UNIVERSITY OF CALIFORNIA,
IRVINE

Performance, Energy and Temperature Considerations for Job Scheduling and for Workload Distribution in Heterogeneous Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical and Computer Engineering

by

Shouq Alsubaihi

Dissertation Committee:
Professor Jean-Luc Gaudiot, Chair
Professor Alexander Veidenbaum
Professor Nader Bagherzadeh

2017

DEDICATION

To my beloved parents, husband and family

For your endless love and support, without which I could never make it this far

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

The completion of this journey is due to the support, assistance, and guidance of many people. It is my greatest pleasure to acknowledge those individuals who've played a significant role in my academic accomplishments.

First and foremost, I am indebted to my loving husband, Nawaf Alsabah, without his endless love, continuous support and patience this work would not have been possible. Thank you for everything with all my heart and soul. There are no words that can express my gratitude and appreciation for all you've done and been for me. I could not have completed this journey without you by my side. I'm very lucky to have you. I love you!

I am extremely grateful to my parents, Ahmed Alsubaihi and Laila Alansari for their love, prayers, caring and sacrifices for educating and preparing me for my future. I'm also thankful to my brothers, Mashari and Mohammed and my sister Jana for their love and support. I would like to thank my daughter Azyan for bringing joy and happiness to my life.

I would like to express my deep and sincere gratitude to my advisor, Professor Jean-Luc Gaudiot, who provided me with encouragement, guidance, and support during my study. It has been an honor to work and study under his guidance. I appreciate how much time and effort he spent to make my Ph.D. experience productive and exciting. I would also like to thank my committee members, Professor Alexander Veidenbaum, and Professor Nader Bagherzadeh for their kind support, encouragement and trust.

In addition, I extend my thanks to all my colleagues in the PArallel Systems & Computer Architecture Lab (PASCAL). I'm also appreciative of my friends near and far, especially Lulwah Alhubail, and Munirah Almatooq for their support throughout this research journey.

Finally, I'm grateful for all the people who have helped me to fulfill my dream directly or indirectly.

CURRICULUM VITAE

Shouq Alsubaihi

EDUCATION

2017 University of California, Irvine
Ph.D. in Electrical and Computer Engineering

2008 Kuwait University
M.Sc. in Computer Engineering

2005 Kuwait University
B.A. in Computer Engineering

EXPERIENCE

2010 – 2011	Developer and Administrator of Computer Programming Grading System
2007 – 2008	Secretary of Women In Engineering committee IEEE, Kuwait Chapter
2006 – 2011	Teaching Assistant, Computer Engineering Department, Kuwait University
2004 – 2005	Member of student consultant committee, Student Affairs, Kuwait University
2003 – 2005	Member of student consultant committee, Collage of Engineering and Petroleum, Kuwait University

HONORS AND AWARDS

2011 Kuwait University scholarship to pursue Ph.D. in Computer Engineering

2008 M.Sc. in Computer Engineering with the highest GPA

2008 Graduated and been honored by the president of the State of Kuwait, M.Sc. Computer Engineering, Kuwait University

2007 The Ideal Engineer Award, Computer Engineering Department, Kuwait University

2005 Graduated and been honored by the president of the State of Kuwait, B.A. in Computer Engineering, Kuwait University

2001 – 2005 Listed in the list of students with honor.

2001 – 2005 Listed in the honorary deans list.

CERTIFICATES

2017 Public Speaking: Activate to Captivate, Graduate Resource Center, University of California, Irvine

2017 Course Design Certificate, Center for Engaged Instruction (CEI), University of California, Irvine

2016 Excellence in Engineering Communications Certificate, Graduate Resource Center, University of California, Irvine

2016 Mentoring Excellence Program, Graduate Resource Center, University of California, Irvine

PUBLICATIONS

S. Alsubaihi and J. L. Gaudiot, "A Runtime Workload Distribution with Resource Allocation for CPU-GPU Heterogeneous Systems," 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Orlando, FL, 2017.

S. Alsubaihi and J. L. Gaudiot, "PETRAS: Performance, Energy and Thermal Aware Resource Allocation and Scheduling for Heterogeneous Systems," In Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'17), ACM, Austin, TX, 2017, pp. 29-38.

S. Alsubaihi and J. L. Gaudiot, "PETS: Performance, Energy and Thermal Aware Scheduler for Job Mapping with Resource Allocation in Heterogeneous Systems," in IEEE 35th International Performance Computing and Communications Conference (IPCCC), Las Vegas, NV, 2016.

S. Al-Subaihi, L. Al-Hubail, P.N. Marimuthu , and S.J. Habib,"Synthesizing Clustered, Secured, and Hierarchical Networks through Genetic Algorithms", International Conference on Intelligent System Modeling and Simulation ISMS Conference, 2010, pp. 385 – 390.

S. Almukhaizim, S. Alsubaihi, and O. Sinanoglu,"On the Application of Dynamic Scan Chain Partitioning for Reducing Peak Shift Power", Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 26, Issue 4, August 2010, pp. 465 – 481.

ABSTRACT OF THE DISSERTATION

Performance, Energy and Temperature Considerations for Job Scheduling and for Workload Distribution in Heterogeneous Systems

By

Shouq Alsubaihi

Doctor of Philosophy in Electrical and Computer Engineering
University of California, Irvine, 2017
Professor Jean-Luc Gaudiot, Chair

Many systems today are heterogeneous in that they consist of a mix of different types of processing units (e.g., CPUs, GPUs). Each of these processing units has different performance and energy consumption characteristics. Job scheduling and workload distribution play a crucial role in such systems as they strongly affect system's performance, energy consumption, peak power and peak temperature. The scheduler maps the entire jobs to processing units, whereas workload distributor maps parts of the job. Allocating resources (e.g., core scaling, thread allocation) is another challenge since different sets of resources exhibit different behavior in terms of performance and energy.

Performance was the dominant factor in job scheduling and workload distribution for years. As processor's design has hit the power-wall, energy consumption also becomes important. Many studies have been conducted on scheduling and workload distribution with an eye on performance improvement. However, few of them consider both performance and energy.

We propose a Performance, Energy and Thermal aware Resource Allocator and Scheduler (PETRAS), which includes core scaling and thread allocation. Since job scheduling is known to be an NP-hard problem, we apply a Genetic Algorithm (GA) to find an efficient job schedule in terms of performance and energy consumption, under peak power and peak CPU temperature constraints. Compared to other schedulers, PETRAS achieves up to 4.7x speedup and energy saving of up to 195%.

The classic workload distribution does not fully utilize the CPUs and the GPUs. It maps the sequential parts of a job to the CPU and the parallel parts to the GPU. We thus propose a Workload Distributor with a Resource Allocator (WDRA), which combines core scaling and thread allocation into a workload distributor. Since workload distribution is known to be an NP-hard problem, WDRA utilizes Particle Swarm Optimization (PSO) to find an efficient workload distribution in terms of performance and energy consumption, under peak power and peak CPU temperature constraints. Compared to other workload distributors, WDRA can achieve up to 1.47x speedup and 82% reduction of energy consumption. WDRA is a well-suited runtime distributor since it only takes up to 1.7% of the job's execution time.

Chapter 1

Introduction

1.1 Background

Attempts at improving uni-core processors have hit a power wall. As a result, the industry has shifted towards multicore processors. Another shift in the industry is that Graphical Processing Units (GPUs) are now being used for more than just image processing. GPUs are now used as general purpose processing units to execute highly parallel jobs [2][37]. GPUs show their capability to run compute intensive jobs efficiently in terms of execution time and energy consumption. Processing units such as multicore processors and general purpose GPUs have emerged, forming heterogeneous systems. Integrating processing units with different performance/energy characteristics on the same machine could enhance a system's performance and energy efficiency [1].

Performance of the system (i.e., execution time) was the dominant metric for evaluating processer design for years. Minimizing the execution time was a major concern for designing software/hardware. As the uni-core processors hit the power wall, power/energy becomes an important metric. Minimizing energy consumption may degrade a system's performance. Hence, the energy and performance trade-off should be considered while designing a system. In the embedded systems field, peak power and peak temperature are considered as additional constraints.

A heterogeneous system can be a computer that consists of a mixture of different types of processing units (e.g., CPUs, GPUs, FPGAs). Each of these processing units has different architectural strengths in execution capabilities and energy consumption. Computational resources in general can be either processing units, number of cores, or number of threads. In a heterogeneous system, selecting the best computational resources to run a job requires an understanding of: 1) the capabilities of the processing units with different number of cores and threads, 2) optimized parameters (i.e., performance and energy consumption), and 3) computational constraints of the problem such as peak power and temperature limits. This is because different computational resources have different strengths and weaknesses with respect to these parameters and constraints.

A classic job scheduler role is to map an entire job to a processing unit. A classic workload distributor role is to map parts of a job to more than one processing unit that cooperate to execute that job. Typically, a GPU handles the parallel parts of a job and a CPU (i.e., host) handles the sequential parts of a job. The CPU also handles CPU/GPU data transfer coordination. The host remains idle waiting for a GPU to finish its part and transfer data back. This classic workload distribution does not fully utilize the CPU and the GPU.

It is true that job scheduling and workload distribution play a crucial role in CPU-GPU heterogeneous systems as they strongly affect the overall system performance, energy consumption, peak power and peak temperature. But selecting the optimal resources (e.g., number of cores, number of threads) to run a job is also important due to their effect on the parameters above. Hence, a resource allocator should be combined with schedulers and workload distributors.

1.2 Research Goals and Our Contributions

The overarching goal of this dissertation is to understand the effectiveness of combining resource allocators with schedulers and workload distributors in improving the performance and reducing the energy consumption of heterogeneous systems. Moreover, the goal is to show the importance of considering both systems' performance and energy consumption while job scheduling or workload distribution. In addition, peak power and peak CPU temperature limits should be taken into account especially if the targeted hardware is an embedded system.

The research goal can be achieved in the following major steps: first, we propose the Performance, Energy and Thermal aware Resource Allocator and Scheduler (PETRAS) [4][19], which combines job mapping, core scaling, and thread allocation into one scheduler. Second, we propose the Workload Distributor with a Resource Allocator (WDRA) [24], which finds a job's efficient workload distribution and resource allocation in terms of both a system's performance and energy consumption. Third, we implement PETRAS and WDRA on CPU-GPU heterogeneous systems. Finally, we evaluate the effectiveness of PETRAS and WDRA on an actual heterogeneous system that is equipped with a multi-core CPU and a GPU. PETRAS and WDRA are evaluated considering the overall performance and energy consumption in both cases: non-constraints, and peak power and peak CPU temperature constraints.

In detail, the contributions of this dissertation are as follows:

- Develop a job scheduler and a workload distributor that has a resource allocator (i.e., core scaling and thread allocation).
- Develop a job scheduler and a workload distributor that considers both a system's performance and energy consumption.
- Introduce scheduling and workload distribution problems as multi-objective problems.
- Identify the problem of only considering one parameter (i.e., performance or energy consumption) while scheduling/workload distribution.
- Explore how running a job on different processing units with different resources exhibits different behaviors in terms of performance, energy consumption, peak power and peak CPU temperature.
- Explore how a job's CPU/GPU map ratio with different resources exhibits different behaviors in terms of performance, energy consumption, peak power and peak CPU temperature.
- Explore the importance of solving scheduling/workload distribution as a multi-objective optimization problem.
- Evaluate PETRAS and WDRA in two cases: non-constraints and constraints (i.e., peak power, peak CPU temperature).
- Evaluate how PETRAS and WDRA can improve a system's performance and reduce energy consumption compared to other classic schedulers and distributors.
- Evaluate the overhead of WDRA to be applied during runtime.
- Propose and evaluate the initial design of PETRAS and WDRA.

1.3 Dissertation Organization

This dissertation is organized as follows:

- Chapter 2 presents an overview of CPU-GPU heterogeneous systems. It also provides a comparison between the CPU and the GPU architectures. It describes the job scheduler and workload distributor for heterogeneous systems. It presents the platform, programming languages and the benchmark that we used in this work.
- Chapter 3 proposes the Performance, Energy and Thermal aware Resource Allocator and Scheduler (PETRAS). It presents the related work on scheduling. It also describes the PETRAS optimization problem in detail. The chapter presents PETRAS methodology and implementation. It then evaluates PETRAS and discusses experimental results.
- Chapter 4 proposes the Workload Distributor and Resource Allocator (WDRA). It discusses the work that has been done on workload distribution. It then gives details on the WDRA optimization problem. The chapter describes WDRA methodology and implementation. It also evaluates WDRA on an actual CPU-GPU heterogeneous system.
- Chapter 5 concludes this work and discusses future work.

Chapter 2

CPU-GPU Heterogeneous Systems

Many computing systems today are heterogeneous in that they consist of a mix of different types of processing units (e.g., CPUs, GPUs). Each of these processing units has different execution capabilities and energy consumption characteristics. These processing units can be fused on the same chip or connected through a network (e.g., PCI express, a ring, or a mesh).

As the processor's design has hit the power wall, the industry has shifted towards multicore processors. Another shift in the industry is that Graphical Processing Units (GPUs) are now being used for general-purpose application computing [2][37]. GPUs show their capability to run compute intensive jobs efficiently in terms of execution time and energy consumption. CPUs and GPUs have different execution capabilities, energy consumption and thermal characteristics. Although GPUs are known to be more energy efficient than CPUs, CPUs are more efficient in some cases, such as when the CPU-GPU communication overhead is too large compared to the computational time.

GPUs have been used as co-processors with CPUs forming heterogeneous systems. Job scheduling and workload distribution play a crucial role in such systems as they strongly affect the overall system performance, energy consumption, peak power and peak temperature. Allocating resources (e.g., core scaling, thread allocation) is another challenge since different sets of resources exhibit different behaviors in terms of performance and energy consumption. A classic job scheduler role is to map an entire job to a processing unit. A classic workload distributor role is to map parts of a job to more than one processing unit that cooperate to execute that job. Typically, the role of the GPU is to execute the parallel parts of a job and the role of the CPU (i.e., host) is to execute the sequential parts and manage the CPU-GPU data transfer. The host remains idle, waiting for the GPU’s execution and data transfer to complete. This classic workload distribution does not fully utilize the CPU and the GPU. It can be concluded that there is a need for efficient schedulers and workload distributors that fully exploit the potential of the CPU-GPU heterogeneous system.

The rest of the chapter is organized as follows. Section 2.1 presents a comparison between the CPU and the GPU architectures. Section 2.2 describes the job scheduler. Section 2.3 presents the workload distributor for heterogeneous systems. The heterogeneous platform that we used in this work is appeared in section 2.4. Section 2.5 lists the programming languages that we utilized in this study. A brief description of the benchmark that we used to test and evaluate our work is appeared in Section 2.6. Finally, we summarize this chapter in section 2.7.

2.1 CPUs vs. GPUs

An industry shift occurred when NVIDIA [2][37] developed the Compute Unified Device Architecture (CUDA), which enables the use of Graphic Processing Units (GPUs) as general-purpose processors (GPGPUs).

CPUs are latency-oriented processers where threads' latency is more important than the system's throughput. GPUs are the opposite; they are throughput-oriented processors that favor the system's throughput over an individual thread's latency. In other words, CPUs are designed to minimize a single thread's latency, whereas GPUs are designed to handle a large number of concurrent, lightweight threads to maximize overall throughput.

Fig. 2.1 shows the architectural overview of CPUs and GPUs. As shown in Fig. 2.1, compared to GPUs, CPUs have fewer but larger ALUs, a larger central data cache memory, and a larger central complicated control unit. Unlike CPUs that have a few large ALUs, GPUs have several hundreds of tiny energy efficient ALUs providing massively parallel computing. GPUs devote more transistors for data computation and less for data cache memory and control units. For this reason, GPUs are well-suited processors for compute intensive, highly parallel processing where memory latency can be hidden with computations instead of using large data caches. Moreover, GPUs are known to be high throughput and energy efficient processors. Although GPUs are more energy efficient than CPUs, they produce very high peak power compared to CPUs. Table 2.1 briefly shows the differences between CPUs and GPUs.

Table 2.1: CPU vs. GPU

CPU	**GPU**
Latency oriented	Throughput oriented
Large data caches	Small data caches
Large ALUs	Energy efficient small ALUs
Large control unit	Small control units

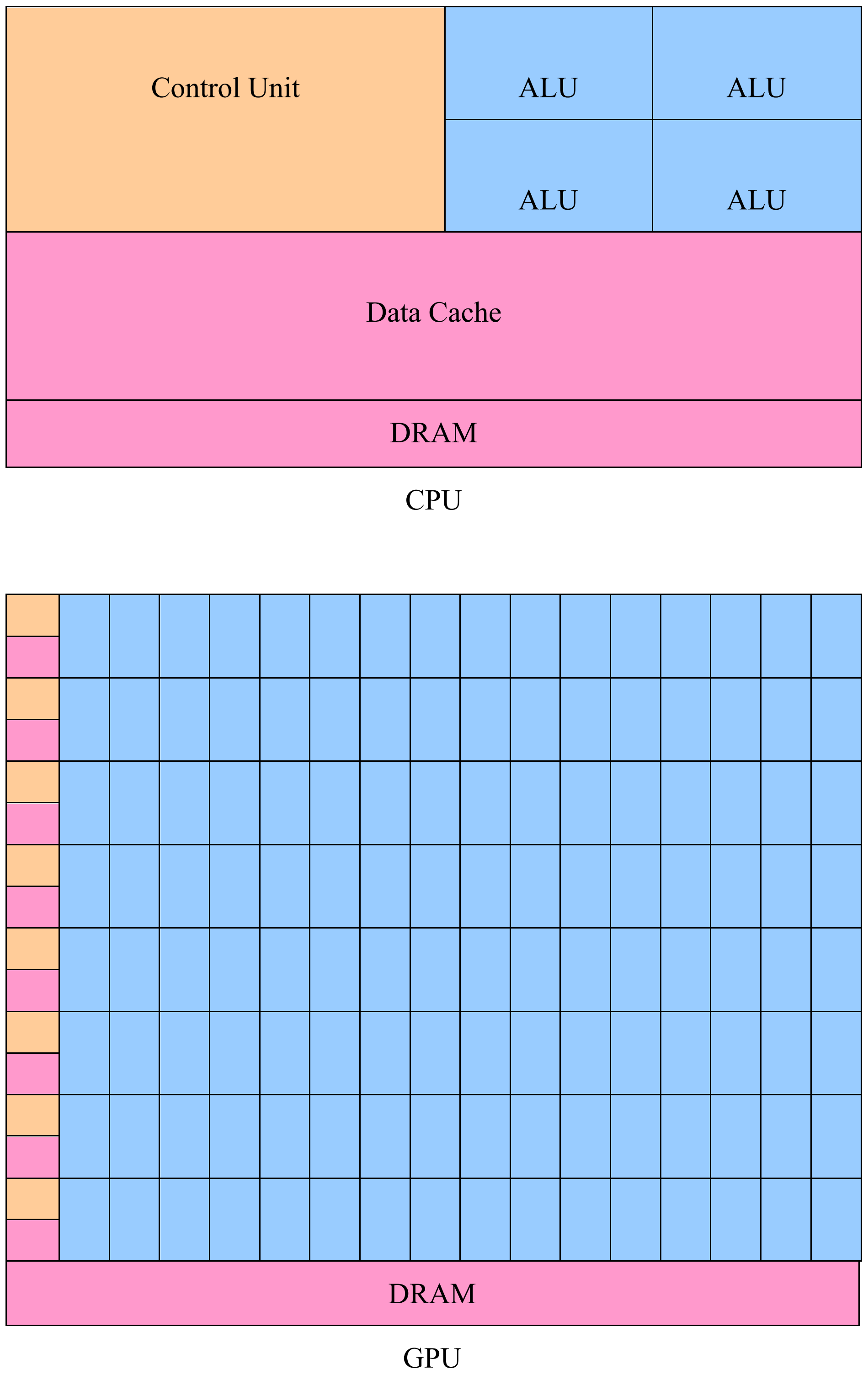


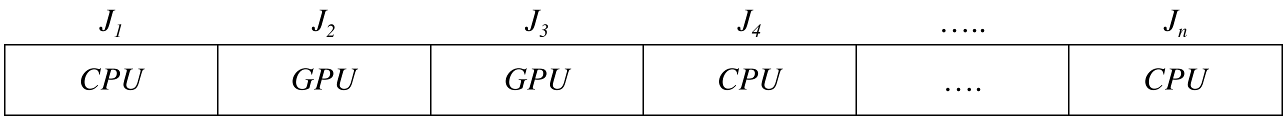
Figure 2.1: A CPU and a GPU Architectural Overview

2.2 Job Scheduler

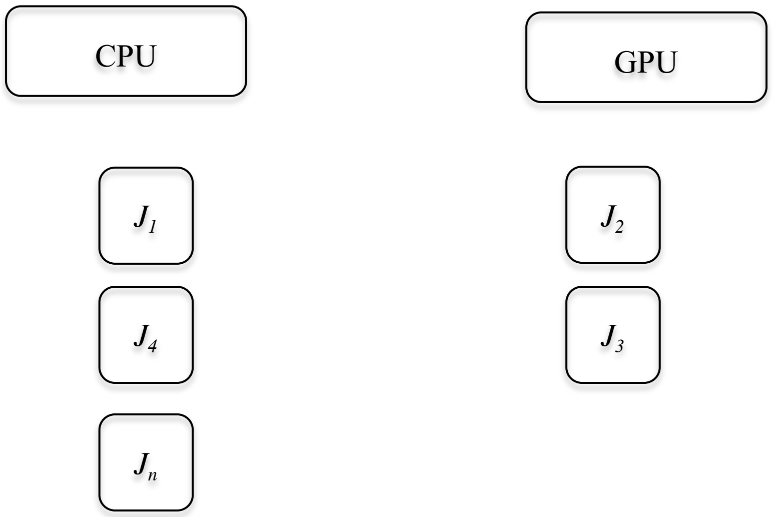
Today's systems from mobiles to servers are heterogeneous. They consist of different types of processing units with different characteristics such as CPUs and GPUs. Due to the architectural differences between the CPU and the GPU, as explained in section 2.1, we will get different job's execution time, energy consumption, peak power and peak CPU temperature if we run that job on a CPU or a GPU. Therefore, a job scheduler plays an important role in such systems where a jobs' schedule affects the overall system's performance, energy consumption, peak power and peak CPU temperature. We thus propose a job scheduler for CPU-GPU heterogeneous systems. The goal of that scheduler is to find an optimal jobs' schedule while taking into account both performance and energy consumption under the peak power and temperature constraints. Given a set of jobs, the job scheduler role is to map the entire jobs to the processing units according to the optimal schedule. Fig. 2.2(a) shows an example of a schedule. Fig. 2.2(b) illustrates that the entire jobs are mapped to execute on the designated processing unit according to Fig. 2.2(a) schedule. It should be noted that our job scheduler is an offline scheduler that provides the jobs' schedule beforehand.

2.3 Workload Distributor

GPUs have been used as co-processors with CPUs forming CPU-GPU heterogeneous systems. A classic workload distributor role is to map the sequential parts of a job to the CPU and map the parallel parts to the GPU. While running the parallel parts on the GPU, the CPU remains idle, waiting for the GPU's execution and data transfer to complete. This classic workload distribution does not fully utilize the CPU and the GPU. Thus, there is a need for an efficient job workload distributor to fully exploit the potential of the CPU and the GPU. Instead of being idle, the CPU can cooperate with the GPU to execute that job. In other words, the workload of a job can be distributed to execute on both a CPU and a GPU. We thus propose a job workload distributor that finds efficient jobs' CPU/GPU map ratios in terms of both performance and energy consumption under peak power and peak CPU temperature constraints. During runtime, if a job arrives, the goal is to find the best job distribution of work between the CPU and the GPU. It is to decide the percentage of job work mapped to the CPU and the percentage of job work mapped to the GPU. There is a possibility that the best CPU/GPU map ratio is to map the entire job to either a CPU or a GPU (e.g., in an extreme case). Fig. 2.3 shows an example of a job workload distribution on a CPU-GPU heterogeneous system. As the job arrives, as shown in Fig. 2.3, a part of the job is mapped to run on the GPU and the rest of the job is mapped to run on the CPU according to the CPU/GPU map ratio.



(a)



(b)

Figure 2.2: An example of jobs' schedule (a) and scheduling illustration (b)

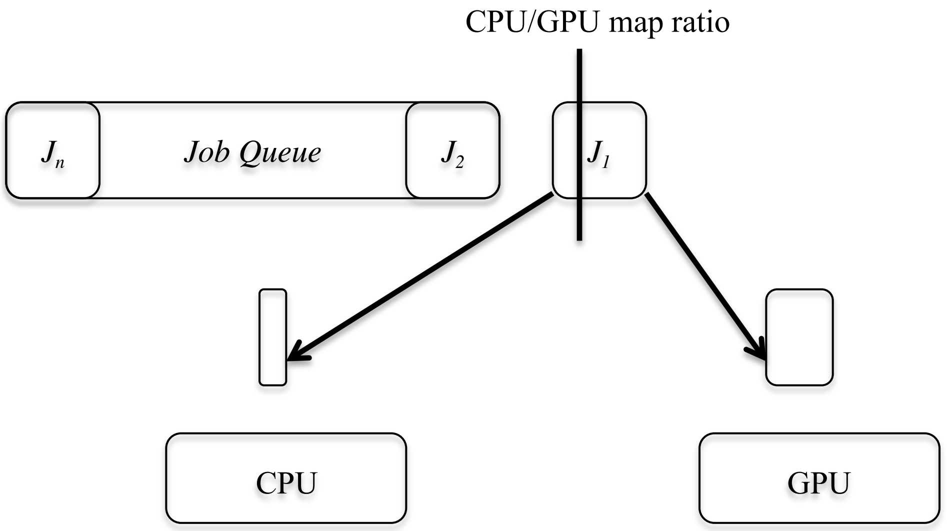


Figure 2.3: An example of job workload distribution

2.4 Experiment Setup

In our experiments, we chose to use an actual CPU-GPU heterogeneous system because it allows us to conduct the job scheduler and the workload distributor experiments in a heterogeneous environment. The system we utilized is equipped with a multi-core CPU and a GPU. As shown in Fig. 2.4, the CPU and the GPU are on different chips that are connected through a bus.

The multi-core CPU is an Intel Core i7-920 that combines four 3.06 GHz computing cores into a single processor [35]. The GPU is an NVIDIA Tesla C2070 with 448 cores in 14 streaming multiprocessors with a frequency of 1.15 GHz [36]. The CPU and the GPU are connected using PCI Express 2.0 bus [43].

We ran and tested the job scheduler and the workload distributor on that machine to demonstrate the benefits of our approaches. We introduce a methodology that is not hardware specific. Even though our results are based on that machine, the methodology we used in our algorithms is generic and can be applied to any heterogeneous system.

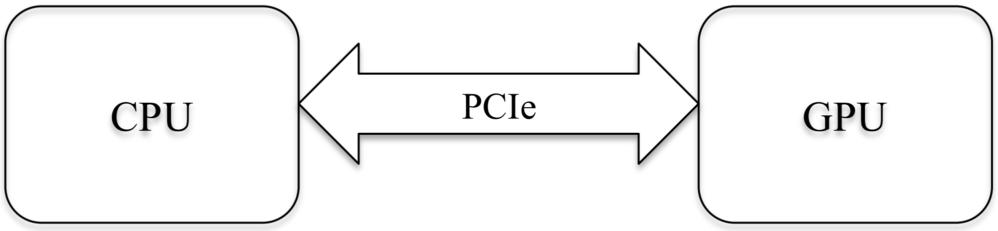


Figure 2.4: A CPU and a GPU connected using PCIe

2.5 Programming languages

In this work we used programming languages for implementation and for testing and running the benchmark on the targeted hardware. We used C++ to implement the job scheduler and the workload distributor. OpenMP (Open Multi-Processing) [30] was used to run the Rodinia benchmark applications [10][32] on the CPU side, whereas CUDA (Compute Unified Device Architecture) [2] was used to run the Rodinia benchmark applications on the GPU side. The following is a brief description of the programming languages that we used.

