CAPITAL UNIVERSITY OF SCIENCE AND
TECHNOLOGY, ISLAMABAD

Load-Balanced Multi-Job Scheduling For Heterogeneous CPU-GPU Systems

by

Yasir Noman Khalid

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosphy

in the

Faculty of Computing
Department of Computer Science

2020

Load-Balanced Multi-Job Scheduling For Heterogeneous CPU-GPU Systems

By

Yasir Noman Khalid

(DCS143001)

Dr. Basel Katt, Associate Professor
Norwegian University of Science and Technology, Norway
(Foreign Evaluator 1)

Dr. Doğan Aydın, Associate Professor
Dumlupınar University, Kutahya, Turkey
(Foreign Evaluator 2)

Dr. Muhammad Aleem
(Thesis Supervisor)

Dr. Nayyer Masood
(Head, Department of Computer Science)

Dr. Muhammad Abdul Qadir
(Dean, Faculty of Computing)

DEPARTMENT OF COMPUTER SCIENCE
CAPITAL UNIVERSITY OF SCIENCE AND TECHNOLOGY
ISLAMABAD
2020

Dedicated to the memory of my grandfather Falak Sher Khan Ghazni Khel

CAPITAL UNIVERSITY OF SCIENCE & TECHNOLOGY ISLAMABAD

Expressway, Kahuta Road, Zone-V, Islamabad
Phone:+92-51-111-555-666 Fax: +92-51-4486705
Email: info@cust.edu.pk Website: https://www.cust.edu.pk

CERTIFICATE OF APPROVAL

This is to certify that the research work presented in the thesis, entitled "**Load-Balanced Multi-Job Scheduling for Heterogeneous CPU-GPU Systems**" was conducted under the supervision of **Dr. Muhammad Aleem**. No part of this thesis has been submitted anywhere else for any other degree. This thesis is submitted to the **Department of Computer Science, Capital University of Science and Technology** in partial fulfillment of the requirements for the degree of Doctor in Philosophy in the field of **Computer Science.** The open defence of the thesis was conducted on **25 June, 2020.**

Student Name :	Mr. Yasir Noman Khalid (DCS143001)	

The Examination Committee unanimously agrees to award PhD degree in the mentioned field.

Examination Committee :

(a)	External Examiner 1:	Dr. Sajjad A. Madani, Professor CUI, Wah Cantt Campus	
(b)	External Examiner 2:	Dr. Ghulam Abbas, Associate Professor GIKI, Topi, Swabi	
(c)	Internal Examiner :	Dr. Nadeem Anjum Assistant Professor CUST, Islamabad	25/6/2020
	Supervisor Name :	Dr. Muhammad Aleem Associate Professor FAST-NUCES, Islamabad	
	Name of HoD :	Dr. Nayyer Masood Professor CUST, Islamabad	
	Name of Dean :	Dr. Muhammad Abdul Qadir Professor CUST, Islamabad	

AUTHOR'S DECLARATION

I, **Mr. Yasir Noman Khalid (Registration No. DCS143001)**, hereby state that my PhD thesis titled, '**Load-Balanced Multi-Job Scheduling for Heterogeneous CPU-GPU Systems**' is my own work and has not been submitted previously by me for taking any degree from Capital University of Science and Technology, Islamabad or anywhere else in the country/ world.

At any time, if my statement is found to be incorrect even after my graduation, the University has the right to withdraw my PhD Degree.

(Mr. Yasir Noman Khalid)

Dated: June, 2020

Registration No : DCS 143001

PLAGIARISM UNDERTAKING

I solemnly declare that research work presented in the thesis titled "**Load-Balanced Multi-Job Scheduling for Heterogeneous CPU-GPU Systems**" is solely my research work with no significant contribution from any other person. Small contribution/ help wherever taken has been duly acknowledged and that complete thesis has been written by me.

I understand the zero tolerance policy of the HEC and Capital University of Science and Technology towards plagiarism. Therefore, I as an author of the above titled thesis declare that no portion of my thesis has been plagiarized and any material used as reference is properly referred/ cited.

I undertake that if I am found guilty of any formal plagiarism in the above titled thesis even after award of PhD Degree, the University reserves the right to withdraw/ revoke my PhD degree and that HEC and the University have the right to publish my name on the HEC/ University Website on which names of students are placed who submitted plagiarized thesis.

(Mr. Yasir Noman Khalid)

Dated: June, 2020

Registration No : DCS 143001

List of Publications

It is certified that following publication(s) have been made out of the research work that has been carried out for this thesis:-

1. **Khalid, Y. N.**, Aleem, M., Prodan, R., Iqbal, M. A., & Islam, M. A. (2018). E-OSched: a load balancing scheduler for heterogeneous multicores. *The Journal of Supercomputing*, 74(10), 5399-5431.

