 3 most principal components to be used when forming a predictive model. The proportion of variance of each principal component is presented in Figure C.2 and shows these 3 principal components can cover 95% of the contributions of difference in a 19-dimensional AIWC feature-space, 5 principal components represent ~98% variance in the data and 6 variables cover more than 99%. Similarly, the t-SNE clustering, from Figure C.3, tell a similar story, namely, 5-6 features convey a majority of the information. There is clearly a cluster structure in the manifold – which is good news for prediction – but there are also 2 interesting linear strings structures (correlations) – which suggests that one method of regression or prediction will not suffice. The visualization and the same methodology can be used to justify the inclusion of a new benchmark, for instance, if an application kernel extends the coverage in the projection.

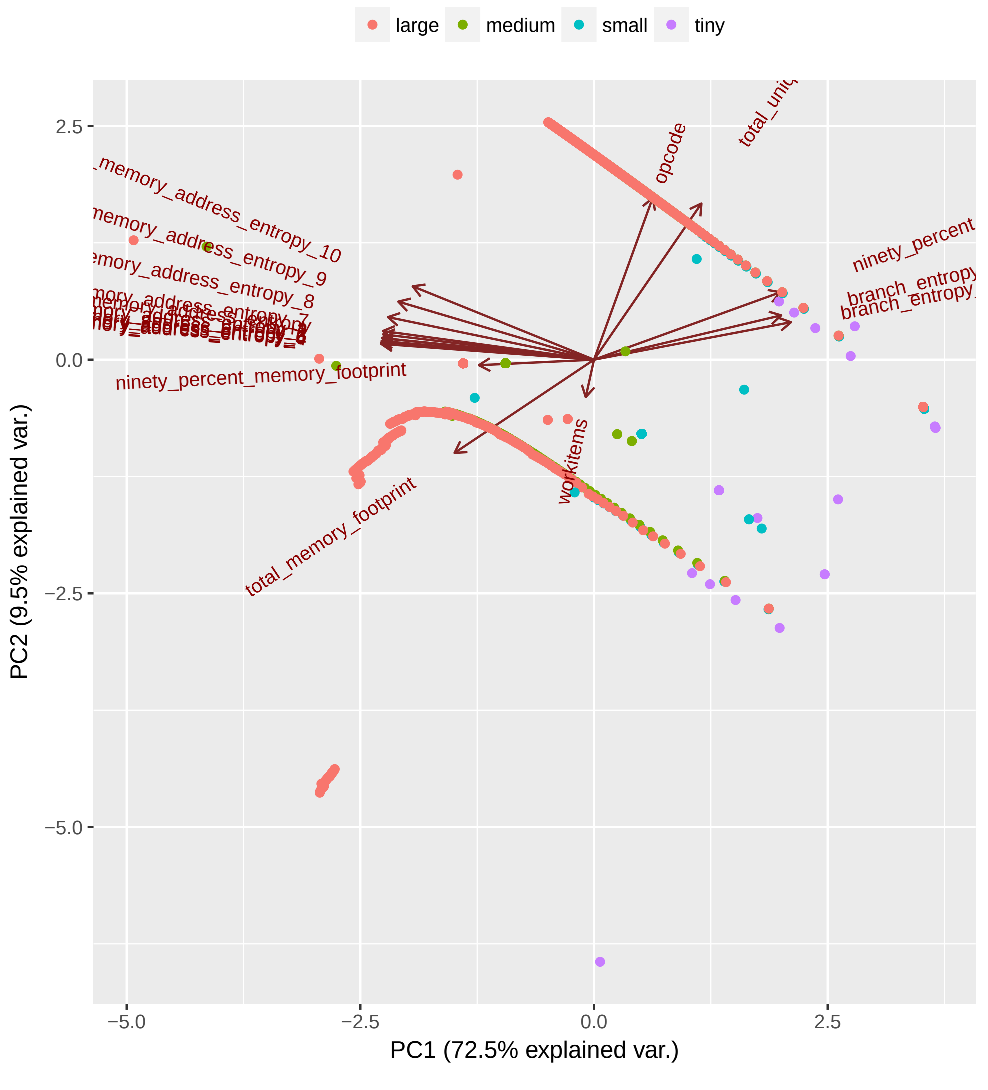


Figure C.1: Biplot Principal Components of AIWC metrics over all application kernels and all problem sizes.