2.5.1 C++

We used the C++ language to implement the job scheduler and the workload distributor algorithms. C++ is a general-purpose object-oriented programming (OOP) language. B. Stroustrup developed C++ in 1979 [31]. C++ was initially standardized in 1998. We chose C++ because it is an efficient and flexible language. It provides both high and low-level language features for program organization. C++ is a C language extension that added object-oriented programming (OOP) features to C. As an OOP language, C++ offers classes, which provide four features: abstraction, encapsulation, inheritance, and polymorphism.

2.5.2 OpenMP

To run the Rodinia benchmark applications on the multi-core CPU side, we utilized OpenMP. OpenMP is an Application Program Interface (API) that supports shared-memory multiprocessor/multi-core programming in C/C++. OpenMP is a multithreaded programming model. A master thread forks and controls a set of slave threads. Parallel parts of the program are divided among these threads using work-sharing constructs. The threads run concurrently to execute the parallel parts of the program. During runtime, threads are allocated to different processors/cores. OpenMP is based on a combination of compiler directives. Programmers must use those directives to identify the parallel parts of a program that can be executed by multiple threads. The sequential parts of a program run by single threads.

2.5.3 CUDA

We utilized CUDA to run the Rodinia benchmark applications on the GPU side. CUDA is the parallel computing platform and programming model developed by NVIDIA. It enables the use of the GPUs for general purpose processing. It works with C/C++ programming languages. The program consists of series of sequential and parallel parts. The CPU, referred to as the host, is responsible for executing the sequential parts of the program. It also controls the CPU-GPU data transfer and the execution of the parallel parts/kernels by the device/GPU. The host/CPU transfers data to the device/GPU and remains idle until the GPU finishes executing its part/kernel and sends its data back to the host.

2.6 Rodinia Benchmark

In our experiments, we used the Rodinia 3.0 benchmark suite [10][32]. Rodinia is a collection of benchmarks designed for parallel processing on heterogeneous systems. It has applications from different domains such as image processing, bioinformatics, pattern recognition, scientific computing, and simulation. These kinds of applications represent different application behavior types that are characterized by Berkeley dwarves [26]. Each Rodinia benchmark application is coded using OpenMP to run on multi-core CPUs and CUDA to run on GPUs.

We selected the Rodinia benchmark suite because it has applications that are parallelized and coded using both OpenMP and CUDA. Therefore, the Rodinina benchmark is well suited for our experiments, which require applications to run on a CPU-GPU heterogeneous system. Applications that are parallelized and coded using OpenMP can run on the CPU side. Applications that are parallelized and coded using CUDA can run on the GPU side. We used the eighteen applications of the Rodinia benchmark that are coded using OpenMP and CUDA in our experiments. Table 2.2 summarizes the Rodinia benchmark applications [10]. It categorizes the applications according to their dwarves and domains. Refer to [10][32] for detailed description of each application.

Table 2.2: Rodinia Benchmark Applications

Applications	Dwarves	Domains
Leukocyte	Structured Grid	Medical Imaging
Heart Wall	Structured Grid	Medical Imaging
MUMmerGPU	Graph Traversal	Bioinformatics
CFD Solver	Unstructured Grid	Fluid Dynamics
LU Decomposition (LUD)	Dense Linear Algebra	Linear Algebra
HotSpot	Structured Grid	Physics Simulation
Back Propagation (BP)	Unstructured Grid	Pattern Recognition
Needleman-Wunsch (NW)	Dynamic Programming	Bioinformatics
Kmeans	Dense Linear Algebra	Data Mining
Breadth-First Search (BFS)	Graph Traversal	Graph Algorithms
SRAD	Structured Grid	Image Processing
Streamcluster (SC)	Dense Linear Algebra	Data Mining
Particle Filter (PFilter)	Structured Grid	Medical Imaging
PathFinder (PFinder)	Dynamic Programming	Grid Traversal
k-Nearest Neighbors (NN)	Dense Linear Algebra	Data Mining
LavaMD	N-Body	Molecular Dynamics
Myocyte	Structured Grid	Biological Simulation
B+ Tree	Graph Traversal	Search

2.7 Summary

In this chapter, we introduced the CPU-GPU heterogeneous systems. We then described the architectural differences between the CPU and the GPU. Because of these differences, running a job on a CPU or a GPU exhibit different behaviors in terms of execution time, energy consumption, peak power and peak CPU temperature. That arises the fact that there is a need for an efficient job scheduler and job workload distributor that exploit the full potential of the CPU-GPU heterogeneous system. Then, we briefly described the job scheduler and workload distributor. Finally, we presented the platform, programming languages and the benchmark that we used to implement and test the job scheduler and the workload distributor.

Chapter 3

Scheduling and Resource Allocation for Heterogeneous Systems

Performance of the system was the dominant factor for evaluating processer design for years. As the processors' design has hit the power wall, energy consumption becomes also important. However, minimizing energy consumption may degrade a system's performance. Hence, the energy and performance trade-off should be considered while designing a system. If the targeted system is an embedded system, peak power and peak temperature should be considered as additional design constraints.

The Graphical Processing Units (GPUs) are energy efficient processors that were originally designed for image processing. However, GPUs are now being used as general purpose processing units to execute highly parallel jobs [2][37]. GPUs show their capability to run compute intensive jobs efficiently in terms of performance and energy consumption. GPUs have been used as co-processors with CPUs forming heterogeneous systems. Integrating CPUs and GPUs that have different performance/energy characteristics on the same machine could enhance system's performance and energy efficiency.

Today's systems from mobiles to servers are heterogeneous. They consist of a mixture of different types of processing units such as CPUs and GPUs. Each of these processing units has different architectural strengths in execution capabilities and energy consumption. System's resources in general can be either processing units, number of cores, or number threads. In a heterogeneous system, selecting the best resources to run a job requires an understanding of the capabilities of the processing units with different resources (i.e., number of cores and threads). In addition, selecting the resources should be subjected to design objectives (i.e., performance and energy consumption) and constraints (i.e., peak power and peak CPU temperature limits). This is because different resources have different strengths and weaknesses with respect to these objectives and constraints.

Job mapping and scheduling play a crucial role in heterogeneous systems as they strongly affect the overall system performance, energy consumption, peak power and peak CPU temperature. Moreover, selecting the optimal resources (e.g., number of cores, number of threads) to run a job is also important due to their impact on the parameters above. Therefore, we propose a scheduler that not only maps and schedules jobs to processing units, but also it finds the number of cores and threads.

Many studies have been done on job scheduling with an eye on performance improvement. However, few of them tackle both performance and energy job scheduling. Thus, to enhance both performance and energy consumption in heterogeneous systems, we propose, as appeared in [19], our novel Performance, Energy and Thermal aware Resource Allocator and Scheduler (PETRAS). A preliminary version of this work appeared in [4] where Performance, Energy and Thermal aware Scheduler (PETS) first introduced. PETRAS is a scheduler that combines job mapping and scheduling, core scaling, and thread allocation into a single scheduler.

Job scheduling is known to be an NP-hard problem in the general case [7]. In addition, resource allocation (i.e., core scaling, thread allocation) makes the optimization problem more complicated. Moreover, PETRAS solves multi-objective problem that takes into account both performance and energy consumption. Thus, we apply an evolutionary algorithm called a Genetic Algorithm (GA). This algorithm promises to find nearly optimal solutions to such NP-hard problems. PETRAS utilizes a power management unit to go through the GA nearly optimal schedule, turning off the idle or low-utilized computational resources. This unit helps in saving energy consumption and freeing low utilized resources for use by other applications.

This chapter presents PETRAS that utilizes GA for resource allocation on a CPU-GPU system. On average, experimental results show that the PETRAS scheduler can achieve up to 4.7x speedup and an energy saving of up to 195% compared to performance based GA and other job schedulers.

The rest of the chapter is organized as follows. In Section 3.1, related work is discussed. Section 3.2 presents the motivation of PETRAS. Sections 3.3, 3.4 and 3.5 describe PETRAS problem in detail. Section 3.6 presents the profiler and the curve fitter. Section 3.7 describes the GA algorithm. PETRAS flowchart is appeared in section 3.8. Section 3.9 evaluates PETRAS on a CPU-GPU heterogeneous system. Finally, we summarize this chapter in Section 3.10.

3.1 Related Work

Several studies have been done on job scheduling to enhance overall performance, but few of them tackle both performance and energy consumption. Many researchers have investigated the use of GA to schedule tasks in heterogeneous systems. For instance, in [3, 14, 15, 18, 33], GA is applied to find an efficient task schedule that enhances overall system performance. But they did not consider the overall energy consumption in scheduling these tasks. Moreover, they solve a classic scheduling problem with no peak power or peak temperature constraints. However, PETRAS is a scheduler that combines both performance and energy consumption to evaluate a schedule with peak power and peak temperature constraints. In addition, PETRAS does not only perform scheduling, it also does core scaling and thread allocation as well.

Chiesi et al. [6] present a power-aware scheduling algorithm based on an efficient distribution of the computing workload on heterogeneous CPU-GPU architectures. The goal of that scheduler is to reduce the peak power of the system. It did not take into account the overall energy consumption. However, PETRAS schedules are optimized in terms of overall performance and energy consumption in addition to peak power constraints.

A power-aware task scheduling has been introduced in [5, 16] for real-time system tasks utilizing a DVFS. They schedule their tasks on a real-time multicore system. PETRAS, however, is targeting a CPU-GPU heterogeneous system.

An adaptive mapping technique has been proposed by Luk et al. [13] to map computations to processing elements on a CPU-GPU machine. It selects processing elements according to the computation input size and execution time. They did not consider energy consumption, peak power, or temperature. PETRAS on the other hand, maps jobs to processing units according the input size, execution time and energy consumption with a peak power and peak temperature as constraints.

Liu et al. [11] propose dynamic voltage frequency scaling (DVFS) [38][39] with core scaling (DVFCS). They utilized GA to minimize the power dissipation of many-core systems under performance constraints by choosing appropriate number of active cores and per-core voltage/frequency levels. Unlike DVFCS, PETRAS optimizes both energy consumption and performance. In addition to core scaling, PETRAS addresses job mapping and scheduling, and thread allocation problems.

In [12], Vega et al. present preliminary characterization data for multi-threaded programs to estimate the potential benefit of power-aware thread placement. PETRAS in turn exploits both performance and energy efficiency of thread allocation with core scaling and job mapping/scheduling.

3.2 Motivation

Our results from preliminary research, as described below motivated this study on developing an efficient job scheduler that takes into account both performance and energy consumption under peak power and peak CPU temperature constraints. We ran the Rodinia 3.0 benchmark suite [10][32] jobs with sizes of 1k up to 64G on a typical CPU-GPU heterogeneous system. We measured the execution time, energy consumption, peak power, and peak CPU temperature by running Rodinia jobs with (1) different processing units on a heterogeneous system consisting of a multicore CPU (4 cores) and a GPU (2) different number of cores, and (3) different number of threads (details of the hardware configuration are given in section 2.4). Our research tackles three problems: (a) job mapping and scheduling, (b) core scaling, and (c) thread allocation.

The results below demonstrate the three problems with different settings. These results illustrate that we do not have to solve each problem independently since the solution of one problem affects the others. We have to solve the three problems as one optimization problem; therefore, we combined job mapping and scheduling, core scaling and thread allocation problems into one scheduler.

The goal of that scheduler is to find the efficient job mapping, number of cores and threads in terms of both performance and energy consumption under peak power and peak CPU temperature constraints. Thus, we propose a Performance, Energy and Thermal aware Resource Allocator and Scheduler (PETRAS).

3.2.1 Scheduling and Core Scaling

We ran the Rodinia 3.0 benchmark suite [10][32] jobs with sizes of 1k up to 64G on a typical CPU-GPU heterogeneous system with different configurations. First, we ran the jobs with different processing units on a heterogeneous system consisting of a multicore CPU (4 cores) and a GPU. Second, we ran the jobs with different numbers of cores. We then measured the overall execution time, energy consumption, peak power, and peak CPU temperature of running these jobs.

On a logarithmic scale, Fig. 3.1(a) and Fig. 3.1(b) show the overall execution time and energy consumption of running each of the Rodinia benchmark jobs with a size of 1k on a GPU, a single core CPU, a dual-core CPU, and a quad-core CPU. Similarly, Fig. 3.1(c) and Fig 3.1(d) show the peak power and peak CPU temperature that are reached by running those jobs on the specified processing unit/number of cores.

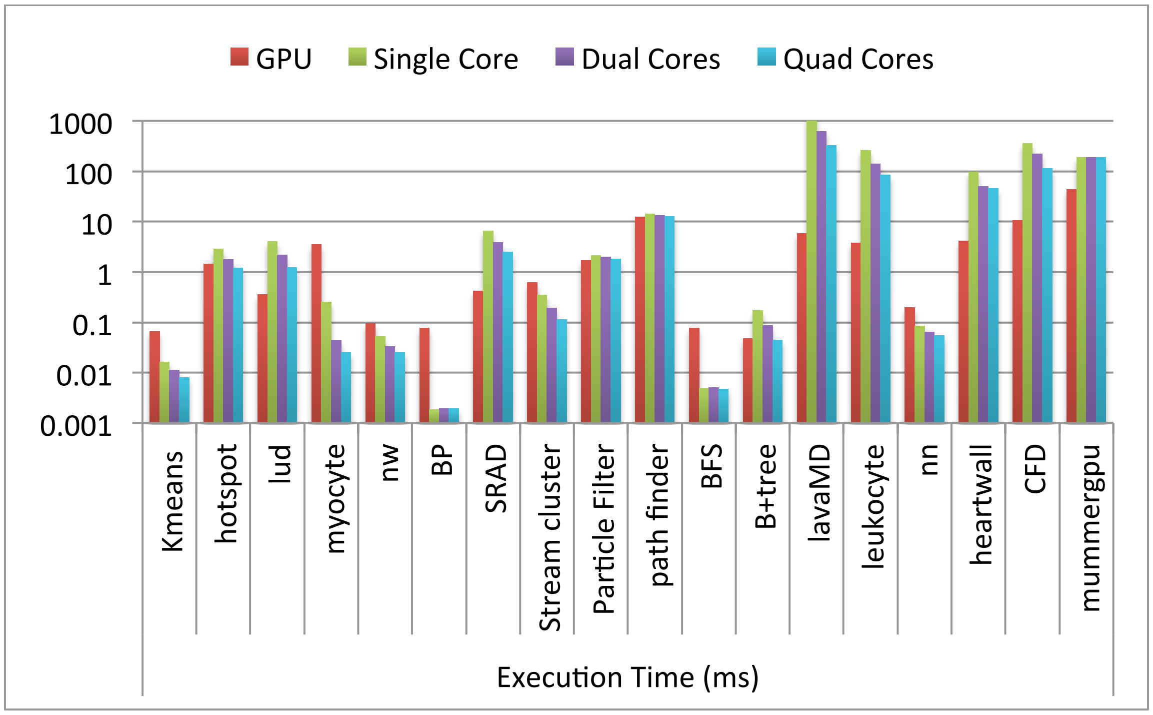
Fig. 3.1(a) shows that using different processing units to run a job results in different values of execution time. From these results it can be concluded that there is no specific processing unit/number of cores that is optimal in terms of performance for all of the jobs. Some of these jobs run faster on the GPU, such as lud, SRAD, lavaMD, leukocyte, heartwall, CFD, and mummergpu. For other jobs, CPU cores are better processing units. In addition, as the number of cores changes, the execution time varies.

Fig. 3.1(b) illustrates that the same job consumes different amounts of energy when it is executed on a different processing unit. GPUs are known to be more energy efficient multi-core CPUs. However, in some cases, the multi-core CPU outperforms the GPU in energy consumption, as shown in kmeans, nw, myocyte, BP, and BFS. Moreover, there is no processing unit or number of cores that consume the least energy for all of the jobs.

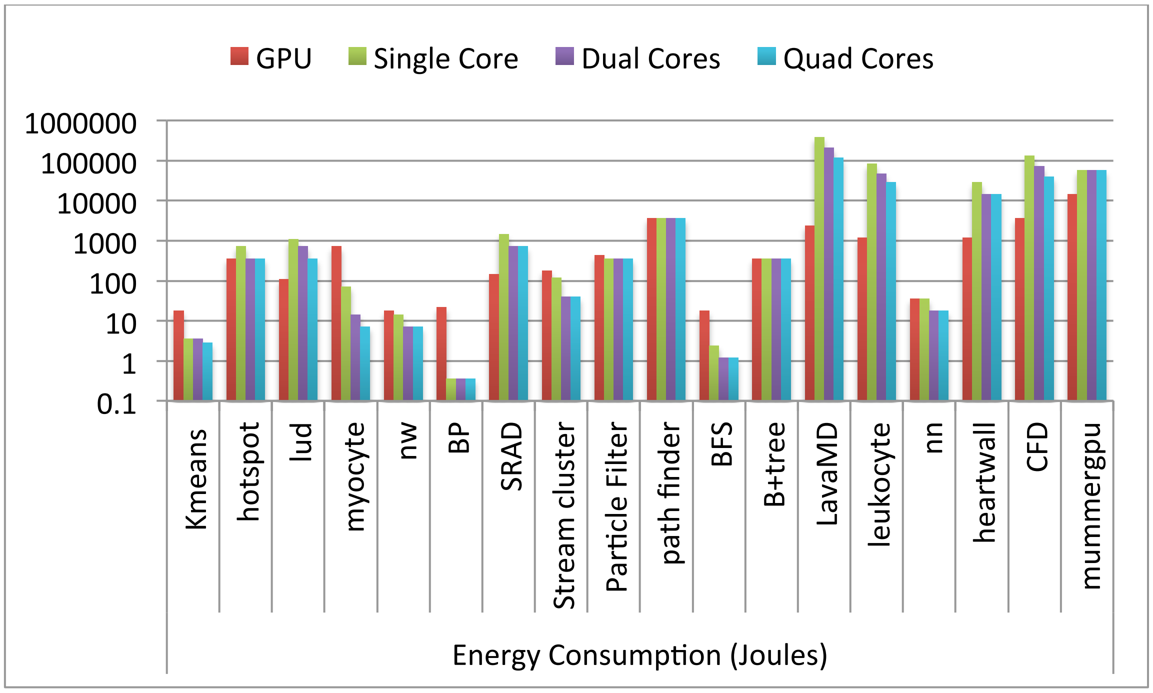
Peak power and peak CPU temperature are important factors for some design fields such as embedded systems. Fig. 3.1(c) and Fig. 3.1(d) show peak power and peak CPU temperature reached by running a job with a size of 1k on different processing units. Although GPU energy consumption is less in SRAD, lavaMD, heartwall, and CFD, the peak power is very high compared to the multi-core CPU. When there is a limitation on the peak CPU temperature, as in embedded systems, peak CPU temperature becomes also an important factor in selecting the right processing unit to run a job. In heartwall, for example, the GPU is excluded from the selection because it imposes a very high peak power. The second best option in terms of performance and energy consumption is the quad-core CPU. However, its peak CPU temperature is very high compared to other configurations. Therefore, peak power and peak CPU temperature should be measured and taken into consideration when selecting a processing unit.

Looking at all of the measured parameters, it can be concluded that choosing the right processing unit to enhance performance and minimize energy is not that easy. It gets more complicated if peak power and peak CPU temperature constraints are considered. For example in SRAD, if the selection criteria are based on only performance and energy consumption then GPU should be selected as the processing unit. But if the peak power is limited to a certain value, GPU may exceed that value and should not be selected. Instead, the quad-core CPU may be selected since it is the second best in terms of performance and energy consumption. But if peak CPU temperature has a limit, CPU quad may not be the best one, etc.

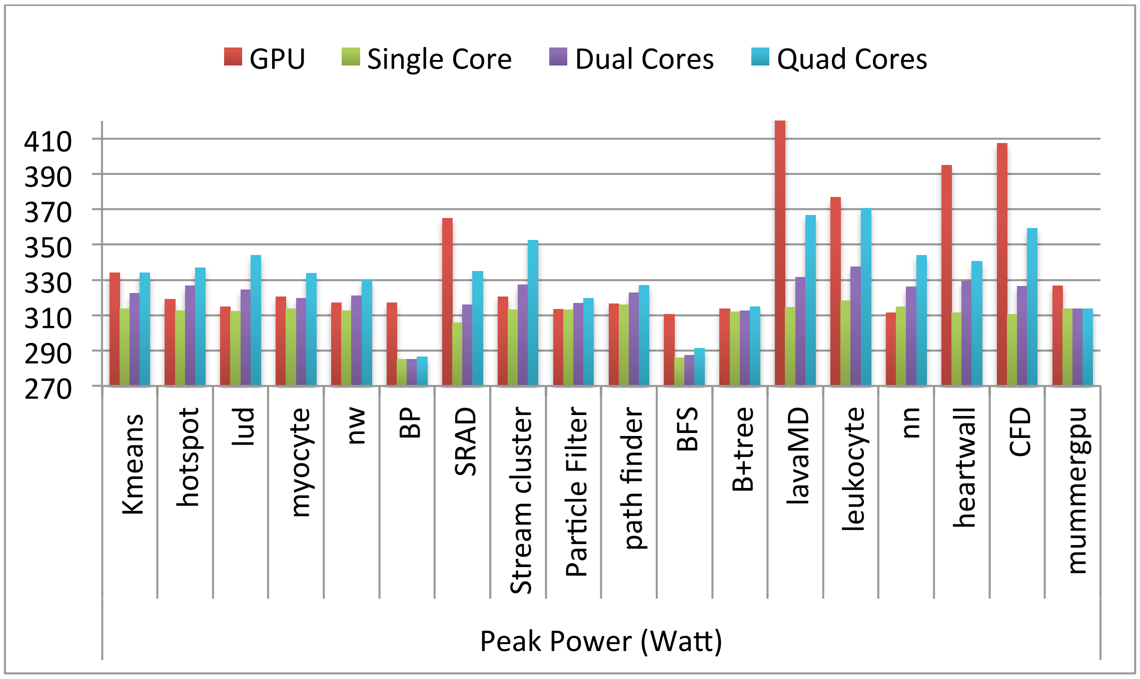
The results below were obtained by running a job with a size of 1k on different processing units/number of cores. Changing the size of a job produces different values of the measured parameters. The results change depending on the job size. For example, results show that GPU is the best processing unit in terms of performance and energy consumption for running the SRAD with a size of 1k. However, changing the size of SRAD (e.g., 16k) may exhibit different values in terms of the measured parameters (e.g., execution time, energy consumption). Therefore, the GPU may not be the best in this case and the multi-core CPU may be the best processing unit.


GPU
Single Core
Dual Cores
Quad Cores
1000
100
10
1
0.1
0.01
0.001
Kmeans
hotspot
lud
myocyte
nw
BP
SRAD
Stream cluster
Particle Filter
path finder
BFS
B+tree
lavaMD
leukocyte
nn
heartwall
CFD
mummergpu
Execution Time (ms)

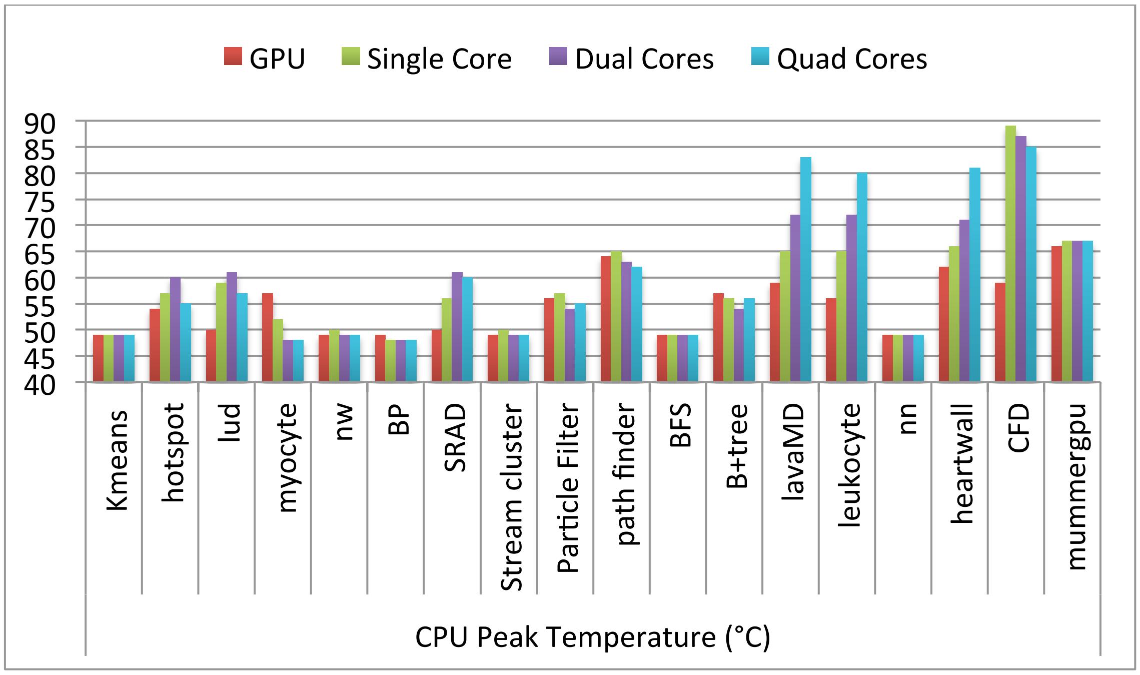
(a)


GPU
Single Core
Dual Cores
Quad Cores
1000000
100000
10000
1000
100
10
1
0.1
Kmeans
hotspot
lud
myocyte
nw
BP
SRAD
Stream cluster
Particle Filter
path finder
BFS
B+tree
LavaMD
leukocyte
nn
heartwall
CFD
mummergpu
Energy Consumption (Joules)

(b)



(c)



(d)

Figure 3.1: Execution time (a), energy consumption (b), peak power (c), and CPU peak temperature (d) of running a job on different processing units/number of cores.

3.2.2 Thread Allocation

To show the effect of changing the number of threads on a system, we ran the Rodinia 3.0 benchmark suite jobs with sizes of 1k up to 64G on a of a multicore CPU (quad-core CPU) with different number of threads. We then measured the overall execution time, energy consumption, peak power, and peak CPU temperature of running Rodinia jobs with the specified number of threads.

An experiment conducted by running the Rodinia benchmark jobs with sizes of 1k on a quad-core CPU with various numbers of allocated threads. On a logarithmic scale, Fig. 3.2(a) and Fig. 3.2(b) show how changing the number of allocated threads changes the execution time and energy consumption respectively. It can be concluded that there is no specific number of threads setting that has the shortest execution time and/or consumes the least energy for all of the jobs.