2. **Khalid, Y. N.**, Aleem, M., Ahmed, U., Islam, M. A., & Iqbal, M. A. (2019). Troodon: A machine-learning based load-balancing application scheduler for CPU–GPU system. *Journal of Parallel and Distributed Computing*, 132, 79-94.

Yasir Noman Khalid
(DCS143001)

Acknowledgements

Looking back at my journey through Ph.D. few people helped me a lot during my Ph.D. Among them, first of all, I would like to express gratitude to my supervisor Dr. Muhammad Aleem for his useful comments, suggestions, persistence and thorough engagement through this doctorate thesis. He mentored me, challenged and pushed me a lot to be the best researcher that I could be.

I am also grateful to Dr. Muhammed Arshad Islam and Dr. Muhammad Azhar Iqbal for their support and guidance throughout the research process. Moreover, I am highly indebted to Dr. Abdul Qadir, Dean Faculty of Computing, CUST for his insightful suggestion during research seminars.
I would also like to acknowledge Dr. Shafiq ur Rehman and Usman Ahmed for all the hours spent in the lab in problem-solving and discussions that broaden my horizon. I am also grateful to all the members of Parallel Computing and Networks (PCN) research lab especially Dr. Muhammad Ibrahim and Omaid Ghayyur for their support during the doctorate studies.

I am very obliged to the Higher Education Commission (HEC) of Pakistan for awarding me with a fully-funded scholarship to pursue my doctoral studies.

Finally, I am extremely grateful to my father for his support during my life. Without his support, none of this would have ever been possible.

Abstract

Heterogeneous systems consisting of a *Central Processing Unit* (CPU) and a *Graphics Processing Unit* (GPU) are prevalent nowadays. The advent of heterogeneous programming models such as *Open Compute Language* (OpenCL) has made it possible to execute data-parallel applications on the GPU. As a result, developers are increasingly porting applications to OpenCL to accelerate application execution. However, mapping all heterogeneous applications to the GPU creates severe load imbalance across CPU and GPU in a multi-job scenario. This results in longer execution time of jobs and lower system throughput.

This thesis is an attempt to resolve the load imbalance problem during scheduling of applications in a heterogeneous environment. Among others, this thesis presents a novel scheduling mechanism, named *Enhanced OpenCL Scheduler* (E-OSched) that maps OpenCL applications on a heterogeneous system in a load-balanced manner. Load balance is achieved by contemplating the computation requirements of applications and processing power of heterogeneous processors in scheduling decisions.

This thesis also examines the impact of applications' device suitability in multi-job scheduling on a heterogeneous CPU-GPU system. A machine learning-based application classifier, called Troodon, has been developed that classifies each application as either suitable for CPU execution or GPU execution. Furthermore, a speedup predictor has also been developed that predict the speedup when an application is executed on a suitable device in comparison to execution on a non-suitable device. Load-balanced mapping of jobs to heterogeneous devices is ensured by adopting the E-OSched scheduling mechanism.

A kernel fusion technique is also part of this thesis that increases GPU device utilization by fusing kernels with small data size. A machine learning-based fusion classifier has been developed that classifies jobs as either fusion suitable or fusion unsuitable. Thereafter, a pair of fusion suitable kernels, producing the highest speedup in comparison to their serial execution, are fused.

Contents

List of Figures

List of Tables

Abbreviations

AA	Alternate Assignment
ANN	Artificial Neural Network
CJQ	CPU job queue
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CuMAS	CUDA Multi-Application Scheduling
DCT	Discrete Cosine Transform
DS	Device Suitability
E-OSched	Enhanced OSched
FCFS	First Come First Serve
FIFO	First In First Out
FLOPS	Floating Point Operations Per Second
FPGA	Field Programmable Gate Array
FPR	False Positive Rate
GJQ	GPU job queue
GOPS	Giga Operations Per Second
GPGPU	General Purpose Programming on GPU
GPU	Graphical Processing Unit
IR	Intermediate Representation
ISA	Instruction Set Architecture
ISG	Input size guided
MIC	Many Integrated Cores
MIMD	Multiple Instruction Multiple Data
MKMD	Multi Kernel on Multi Devices

ML	Machine Learning
OpenCL	Open Compute Language
OpenMP	Open Multi Processing
OSched	OpenCL Scheduler
pthreads	POSIX threads
ROC	Receiver Optimization Characteristics
SIMD	Single Instruction Multiple Data
SVM	Support Vector Machine
TBB	Thread Building Block
TPOT	Tree-based Pipeline Optimization Tool
TPR	True Positive Rate