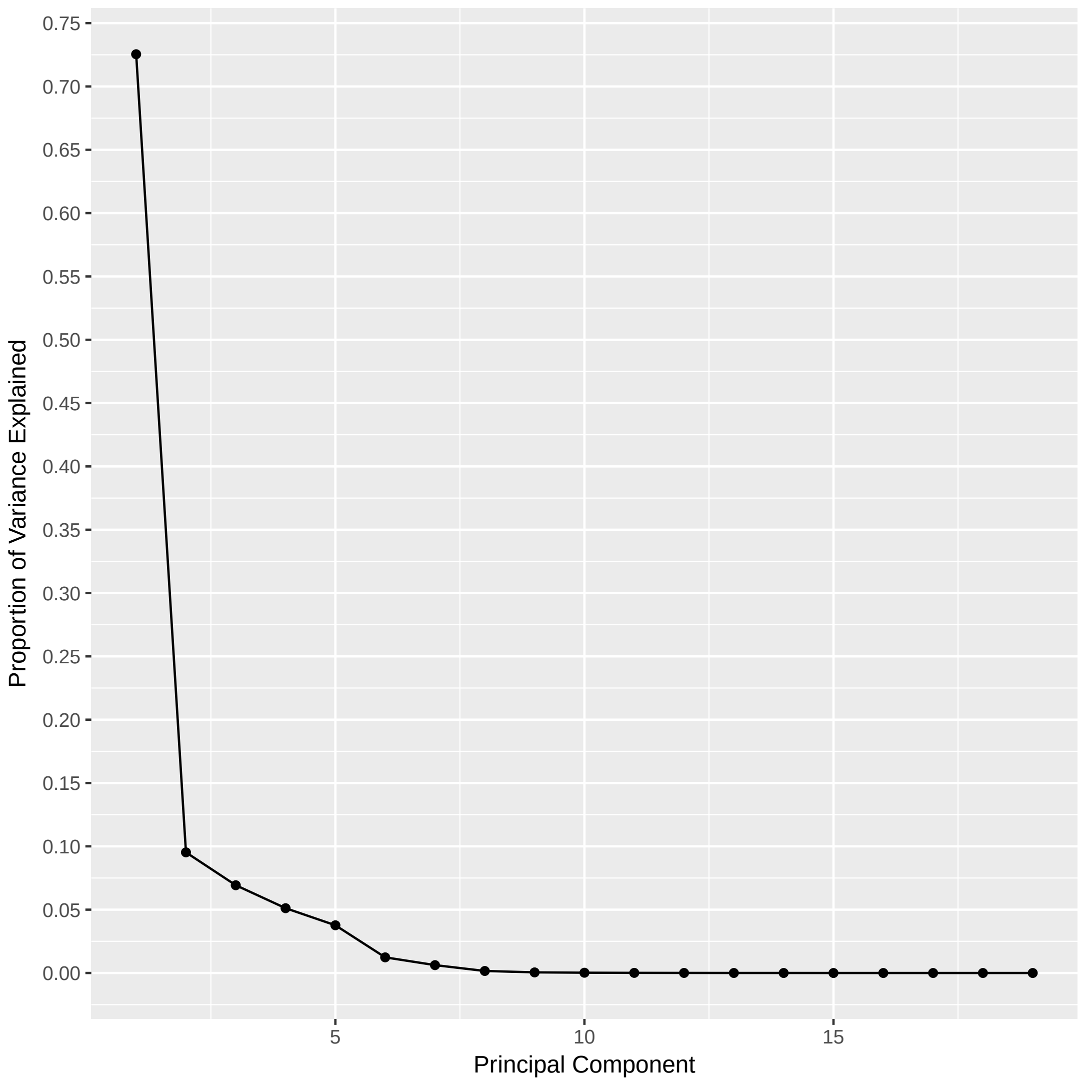


Figure C.2: The proportion of explained variance of each Principal Component.