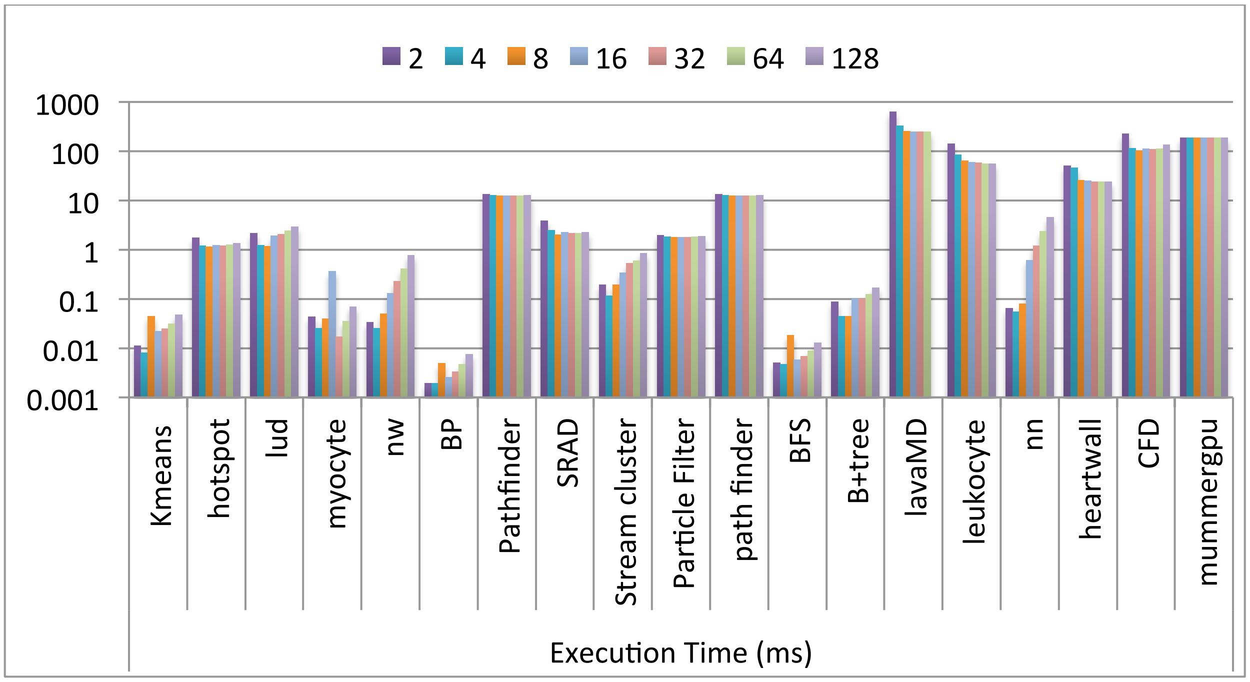
Fig. 3.2(c) shows the peak power and Fig. 3.2(d) presents the peak CPU temperature. The results show the effect of changing the number of threads on these parameters. From these results, it is clear that deciding on the number of threads has high impact on performance, energy consumption, peak power and peak CPU temperature.

Looking at all of the measured parameters, it can be concluded that choosing the right number of threads to enhance performance and minimize energy is hard. It gets more complicated if peak power and peak CPU temperature constraints are considered. For example in kmeans, if the selection criteria are based on only performance and energy consumption then 4 threads should be allocated. But if the peak power is limited to a certain value (e.g., 330), running kmeans with 4 threads exceeds that value and should not be selected. Instead, we have to select another number of threads setting that satisfies our needs.

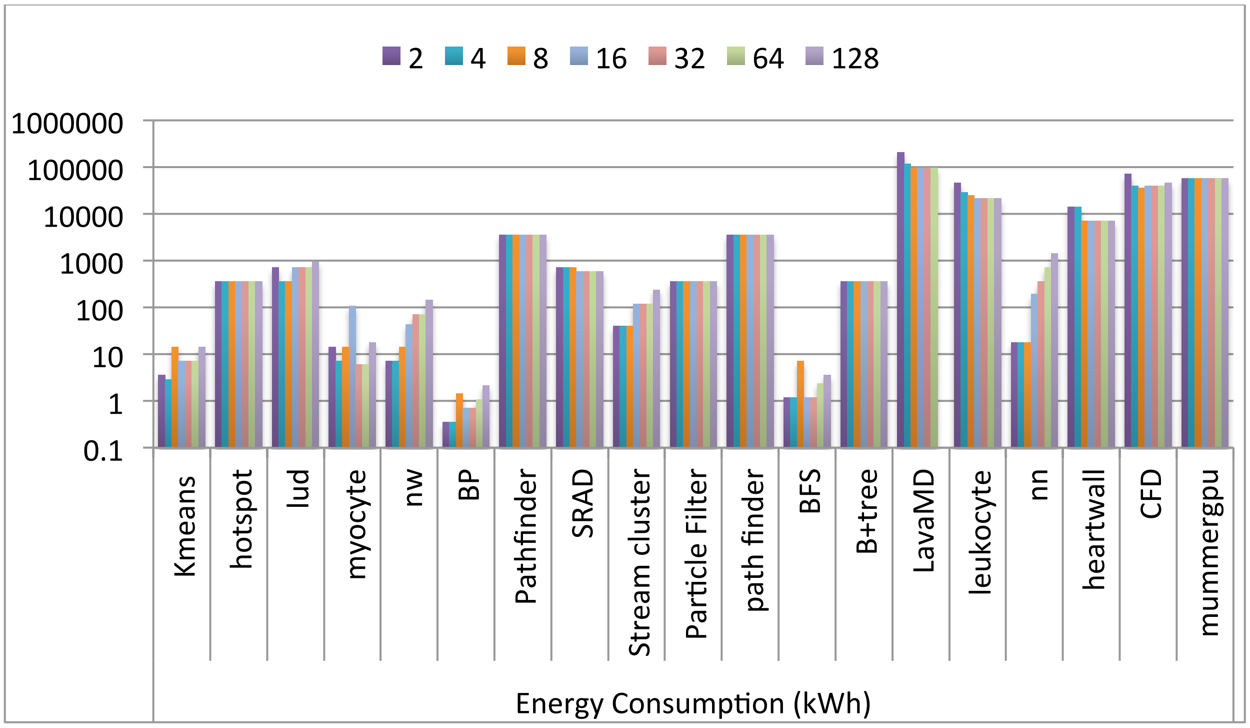
Moreover, these results were obtained by running the Rodinia jobs of a specific size (i.e., 1k) on the quad-core CPU with different thread number settings. The results change depending on the job size as well as on whether that job runs on single core, dual-core or quad-core CPU, etc. For example, results below show that running BP of a size 1k with 2 threads on a quad-core CPU is the best in terms of performance and energy consumption. However, changing the size of BP (e.g., 16k) may exhibit different values in terms of the measured parameters (e.g., execution time, energy consumption). Therefore, the allocating 2 threads may not be the best in this case and another number of threads setting should be selected. Moreover, if we run BP with a size of 1k on different number of cores such as dual-core CPU, we will get different values of the measured parameters (e.g., execution time, energy consumption). Therefore, the number of threads that is allocated to a job to run on the quad-core CPU may not be the best for the dual-core CPU.

3.2.3 Summary

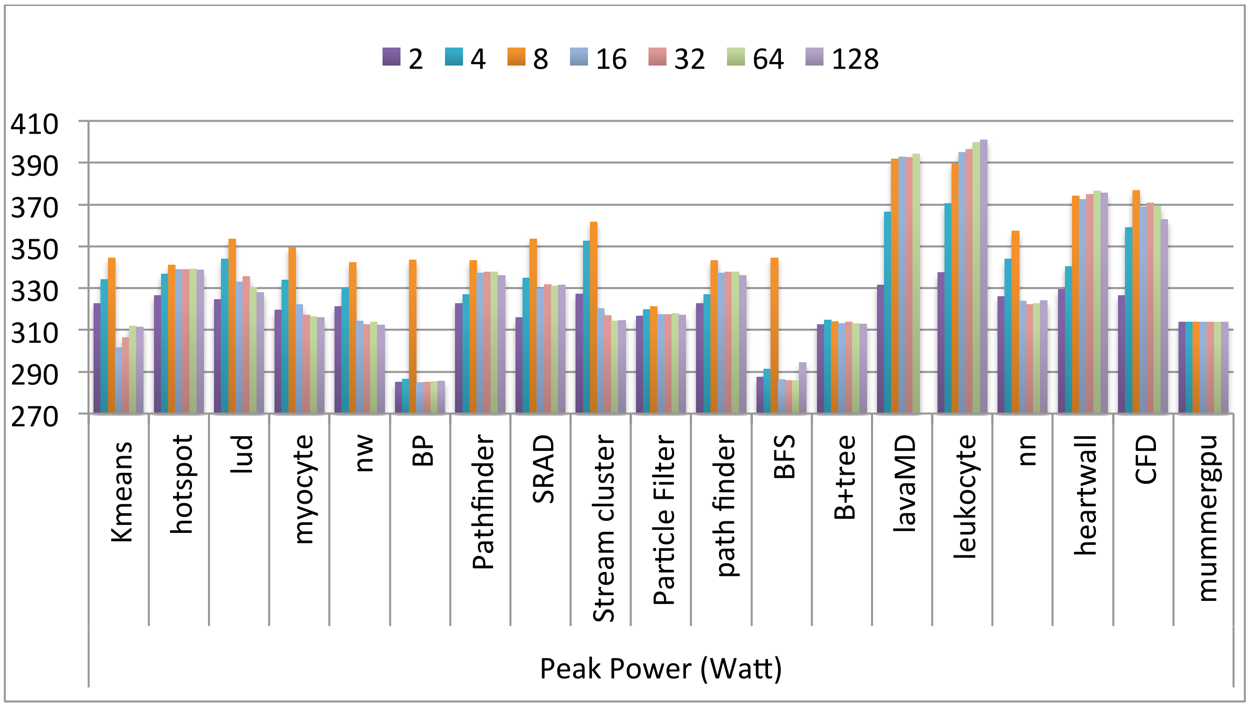
The results show the three problems (processing units mapping, core scaling, and thread allocation) with different settings. These results illustrate that we do not have to solve each problem independently since the solution of one problem affects the others. We have to solve the three problems as one optimization problem; therefore, we combined processing units mapping, core scaling and thread allocation problems into one scheduler. The goal of that scheduler is to find an efficient schedule, processing unit mapping, core scaling and thread allocation that try to enhance performance and minimize energy consumption under peak power and peak CPU temperature constraints.


2
4
8
16
32
64
128
1000
100
10
1
0.1
0.01
0.001
Kmeans
hotspot
lud
myocyte
nw
BP
Pathfinder
SRAD
Stream cluster
Particle Filter
path finder
BFS
B+tree
lavaMD
leukocyte
nn
heartwall
CFD
mummergpu
Execution Time (ms)

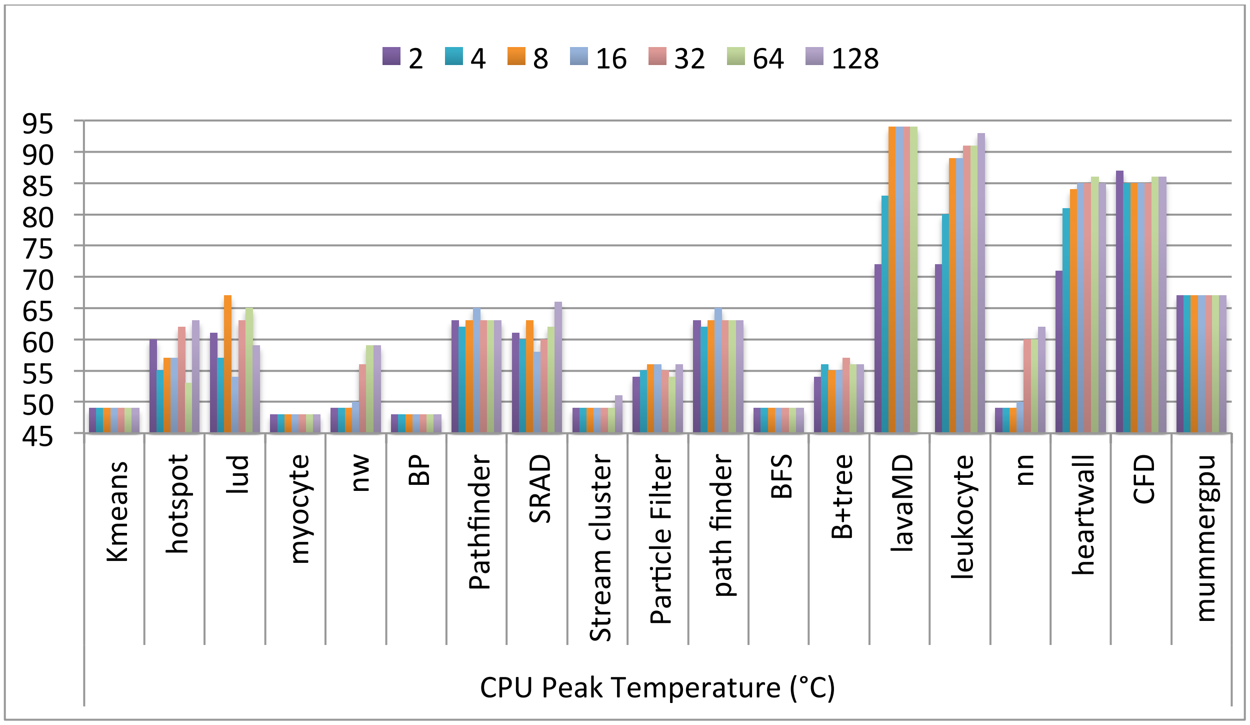
(a)


2
4
8
16
32
64
128
1000000
100000
10000
1000
100
10
1
0.1
Kmeans
hotspot
lud
myocyte
nw
BP
Pathfinder
SRAD
Stream cluster
Particle Filter
path finder
BFS
B+tree
LavaMD
leukocyte
nn
heartwall
CFD
mummergpu
Energy Consumption (kWh)

(b)



(c)



(d)

Figure 3.2: Execution time (a), energy consumption (b), peak power (c), and CPU peak temperature (d) of running a job with different number of threads

3.3 Performance, Energy and Thermal Aware Resource Allocator and Scheduler (PETRAS)

PETRAS is a performance, energy and thermal aware scheduling framework for managing jobs, resources and power in a heterogeneous system. This scheduler not only determines where to run jobs, it also provides more information regarding number of cores, number of threads and power management. PETRAS has many useful features. It is not a system-specific; it can be applied on any heterogeneous environment including embedded system because it takes into account peak power and peak temperature. It utilizes a system profiler and a curve fitter to predict jobs' execution time, energy consumption, peak power and CPU peak temperature.

The goal of our scheduler is to find a schedule that takes into account the overall performance and energy consumption simultaneously while not exceeding peak power and peak CPU temperature limits. Because this is a multi-objective optimization problem, there is no single schedule that can simultaneously optimize performance and energy consumption. Instead, there exists a set of Pareto optimal schedules. A schedule is called Pareto optimal if none of the objective functions can be improved in value (e.g., performance) without degrading the other objective value (e.g., energy consumption).

PETRAS does not only solve classic scheduling problems. Instead, it solves all the following problems at the same time: job mapping and scheduling, core scaling, and thread allocation. It also has a power management unit. To the best of our knowledge, this scheduler is the first attempt to combine all these into one optimization problem to consider both performance and energy consumption simultaneously under peak power and peak CPU temperature limits.

3.4 PETRAS Problems

PETRAS is not a classic job scheduler. It has a resource allocator that determines the number of cores and threads needed if the CPU is the processing unit. Hence, it combines core scaling and thread allocation into the job scheduling optimization problem.

3.4.1 Job Mapping and Scheduling

Our goal is to find optimal jobs' mapping and scheduling while taking into account both performance and energy consumption under the peak power and peak CPU temperature constraints. Given a set of jobs and processing units, job mapping problem is to decide which processing unit is responsible to run a job. (i.e., job mapping is based on processing unit affinity). Because these processing units are heterogeneous, different processing units have different strengths and weaknesses in respect to the following parameters: performance, energy consumption, peak power, and thermal activity. Hence, selecting the right processing unit that satisfies our objective and constraints is not straightforward. On the other hand, jobs scheduling can be defined as choosing an optimal execution order of jobs for a heterogeneous system.

3.4.2 Core Scaling

It is not always true that adding more cores to run a job will reduce its execution time. It may degrade its performance due to communication latency between cores. Moreover, some jobs do not have enough parallelism to utilize all of the cores provided. Hence, it is important to decide number of cores needed to run a job. PETRAS's core scaling aims to find the optimal number of cores needed to run a job. It takes into account both performance and energy consumption under peak power and peak CPU temperature constraints. Moreover, after finding the optimal number of cores, we turn off idle cores to save power and free these resources to be used by other applications that share the same hardware.

3.4.3 Thread Allocation

Multithreading is a programming and execution model that allows multiple threads to cooperate and utilize a multiprocessing system to execute a job. Jobs run on multicore processors or multiprocessing units can utilize multithreading to enable their parallel execution. Moreover, using multithreading helps to hide memory latency and enhance processing unit utilization by having more than one thread running on the same processing unit. If a thread is blocked waiting for resources, other threads can proceed, keeping the processing unit busy.

The number of threads allocated to a job highly affects its execution time, energy consumption, peak power, and CPU peak temperature. Using too many threads may degrade its performance and increase energy consumption due to high communication. And using too few threads may not be enough to achieve full parallelism, which in turn increases execution time and energy consumption. Hence, the number of threads should be carefully chosen. The optimal number of threads might be different for each job on a different processing unit. Given jobs and heterogeneous processing units, PETRAS's thread allocation goal is to find the optimal number of threads of a job that runs on a processing unit while taking into account both performance and energy consumption under peak power and peak CPU temperature constraints.

3.5 PETRAS Optimization Problem Formulation

The objective is to map jobs to processing units, decide the number of cores, set the number of threads and schedule their executions such that the overall schedule execution time and energy consumption are considered while not violating peak power budget and peak CPU temperature limit. We consider a heterogeneous system that has *m* processing units $P=\{P_1, P_2,\dots, P_m\}$ and a set of *r* resources $R=\{ R_1,R_2,\dots, R_r\}$. Processing units can be single core CPUs, multicore CPUs, GPUs, FPGAs, etc. These processing units can be fused on the same chip or connected through a network (e.g., PCI Express, a ring, or a mesh). Resources can be GPU's block size, CPU's number of threads, number of cores, etc. There are *n* compute intensive jobs $J=\{J_1, J_2,\dots, J_n\}$ competing for system resources. Estimation models of the overall execution time, energy consumption, peak power and peak CPU temperature of a given job J_i on processing unit P_j are available by profiling and curve fitting. Thus, the communication cost of transferring data between *PUs* is included in the estimation models. The PETRAS problem is formalized as follows:

3.5.1 Input

The input of PETRAS optimization problem consists of the following:

- A set of *n* compute intensive jobs $J=\{J_1, J_2,\dots, J_n\}$; where S_i is the input size of job J_i and $S_i =(0, S_{max}]$.
- A set of *m* processing units $P=\{P_1, P_2,\dots, P_m\}$.
- A set of *r* resources $R=\{ R_1,R_2,\dots, R_r\}$.
- The profiler training set's input sizes $TS=\{TS_1, TS_2,\dots, TS_m\}$; where $TS= [1, S_{max}]$.

- S_{max}; Jobs' maximum size.
- U; Utilization threshold.
- PP_{max}; Peak Power budget.
- PT_{max}; Peak temperature limit.
- An execution time model $T(ts_i, P_j)$ which determines the estimated execution time T_i of running a job J_i of a size ts_i on a processing unit P_j.
- An energy consumption model $E(ts_i, P_j)$ which determines the estimated energy consumption E_i of running a job J_i of a size ts_i on a processing unit P_j.
- A peak power model $PP(ts_i, P_j)$ which determines the estimated peak power PP_{ij} of running a job J_i of a size ts_i on a processing unit P_j.
- A peak CPU temperature model $PT(ts_i, P_j)$ which determines peak CPU temperature PT_{ij} of running a job J_i of a size ts_i on a processing unit P_j.

3.5.2 Objective Function

PETRAS is a multi objective optimization problem that takes into account both the total execution time (1) and total energy consumption (2) of a schedule. Hence, a weighted fitness function is used to combine them (3).

$$f_1 = min \sum_{ij} \Omega_{ij} T_{ij} \; ; \forall job \; i = 0, n \; and \; P \; j = 0, m \qquad (1)$$

$$f_2 = min \sum_{ij} \Omega_{ij} E_{ij} \; ; \forall job \; i = 0, n \; and \; P \; j = 0, m \qquad (2)$$

;where Ω_{ij} represents mapping a job i to a processing unit j. $\Omega_{ij} = 1$ is when a job i is scheduled to a processing unit j, otherwise $\Omega_{ij} = 0$. The parameter T_{ij} represents the execution time of a job *i* running on a processing unit j. The parameter E_{ij} represents the energy consumed by a job *i* running on a processing unit j.

The overall objective function is as follows:

$$\min f_{total} = wf_1 + (1-w)f_2 \quad ; w = [0,1] \tag{3}$$

;where *f1* and *f2* are normalized. *w* is a predefined weight value that determines the importance of each of the objective functions. If *w*= 0.5, both objective functions are equally important. If *w*=1 or 0, (3) is a single objective function that optimizes execution time or energy consumption respectively. Moreover, if *w*>0.5, execution time is more important than energy consumption. If *w*<0.5, energy consumption is more important than execution time.

3.5.3 Constraints

The objective function is subject to the following constraints:

- Each job should be mapped to a single processing unit

$$\sum_i \Omega_{ij} = 1 \, ; \forall job\ i = 0, n$$

- The peak power should be less than a peak power budget

$$\forall \Omega_{ij} * PP_{ij} < PPmax \,; \forall job\ i = 0, n\ and\ P\ j = 0, m$$

- The peak temperature should be less than a peak temperature limit

$$\forall \Omega_{ij} * PT_{ij} < PTmax \,; \forall job\ i = 0, n\ and\ P\ j = 0, m$$

- The processing unit utilization should be larger than a threshold

$$\forall U_{Pi} > U; \forall P\ i = 0, m$$

;where the parameter $PP_{ij}\ and\ PT_{ij}$ represent the peak power and peak temperature of a job i running on a processing unit *j* respectively.

3.6 Profiler and Curve Fitter

To evaluate a schedule and compare it to other schedules, we have to measure the overall performance, energy consumption, peak power and peak temperature after mapping jobs to processing units, setting the number of cores and the number of threads, and the order of jobs. However, there are no accurate models that can be used to estimate all these parameters. The expected performance, energy consumption, peak power and peak temperature of a job running on a processing unit are hard to predict. Moreover, the prediction becomes more complicated when the number of cores and threads change. Hence, PETRAS uses a profiler and curve fitter to find estimation models for the parameters above. The curve fitting method is described in detail in [13]. Profiling is done by measuring the system overall execution time, overall energy consumption, peak power and peak temperature of running a given job J_i on processing unit P_j for various input sizes, number of cores and number of threads. Since our profiler is based on the overall system parameters' readings (e.g., overall execution time in (4) and overall energy consumption in (5)), it includes memory latency and communication overhead between processing units. After these samples are collected, we use a curve fitter to produce estimation models that can be used for prediction.

$$T = T_{computation} + T_{memory} + T_{Communication};$$

$$T_{computation} = T_{CPU} + T_{GPU} + \ldots \qquad (4)$$

$$E = E_{computation} + E_{memory} + E_{Communication};$$

$$E_{computation} = E_{CPU} + E_{GPU} + .. \qquad (5)$$

3.7 Genetic Algorithm

PETRAS has a job scheduler, and scheduling is known to be an NP-hard problem [7]. It does not only solve the classic job scheduling problem. Instead, it solves all the following problems at the same time: core scaling, thread allocation, job mapping and scheduling. Moreover, PETRAS solves a multi-objective problem in which both performance and energy consumption must be considered. That makes the problem more complicated, and the search space to find the nearly optimal schedule is very large. Hence, we applied an evolutionary algorithm called a Genetic Algorithm (GA) to find nearly optimal schedules.

GA [8][17] is a search algorithm inspired by natural selection and genetics that is based on the survival of the fittest theory. It is an iterative search technique that is applied to solve optimization problems to find a nearly optimal solution. Unlike traditional random search, it does not examine a single solution/schedule, it is a population-based algorithm which makes exploring the search space faster. Being a population-based approach, GA is well suited to solve multi-objective optimization problems. GA applies different operators (e.g., crossover, mutation) to evolve from one population to another. These operators help in exploiting and exploring the search space without getting stuck in a local optimum.

As shown in Fig. 3.7, GA starts with an initial random population of *S* solutions (schedules). Fig. 3.3 shows an example of a population. Each solution/schedule is represented by a one-dimensional array of objects. A fitness function is used to evaluate the solution. In each iteration, a new generation of solutions (offsprings) is produced by performing crossover and mutation operators on the previous generation (parents). If the offspring that is produced is better than its parents, it replaces its parents. Otherwise, parents remain in the next generation.

GA ensures feasibility of the produced schedules of each generation. A solution/schedule is said to be feasible if it meets problem constraints (e.g., peak power, peak temperature). If a solution violates any of the constraints, GA identifies the violating point(s) and modifies them randomly, ensuring solution feasibility. For example, if running a job *x* on processing unit *y* violates the peak power constraint, another processing unit *z* is assigned randomly instead of *y*. The attempts to ensure solution feasibility are limited to *c* times (i.e., input), otherwise the parent is selected instead of the offspring. The algorithm stops after a specified number of iterations or if the best solution is not changed for a number of iterations.

To apply GA on the PETRAS multi-objective optimization problem, we use the weighted sum method for the fitness function [9]. This method transforms both objectives (e.g., performance and energy consumption) into an aggregated objective fitness function by multiplying each objective function by a pre-defined weight and summing up all weighted objective functions.

	J_1	J_2	J_3	J_4	J_5		J_n	
Solution 1	4	3	1	5	8	…	3	65
	2	1	8	2	4	…	2	380
	16	4	2	8	32	…	64	65.5
Solution 2	6	3	7	9	8	…	5	128
	2	1	8	1	4	…	4	367
	2	8	64	8	32	…	128	70.1
Solution 3	1	4	3	9	6	…	5	57
	2	1	4	4	4	…	4	383
	16	4	8	8	32	…	32	75.2
…						…		….
Solution *S*	2	5	7	2	2	…	2	30
	2	1	2	4	4	…	4	374
	16	4	32	32	32	…	8	68.9

Figure 3.3: GA population sample

3.7.1 Solution Representation

GA is a population-based algorithm that has *S* solutions/schedules. As shown in Fig. 3.4, a schedule is represented by a one-dimensional array of objects of size *n* where *n* is the total number of jobs. Each object of the array refers to a job. The object has three attributes: first, an integer value that identifies the processing unit to which that job is scheduled (e.g., CPU or GPU). Second, a number of cores as an integer (e.g., 1, dual, or quad). Third, a number of threads as an integer value. In Fig. 3.4 for example: Job 1 is assigned to processing unit 4, which has dual cores and 16 threads. Each solution has its normalized fitness value, peak power in Watts and peak temperature in Celsius. The fitness value is used to evaluate a solution, whereas peak power and peak temperature are used to ensure the feasibility of a solution by not violating constraints (e.g., peak power and peak temperature limits).