Symbols

α	Adjustment factor
A	Merge unsuitable job list
A_{cpu}	Set of CPU suitable jobs
A_{gpu}	Set of GPU suitable jobs
B	Merge suitable job list
$\mathbb{C}$	Set of submitted jobs
C	Combination of each job from B with every other job from B
CA	Computational assessment module
$Ceiling_{cpu}$	Upper boundary for the number of jobs in J_{cpu}, assigned to the CPU
$Ceiling_{gpu}(i)$	The upper boundary for the number of jobs from J_{gpu}, assigned to i^{th} GPU
CF	Count of a code feature
CJQ	CPU Job Queue
CRC_i	Computation requirement of a job c_i
CRJ	Computation requirement of a job
CS_{cpu}	The computational share of CPU
$CS_{gpu}(i)$	The computational share of i^{th} GPU
$DS(c_i)$	Device suitability of a job c_i
F	Set of 23 code features
GJQ	GPU Job Queue
J	Set of jobs after final arrangement in the job pool
J_{cpu}	A subset of 1^{st} q jobs from job pool J

J_{CR}	Total computation requirement of all the jobs in J
J_{gpu}	A subset of 1^{st} r jobs from J_{update}
J_{update}	A subset of J, formed by subtracting CJQ from J
JP_{gpu}	Final GPU job pool
KFE	Kernel feature extractor module
ML_{DS}	Machine Learning based device suitability model
ML_{MS}	Machine learning based merged suitability classifier
ML_{MSP}	Machine Learning based Merge Speedup Predictor
ML_{SP}	Machine learning based speedup predictor
$MS\{A_{gpu}\}$	Merge suitability of jobs in A_{gpu} along with their predicted speedup
$\aleph$	Set of code features count in the kernel code of a job
N_x	Data size of a job x
N_y	Data size of a job y
P	Set of processors
P_C	The processing speed of a CPU
P_G(i)	The processing speed of i^{th} GPU
PS	Processing speed
Q	A family of set in which each element is a set containing elements from J
SP{xy}	Predicted speedup when a pair of jobs (x and y) in B are merged
SPC_i	Speedup of a job c_i on the suitable device
T_{cpu}	The total processing speed of all processors

Chapter 1

Overview

In 1965, Gordon E. Moore observed that the number of transistors in an integrated circuit has doubled every two years and this trend is set to continue for the next ten years [1]. This observation is known as "Moore's Law" and had held true for 40 years. As a result, software developers relied on the ever-increasing *Central Processing Speed* (CPU) speed due to Moore's Law to speed up execution of their softwares. Newer applications requiring more compute-power were developed that benefited from the growing clock speeds of new processors. Therefore, the developers relied on the hardware technology to speedup execution of compute-intensive applications.

Towards the end of 2004, transistors growth on an integrated circuit began to stall due to exponential power-consumption and heat dissipation. These problems occurred due to increasing clock frequency with every new generation of processors. To overcome these limitations, multi-core architectures were developed as a solution to problems faced by the single-core processors. In multi-core architecture, multiple similar CPUs are packaged together on the same integrated circuit. In this era, the architectural trend is to increase the cores of a single CPU or increase number of CPUs in the system to meet the computation demands of applications. The application developer can harness the full potential of multi-core architecture by introducing parallelism in their applications [2].

In the contemporary era, application developers cannot rely solely on the processor technology to speed up application execution. Instead, they have to re-engineer and parallelize their applications to exploit available computing power of multi-core processors. This gave rise to parallel programming paradigm where a single application is partitioned into parallel portions each executing on a separate processor-core. To parallelize application on multi-core CPUs, several frameworks and compilers are available such as *POSIX threads* (pthreads) [3], *OpenMP* (Open Multi Processing) [4], Intel's *Thread Building Block* (TBB) [5]. These frameworks usage have aided programmers to speedup the application's execution.

However, the usage of parallel programming models is becoming increasingly insufficient to meet the execution performance demands of many scientific applications due to physical limitations on increasing the number of cores. Moreover, multi-core systems usage is progressively failing to meet the power and energy budget in the data center and cloud environment. To overcome limitations of the multi-core architecture, recent years have witnessed the emergence of heterogeneous systems i.e., a computing system that consists of more than single kind of processor. Heterogeneous systems consist of a multi-core CPU and a co-processor. Co-processor may be *Graphical Processing Unit* (GPU), *Many Integrated Cores* (MIC), *Field Programmable Gate Array* (FPGA) etc. Among the co-processors, GPUs have become an integral part of most heterogeneous systems. The usage of GPU in heterogeneous systems ranges from general purpose computers to systems in data centers and super computers. According to a recent survey conducted by the TOP500 organization, 05 of the top 10 supercomputers include GPU in their system [6]. GPUs are generally favored in the heterogeneous systems due to providing high throughput for data parallel applications and excellent performance per watt ratio [7].