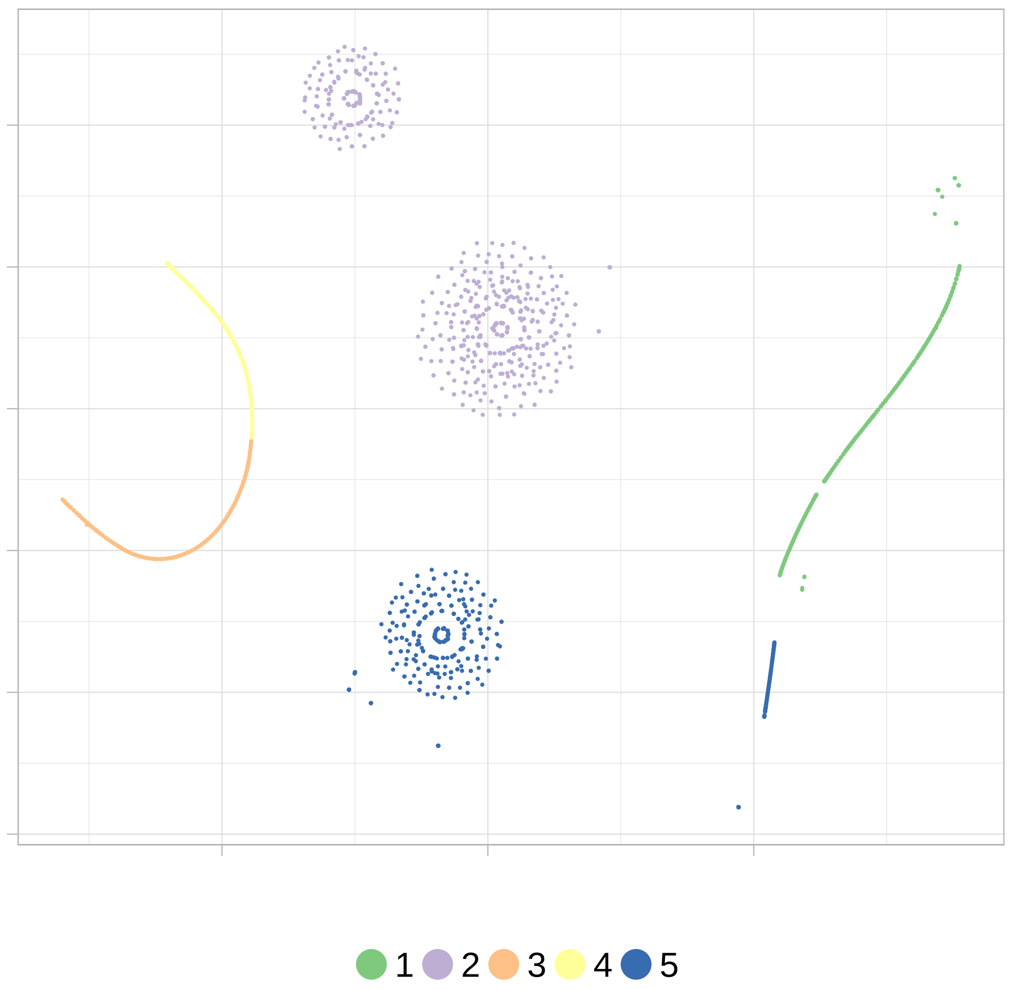


Figure C.3: The t-SNE with k-means cluster results to show Principal Components.

Appendix D

Abbreviations

AIWC **A**rchitecture **I**ndependent **W**orkload **C**haracterization
ASIC **A**pplication-**S**pecific **I**ntegrated **C**ircuit
AVX **A**dvanced **V**ector E**x**tensions
CLgen Open**CL** kernel **gen**eration framework
CPE **C**omputer **P**rocessing **E**lements
CPU **C**entral **P**rocessing **U**nit
DSP **D**igital **S**ignal **P**rocessor
EOD **E**xtended **O**pen **D**warfs Benchmark Suite
FPGA **F**ield-**P**rogrammable **G**ate **A**rray
GPU **G**raphics **P**rocessing **U**nit
GLM **G**eneralized **L**inear **M**odel
HASS **H**eterogeneity-**A**ware **S**ignature-**S**upported scheduler
HCC **H**eterogeneous **C**ompute **C**ompiler
HPC **H**igh **P**erformance **C**omputing
ILP **I**nstruction **L**evel **P**arallelism
IPC **I**nstructions **P**er **C**ycle
ISA **I**nstruction **S**et **A**rchitecture
ITB **I**nstructions **T**o **B**arrier
KNL **K**nights **L**anding
LMAE **L**ocal **M**emory **A**ddress Access **E**ntropy
LSB **L**east **S**ignificant **B**its
MIC **M**any **I**ntegrated **C**ore
MICA **M**icroarchitecture-**I**ndependent Workload **C**haracteristics
MPE **M**anagement **P**rocessing **E**lements
NVML **Nv**idia **M**anagement **L**ibrary
OpenCL **O**pen **C**omputing **L**anguage
OpenDwarfs **Open**CL and the 13 **Dwarfs**
PAPI **P**erformance **A**pplication **P**rogramming **I**nterface
PARSEC **P**rinceton **A**pplication **R**epository for **S**hared-Memory **C**omputers
PCA **P**rincipal **C**omponent **A**nalysis