		J_1	J_2		J_n			
Processing unit	→	4	2	…	3	65	←	Fitness
Number of cores	→	2	1	…	2	375	←	Peak power (Watts)
Number of threads	→	16	4	…	64	65.4	←	Peak temperature (°C)

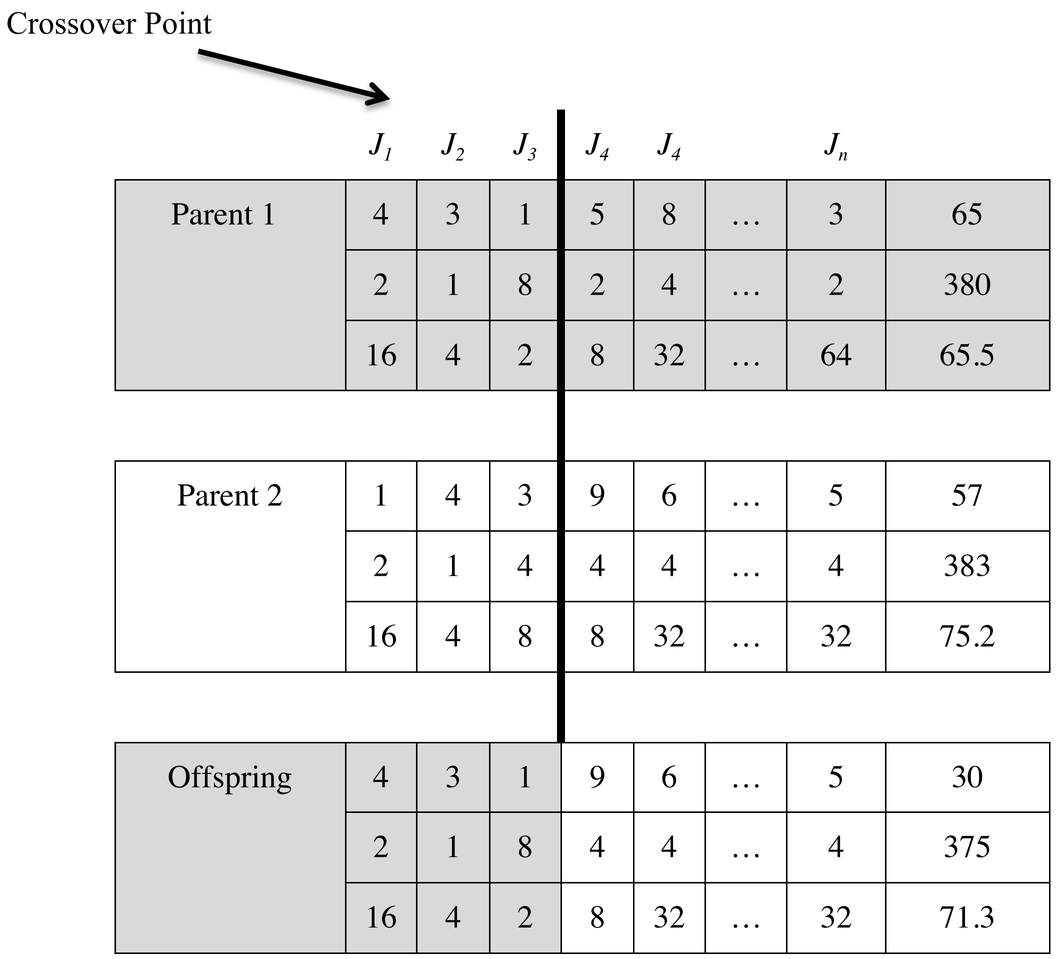
Figure 3.4: GA solution representation

3.7.2 Fitness Function

GA uses the fitness function to evaluate a schedule. A schedule is better if it has a lower fitness value. Since PETRAS solves a multi-objective optimization problem, it uses the weighted sum fitness function to consider both performance and energy consumption. To calculate the total execution time of a schedule as shown in (1), we used performance estimation models that were obtained by profiling and curve fitting to predict the execution time T_{ij} of each job J_i run on a processing unit P_j with the specified number of cores and threads. Then we calculated the summation. The same method was used to calculate the total energy consumption of a schedule as shown in (2). As in (3), the total execution time and total energy consumption of a solution were calculated and normalized then multiplied by a specified weight to calculate the fitness value of a solution.

3.7.3 Crossover

A crossover operator is performed on a current generation population to produce a new generation of solutions. The crossover operator helps to exploit the search space. It is applied on two solutions of a population that called parents to produce a new solution/offspring. The crossover operator has various forms, but in PETRAS, we selected the single point crossover in our algorithm. The single point crossover selects two parents randomly from the population based on their fitness, chooses a random cut point, and creates an offspring that has the right part of that point of its first parent and the left part of its second parent. Fig. 3.5 is an illustration of a crossover operator.



	J_1	J_2	J_3	J_4	J_4		J_n	
Parent 1	4	3	1	5	8	…	3	65
	2	1	8	2	4	…	2	380
	16	4	2	8	32	…	64	65.5

	J_1	J_2	J_3	J_4	J_4		J_n	
Parent 2	1	4	3	9	6	…	5	57
	2	1	4	4	4	…	4	383
	16	4	8	8	32	…	32	75.2

	J_1	J_2	J_3	J_4	J_4		J_n	
Offspring	4	3	1	9	6	…	5	30
	2	1	8	4	4	…	4	375
	16	4	2	8	32	…	32	71.3

Figure 3.5: GA crossover operator

3.7.4 Mutation

A mutation operator occurs according to a mutation probability that should be very low. It makes a tiny random change to a solution to introduce diversity into population and avoid local minima. Therefore, the new solution produced will not be very different from the original one. After an offspring is produced from the crossover operator, mutation is applied with a very low probability. PETRAS's mutation operator as in Fig. 3.6 selects a random point and switches the values of processing units, number of cores and threads with values selected randomly.

Mutation Point

	J_1	J_2	J_3	J_4	J_5		J_n	
Solution 2	1	4	3	9	6	…	5	65
	2	1	4	4	4	…	4	380
	16	4	8	8	32	…	32	65.5

	J_1	J_2	J_3	J_4	J_5		J_n	
Offspring	1	8	3	9	6	…	5	57
	2	4	4	4	4	…	4	372
	16	32	8	8	32	…	32	67.2

Figure 3.6: GA mutation operator

3.8 Power Management Unit

PETRAS's power management unit takes the efficient schedule as an input. This unit has two functionalities. First, it turns off all idle processing units. Second, it is responsible for shutting off low-utilized processing units and re-assigning their jobs to other processing units randomly. As shown in (6), to determine processing unit utilization, we divide the number of assigned jobs to that processing unit by the total number of system jobs. Processing unit utilization categorization is shown in (7). A low-utilized processing unit is a processing unit that has a lower utilization than a predefined utilization *U*. On the other hand, a high-utilized processing unit is a processing unit that has a higher utilization than *U*. A processing unit is said to be idle when it has zero jobs assigned to it. To turn a processing unit off we set its frequency to zero. The goal of power management unit is to save energy consumption and free idle and low-utilized processing units to be used by other applications that share the same hardware.

$$U_{Pi} = \frac{total\ number\ of\ assigned\ jobs\ to\ Pi}{totat\ number\ of\ system\ jobs} \qquad (6)$$

Processing unit utilization can be categorized as

$$Pi = \begin{cases} Idle & if\ U_{Pi} = 0 \\ Low\_utilized & if\ U_{Pi} < U \\ High\_utilized & if\ U_{Pi} \geq U \end{cases} \qquad (7)$$

3.9 PETRAS flowchart

As in Fig. 3.7, PETRAS starts with jobs and processing resources information as inputs. It utilizes GA that uses estimation models that are generated by profiling and curve fitting for schedule evaluation. When GA terminates, the efficient schedule, processing unit mapping, number of cores and number of threads are set. Then, power management unit turns off the idle and low-utilized processing units of the GA output schedule.

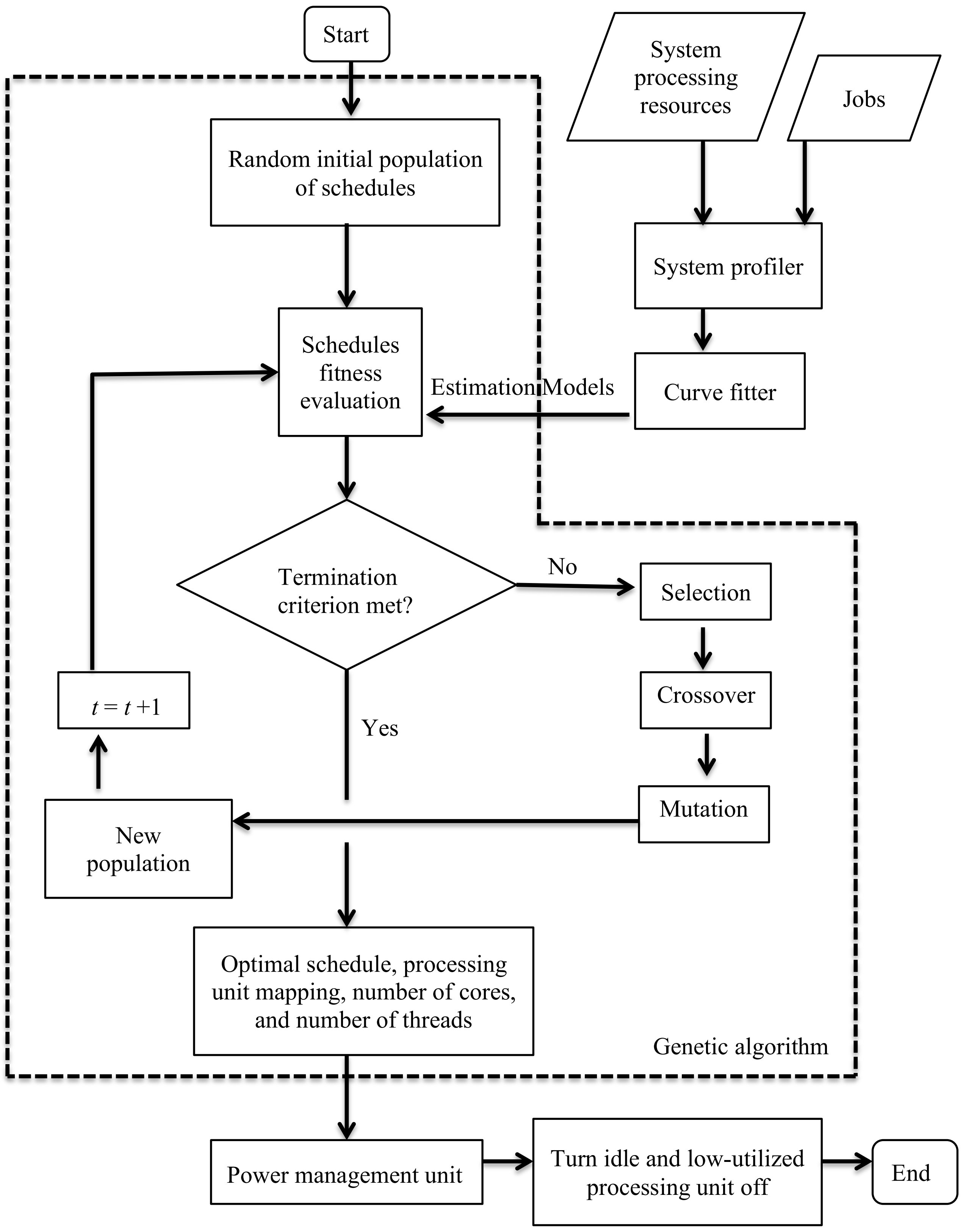

Start
System processing resources
Jobs
Random initial population of schedules
System profiler
Schedules fitness evaluation
Estimation Models
Curve fitter
Termination criterion met?
No
Selection
Yes
Crossover
t = t +1
Mutation
New population
Optimal schedule, processing unit mapping, number of cores, and number of threads
Genetic algorithm
Power management unit
Turn idle and low-utilized processing unit off
End

Figure 3.7: PETRAS flowchart

3.10 Evaluation

This section presents the evaluation methodology and the results of PETRAS on an actual CPU-GPU heterogeneous system.

3.10.1 Experimental Setup and Benchmark

We conducted our experiments on an actual system equipped with a multicore CPU and a GPU connected via PCI-e. Table 3.1 shows the detailed architectural parameters. We used a quad-core Intel i7 processor that has 4 physical cores and 8 threads which is seen by the operating system as 8 cores.

For testing, we used the Rodinia 3.0 benchmark suite [10][32], which is a collection of benchmarks for parallel processing on heterogeneous systems. It contains parallel applications from various domains such as medical imaging, bioinformatics, data mining, and scientific computing. Each application is parallelized and coded using OpenMP [30] (Open Multi-Processing) for multi-core CPUs and CUDA for GPUs. We ran applications (18 applications) that have both OpenMP and CUDA versions of Rodinia 3.0 benchmark to evaluate PETRAS.

Although PETRAS was tested on a heterogeneous CPU-GPU system, PETRAS is generic in that it can be applied on any platform with different processing units. The key idea is to show the capability of PETRAS when it comes to scheduling jobs on heterogeneous systems (regardless of the hardware) with different types of processing units with various characteristics, such as CPUs, GPUs, FPGAs, etc.

Table 3.1: Testbed Configuration

CPU	Intel Core i7-920 @3.06 GHz 4 cores
C Compiler	gcc 4.4.3
GPU	NVIDIA Tesla C2070
CUDA	Compiler CUDA 3.2, V0.2.1221
Operating system	UBUNTU 10.04 (X64)
PCI-e version	2.0

3.10.2 Profiling and Curve Fitting

For profiling, we ran each application of the Rodinia benchmark on both the CPU and the GPU with different resources (input size, number of cores, and number of threads). We ran each application and change one of the resources and fix the rest. For example, we ran application *A* with a fixed size *x* and fixed number of threads *z* but with different number of cores (1, 2, 4). We used a Linux command to change the boot arguments to disable/enable cores. Then we ran A with size *x* on a fixed number of cores *y* with different numbers of threads (1, 2, 4, 8, etc.). Then we ran *A* with a fixed number of cores *y* and a fixed number of threads *z* but with different input sizes from 1K up to 64G. For each of the experiments above we ran the application 1000 times, measured the parameters and took the averages.

We connected a Kill-A-Watt meter [40] to the system power supply (PSU) to measure the overall energy consumption (kWh), and the peak power (Watt). We used time Linux command [42] to measure the overall wall clock execution time (seconds). Therefore, all measurements of execution time and energy consumption are for the entire system where memory delay and CPU-GPU communications are included. To measure peak CPU/GPU temperatures, we used the lm_sensors application (Linux monitoring sensors) [41]. If a job runs on a multi-core CPU, lm_sensors monitors each core's temperature separately and we record the highest core temperature as the peak CPU temperature. If it runs on a GPU, we capture the highest GPU temperature value reached. Since running a job on a GPU needs one of the CPU cores to control the execution, we recorded CPU peak temperature as well. Curve fitting is performed on the data measured by profiling. It tries to find the best model that fits the gathered data points to produce estimation models. These estimation models are used in PETRAS to predict performance, energy consumption, peak power, and peak temperature for future jobs.

3.10.3 PETRAS vs. Other Scheduling Algorithms

We compared PETRAS to the following schedulers:

- Minimum energy greedy algorithm (MinE): for each job, it performs an exhaustive search to find what resources (processing unit, number of cores and number of threads) consume the least energy by running that job.
- Minimum execution time greedy algorithm (MinT): for each job, it performs an exhaustive search to find what resources (processing unit, number of cores and number of threads) have the shortest execution time by running that job.
- Round Robin algorithm (RR): each job is scheduled to use the next available resources in a round robin fashion.
- GPU-Only: all jobs are scheduled to run on the GPU.
- CPU-Only: all jobs are scheduled to the CPU with random resources (number of cores, number of threads).
- Random: for each job, it selects resources randomly.
- Performance-based GA (GAP): a scheduler that utilizes GA and uses (1) as a fitness function to minimize the overall execution time. GAP accounts for schedulers that are implemented by [3, 14, 15, 18].
- Energy-based GA (GAE): a scheduler that utilizes GA and uses (2) as a fitness function to minimize the overall energy consumption. Although [5, 16] schedulers do not consider heterogeneous systems, GAE considers these schedulers on a heterogeneous system.

Although PETRAS profiler and curve fitter are inspired by Qilin [13], PETRAS cannot be compared to Qilin because Qilin distributes threads of a job into both a CPU and a GPU whereas PETRAS considers mapping an entire job with its threads to either a CPU or a GPU.

3.10.4 Complexity

For a fair evaluation, PETRAS and the other algorithms we evaluated use the same profiler. We perform profiling once and offline. The results were used by PETRAS and the other algorithms to estimate the execution time, energy consumption, peak power, and peak temperature. Hence, the profiler overhead is neglected. Note that profiling and curve fitting are tools that we used for estimation and can be replaced by any other estimation models or tools. For PETRAS, GAP and GAE, we used a classic GA that has a complexity of O(Number of iterations*population size*n). MinT and MinE have O(n*n) complexity. Whereas, CPU-Only, GPU-Only and RR have O(n) complexity.

3.10.5 Implementation

We implemented PETRAS and the other scheduling algorithms using C++. PETRAS is a GA scheduler that is based on the weighted sum fitness function (3). If the w=1 or 0, PETRAS acts as GAP or GAE respectively. We tested different values of w but we selected w={0.25, 0.5, 0.75} because these values cover all weight scenarios. If w=0.5 both execution time and energy consumption are equally important. But when w=0.75 it favors execution time over energy consumption and the opposite if w=0.25. After PETRAS finds the efficient schedule, a power management unit checks if any of the processing units violate the utilization constraint U, and if so it turns them off and reschedules their jobs. We tested PETRAS for different values of U but we selected *U*={5%, 7.5%, 10%} because higher values of *U* will shut down needed processing units. GA setup parameters are: population size 250, number of iterations 1000 and mutation rate 0.05.

3.10.6 Results

We tested each of the schedulers by running 5000 randomly generated jobs for applications of the Rodinia benchmark and then we took the average. To evaluate the schedulers we tested the following cases: schedulers with no constraints (peak power, peak CPU temperature) as in Fig. 3.8, with only peak power constraint as in Fig. 3.9, and with both constraints as in Fig. 3.10.

To select the peak power budget PPmax, we measured the peak power reached by each Rodinia application using the random scheduler and reduced it by 5% or 10%. We did the same to select peak temperature PTmax. Fig. 3.9 and Fig. 3.10 show PPmax and PTmax with a 10% reduction. Moreover, some of the applications were executed with a low peak power or peak temperature, so reducing them would have prevented the processing units from operating. Therefore, they were discarded from the results.

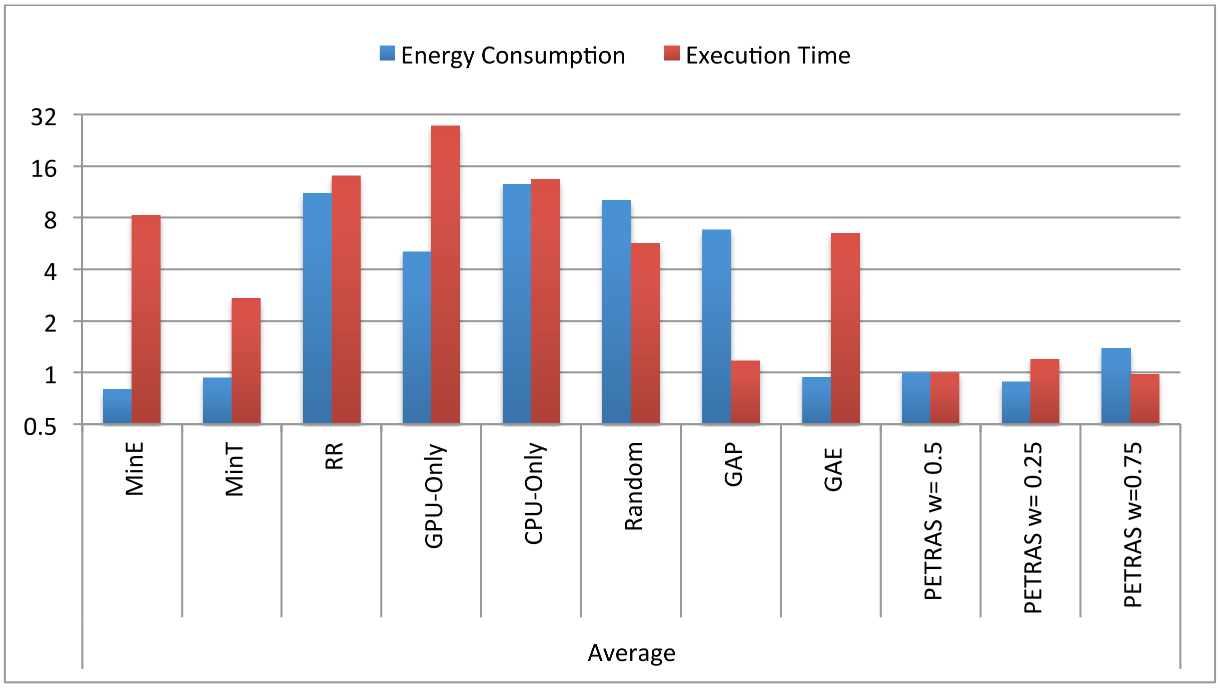


Figure 3.8: Comparison of PETRAS to the other scheduling algorithms in terms of average energy consumption and execution time normalized to PETRAS *w*=0.5 with no constraints

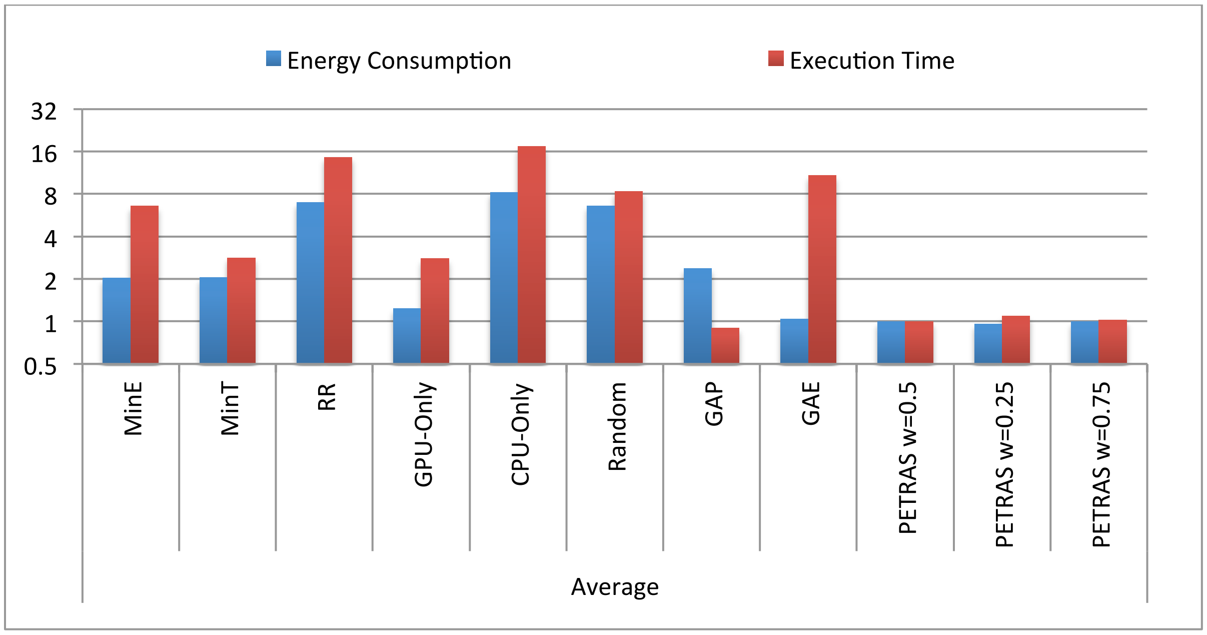


Figure 3.9: Comparison of PETRAS to the other scheduling algorithms in terms of average energy consumption and execution time normalized to PETRAS *w*=0.5 with peak power constraint only

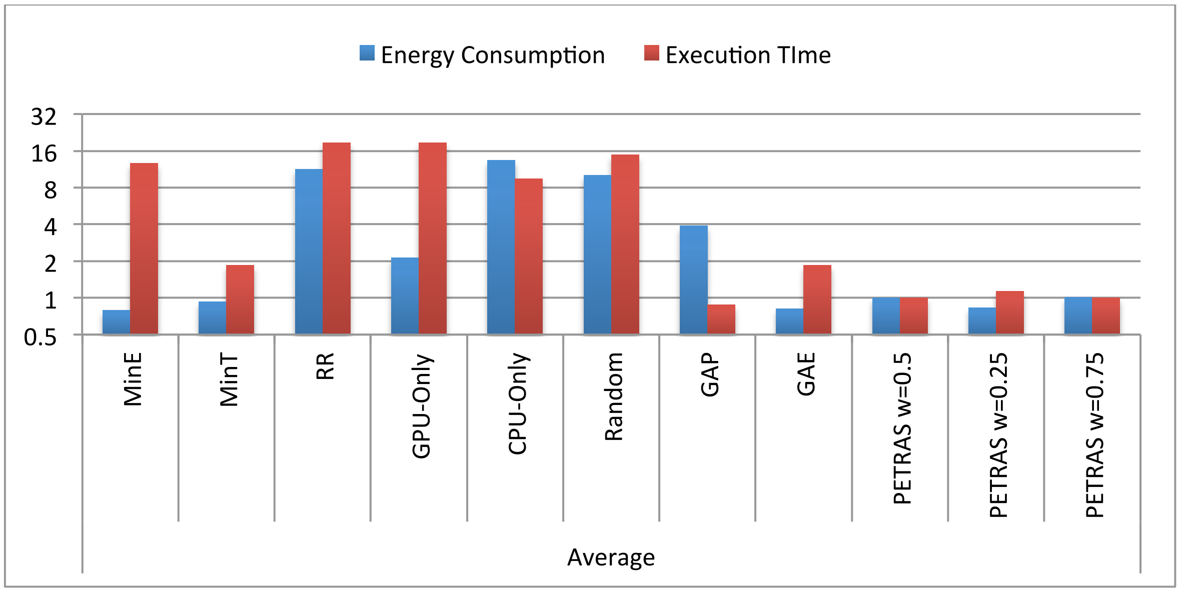


Figure 3.10: Comparison of PETRAS to the other scheduling algorithms in terms of average energy consumption and execution time normalized to PETRAS *w*=0.5 with peak power and peak CPU temperature constraints

On a logarithmic scale, Fig. 3.8, Fig. 3.9, and Fig. 3.10 follow the same trend. They show the average energy consumption and average execution time of the Rodinia benchmark applications using different scheduling algorithms normalized to the PETRAS with *w*=0.5. Although MinE, GAE, and PETRAS with *w*=0.25 have less average energy consumption, PETRAS with *w*=0.5 beats the rest of the schedulers if we consider both energy consumption and execution time.

Since the MinE and GAE schedulers focus on minimizing energy consumption, their output schedules have a very high execution time compared to PETRAS with all *w* values. Compared to PETRA with *w*=0.5, the MinT scheduler could not reach a nearly optimal solution in terms of performance because it is a greedy algorithm that finds the shortest execution time of each job at a time without considering whole schedule length.

The RR and the random scheduler perform poorly, which is expected due to their randomness. The poor results of the GPU-only and CPU-only schedulers demonstrate the need of heterogeneous processing units to run an application.

As for GAP, its objective function is to minimize execution time, but PETRAS with *w*=0.5 and *w*=0.75 outperform it in terms of average performance. We ran all the GA algorithms for the same number of iterations, but GAP may find the nearly optimal execution time if we had given it more time, but that schedule would have a very high energy consumption. From Fig. 3.9 and Fig. 3.10, it can be concluded that PETRAS works well in finding a nearly optimal schedule with peak power and peak CPU temperature constraints.

Fig. 3.11 shows detailed Rodinia applications results of GAP and GAE normalized to PETRAS with *w*=0.5. It can be concluded that PETRAS outperforms both GAP and GAE if we consider both execution time and energy consumption.