PDB **P**rotein **D**ata **B**ank format
RAM **R**andom **A**ccess **M**emory
RAPL **R**unning **A**verage **P**ower **L**imit
RISC **R**educed **I**nstruction-**S**et **C**omputers
SC **S**uper **C**omputing
SHOC **S**calable **H**eter**o**geneous **C**omputing benchmark suite
SIMD **S**ingle **Instruction** **M**ultiple **D**ata
SIMT **S**ingle **I**nstruction **M**ultiple **T**hread
SMaC **S**calable **Ma**ny **C**ore
SoC **S**ystem-**o**n-a-**C**hip
SPIR **S**tandard **P**ortable **I**ntermediate **R**epresentation
TLB **T**ranslation **L**ook-aside **B**uffer
TLP **T**hread **L**evel **P**arallelism
TPU **T**ensor **P**rocessing **U**nits
t-SNE **t**-Distributed **S**tochastic **N**eighbor **E**mbedding
VPU **V**ector **P**rocessing **U**nit

References

1. TOP500, "TOP500 list - June 2018," *TOP500.org*. https://www.top500.org/list/2018/06/; TOP500.org, June-2018.

2. J. Stuecheli *et al.*, "IBM POWER9 opens up a new era of acceleration enablement: OpenCAPI," *IBM Journal of Research and Development*, vol. 62, no. 4/5, pp. 8–1, 2018.

3. D. Foley and J. Danskin, "Ultra-performance Pascal GPU and NVLink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.

4. S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke, "IBM Power9 processor architecture," *IEEE Micro*, vol. 37, no. 2, pp. 40–51, 2017.

5. C. Nvidia, "Compute unified device architecture programming guide," 2007.

6. M. Breternitz, "Machine learning at AMD foundations and support."

7. L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

8. J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

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

10. K. Rupp, "The OpenCL library ecosystem: Current status and future perspectives," in *Proceedings of the 4th international workshop on OpenCL*, 2016, p. 13.

11. 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.

12. 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.

13. P. Colella, "Defining software requirements for scientific computing, 2004," *DARPA HPCS presentation*.

14. 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.

15. 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.

16. V. W. Lee *et al.*, "Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 451–460, Jun. 2010.

17. J. Dongarra, "Report on the Sunway TaihuLight system," *PDF). www. netlib. org. Retrieved June*, vol. 20, 2016.

18. 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.

19. 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.

20. 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.

21. 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.

22. 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.

23. 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.

24. 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.

25. R. Nicolescu, "Structured grid algorithms modelled with complex objects," in *International conference on membrane computing*, 2015, pp. 321–337.

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

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

28. 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.

29. 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.

30. 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.

31. A. Sodani *et al.*, “Knights landing: Second-generation Intel Xeon Phi product,” *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

32. 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.

33. M. Rajan, D. Doerfler, and S. Hammond, “Trinity benchmarks on Intel Xeon Phi (Knights Corner).”

34. 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.

35. 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.

36. 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.

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

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

39. 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.

40. B. Reagen, Y. S. Shao, G.-Y. Wei, and D. Brooks, “Quantifying acceleration: Power/performance trade-offs of application kernels in hardware,” in *Low power electronics and design (ISLPED), 2013 ieee international symposium on*, 2013, pp. 395–400.

41. 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.

42. 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.

43. 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.