Table 3.2 illustrates a sample output of PETRAS with *w*=0.5 applied on 250 randomly generated jobs for the two cases: 1) no constraints and 2) both peak power and peak temperature constraints. The execution time and energy consumption of case 2 are higher than that of case 1 due to peak power and peak CPU temperature constraints. Thus, jobs are mapped to slower processing units that consume higher energy. Moreover, these constraints force the resources that violate them to be turned off. Therefore, PETRAS avoids selecting these resources.

Compared to all other schedulers, on average, PETRAS schedules can achieve up to 8.7x speedup and an energy saving of up to 627%. If we do not consider schedulers that perform badly (RR, CPU-only, GPU-only, and random schedulers), PETRAS schedules can achieve up to 4.7x speedup and an energy saving of up to 195%.

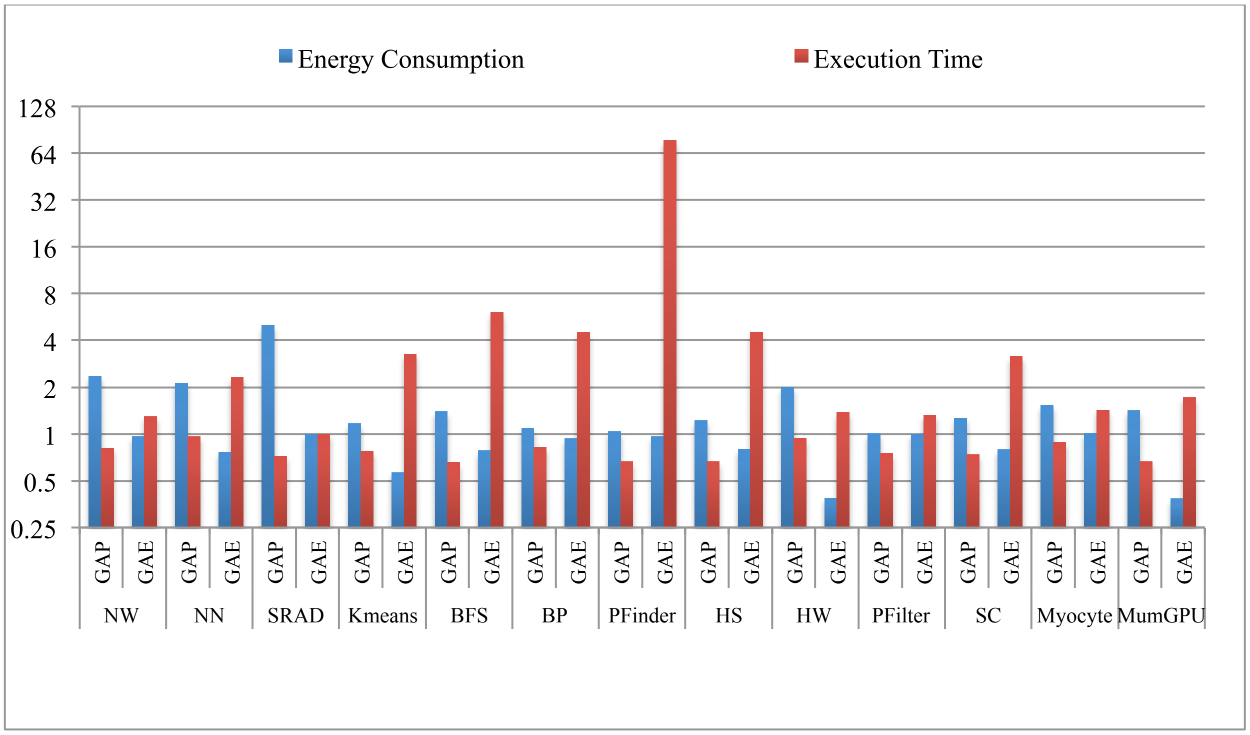


Figure 3.11: GAP and GAE average energy consumption and execution time normalized to PETRAS *w*=0.5

Table 3.2: Sample Results of PETRAS *w*=0.5

Selected Benchmark	No constraints				Peak Power and Peak Temperature Constraints			
Overall	Energy (KJ)	Execution Time (s)	PP (Watt)	PT (°C)	Energy (KJ)	Execution Time (s)	PP (Watt)	PT (°C)
NW	642.59	775.72	345.4	65.5	2790	15195	323.5	60.5
NN	142.20	111.016	360.5	64.5	192.51	567.694	323.05	60
CFD	582.3	1542.05	422.6	88.5	4415.4	3232.17	370.5	83
LUD	117.77	337.774	337.1	58	118.28	338.111	317.8	58
SRAD	178.16	517.763	399.95	56.5	928.05	837.329	354.75	67
Kmeans	220.10	112.106	367.9	70.5	619.62	2229.89	330.15	63
BFS	822.75	796.197	345.7	64.5	1036.9	5932.56	319.1	63.5
BP	5.5441	2.31291	343.55	49	5.6409	3.23245	327.15	49
Pathfinder	392.61	270.996	340.6	64	394.57	449.486	325.95	63
leukocyte	130.95	452.638	376.7	61	5304.6	15140.3	356.6	84
hotspot	1840.0	1523.55	373.6	78.5	2888.5	15806.6	349.4	68.5
heartwall	445968	392.552	394.6	85.5	1235400	2839.11	343.25	80.5
lavaMD	123.12	335.666	462.7	63	6428.2	12530.6	408.2	89
Particlefilter	2568.0	1148.37	342.7	61.5	2568.3	1139.39	331.65	61.5
StreamCluster	413.33	276.352	366.25	71	1524.9	5022.51	328.25	64
myocyte	1.5528	1.03764	351.15	48	1.9079	2.80564	316.05	48
B+tree	9000	1.163	315.2	61	9000	3.514	312	57
mummergpu	0.83	5905	326	67	1.36	4689	319	67

3.10.7 Power Management Unit Effect

The GA nearly optimal schedule goes through the power management unit that is responsible for turning off the idle or low-utilized processing units. Table 3.3 shows the effect of adjusting the processing unit utilization threshold *U* on the average energy saving and the percentage of processing units that are turned off. The goal is to save energy consumption by turning the processing units off or freeing these processing units for use by other applications. Note that some of the schedules remain the same since their processing units are filled up by jobs with utilization greater than the utilization threshold. As the utilization threshold *U* increases, more processing units are turned off and the energy consumption is reduced. On average, the power management unit helps to reduce the overall energy consumption up to 6.5% and free up to 24.2% of low-utilized processing units.

Table 3.3: Power Management Unit effect

Processing unit utilization threshold U	Energy consumption reduction	Turned off processing units
5%	-1%	16.6%
7.5%	-3%	20%
10%	-6.5%	24.2%

3.11 Summary

We have presented the Performance, Energy and Thermal aware Resource Allocator and Scheduler (PETRAS) utilizing a Genetic Algorithm (GA) to find efficient schedules in a heterogeneous system. PETRAS can be applied on any heterogeneous system where processing units are fused on the same chip or connected through a bus. The proposed scheduler is not a classic scheduler, it combines the following problems into one scheduler: job mapping and scheduling, core scaling, and thread allocation. PETRAS shows its capability to find efficient solutions in terms of both execution time and energy consumption under peak power and peak CPU temperature constraints. Therefore, PETRAS can be used for embedded system applications. To test PETRAS, we have implemented PETRAS on an actual system equipped with a multi-core CPU and a GPU. We have demonstrated that the PETRAS scheduler outperforms performance-based schedulers and other schedulers in both execution time and energy consumption. We believe that we have to consider all problems (e.g., job mapping and scheduling, core scaling, thread allocation) into one scheduling optimization problem. Moreover, schedules should be selected based on both execution time and energy consumption.

Chapter 4

Workload Distribution and Resource Allocation for CPU-GPU Heterogeneous Systems

As uni-core processors have hit the power wall, multicore processors have emerged in the computing industry. Another industry shift occurred when NVIDIA [2][37] developed the Compute Unified Device Architecture (CUDA), which enables the use of Graphic Processing Units (GPUs) as general-purpose processors (GPGPUs). Unlike multicore CPUs that have a few large cores, GPUs have several hundreds of tiny cores providing massively parallel computing. Therefore, GPUs are known to be high throughput and energy efficient processors. Although GPUs are more energy efficient than CPUs, they produce very high peak power compared to CPUs. To overcome the performance and energy limitations of conventional systems, most systems, from mobiles to servers, have become heterogeneous systems. These systems combine traditional processers (CPUs) and accelerators such as GPUs, processing units with different performance/energy consumption characteristics, into the same machine. However, these systems use a GPU to handle the parallel parts of a job and a CPU (i.e., host) to handle the sequential parts of a job. The CPU also handles CPU/GPU data transfer coordination. After the data transfer, the host remains idle until a GPU finishes its part of a job and transfers data back. If the data of an application is large, most of the CPU time is wasted busy waiting for a GPU to finish execution and data transfer. This classic workload distribution does not fully utilize the CPU and the GPU. Thus, there is a need for a cooperate CPU-GPU heterogeneous computing framework that efficiently distributes work to fully exploit the potential of the CPU and the GPU.

Due to the performance and energy consumption differences between CPUs and GPUs, cooperative CPU-GPU heterogeneous computing may deliver high performance and energy efficiency [1]. Instead of mapping the entire job to run on either a CPU or a GPU, a job is distributed to be executed on both a CPU and a GPU, where part of a job is mapped to a CPU and the remaining part is mapped to a GPU. Finding the best CPU/GPU workload map ratio of a job to fully exploit the system's potential is difficult because of the performance/energy tradeoff. Moreover, the number of cores and threads on the CPU highly affects the performance and energy consumption of the part that run on the CPU. This in turn affects overall system performance and energy consumption.

System's performance was the dominant factor in computer design for years, but as the industry hits the power wall, energy consumption also becomes important. Peak power and peak CPU temperature are also important factors especially in the embedded systems field. Thus, we believe we have to consider all of these factors in systems' design.

In this work that is appeared in [34], we introduce a runtime Workload Distributor with a Resource Allocator (WDRA) that finds the best CPU/GPU map ratio, number of cores, and number of threads in terms of both performance and energy consumption under peak power and peak CPU temperature constraints. WDRA solves a multi-objective optimization problem that combines job workload distribution and resource allocation. The general problems of resource allocation and workload distribution are known to be NP-Hard [15]. Therefore, WDRA utilizes an evolutionary algorithm called Particle Swarm Optimizer (PSO) [16] to find an approximate solution of the WDRA problem. PSO algorithms have shown its capability to solve such problems.

There have been several studies conducted on workload distribution to exploit parallelism in CPU and GPU heterogeneous systems. However, WDRA is the first attempt to combine a workload distributor and a resource allocator. To the best of our knowledge, none of the current workload distributors map work in terms of both performance and energy consumption under peak power and peak CPU temperature constraints. In addition, WDRA mapping is adaptive to the runtime changes of the application, input sizes, and underlying hardware/software settings because they highly affect the overall performance, energy consumption, peak power and peak CPU temperature. Our results show that WDRA achieves up to 1.47x speedup and 82% reduction in energy consumption compared to other distributors on average.

The rest of the chapter is organized as follows. In Section 4.1, related work is discussed. Section 4.2 presents the motivation of the WDRA. Sections 4.3, 4.4 and 4.5 describe WDRA problem in detail. Section 4.6 presents the profiler and the curve fitter. Section 4.7 describes the PSO algorithm. WDRA flowchart is appeared in section 4.8. Section 4.9 evaluates WDRA on a CPU-GPU heterogeneous system. Finally, we summarize this chapter in Section 4.10.

4.1 Related Work

Several studies [13, 20, 25] have been conducted on workload distribution to enhance overall performance. Luk et al. [13] have proposed the Qilin adaptive mapping to map computations to processing elements on a CPU-GPU machine. Qilin framework maps computations to CPUs and GPUs based on an empirical performance model produced by profiling and curve fitting. Although our WDRA profiler and curve fitter are inspired by their profiling and curve fitting techniques for estimation purposes, any other estimation models or tools can be used instead. Qilin work distribution is based on execution time only. Lee et al. [20] proposed a cooperative heterogeneous computing framework that efficiently utilizes the idle host CPU cores for CUDA kernels, which are supposed to execute only on GPUs. Its workload distribution is also based on predicted execution time. To improve the performance, Lee et al. [25] developed a framework that coordinates collaborative execution of a single data-parallel kernel in CPU-GPU heterogeneous systems. All of these studies were focusing on performance improvement without considering energy consumption or other design factors (e.g., peak power, peak CPU temperature).

Some studies [21, 22, 23] have considered energy consumption or peak power for workload distribution in CPU-GPU heterogeneous systems. Ge et al. [22] studied the performance and energy impact of workload distribution in order to optimize either the system performance or energy consumption. Wang et al. [21] presented a power-efficient work distribution technique that coordinates work distribution and per-processor's frequency scaling to optimize energy consumption under performance constraint. On the other hand, Komoda et al. [23] proposed a power capping technique that orchestrates Dynamic Voltage Frequency Scaling (DVFS) and workload distribution without violating peak power budget. None of these studies considered both performance and energy consumption; they optimized one parameter and constrained the other.

Some research [11, 12] has investigated resource allocation such as core scaling and thread allocation. Liu et al. [11] proposed DVFS with core scaling which selects number of active cores and per-core voltage/frequency levels in order to minimize the overall system power under performance constraints. In [12], Vega et al. presented preliminary characterization data for multi-threaded programs to estimate the potential benefit of power-aware thread placement. These studies showed the effect of core scaling and thread allocation on energy consumption.

Alsubaihi et al. [4, 19] illustrated the importance of combining resource allocation (e.g., core scaling and thread allocation) with the scheduling problem. The goal of that scheduler is to find an efficient job schedule, number of cores and number of threads while considering both performance and energy consumption under peak power and peak CPU temperature constraints. It maps an entire job to either a GPU or a CPU and does not distribute the work of a job between the CPU and the GPU for collaborative execution.

WDRA differs from prior work in that it combines workload distribution and resource allocation problems. For a given job, the goal is to find an efficient CPU/GPU map ratio, number of cores, and number of threads needed while considering both the overall system performance, energy consumption under the peak power and peak CPU temperature constraints. Previous studies have only considered overall system either performance or energy consumption. Moreover, WDRA is adaptive to changes in job’s input size and underlying hardware during runtime.

4.2 Motivation

Our results from preliminary research, as described below motivated this study on developing an efficient workload distributor that takes into account both performance and energy consumption under peak power and peak CPU temperature constraints. We ran the Rodinia 3.0 benchmark suite [10][32] jobs with sizes of 1k up to 64G on a typical CPU-GPU heterogeneous system. We measured the execution time, energy consumption, peak power, and peak CPU temperature by running Rodinia jobs with (1) different CPU/GPU map ratios on a heterogeneous system consisting of a multicore CPU (4 cores) and a GPU (2) different number of cores, and (3) different number of threads (details of the hardware configuration are given in Section 2.4). Our research tackles three problems: (a) CPU/GPU workload distribution, (b) core scaling, and (c) thread allocation.

The results below demonstrate the three problems with different settings. These results illustrate that we do not have to solve each problem independently since the solution of one problem affects the others. We have to solve the three problems as one optimization problem; therefore, we combined CPU/GPU workload distribution, core scaling and thread allocation problems into one distributor. The goal of that distributor is to find the efficient CPU/GPU map ratio, number of cores and threads in terms of both performance and energy consumption under peak power and peak CPU temperature constraints. Moreover, the results show that static workload distribution, core scaling and thread allocation techniques would not be sufficient. What needed is a dynamic distributor that can automatically adapt to the job and the runtime environment. Thus, we propose a runtime Workload Distributor with Resource Allocator (WDRA).

4.2.1 CPU/GPU Map Ratio

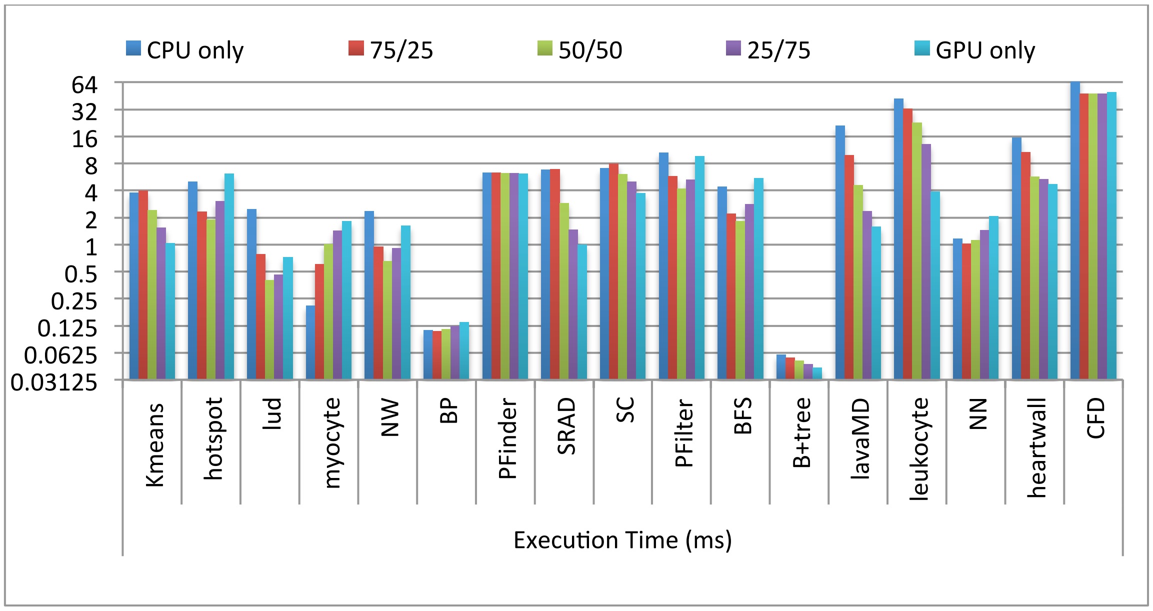
In the following experiments, we varied the CPU/GPU map ratio to show the impact of different workload distributions on the following measured parameters: performance, energy consumption, peak power and peak CPU temperature. The *y*-axis represents each of the measured parameters. The *x*-axis is the distribution of work between the CPU and GPU, where the notation *"X/Y"* means *X%* of work is mapped to the CPU and *Y%* of work is mapped to the GPU. There are two extreme cases where all the work is entirely mapped to either the GPU or the CPU. For illustration, we show the results of the CPU/GPU map ratio with a granularity of 25. Since the selected maps have this granularity not all values are shown. Therefore, there maybe a map in-between these values that is better in terms of the measured parameter.

On a logarithmic scale, Fig. 4.1(a) shows that different CPU/GPU workload distributions for running a job with a size of 4k results in different values of execution time. Some of these jobs run faster if we map all the work to the GPU such as kmeans, SRAD, Streamcluster, B+tree, LavaMD, Leukocyte, and heartwall. For other jobs such as myocyte and backprob, CPU only is better. For the rest of the jobs, the execution time is shorter if the work is distributed across the CPU and GPU. This experiment was performed on a quad-core CPU. It should be noted that as the number of the cores changes, the execution time varies.

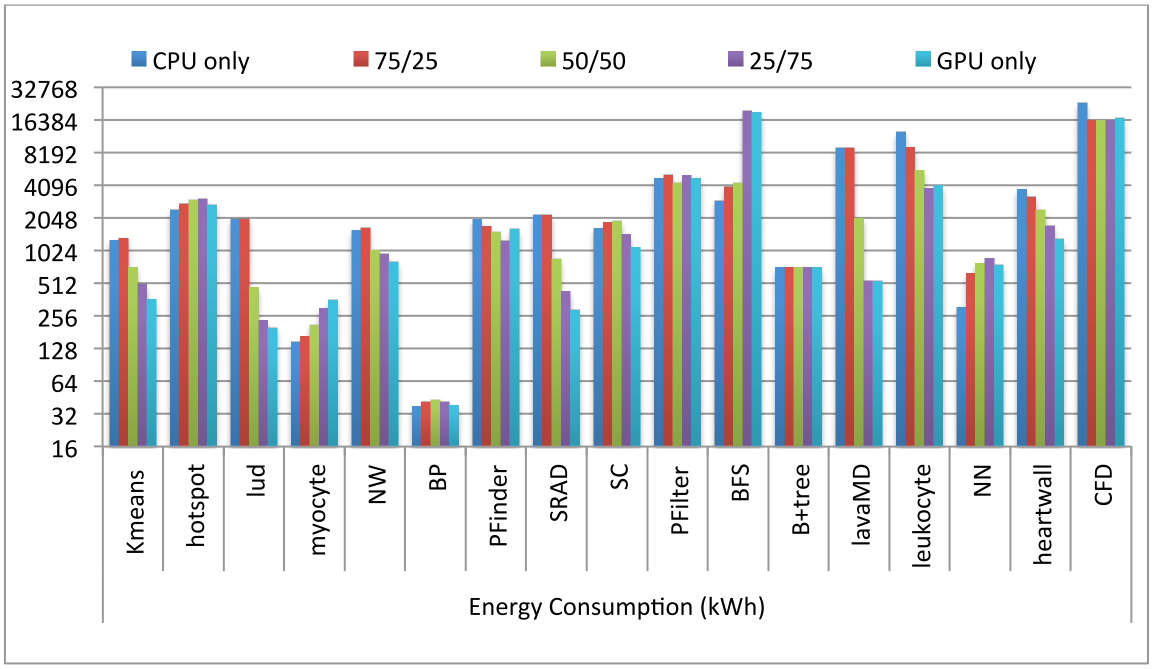
On a logarithmic scale, Fig. 4.1(b) illustrates that energy consumption differs if CPU/GPU map ratios change. Even though GPUs are known to be more energy efficient than multi-core CPUs, multi-core CPUs outperform GPUs in energy consumption as shown for hotspot, myocyte, BP, BFS and NN jobs. From Fig. 4.1(a) and Fig. 4.1(b), it can be concluded that both performance and energy consumption should be considered when deciding the CPU/GPU map ratio. For example, if we only consider energy consumption for mapping NW, then the entire job should be mapped to the GPU. But if we also consider performance, a 50/50 workload map ratio may be the best mapping.

Fig. 4.1(c) and Fig. 4.1(d) show the peak power and peak CPU temperature reached by running a job with a size of 4k with different CPU/GPU map ratios. The graphs show that changing CPU/GPU map ratios change the peak power and peak CPU temperature values. Hence, peak power and peak CPU temperature should be examined when selecting the CPU/GPU map ratio.

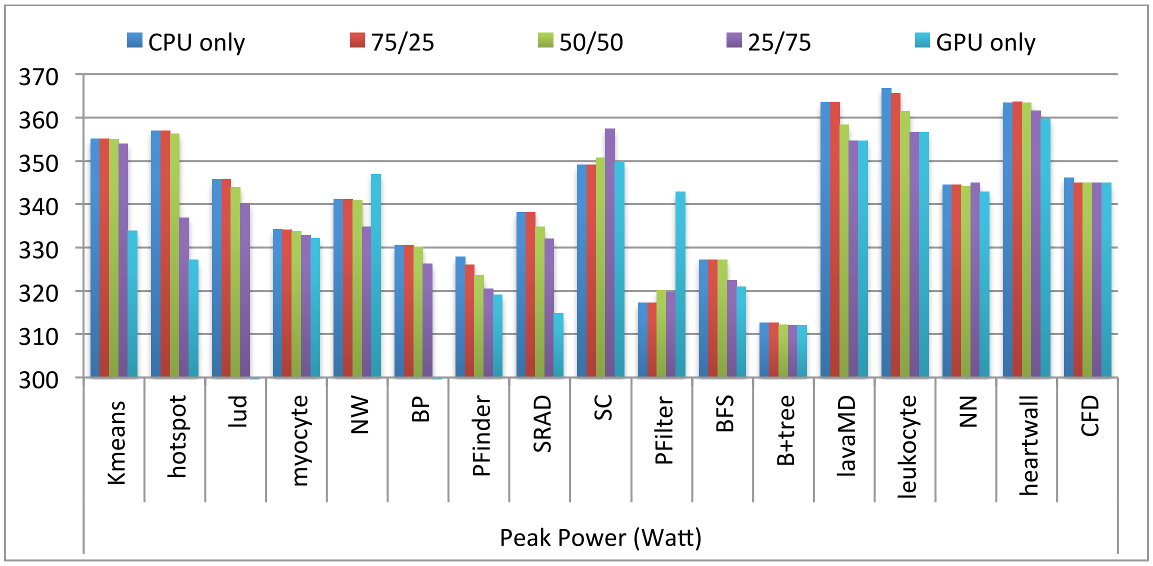
Note that the presented results show an instance of a job of a fixed size. These results change as the job size varies. If a CPU/GPU map ratio *x* is the best for a job of a specific size, changing the size of this job might mean that *x* would no longer the best for it. Hence, the CPU/GPU map ratio highly depends on the input problem size and the underlying hardware. Therefore, it can be concluded that choosing the right CPU/GPU map ratio to enhance performance or minimize energy is difficult. It becomes more complicated if peak power and peak CPU temperature constraints are considered.