44. M. Feldman, “Cray to deliver ARM-powered supercomputer to UK consortium,” *TOP500 Supercomputer Sites*, Jan. 2017.

45. 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.

46. S. McIntosh-Smith, J. Price, T. Deakin, and A. Poenaru, “Comparative benchmarking of the first generation of HPC-optimised ARM processors on Isambard.”

47. 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.

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

49. 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.

50. 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.

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

52. T. Morgan, “NVLink takes GPU acceleration to the next level,” *The Next Platform*, May 2016.

53. 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.

54. 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.

55. 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.

56. 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.

57. 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.

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

59. 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.

60. 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.

61. 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.

62. 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.

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

64. 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.

65. Y. Sun *et al.*, "Hetero-Mark, a benchmark suite for CPU-GPU collaborative computing," in *IEEE international symposium on workload characterization (iiswc)*, 2016.

66. 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.

67. 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.

68. B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *Workload characterization (IISWC), 2014 IEEE international symposium on*, 2014, pp. 110–119.

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

70. 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.

71. 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.

72. 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.

73. 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.

74. 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.

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

76. 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.

77. 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.

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

79. 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.

80. 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.

81. 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.

82. 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.

83. 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.

84. 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.

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

86. 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.

87. 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.

88. 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.

89. J. Filipovič, F. Petrovič, and S. Benkner, "Autotuning of OpenCL kernels with global optimizations," in *Proceedings of the 1st workshop on autotuning and aDaptivity approaches for energy efficient HPC systems*, 2017, pp. 2:1–2:6.

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

91. S. McIntosh-Smith, J. Price, R. B. Sessions, and A. A. Ibarra, "High performance in silico virtual drug screening on many-core processors," *The international journal of high performance computing applications (IJHPCA)*, vol. 29, no. 2, pp. 119–134, 2015.

92. L. Eeckhout, J. Sampson, and B. Calder, "Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation," in *Workload characterization symposium, 2005. proceedings of the IEEE international*, 2005, pp. 2–12.

93. 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.

94. 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.

95. 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.

96. 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.

97. 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.

98. J. Kessenich, "A Khronos-defined intermediate language for native representation of graphical shaders and compute kernels." 2015.

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

100. 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.

101. 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.

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

103. 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.

104. 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.

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

106. 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.

107. 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.

108. 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.

109. 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.

110. 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.

111. 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.

112. 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.

113. 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.

114. J. Fowers, G. Brown, J. Wernsing, and G. Stitt, "A performance and energy comparison of convolution on GPUs, FPGAs, and multicore processors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, p. 25, 2013.

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

116. 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.

117. 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.

118. 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.

119. 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.

120. 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.

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

122. 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.

123. 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.

124. 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.

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

126. 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.

127. 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.

128. 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.

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

130. 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.

131. 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.

132. 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.

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

134. 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.

135. H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: Memory power estimation and capping," in *Proceedings of the 16th ACM/IEEE international symposium on low power electronics and design*, 2010, pp. 189–194.

136. K. Kasichayanula, D. Terpstra, P. Luszczek, S. Tomov, S. Moore, and G. D. Peterson,

"Power aware computing on GPUs," in *Application accelerators in high performance computing (SAAHPC), 2012 symposium on*, 2012, pp. 64–73.

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

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

139. 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.

140. 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.

141. 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.

142. 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.

143. 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.

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

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

146. 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.

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

148. C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

149. A. Duran *et al.*, "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

150. T. R. Scogland, W.-c. Feng, B. Rountree, and B. R. de Supinski, “Coretsar: Adaptive worksharing for heterogeneous systems,” in *International supercomputing conference*, 2014, pp. 172–186.

151. 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.

152. L. Breiman, “Random forests,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

153. 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.

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

155. S. Kumar, V. Srinivasan, A. Sharifian, N. Sumner, and A. Shriraman, “Peruse and profit: Estimating the accelerability of loops,” in *Proceedings of the 2016 international conference on supercomputing*, 2016, pp. 21:1–21:13.

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

157. H. Perkins, “CUDA-on-CL: A compiler and runtime for running NVIDIA CUDA C++11 applications on OpenCL 1.2 devices,” in *Proceedings of the 5th international workshop on OpenCL*, 2017, pp. 6:1–6:4.