CPU only
75/25
50/50
25/75
GPU only
64
32
16
8
4
2
1
0.5
0.25
0.125
0.0625
0.03125
Kmeans
hotspot
lud
myocyte
NW
BP
PFinder
SRAD
SC
PFilter
BFS
B+tree
lavaMD
leukocyte
NN
heartwall
CFD
Execution Time (ms)

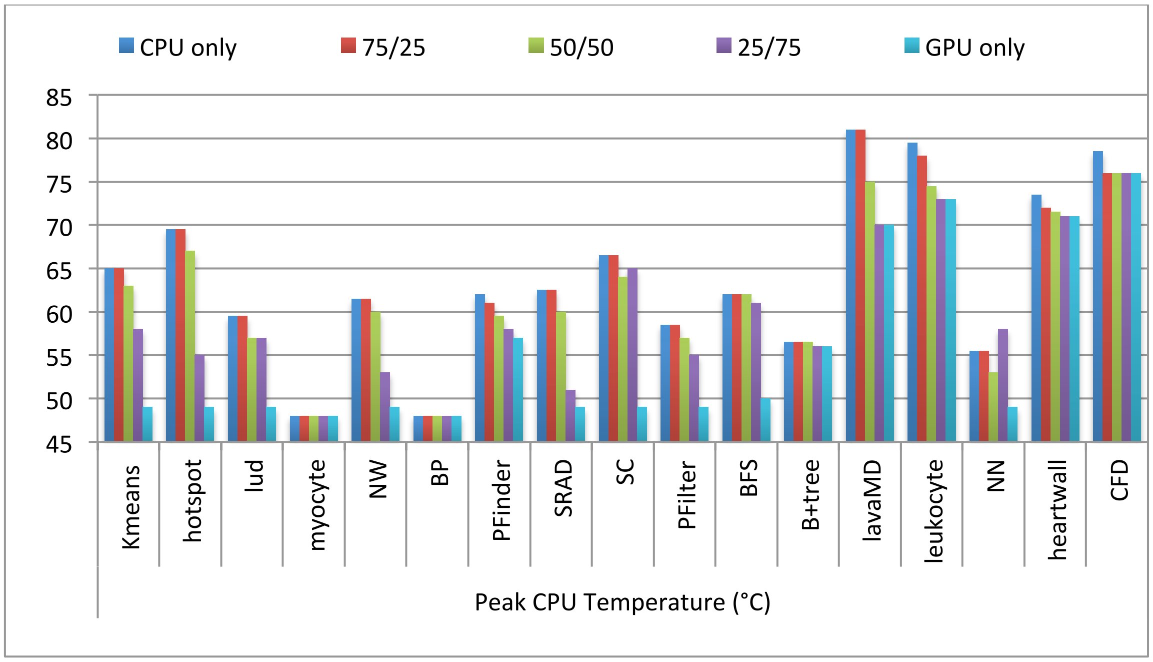
(a)


CPU only
75/25
50/50
25/75
GPU only
32768
16384
8192
4096
2048
1024
512
256
128
64
32
16
Kmeans
hotspot
lud
myocyte
NW
BP
PFinder
SRAD
SC
PFilter
BFS
B+tree
lavaMD
leukocyte
NN
heartwall
CFD
Energy Consumption (kWh)

(b)



(c)



(d)

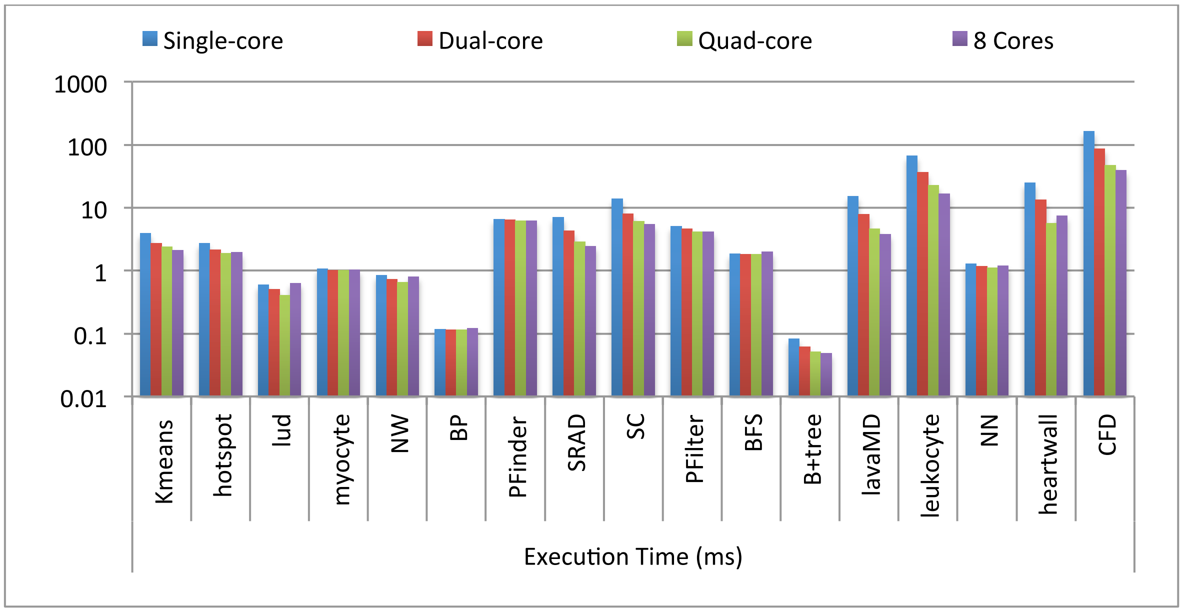
Figure 4.1: Execution time (a), energy consumption (b), peak power (c), and peak CPU temperature (d) of running a job with different CPU/GPU map ratio

4.2.2 Number of Cores

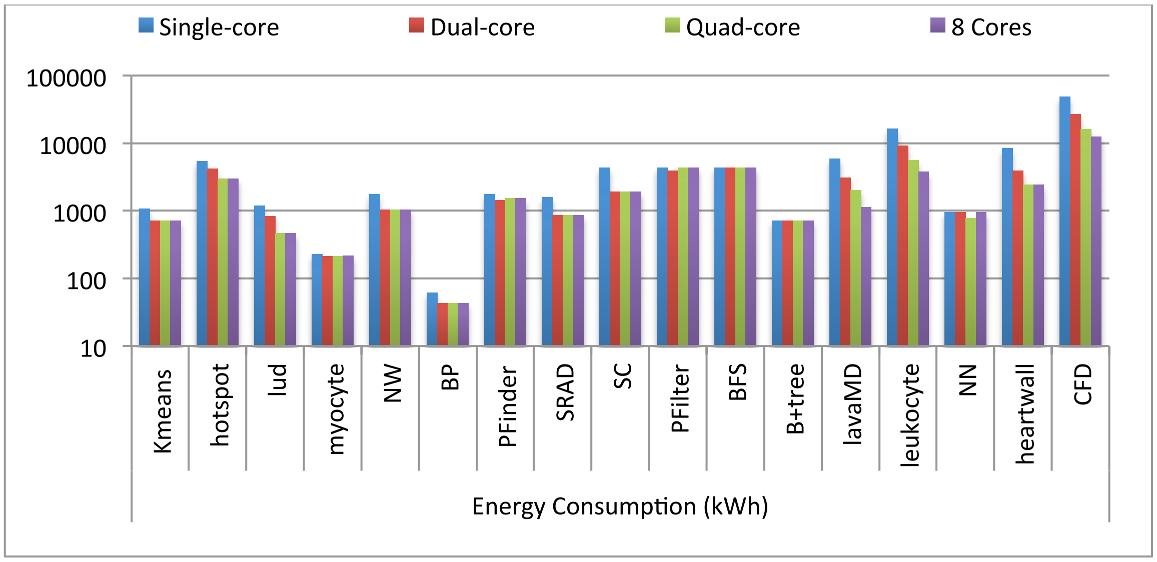
We ran Rodinia benchmark jobs with fixed sizes of 4k and fixed CPU/GPU map ratio (i.e., 50/50) to examine the effect of varying number of cores on the following measured parameters: execution time, energy consumption, peak power and peak CPU temperature.

On a logarithmic scale, Fig. 4.2(a) and Fig. 4.2(b) show execution time and energy consumption respectively for various number of cores. On the other hand, Fig. 4.2(c) and Fig. 4.2(d) show the measured peak power and peak CPU temperature. It can be concluded that changing number of cores affects the measured parameters. Moreover, there is no specific number of cores that optimizes the measured parameter for all of the jobs.

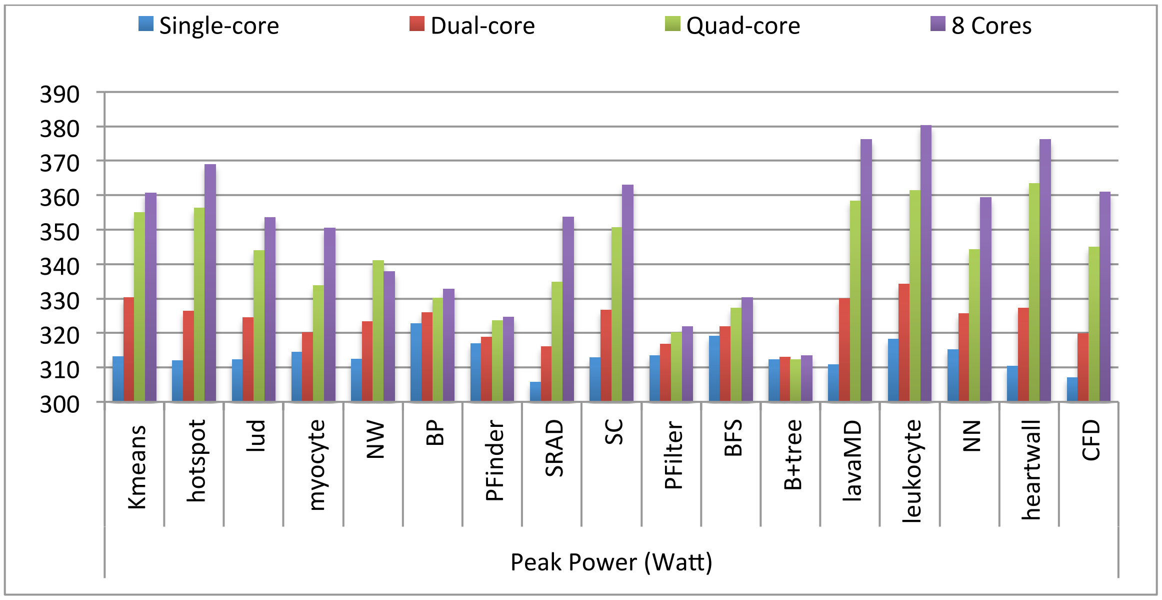
Not only that, if the decision on number of cores is made by only evaluating one parameter, it may result in higher values of the rest of the parameters. In heartwall, for example, if we are looking to execution time and energy consumption, 8-core CPU is the best in that case. But running heartwall on 8-core CPU produces higher peak power and peak CPU temperature compared to the rest of the configurations. Hence, number of cores should be carefully selected and evaluated according to all of the measured parameters.


Single-core
Dual-core
Quad-core
8 Cores
1000
100
10
1
0.1
0.01
Kmeans
hotspot
lud
myocyte
NW
BP
PFinder
SRAD
SC
PFilter
BFS
B+tree
lavaMD
leukocyte
NN
heartwall
CFD
Execution Time (ms)

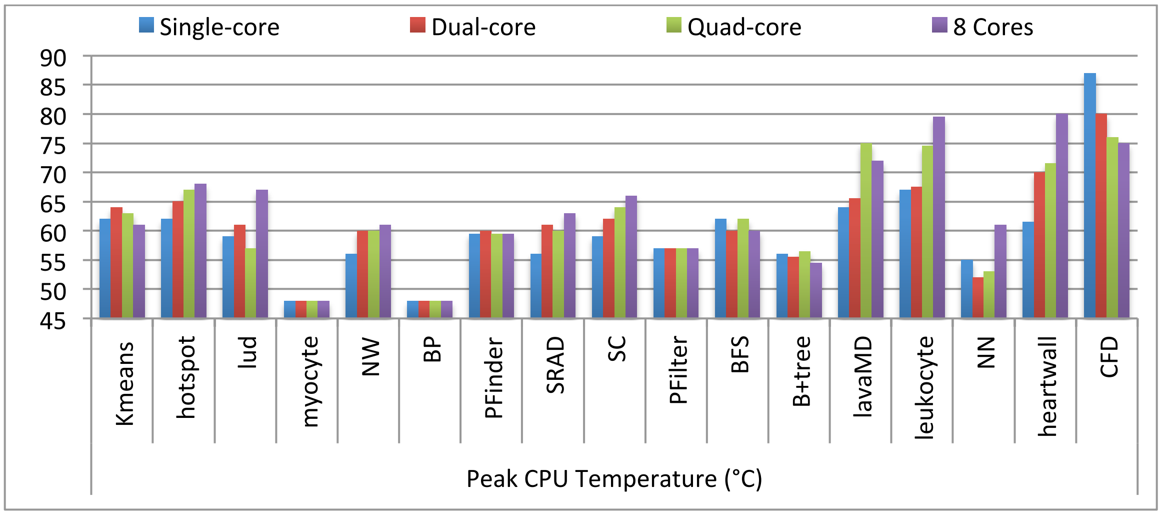
(a)


Single-core
Dual-core
Quad-core
8 Cores
100000
10000
1000
100
10
Kmeans
hotspot
lud
myocyte
NW
BP
PFinder
SRAD
SC
PFilter
BFS
B+tree
lavaMD
leukocyte
NN
heartwall
CFD
Energy Consumption (kWh)

(b)



(c)



(d)

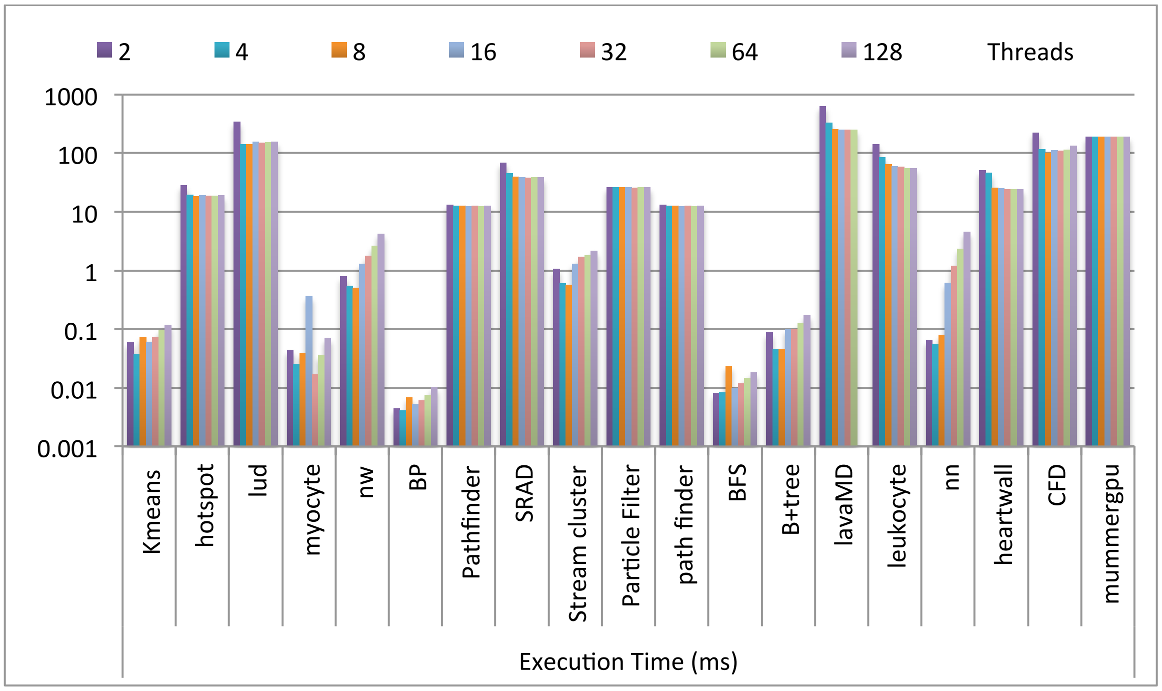
Figure 4.2: Execution time (a), energy consumption (b), peak power (c), and peak CPU temperature (d) of running a job with different number of cores

4.2.3 Number of Threads

We ran jobs of the Rodinia benchmark with fixed sizes of 4k on a quad-core CPU and a fixed CPU/GPU map ratio (i.e., 50/50) with various numbers of allocated threads.

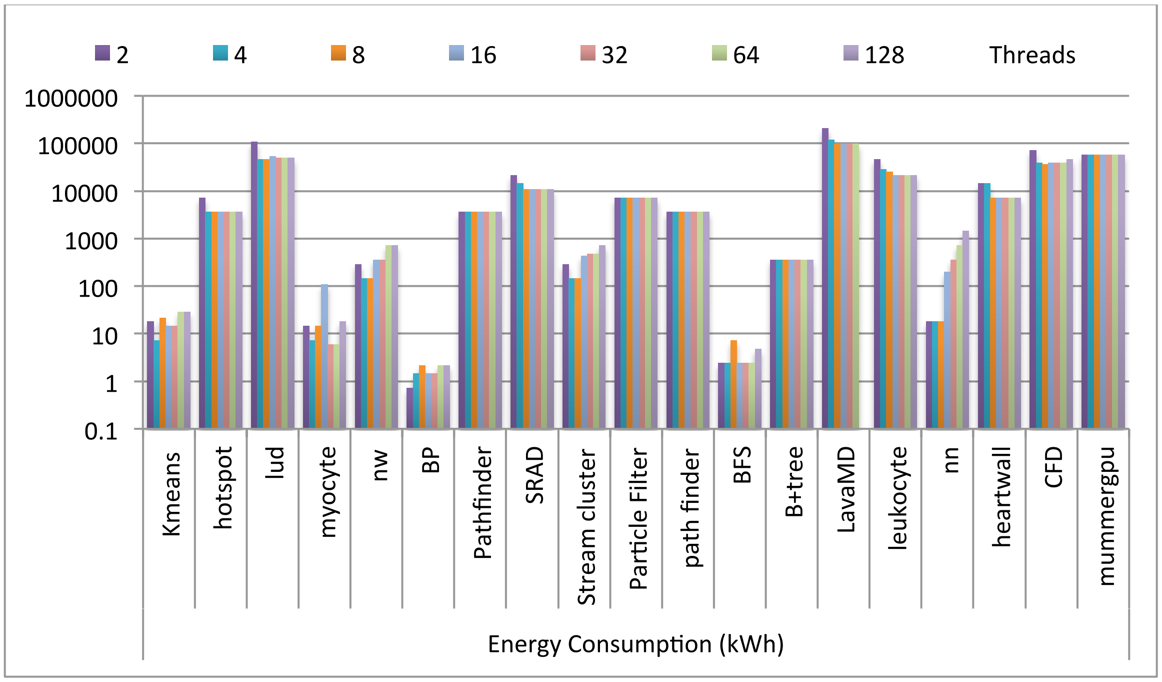
On a logarithmic scale, Fig. 4.3(a) and Fig. 4.3(b) show how changing the number of allocated threads changes the execution time and the energy consumption of a specific job. Results of peak power and peak CPU temperature of a different number of allocated threads show the same trend as shown in Fig. 4.3(c) and Fig. 4.3(d). It can be concluded that there is no specific number of threads that is optimal in terms of the measured parameter for all of the jobs.

From these results, it is clear that deciding on the number of threads has high impact on performance, energy consumption, peak power and peak CPU temperature. Moreover, these results were obtained by running a job of a specific size on a quad-core CPU with different thread number settings. The results change depending on the job size as well as on whether that job runs on a single-core, dual-core or quad-core CPU, etc. In addition, running these jobs with different CPU/GPU map ratios gives different values of the measured parameters.



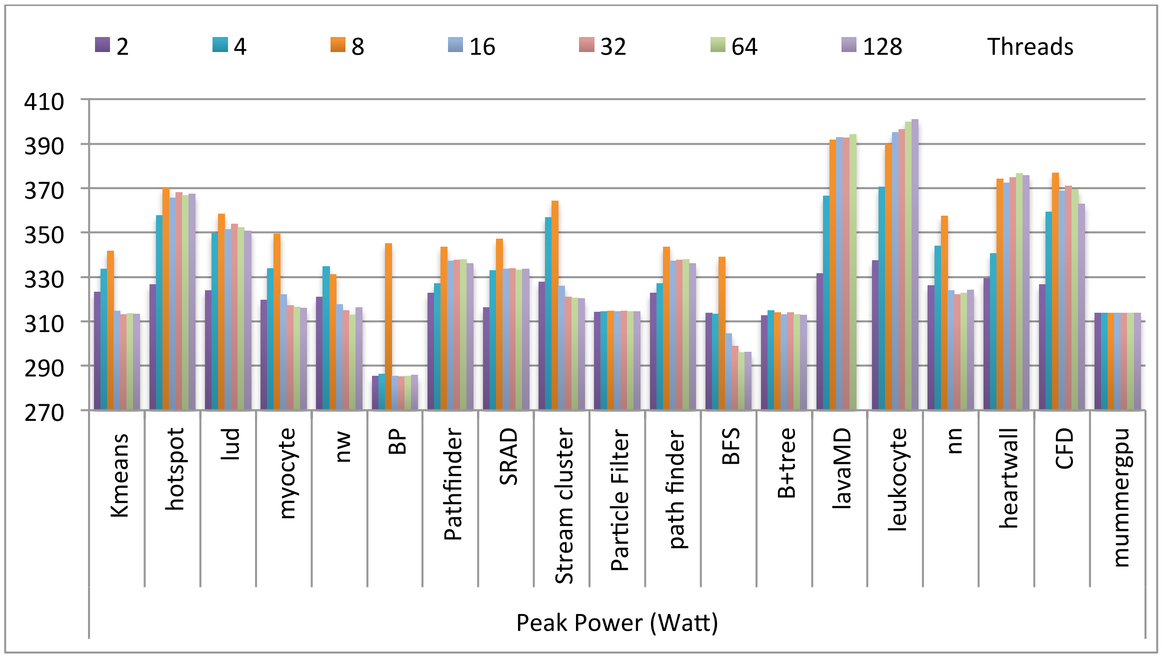
2
4
8
16
32
64
128
Threads
1000
100
10
1
0.1
0.01
0.001
Kmeans
hotspot
lud
myocyte
nw
BP
Pathfinder
SRAD
Stream cluster
Particle Filter
path finder
BFS
B+tree
lavaMD
leukocyte
nn
heartwall
CFD
mummergpu
Execution Time (ms)

(a)

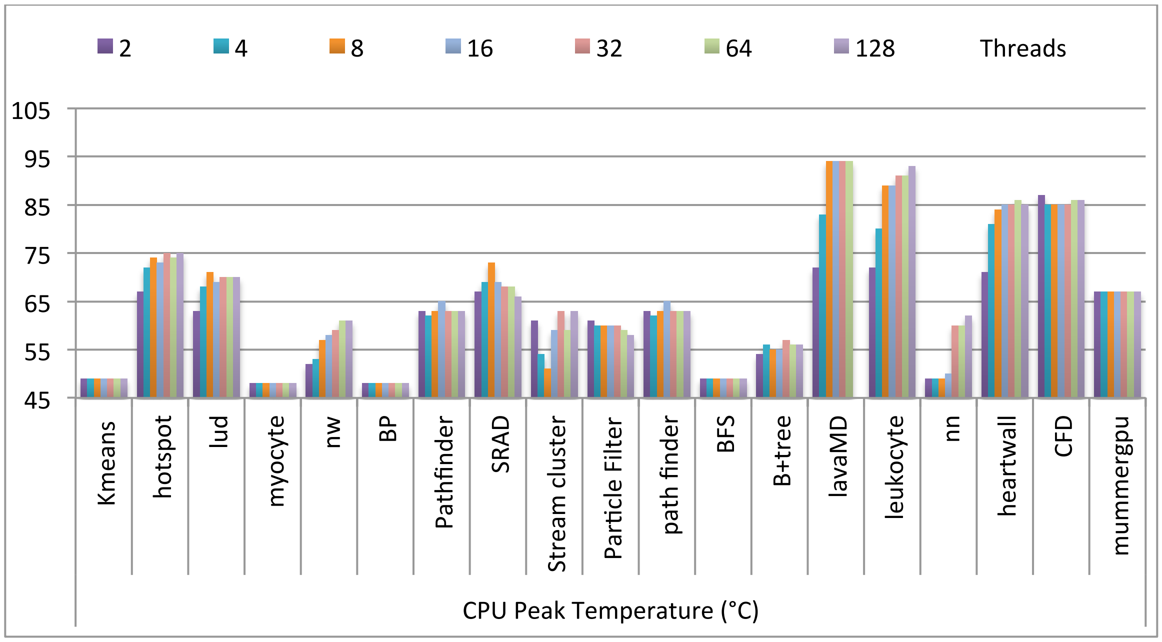


2
4
8
16
32
64
128
Threads
1000000
100000
10000
1000
100
10
1
0.1
Kmeans
hotspot
lud
myocyte
nw
BP
Pathfinder
SRAD
Stream cluster
Particle Filter
path finder
BFS
B+tree
LavaMD
leukocyte
nn
heartwall
CFD
mummergpu
Energy Consumption (kWh)

(b)



(c)



(d)

Figure 4.3: Execution time (a), energy consumption (b), peak power (c), and peak CPU temperature (d) of running a job with different number of threads

4.3 Workload Distributor with Resource Allocator (WDRA)

Unlike other workload distributors, WDRA is not just a workload distributor; it is also a resource allocator for a CPU-GPU heterogeneous system that considers the following questions during the run-time when a job arrives: (a) what is the efficient CPU/GPU map ratio to run a job? (b) how many cores are needed to run a job on a CPU? (c) how many threads are needed to run a job?. WDRA's goal is to find efficient solutions to the three questions above while considering efficiency in both overall execution time and overall energy consumption without violating peak power and peak CPU temperature constraints. Hence, WDRA solves a multi-objective optimization problem. But due to the performance and energy consumption trade-off, there is no single solution that can optimize them simultaneously. Therefore, WDRA aims to find a Pareto optimal solution: a solution in which an objective value cannot be improved (e.g., performance) without degrading another objective value (e.g., energy consumption).

WDRA is not system-specific; it can be applied on any heterogeneous system from embedded systems (e.g., mobiles) to servers because it takes into account peak power and peak CPU temperature.

To estimate a job's execution time, energy consumption, peak power and peak CPU temperature, WDRA utilizes estimation models provided by a system profiler and a curve fitter that were developed by Luk et al. [13]. Any other estimation models or tools could be used instead of the profiler and the curve fitter. However, we chose to use the profiler presented in [13] that have proved its effectiveness in parameter's prediction during runtime.

WDRA does not only solve a classic workload distribution problem. In addition, it has a resource allocator that solves core scaling and thread allocation problems at the same time. To the best of our knowledge, WDRA is the first attempt to combine a workload distributor and a resource allocator into one optimization problem that takes into account both performance and energy consumption under peak power and peak CPU temperature limits.

4.4 WDRA Problems

WDRA is not a classic job workload distributor. It has a resource allocator that determines the number of cores and threads needed if the CPU is the processing unit. Hence, it combines core scaling and thread allocation into the job workload distribution optimization problem. The WDRA solves the following problems:

4.4.1 Workload distribution

Our goal is to find efficient jobs' CPU/GPU map ratios in terms of both performance and energy consumption under peak power and peak CPU temperature constraints. Given a job and a CPU-GPU heterogeneous system, the problem is to find the best job distribution of work between the CPU and the GPU; it is to decide the percentage of job work mapped to the CPU and the percentage of job work mapped to the GPU. There is a possibility that the best CPU/GPU map ratio is to map the entire job to either a CPU or a GPU (e.g., in an extreme case).

4.4.2 Core Scaling

The problem is to find the best number of cores needed to run a job in terms of both performance and energy consumption under the peak power and peak CPU temperature constraints. As concluded in section 4.2.2, the number of cores highly affects the parameters above. Hence, number of cores should be carefully selected. For example, if we allocate a fewer cores than needed, the performance degrades due to the serial execution of the job. On the other hand, if we allocate a higher number of cores than needed, performance may also degrade due to cores communication latency or because jobs do not have enough parallelism to utilize all of the cores provided.

The goal of WDRA core scaling is to find an efficient number of cores needed by the percentage of work mapped to the CPU while considering both overall execution time and energy consumption under the peak power and peak CPU temperature constraints.

4.4.3 Thread Allocation

Jobs run on multicore CPUs can exploit their parallelism utilizing multithreading programming and an execution model. The cooperation of multiple threads to run a job helps to hide memory latency and enhance processor utilization. Having more than one thread running on a processor keeps that processor busy so if a thread is blocked waiting for resources, other threads can proceed.

As concluded in section 4.2.3, selecting the number of threads allocated to run a job is important due to its effect on execution time, energy consumption, peak power, and peak CPU temperature. Hence, the number of allocated threads should be carefully chosen. Allocating more threads than needed to run a job may degrade its performance and increase energy consumption due to high thread communication latency. On the other hand, allocating fewer threads than needed to achieve job's full parallelism may also have a negative impact on its performance and energy consumption. Hence, the goal of the WDRA thread allocator is to find an efficient number of threads needed to run a job while taking into account both performance and energy consumption under the peak power and peak CPU temperature constraints.

4.5 WDRA Optimization Problem Formulation

WDRA optimization problem is for a heterogeneous system that is equipped with a CPU and a GPU either fused on the same chip or connected through a network (e.g., PCI Express, a ring, or a mesh). A CPU can have different resources settings such as number of threads and number of cores. There are *n* compute intensive jobs $J=\{J_1, J_2,..., J_n\}$ with different workload sizes competing for system resources. Estimation models of the overall execution time, energy consumption, peak power and peak CPU temperature of a given job J_i on a CPU and a GPU are predicted by profiling and curve fitting. Thus, the communication cost of transferring data between the CPU and the GPU is included in the estimation models. The objective is to find an efficient job's CPU/GPU map ratio, decide the number of cores, set the number of threads such that taking into account the overall execution time and energy consumption while not violating peak power budget and peak CPU temperature limit. The WDRA problem is formalized as follows:

4.5.1 Input

The input consists of the following:

- A set of *n* compute intensive jobs $J=\{J_1, J_2,..., J_n\}$; where S_i is the input size of job J_i and $S_i =(1, S_{max}]$.
- Processing units PUs={CPU, GPU}.
- A set of number cores C={1, 2, 4,...}.
- A set of number of threads Trd={2, 4, 8, 16, 32,…}
- The profiler training set's input sizes $TS=\{TS_1, TS_2,..., TS_m\}$; where $TS= [1, S_{max}]$.

- S_{max}; Jobs' maximum size.
- PP_{max}; Peak Power budget.
- PT_{max}; Peak temperature limit.
- Execution time models *T(ts_i, CPU)* and *T(ts_i, GPU)* which determine the estimated execution time T_i of running a job J_i of a size ts_i on a CPU and a GPU respectively.
- Energy consumption models *E(ts_i, CPU)* and *E(ts_i, GPU)* which determine the estimated energy consumption E_i of running a job J_i of a size ts_i on a CPU and a GPU respectively.
- Peak power models *PP(ts_i, CPU)* and *PP(ts_i, GPU)* which determine the estimated peak power PP_i of running a job J_i of a size ts_i on a CPU and a GPU respectively.
- Peak CPU temperature models *PT(ts_i, CPU)* and *PT(ts_i, GPU)* which determine peak CPU temperature PT_i of running a job J_i of a size ts_i on a CPU and a GPU respectively.

4.5.2 Objective Function

WDRA is a multi objective optimization problem that considers both the total execution time as in (8) and total energy consumption as in (9) of a solution. Hence, a weighted fitness function is used to combine them as in (10).

$$f_1 = \min \sum_{\mathrm{i}} \max\left(\Omega_i\, T_{i\,CPU}, (1-\Omega_i)\, T_{i\,GPU}\right) \tag{8}$$

$$f_2 = \min \sum_{\mathrm{i}} \Omega_i\, E_{i\,CPU} + (1-\Omega_i) E_{i\,GPU} \tag{9}$$

$$;\forall \text{job } i = 1, n \text{ and } \Omega_i = [0,1]$$

;where Ω_{i} represents the CPU/GPU mapping ratio of a job *i* to a CPU and a GPU. If $\Omega_{\mathrm{i}} = 0$ the entire job *i* is mapped to the GPU and if $\Omega_{\mathrm{i}} = 1$ the entire job is mapped to the CPU. If the Ω_{i} value is between 0 and 1, Ω_{i} of a job *i* is mapped to the CPU and the rest of that job is mapped to the GPU. The parameters $T_{i\,CPU}$ and $T_{i\,GPU}$ represent the execution time of a job *i* running on the CPU and the GPU respectively. The parameters $\mathrm{E}_{\mathrm{i\,CPU}}$ and $\mathrm{E}_{\mathrm{i\,GPU}}$ represent the energy consumed by a job *i* running on the CPU and the GPU respectively. The overall objective function is as follows:

$$f_{total} = w f_1 + (1-w) f_2 \quad ; w = [0,1] \tag{10}$$

;where *f1* and *f2* are normalized. *w* is a predefined weight value that determines the importance of each of the objective functions. If *w*= 0.5, both objective functions are equally important. If *w*=1 or 0, (10) is a single objective function that optimizes execution time or energy consumption respectively. Moreover, if *w*>0.5, execution time is more important than energy consumption and if *w*<0.5, energy consumption is more important than execution time.

4.5.3 Constraints

The objective function is subject to the following constraints:

- The CPU/GPU map ratio should be [0, 1]

$$0 \leq \forall \Omega_{\mathrm{i}} \leq 1 \text{ ; } \forall \text{job } i = 1, n$$

- The peak power should be less than a peak power budget

$$\forall \Omega_{\mathrm{i}} PP_{i\,CPU} \text{ and } \forall (1 - \Omega_{\mathrm{i}}) PP_{i\,GPU} < PP_{max}$$

$$\text{; } \forall \text{job } i = 1, n$$

- The peak CPU temperature should be less than a peak CPU temperature limit

$$\forall \Omega_{\mathrm{i}} PT_{i\,CPU} < PT_{max} \text{ ; } \forall \text{job } i = 1, n$$

;where the parameters $PP_{i\,CPU}$ *and* $PP_{i\,GPU}$ represent the peak power of running a job *i* on the CPU or GPU respectively. The parameter $PT_{i\,CPU}$ represents the peak CPU temperature of running a job *i* on the CPU.

4.6 Profiler and Curve Fitter

To evaluate a job’s workload distribution, we have to measure the overall performance, energy consumption, peak power, and peak CPU temperature after selecting a CPU/GPU work map ratio and allocating the resources (i.e., the number of cores and the number of threads). Due to the absence of accurate models that estimate all these parameters, we instead use prediction. Predicting the execution time, energy consumption, peak power, and peak CPU temperature for a given job workload distribution is difficult and it becomes more complicated when the number of cores and threads change. Profiling and curve fitting [13] have proved to be useful techniques for estimating workload distribution parameters during runtime. Hence, WDRA utilizes the same profiler and curve fitter presented by [13] to find estimation models for the parameters above.

To explain profiling and curve fitting in detail, we use execution time in the following example. The same technique was applied to the rest of the parameters. First, we run a benchmark job (e.g., hotspot) on a quad-core CPU with 32 threads. We run it for the workload sizes {1k, 2k, .., 64G} and measure the overall execution time of each workload size. Then, we apply curve fitting to the collected points to construct an execution time estimation model that has the best fit to these points. This model can be used by any hotspot job with *x* size to predict its execution time on a quad-core CPU with 32 threads. This process is repeated for each benchmark job to produce estimation models for a GPU and a CPU with different number of cores and threads.

Initially, the profiler and curve fitter start using the training set that we collected offline by the method explained above. During runtime, as a job finishes its execution, its real execution time, energy consumption, peak power, and peak CPU temperature are recorded and added to the training set. These added values can enhance the accuracy of the curve fitter.

Our profiler is based on the overall system parameters' measurement (i.e., overall execution time in (11) and overall energy consumption in (12)) that include memory access/transfer latency and CPU-GPU communication overhead.

$$T = \max\left(T_{CPU} + T_{GPU}\right) + T_{memory} + T_{Comm} \tag{11}$$

$$E = E_{CPU} + E_{GPU} + E_{memory} + E_{Communication} \tag{12}$$

4.7 Particle Swarm Optimization

WDRA is not just a workload distributor that finds the best CPU/GPU map ratio to run a job. It is also a resource allocator that provides a job with the needed number of cores and threads. In other words, WDRA is a combination of a workload distributor and a resource allocator. Workload distribution and resource allocation are considered to be as NP-Hard problems in the general case [27]. Moreover, WDRA solves a multi-objective optimization problem and aims to efficiently allocate resources to jobs while considering both performance and energy consumption. Hence, the problem becomes more complex and the search space to find the optimal solution is very large. Therefore, WDRA utilizes an evolutionary algorithm called Particle Swarm Optimization (PSO) that had been proved to be effective for finding the Pareto optimal solutions of such optimization problems.

PSO [28][29] is a bio-inspired search algorithm that simulates the social behavior of a flock of birds where each individual profits from shared knowledge that is based on the discoveries and previous experiences of other individuals of the population to search for food. Each individual, referred to as a particle, in the population, which is also called swarm, represents a solution in the search space. Each particle adjusts its movement according to its experience and that of other particles.

PSO is an iterative search technique that is applied to solve optimization problems to find Pareto optimal solutions. PSO differs from traditional random search methods in that instead of examining a single solution; it is a population-based search algorithm that makes the exploration of the search space faster.

Compared to other evolutionary algorithms, such as Genetic Algorithms (GA), the advantages of PSO are that it is simple, has low computational expense, and converges faster. PSO's fast convergence makes PSO a well-suited optimizer for the runtime workload distribution problem. In order to solve the WDRA multi-objective problem we used the weighted aggregated approach multi-objective PSO optimizer that appeared in [29].

4.7.1 PSO Algorithm

As shown in Fig. 4.4, PSO starts with an initial random swarm/population of *p* particles/solutions. Fig. 4.5(b) shows an example of a swarm. During the PSO iterations, the personal best (*pbest*) and the global best (*gbest*) values are recorded. *pbest* is the best position reached by a given particle so far; whereas, *gbest* is the position of the best particle of the entire population. In each iteration, a fitness function is used to evaluate each particle. If the fitness value of the particle is better than its *pbest*, the *pbest* is set to the current fitness value. Then, we set the value of *gbest* as *pbest* if the *pbest* has a better value. These values are used to calculate the velocity of each particle of the swarm that is necessary for updating its position in the next iteration. The algorithm terminates after a maximum number of iterations is reached.

4.7.2 Particle/Solution Representation

PSO is a population-based algorithm that has *p* particles/solutions. As shown in Fig. 4.5(a), a particle/solution is represented by an object that has three attributes: CPU/GPU map ratio, number of cores, and number of threads. Each particle/solution has its normalized fitness value, peak power in Watts and peak CPU temperature in Celsius. In Fig. 4.5(b) for example: in particle 2, 41% of the job is mapped to the dual-core CPU with 16 threads and the remainder of the job (i.e., 59%) is mapped to the GPU.

We utilize the estimation models to predict the execution time, energy consumption, peak power, and peak CPU temperature for each particle according to its CPU/GPU map ratio, number of cores and number of threads. The particle fitness value is calculated using the weighted sum of the normalized values of the execution time and energy consumption as in (10). We use peak power and peak CPU temperature to ensure the feasibility of a solution so that these parameters should not exceed peak power and peak temperature constraints. A particle is said to be feasible if it does not violate any of the problem constraints (i.e., CPU/GPU map ratio, peak power, peak temperature). If a solution violates any of the constraints, PSO identifies the violation and ensures particle feasibility. For example, if a particle map ratio falls out of the range [0, 1], PSO replaces it with a random number within the range. Moreover, if a particle violates the peak power or peak CPU temperature constraints, the position of the particle is modified randomly for a *z* number of attempts and if all attempts fail it is moved back to its previous position.

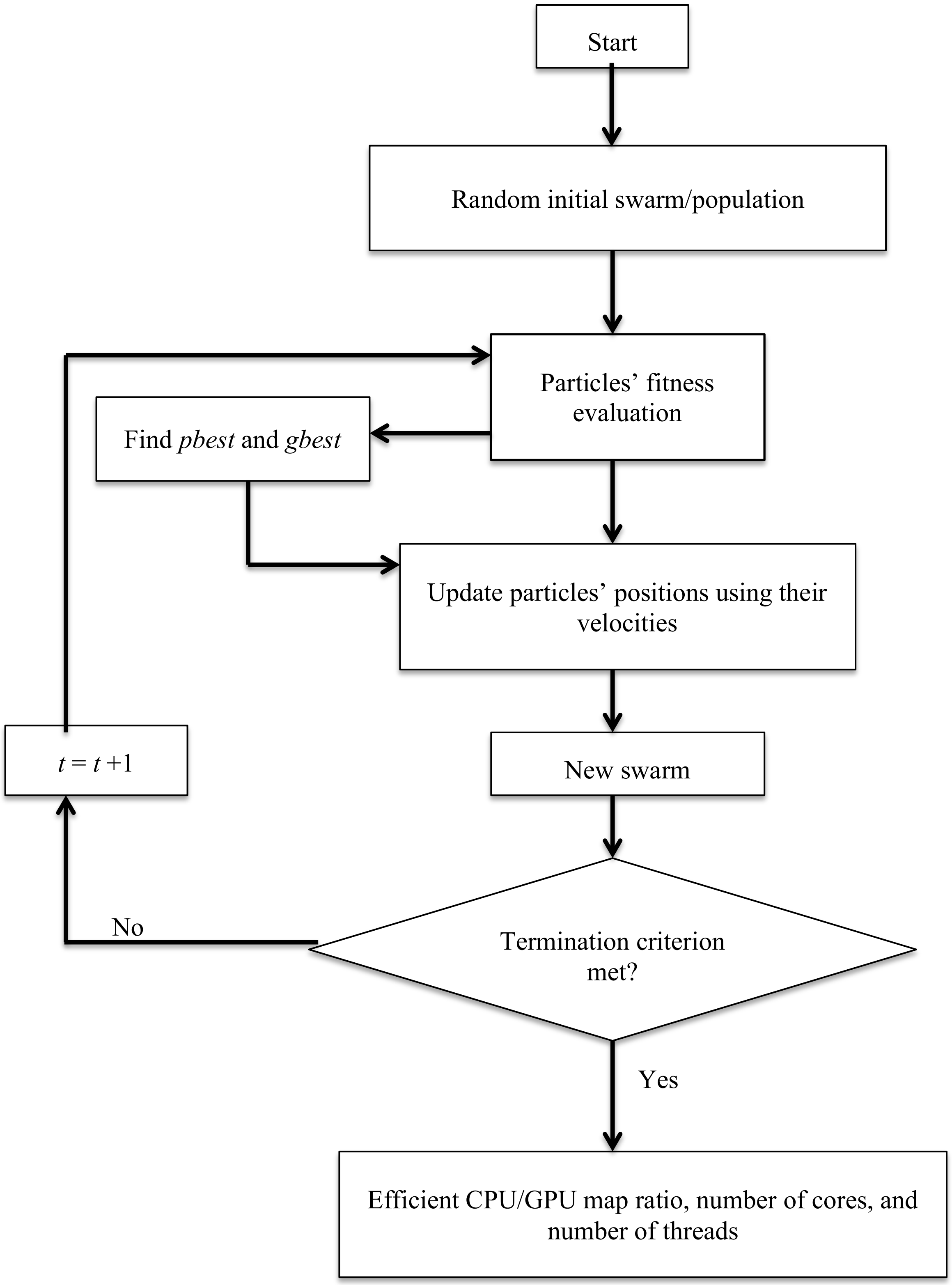

Start
Random initial swarm/population
Particles' fitness evaluation
Find *pbest* and *gbest*
Update particles' positions using their velocities
New swarm
Termination criterion met?
No
Yes
$t = t + 1$
Efficient CPU/GPU map ratio, number of cores, and number of threads

Figure 4.4: PSO flowchart

	Particle		
CPU/GPU map ratio →	0.23	65	← Fitness
Number of cores →	2	375	← Peak power (Watts)
Number of threads →	16	65.4	← Peak temperature (°C)

(a)

P_1		*P_2*		*P_3*		*P_4*			*P_p*	
0.23	55	0.41	49	0.72	47.5	0.91	65.5	…	0.05	64
4	368	2	371	4	372	2	365		1	379
32	70.1	16	71.2	128	73.2	64	71.4		2	49.2

(b)

Figure 4.5: Sample of particle/solution representation (a) and sample of population/swarm (b)

4.7.3 Fitness Function

PSO uses the fitness function to evaluate a particle. A particle is better if it has a lower fitness value. Since WDRA solves a multi-objective optimization problem that considers both performance and energy consumption, it uses the weighted sum fitness function in (10).

To calculate a particle’s fitness with a CPU/GPU map ratio: we first have to predict the total execution time of the part of the job mapped to the CPU with the specified settings (i.e., number of cores and threads). Then, we have to predict the execution time of the remaining part of the job that is mapped to the GPU. The execution time of the entire job is equal to the execution time of the part (i.e., CPU or GPU) that takes the longest execution time, as in (8). The same method is used to calculate the total energy consumption as in (9) except we take the summation of the energy consumption of the entire job (i.e., CPU and GPU parts).

Predicted values are obtained using performance and energy consumption estimation models that were produced by profiling and curve fitting. Finally, the execution time and the total energy consumption are normalized then multiplied by a specified weight as in (10) to calculate the fitness value of a solution.

4.7.4 Particle Position and Velocity

To explore the search space and find the Pareto solution, particles of the swarm move from one position to another based on their experiences and that of other particles. Since *pbest* is the best value reached by a particle so far, it represents a particle's own experience. *gbest* represents the experience of all particles of the swarm since it records the best value reached by the entire swarm. These two values, considered to be social information, are used to calculate particles' velocities. At iteration t, $x_i(t)$ denotes the position of particle p_i and $v_i(t)$ denotes the velocity that represents the shared social information. To change the position of p_i, a velocity $v_i(t)$ is added to the current position as shown in (13).

$$x_i(t) = x_i(t-1) + v_i(t) \tag{13}$$

$$v_i(t) = v_i(t-1) + C_1 r_1 \left(x_{pbest_i} - x_i(t)\right) + C_2 r_2 \left(x_{gbest_i} - x_i(t)\right) \tag{14}$$

;where r_1 and r_2 are random values between [0,1] and C_1, C_2 are constant learning factors. In the WDRA PSO, the position of the particle x_i is the CPU/GPU map ratio that can be updated using its velocity as in (13). A particle velocity is calculated as shown in (14) using its best CPU/GPU map ratio reached so far (i.e., *pbest*) and the best CPU/GPU map ratio that is reached by all particles of the swarm (i.e., *gbest*).

4.8 WDRA Flowchart

As in Fig. 4.6, WDRA starts by performing profiling and curve fitting on an initial training set (i.e., input) of jobs with different sizes running on the CPU and the GPU. During runtime, for each job that arrives, the PSO-based algorithm utilizes these estimation models to predict a job's execution time, energy consumption, peak power and peak CPU temperature based on a job's size. When the PSO algorithm terminates, the efficient CPU/GPU map ratio, and number of cores and threads are set. Then, each job starts running on the CPU and the GPU according to the specified CPU/GPU map ratio, number of cores and threads (i.e., PSO output). When a job's execution is done, the measured (i.e., real) execution time, energy consumption, peak power and peak CPU temperature are recorded into the training set to be used by the curve fitter to enhance prediction accuracy.

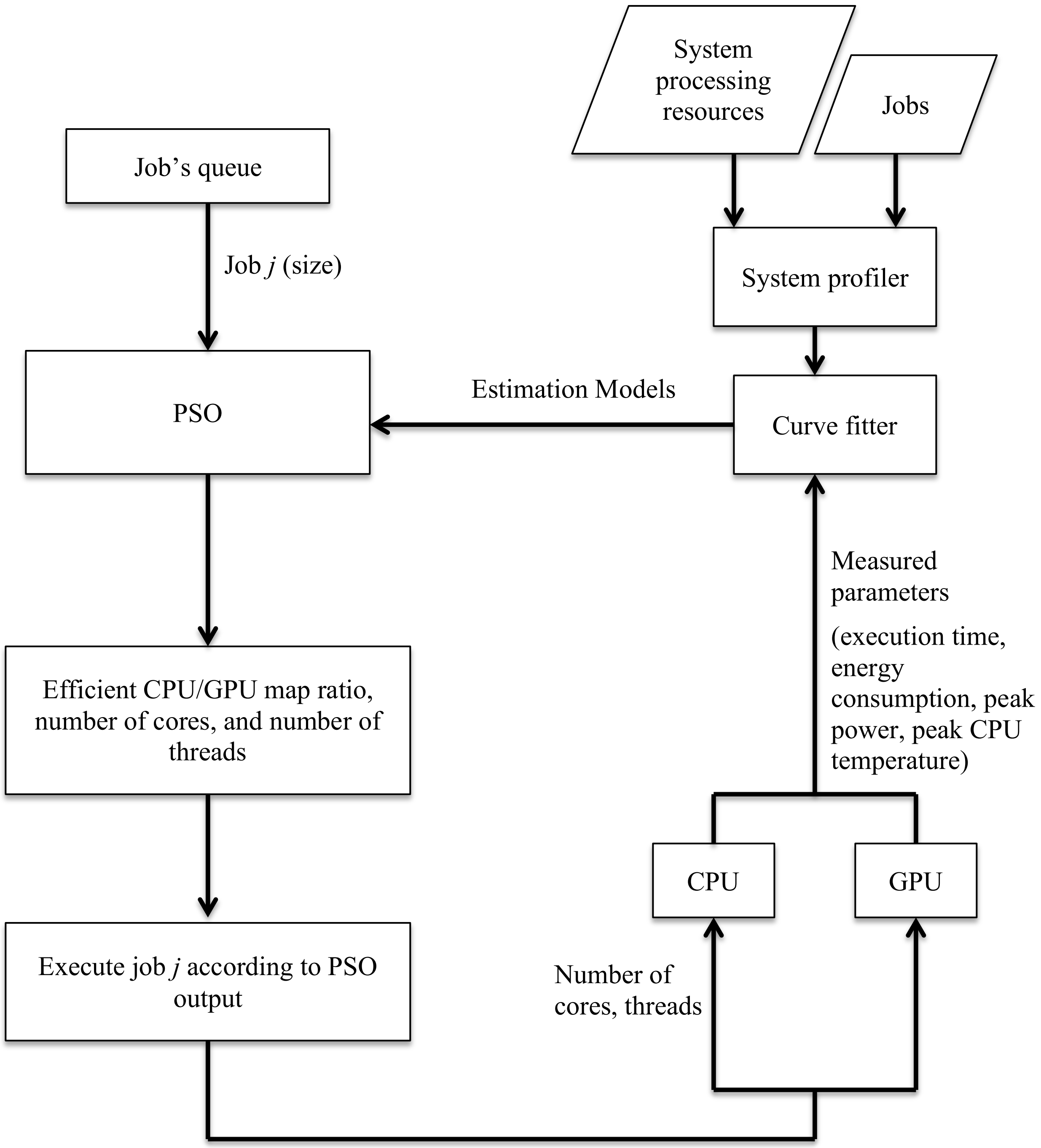

System processing resources
Jobs
Job's queue
System profiler
Job *j* (size)
PSO
Estimation Models
Curve fitter
Measured parameters
(execution time, energy consumption, peak power, peak CPU temperature)
Efficient CPU/GPU map ratio, number of cores, and number of threads
CPU
GPU
Execute job *j* according to PSO output
Number of cores, threads

Figure 4.6: WDRA flowchart

4.9 Evaluation

This section presents the evaluation methodology and the results of WDRA on an actual CPU-GPU heterogeneous system.

4.9.1 Experimental Setup and Benchmark

We conducted our experiment on a real heterogeneous system equipped with a quad-core CPU and a GPU connected via PCI-e. The hardware specifications are shown in detail in Table 3.1. To test WDRA and the other algorithms, we carried out a set of experiments using the Rodinia 3.0 benchmark suite [10][32].

Rodinia is a collection of benchmarks designed for parallel processing on heterogeneous systems. It has applications from different domains such as image processing, bioinformatics, pattern recognition, scientific computing, and simulation. These kinds of applications represent different application behavior types that characterized by Berkeley dwarves [26]. Rodinia applications are implemented to run on both multicore CPUs and GPUs. Therefore, it contains applications that are parallelized and coded using Open Multi-Processing (OpenMP) to run on multicore CPUs and CUDA to run on GPUs.

We evaluated WDRA and the other workload distribution algorithms using applications that have both OpenMP and CUDA versions of the Rodinia 3.0 benchmark. The parallelism of the Rodinia benchmark applications makes them easy to partition. To distribute the work of a job we multiply number of parallel computations/threads of a job by the CPU/GPU map ratio. Then we map the computations/threads to the CPU and the GPU according to the CPU/GPU map ratio.

4.9.2 Profiling and Curve Fitting

To produce estimation models (e.g., an execution time estimation model), we have to apply profiling and curve fitting on each application of the Rodinia benchmarks running on a GPU or a CPU with different configurations of problem sizes, and number of cores and threads. For profiling, we used the configuration set provided in Table 4.1.

To construct an estimation model for each of the configurations, we have to run each application and change one configuration parameter and fix the rest. For example, we ran application *x* with a fixed size (e.g., 4k) and fixed number of threads (e.g., 16) for all possible number of cores (i.e., 1, 2, 4, 8). We used a Linux command to change the boot arguments to disable/enable cores. Then we ran application *x* with specific size (e.g., 4k) on a fixed number of cores (e.g., 4) using different numbers of threads provided in Table 4.1 (e.g., 1, 2, 4, 8). Then we ran application *x* with a fixed number of cores (e.g., 4) and a fixed number of threads (e.g., 16) but with different input sizes as shown in Table 4.1 (e.g., 1k, 2k). For each configuration, we performed the above experiment 1000 times, measured the overall system parameters (e.g., execution time), and took the average.

We used time Linux command [42] to measure the overall wall clock execution time (seconds). We connected a Kill-A-Watt meter [40] to the system's Power Supply Unit (PSU) to measure the overall energy consumption (kWh) and the peak power (Watt). Using that meter and time linux command, we were able to measure the overall system execution time and energy consumption. Therefore, memory access/transfer delay and CPU-GPU communications overhead were also measured.

To measure peak CPU temperature, we utilized the lm_sensors application (Linux monitoring sensors) [41] that monitors each core's temperature separately and we recorded the highest core temperature as the peak CPU temperature.

Curve fitting was performed on the data measured by profiling to find the best model that fits the collected data. These estimation models were used by WDRA to predict execution time, energy consumption, peak power, and peak temperature during runtime.

Table 4.1: Profiling Set

Parameter	**Values**
CPU Cores	1, 2, 4, 8
Number of Threads	1, 2, 4, 8, 16, 32, 64, 128
Problem Size	1k, 2k, 4k, 8k, ….., 64G

4.9.3 WDRA vs. Other Workload Distribution Algorithms

We compared WDRA to the following algorithms:

- Manual Combined Greedy algorithm (MCG): for each job, it performs an exhaustive search to try all possible CPU/GPU map ratios at a granularity of 1% with all possible number of cores and threads. It uses (10) to evaluate all configurations based on both performance and energy with *w*=0.5. The goal is to find the best CPU/GPU map ratio, number of cores, and number of threads in terms of both performance and energy consumption.
- Manual Performance Greedy algorithm (MPG): for each job, it performs an exhaustive search to find what configurations (CPU/GPU map ratio, number of cores and number of threads) have the shortest execution time by running that job. The CPU/GPU map ratio is done at a granularity of 1%. This algorithm accounts for algorithms that are implemented by [13, 20, 25].
- Manual Energy Greedy algorithm (MEG): for each job, it performs an exhaustive search to try all possible configurations (CPU/GPU map ratio at a granularity of 1%, number of cores and number of threads) consume the least energy by running that job. This algorithm represents algorithms appeared in [4, 5, 6].

For fairness, WDRA and the other algorithms that are evaluated use identical profiler. Profiling is done once and offline. A curve fitter was performed on the collected results to produce estimation models. These models were used by all algorithms to estimate the execution time, energy consumption, peak power, and peak CPU temperature. Hence, the overhead of the profiler and the curve fitter is neglected. Note that profiling and curve fitting are tools that we used for prediction and can be replaced by any other estimation methods or tools.

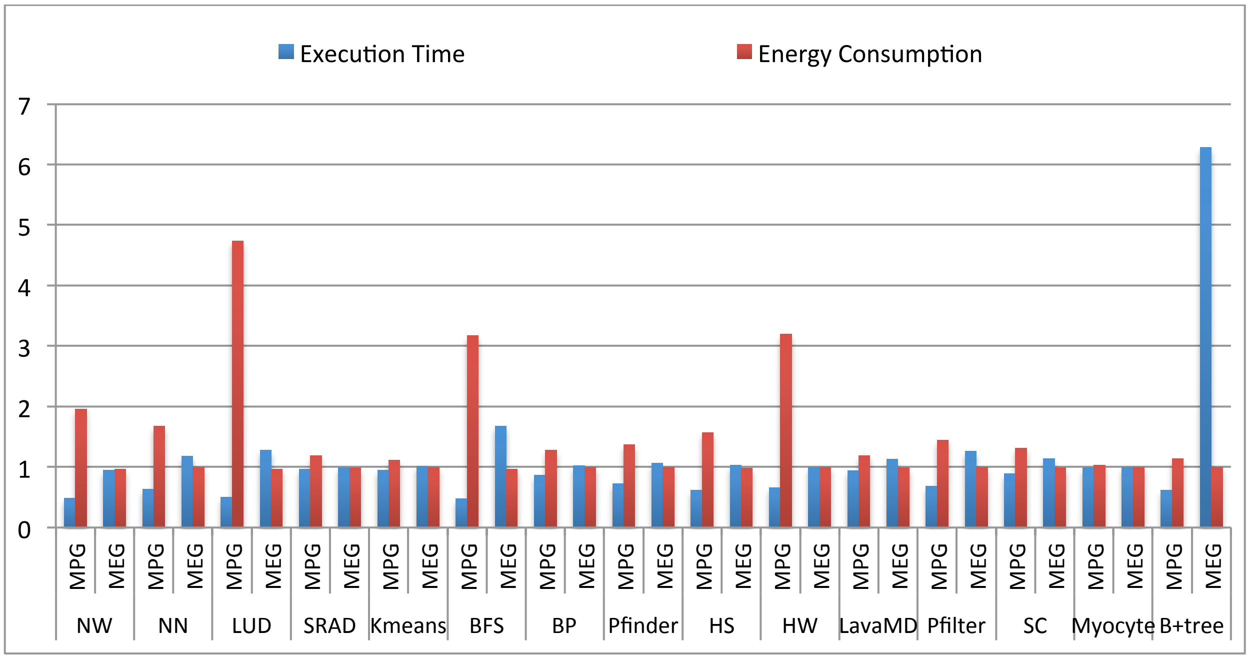
We implemented WDRA and the other greedy algorithms using C++. WDRA utilizes a PSO that is based on the weighted sum fitness function as in (10). If the w=1, WDRA acts as the manual performance greedy algorithm. If the w=0, WDRA acts as the manual energy greedy algorithm. We tested different values of w but we selected w=0.5 for WDRA and the manual combined greedy algorithm because this value represents the balanced scenario between the performance and energy consumptions; when w=0.5 both execution time and energy consumption are equally important. But when w>0.5 it favors execution time over energy consumption and the opposite if w<0.5.

4.9.4 Results

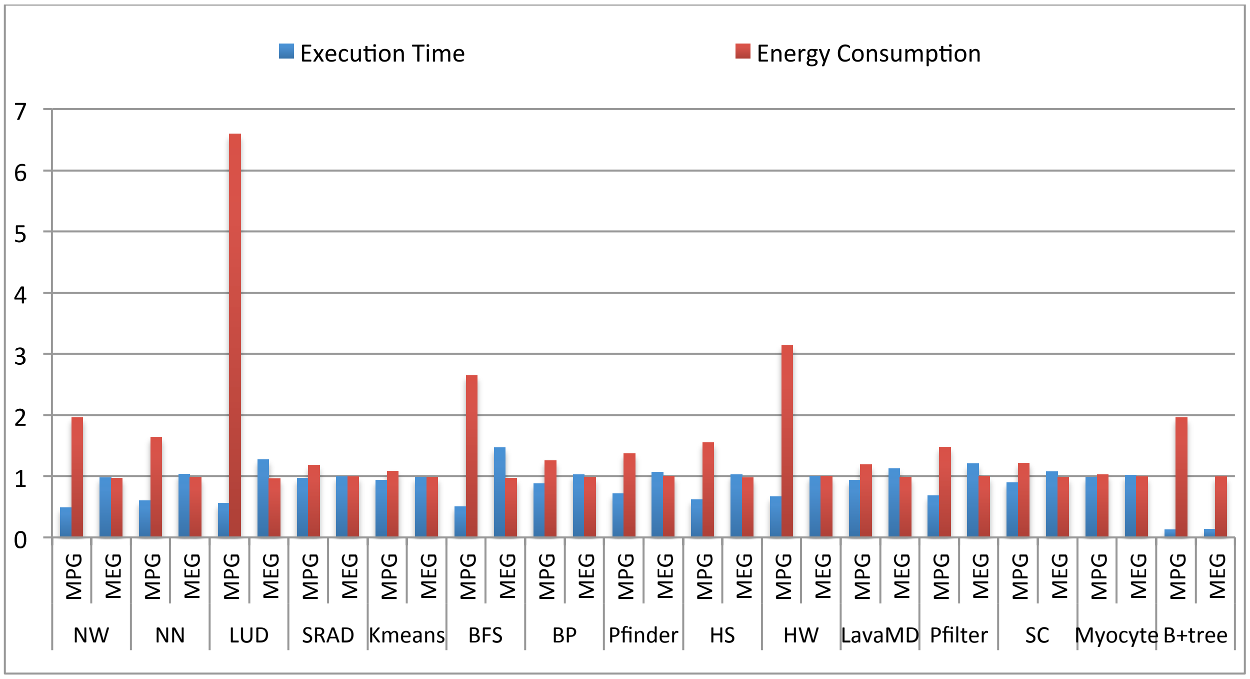
We tested each of the workload distribution algorithms by running 1000 randomly generated jobs of the Rodinia benchmark and then we took the average. For WDRA PSO, we ran PSO with 20 particles for 10 generations. We set C_1 and C_2 to 2. It was proved in [28] that when $C_1=C_2=2$, PSO converges in about half the time that it would when $C_1=C_2=1$. To evaluate the algorithms, we tested them with no constraints as in Fig. 4.7(a), and with both constraints (i.e., peak power, peak CPU temperature) as in Fig. 4.7(b).

Fig. 4.7(a) shows the average energy consumptions and execution time of the benchmarks using the manual greedy algorithms (i.e., performance-based and energy-based) normalized to the WDRA algorithm. The MPG beats WDRA in performance for the following jobs: NW, NN, LUD, BFS, BP, PFinder, HS, HW, PFilter, SC, and B+tree. However, WDRA outperforms it in terms of energy consumption. On the other hand, as shown in SRAD, Kmeans, LavaMD, and Myocyte, WDRA has the same average performance as the MPG algorithm but with a better reduction in energy consumption. In some cases, distributing the work according to the least energy consumption happens to be the workload distribution with the shortest execution time as well; WDRA and the MEG algorithm have the same results in both performance and energy consumption for NW, SRAD, Kmeans, BP, HS, HW, and Myocyte. But in other cases (i.e. the rest of the benchmark jobs), distributing the workload according to the best energy consumption degrades its performance (i.e., higher execution time). Because the performance-based algorithm focuses on only minimizing a job's execution time, it may increase the energy consumption of that job. The energy-based algorithm cannot reach the best solution in terms of performance because it aims to find the least energy consumed by each job without considering its execution time.

As it shown in Fig. 4.7(a), it can be concluded that WDRA outperforms both algorithms because it distributes the work according to both jobs' execution time and energy consumption. On average, as shown in Fig. 4.8, compared to the other greedy distribution algorithms, WDRA can achieve up to 1.47x speedup and energy saving of up to 82%.



(a)



(b)

Figure 4.7: MPG and MEG average execution time and energy consumption normalized to WDRA *w*=0.5 with no constraints (a) with peak power and peak CPU temperature constraints (b)

We tested WDRA and other workload distributors with no constraints as in Fig. 4.7(a), and with both constraints (i.e., peak power, peak CPU temperature) as in Fig. 4.7(b). To select the peak power budget PP_{max}, we measured the peak power reached by each Rodinia job using the random workload distributor and reduced it by 5% or 10%. We did the same to select peak temperature PT_{max}. Fig. 4.7(b) shows PP_{max} and PT_{max} with a 5% reduction.

As shown in Fig. 4.7(b), results of the algorithms with constraints follow the same trend of the case with no constraints, except that the constraints limit the search space. Therefore, it can be concluded that WDRA works well in finding the best workload distribution with peak power and peak CPU temperature constraints.

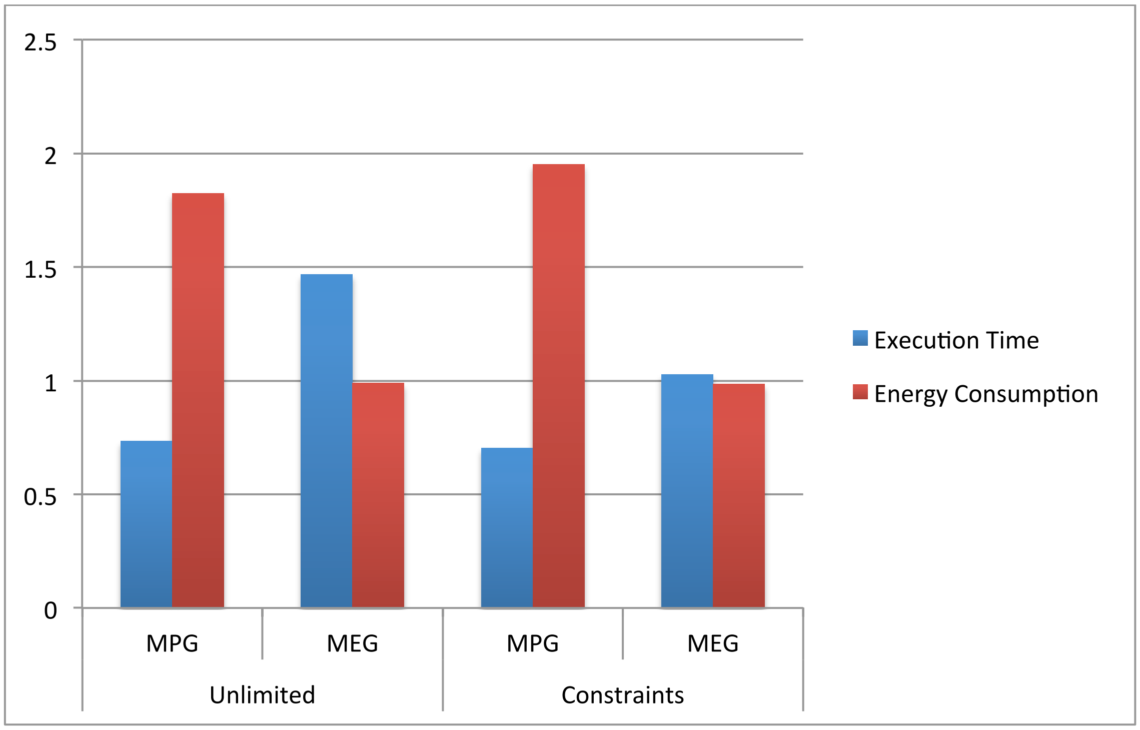


Figure 4.8: Rodinia Benchmark's Average of MPG and MEG execution time and energy consumption normalized to WDRA *w*=0.5 with no constraints and with peak power and peak CPU temperature constraints

4.9.5 WDRA Accuracy, Speedup and Overhead

To check WDRA PSO results' accuracy, we implemented the MCG algorithm that utilizes the weighted sum fitness function that considers both performance and energy consumption with w=0.5.

Table 4.2 shows that the accuracy of WDRA's results is about 99% compared to the MCG. As for the speedup, we measure the duration of WDRA and other greedy algorithms. As shown in Table 4.2, on average, the WDRA (i.e., PSO-based) algorithm runs 76% faster compared to the manual greedy algorithms. Note that, if we run WDRA with a fewer number of particles or generations than what we used in our experiment, we could enhance the PSO speedup; however, the results would be less accurate.

To measure the WDRA overhead, for each job we divided the duration of WDRA by the job execution time and took the average. According to our experiments on Rodinia jobs that are presented in Table 4.2, WDRA only takes up to 1.7% of the job execution time. Therefore, WDRA is suitable for runtime workload distribution.

Table 4.2: WDRA Accuracy, Speedup and Overhead

Rodinia Jobs	Parameters		
	Accuracy	*Speedup*	*Overhead*
NW	0.9822	1.6860	0.0043
NN	0.9971	1.7820	0.0020
LUD	0.9821	1.6697	0.0069
SRAD	0.9889	1.8007	0.0031
Kmeans	0.9982	2.1526	0.0031
BFS	1.0000	1.9178	0.0018
BP	0.9966	1.9663	0.0610
Pathfinder	1.0000	1.5419	0.0004
Leukocyte	1.0000	1.7802	0.0011
Hotspot	1.0000	1.7776	0.0016
Heartwall	1.0000	1.7223	0.0008
LavaMD	1.0000	1.6462	0.0026
PFilter	1.0000	1.6139	0.0006
SC	0.9970	1.9317	0.0008
Myocyte	1.0000	1.6363	0.0943
B+tree	0.9967	1.5924	0.1194
CFD	1.0000	1.5713	0.0001
Average	0.9969	1.7625	0.0179

4.10 Summary

We have introduced a runtime Workload Distributor with a Resource Allocator (WDRA) that finds the best CPU/GPU map ratio, number of cores, and number of threads in terms of both performance and energy consumption under peak power and peak CPU temperature constraints. Since resource allocation and workload distribution are known as NP-hard problems, WDRA utilizes a multi-objective Particle Swarm Optimization (PSO) algorithm. WDRA can be applied to any CPU-GPU heterogeneous system from mobile systems to servers. WDRA is not a classic CPU-GPU workload distributor; it is also a resource allocator (i.e., number of cores and threads). Experimental results show WDRA's capability to find efficient solutions in terms of both execution time and energy consumption under peak power and peak CPU temperature constraints. Therefore, WDRA can be used in embedded systems fields. For evaluation, we have implemented WDRA on an actual system equipped with a multi-core CPU and a GPU. WDRA outperforms performance-based and other distribution algorithms in both execution time and energy consumption for constraint (i.e., peak power and peak CPU temperature limits) or non-constraint problems. Since WDRA takes only 1.7% of a job's execution time, our algorithm is well suited for workload distribution during runtime. We believe that it is important to combine all problems (i.e., workload distribution, core scaling, and thread allocation) into one optimization problem. Moreover, both execution time and energy consumption should be considered while distributing a job's workload.

Chapter 5

Conclusions and Future Work

In this chapter, we give our conclusions on the research presented in this thesis and discuss future work and other possible extensions to the research.

5.1 Conclusions

The goal of this dissertation is to study the effectiveness of combining resource allocators with schedulers and workload distributors in improving the performance and reducing the energy consumption of heterogeneous systems. Moreover, the goal is to show the importance of considering both systems' performance and energy consumption while job scheduling or workload distribution. In addition, peak power and peak CPU temperature limits should be taken into account especially if the targeted hardware is an embedded system.

The research goal has been achieved in the following major steps: first, we have proposed the Performance, Energy and Thermal aware Resource Allocator and Scheduler (PETRAS), which combines job mapping, core scaling, and thread allocation into one scheduler. Second, we have proposed the Workload Distributor with a Resource Allocator (WDRA), which finds a job's efficient workload distribution and resource allocation in terms of both a system's performance and energy consumption. Third, we have implemented PETRAS and WDRA on CPU-GPU heterogeneous systems. Finally, we have evaluated the effectiveness of PETRAS and WDRA on an actual heterogeneous system that is equipped with a multi-core CPU and a GPU. PETRAS and WDRA were evaluated considering the overall performance and energy in both cases: non-constraints, and peak power and peak CPU temperature constraints.

5.1.1 Scheduling and Resource Allocation for Heterogeneous Systems

In chapter 3, we have presented the Performance, Energy and Thermal aware Resource Allocator and Scheduler (PETRAS), which utilizes a Genetic Algorithm (GA) to find efficient schedules in a heterogeneous system. PETRAS combines the following problems into one scheduler: job mapping and scheduling, core scaling, and thread allocation. To evaluate PETRAS, we have implemented PETRAS on an actual system equipped with a multi-core CPU and a GPU. We have demonstrated that the PETRAS scheduler outperforms performance-based schedulers and other schedulers in both execution time and energy consumption. We have tested PETRAS for the two cases of with and without peak power and peak CPU temperature constraints. PETRAS shows its capability to find efficient schedules in terms of both execution time and energy consumption for the two cases. Therefore, PETRAS can be used for embedded systems applications. Even though we have tested PETRAS on a CPU-GPU heterogeneous system, PETRAS's methodology is generic and can be applied on any heterogeneous system where processing units are fused on the same chip or connected through a bus.

We have added a power management unit that turns off the idle and low-utilized processing units of the GA efficient schedules. We have found that this step helps in saving energy consumption and freeing idle and low-utilized processing units to be used by other applications that share the same hardware.

We have shown the importance of combining resource allocation (i.e., core scaling, thread allocation) and scheduling into one optimization problem. Therefore, we believe that we have to consider all problems (i.e., job mapping and scheduling, core scaling, thread allocation) into one scheduling optimization problem. Moreover, we have compared performance-based and energy-based schedulers to PETRAS, which considers both execution time and energy consumption. PETRAS as a multi-objective scheduler outperforms single-objective schedulers (i.e., energy-based, performance-based). Hence, we believe that schedules should be selected based on both execution time and energy consumption.

5.1.2 Workload Distribution and Resource Allocation for Heterogeneous Systems

In chapter 4, We have introduced a runtime Workload Distributor with a Resource Allocator (WDRA) that finds the best CPU/GPU map ratio, number of cores, and number of threads in terms of both performance and energy consumption under peak power and peak CPU temperature constraints. In WDRA, we have applied a multi-objective Particle Swarm Optimization (PSO) algorithm because resource allocation and workload distribution are known as NP-hard problems. WDRA's methodology is generic and can be applied to any CPU-GPU heterogeneous system from mobile systems to servers.

We have shown the impact of combining workload distribution and resource allocation problems on the overall performance and energy consumption. Therefore, we have proposed a WDRA, which is not a classic CPU-GPU workload distributor; it is also a resource allocator (i.e., number of cores and threads).

To test WDRA, we have implemented WDRA on an actual system equipped with a multi-core CPU and a GPU. We have found that WDRA outperforms performance-based and other workload distribution algorithms in both execution time and energy consumption. We have evaluated WDRA for the two cases: constraint (i.e., peak power and peak CPU temperature limits) and non-constraint problems. By analyzing experimental results, we have demonstrated WDRA's capability to find efficient solutions in terms of both execution time and energy consumption under peak power and peak CPU temperature constraints. Therefore, WDRA can also be used in embedded systems fields.

We have measured WDRA algorithm's duration and compared it to jobs' execution times. WDRA takes only up to 1.7% of a jobs' execution time. Thus, WDRA algorithm is well suited for workload distribution during runtime.

We have shown the importance of combining all problems (i.e., workload distribution, core scaling, and thread allocation) into one optimization problem. Therefore, we believe that we have to consider resource allocation while performing job workload distribution. Moreover, we have shown that WDRA outperforms performance-based and energy-based greedy workload distribution algorithms in terms of both execution time and energy consumption. Hence, we believe that both execution time and energy consumption should be considered while distributing a job's workload.

5.2 Future Work

The research of this thesis can be extended in many possible directions. In PETRAS and WDRA, we have demonstrated the importance of combining resource allocation (i.e., core scaling, thread allocation) with job scheduling or workload distribution. We plan to extend these algorithms by exploring more resources such as GPU block size, cache memory size, etc.

We have tested PETRAS and WDRA on a CPU-GPU heterogeneous system. But we also believe that PETRAS and WDRA have significant potential in large-scale systems such as cloud computing systems. Moreover, PETRAS and WDRA can be applied for embedded systems applications since we have demonstrated their effectiveness while considering peak power and peak CPU temperature constraints. Therefore, we would like to apply and evaluate PETRAS and WDRA on the cloud and the embedded systems. We plan to study how these algorithms improve performance and reduce energy consumption on such platforms.

PETRAS has a power management unit that turns off the idle or low-utilized processing units by setting their frequency to zero. We plan to add Dynamic Voltage Frequency Scaling (DVFS) to PETRAS to adjust processing units frequency where low-utilized processing units would operate with low frequency set by DVFS. We believe that by adding DVFS we will save more energy. Moreover, we would like to add DVFS to the WDRA and study the impact of applying DVFS on reducing energy consumption.

REFERENCES

[1] R. Kumar, D. Tullsen, N. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *Computer*, vol. 38, no. 11, pp. 32–38, Nov 2005.

[2] NVIDIA Corporation, NVIDIA CUDA Compute Unified Device Architecture Programming Guide. NVIDIA Corporation, 2008.

[3] S. Ahmad, E. Munir, and W. Nisar, "Pega: A performance effective genetic algorithm for task scheduling in heterogeneous systems," in High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCCICESS), 2012 IEEE 14th International Conference on, June 2012, pp. 1082–1087.

[4] S. Alsubaihi and J. L. Gaudiot, "PETS: Performance, Energy and Thermal Aware Scheduler for Job Mapping with Resource Allocation in Heterogeneous Systems," in IEEE 35th International Performance Computing and Communications Conference (IPCCC), Las Vegas, NV, 2016.

[5] K. Ansari, P. Chitra, and P. Sonaiyakarthick, "Power-aware scheduling of fixed priority tasks in soft real-time multicore systems," in Emerging Trends in Computing, Communication and Nanotechnology (ICE-CCN), 2013 International Conference on, March 2013, pp. 496–502.

[6] M. Chiesi, L. Vanzolini, C. Mucci, E. Franchi Scarselli, and R. Guerrieri, "Power-aware job scheduling on heterogeneous multicore architectures," pp. 1–1, 2014.

[7] M. R. Garey and D. S. Johnson, Computers and Intractability; A Guide to the Theory of NP-Completeness. New York, NY, USA: W. H. Freeman & Co., 1990.

[8] J. H. Holland, Adaptation in Natural and Artificial Systems. Cambridge, MA, USA: MIT Press, 1992.

[9] A. Konak, D. Coit, A. Smith Multi-objective optimization using genetic algorithm: a tutorial Reliability Engineering and System Safety, 91 (2006), pp. 992–1007.

[10] Che, Shuai, et al. "Rodinia: A benchmark suite for heterogeneous computing." Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on. IEEE, 2009.

[11] B. Liu, M. H. Foroozannejad, S. Ghiasi and B. M. Baas, "Optimizing power of many-core systems by exploiting dynamic voltage, frequency and core scaling," 2015 IEEE 58th International Midwest Symposium on Circuits and Systems (MWSCAS), Fort Collins, CO, 2015, pp. 1-4.

[12] A. Vega, A. Buyuktosunoglu and P. Bose, "SMT-centric power-aware thread placement in chip multiprocessors," Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, 2013, pp. 167-176.

[13] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on, Dec 2009, pp. 45–55.

[14] G. Menghani, "A fast genetic algorithm based static heuristic for scheduling independent tasks on heterogeneous systems," in Parallel Distributed and Grid Computing (PDGC), 2010 1st International Conference on, Oct 2010, pp. 113–117.

[15] A. Page and T. Naughton, “Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing,” in Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International, April 2005, pp. 189a–189a.

[16] P. Rong and M. Pedram, “Power-aware scheduling and dynamic voltage setting for tasks running on a hard real-time system,” in Design Automation, 2006. Asia and South Pacific Conference on,Jan 2006, pp.6.

[17] M. Srinivas and L. Patnaik, “Genetic algorithms: a survey,” Computer, vol. 27, no. 6, pp. 17–26, June 1994.

[18] R. Al Na’mneh and K. Darabkh, “A new genetic-based algorithm for scheduling static tasks in homogeneous parallel systems,” in Robotics, Biomimetics, and Intelligent Computational Systems (ROBIONETICS), 2013 IEEE International Conference on, Nov 2013, pp. 46–50.

[19] S. Alsubaihi and J. L. Gaudiot, "PETRAS: Performance, Energy and Thermal Aware Resource Allocation and Scheduling for Heterogeneous Systems," In Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'17), ACM, Austin, TX, 2017, pp. 29-38.

[20] C. Lee, W.Ro, J.Gaudiot, “Cooperative Heterogeneous Computing for Parallel Processing on CPU/GPU Hybrids,” Workshop on Interaction between Compilers and Computer Architectures, 2012.

[21] G. Wang and X. Ren, "Power-Efficient Work Distribution Method for CPU-GPU Heterogeneous System," International Symposium on Parallel and Distributed Processing with Applications, Taipei, 2010, pp. 122-129.

[22] R. Ge, X. Feng, M. Burtscher and Z. Zong, "Performance and Energy Modeling for Cooperative Hybrid Computing," 2014 9th IEEE International Conference on Networking, Architecture, and Storage, Tianjin, 2014, pp. 232-241.

[23] T. Komoda, S. Hayashi, T. Nakada, S. Miwa and H. Nakamura, "Power capping of CPU-GPU heterogeneous systems through coordinating DVFS and task mapping," 2013 IEEE 31st International Conference on Computer Design (ICCD), Asheville, NC, 2013, pp. 349-356.

[24] B. Liu, M. H. Foroozannejad, S. Ghiasi and B. M. Baas, "Optimizing power of many-core systems by exploiting dynamic voltage, frequency and core scaling," 2015 IEEE 58th International Midwest Symposium on Circuits and Systems (MWSCAS), Fort Collins, CO, 2015, pp. 1-4.

[25] J. Lee, M. Samadi, Y. Park and S. Mahlke, "Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems," Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, 2013, pp. 245-255.

[26] K. Asanovic et al. “The landscape of parallel computing research: A view from Berkeley. Technical Report,” UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[27] D. S. Hochbaum. “Approximation algorithms for NP Hard problems,” PWS publishing company, 1998, pp. 1-17.

[28] J. Kennedy and R. Eberhart, "Particle swarm optimization," Neural Networks, 1995. Proceedings., IEEE International Conference on, Perth, WA, 1995, pp. 1942-1948 vol.4.

[29] K. E. Parsopoulos and M. N. Vrahatis. “Particle swarm optimization method in multiobjective problems,” In Proceedings of the 2002 ACM Symposium on Applied Computing (SAC’2002), pages 603– 607, Madrid, Spain, 2002. ACM Press.

[30] OpenMP Application Programming Interface. Version 3.0. OpenMP. 2008. url: http://www.openmp.org/wp-content/uploads/spec30.pdf

[31] B. Stroustrup, The C++ Programming Language (4th ed.), Addison-Wesley (2013).

[32] Che, Shuai, et al. "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," In Proceedings of the IEEE International Symposium on Workload Characterization, Dec. 2010.

[33] Y. Xu, K. Li, T. T. Khac, and M. Qiu, “A multiple priority queueing genetic algorithm for task scheduling on heterogeneous computing systems,” In High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on, June 2012, pp. 639–646.

[34] S. Alsubaihi and J. L. Gaudiot, "A Runtime Workload Distribution with Resource Allocation for CPU-GPU Heterogeneous Systems," 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Orlando, FL, 2017.

[35] 7th Generation Intel Core i7 Processor. url: http://www.intel.com/content/www/us/en/products/processors/core/i7-processors.html

[36] TESLA™ C2050 / C2070 GPU Computing Processor. url: http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf

[37] J. Nickolls and W. J. Dally, "The GPU Computing Era," in IEEE Micro, vol. 30, no. 2, pp. 56-69, March-April 2010.

[38] T. Pering, T. Burd and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algorithms," Proceedings. 1998 International Symposium on Low Power Electronics and Design (IEEE Cat. No.98TH8379), Monterey, CA, USA, 1998, pp. 76-81.

[39] M . Weiser. B. Welch. A. Demers. and S.Shenker. "Scheduling for reduced CPU energy." In 1st OSDI (Monterey, CA, USA, Nov 1994), pp. 13–23.

[40] Kill-A-Watt. url: http://www.p3international.com/products/energy-savers.html

[41] Discovering and Monitoring Hardware in Linux. url: http://www.linux.com/learn/discovering-and-monitoring-hardware-linux

[42] Linux User's Manual – Time Command. url: http://man7.org/linux/man-pages/man1/time.1.html

[43] PCI Express. url: http://www.nvidia.com/page/pci_express.html