Open Compute Language (OpenCL) [8] and *Compute Unified Device Architecture* (CUDA) have become de facto industry standards to program GPUs for *General Purpose Programming on GPU* (GPGPU). Among the two, OpenCL has emerged as an industry standard to program heterogeneous devices. Due to its portable

nature as a single OpenCL application can be executed on many types of accelerators including GPUs, CPUs etc. [9], [10]. An OpenCL program comprises of two parts, host program and kernel function. Supervision of kernel function execution on accelerator devices (i.e., CPU, GPU etc.) is the responsibility of a host program that executes on a CPU device [11]. The kernel function deals with the compute-intensive data-parallel code. Within an OpenCL program, buffer creation, data transfer, and kernel mapping to the devices are manually managed by the application developer [12].

1.1 Scheduling on Heterogeneous Machines

Task scheduling is a non-trivial problem that requires best mapping of tasks to a processor so that the overall execution time of applications is reduced [13]. The scheduling decision becomes more difficult when dealing with a heterogeneous (CPU-GPU) system in which each compute unit has different characteristics. CPU has a limited number of powerful and complex cores that are generalized to execute different types of applications efficiently whereas GPU contains a large number of simplified cores that are mainly specialized to execute data-parallel portions of the program. Therefore when scheduling applications on CPU-GPU system, heterogeneity of computing devices should be considered to effectively map computation to processors. It should also be considered that while one application may enjoy considerable speed-up on GPU than on CPU, another application's performance might degrade significantly if assigned to GPU while attaining speed-up on CPU. Programmers normally assign tasks to either a CPU or a GPU that results in the other processing-unit remaining idle during execution.

1.2 Load-Balance in Heterogeneous Machines

Load-balanced scheduling aims to maximize resource utilization and increase machine throughput while minimizing the execution time. Load balancing is usually

achieved by using mechanisms such as work stealing and work-sharing. In work stealing, an idle core or processor (thief-device) acquires tasks from another processor (victim-device with heavy-load) [14]. In work sharing, scheduler maintains a centralized list of all pending tasks and idle resources (core or processor) to map tasks to cores or processor [15].

Achieving a load-balanced schedule of jobs in heterogeneous machines (based on CPUs and GPUs) is a challenging task [16] because of heterogeneous architecture, varying clock-speed, identifying code-parallelism, and presence of multiple memory levels. In a heterogeneous environment, an optimal load-balanced schedule of tasks represents the assignment of tasks to both processors in such a way that CPU and GPU complete their work at approximately the same time. Another aspect of the load-balance is to look to minimize the idle time of a processor during the program's execution. A load-balancing mechanism should incorporate architectural characteristics, data-transfer overheads, and application needs.

1.3 Scheduling Challenges

Scheduling of tasks to a processor is a NP-hard problem [17, 18] and therefore it is difficult to optimally schedule tasks to a device in polynomial time. Task scheduling becomes even more difficult when heterogeneous processors are involved with vastly different characteristics [19].

The present trend that is prevalent in the field of heterogeneous computing is to burden programmer with the scheduling of tasks. Programmers normally use the default scheduling strategy i.e., assigning the parallel portion of a program (kernel) to a GPU while CPU executes serial portion (kernel management) of a program. This tendency has led to the following important observations.

1. **Wastage of computing resources:** CPU remains idle while all the computation is performed by GPU although by using OpenCL, a program can be executed on both CPU and GPU. This is wastage of precious CPU resources

which is consuming power and energy but performing no useful task [19–24]. Although kernels are normally developed to be optimally executed on GPU, CPU can also provide good performance on kernels where little computation is performed and data size is small. The reason is that today's CPUs consist of the multiple number of cores, have higher clock frequency, have efficient cache hierarchy, and no data transfer overhead over PCIe interconnect (as is the case for GPU), etc. In a job pool consisting of multiple kernels with different computation and data requirement, this means that kernels with small computation requirement and data size can be scheduled to CPU while kernels with large computation requirement and data size can be scheduled to GPU in order to reduce wastage of CPU resources.

2. **Longer execution time (throughput):** As computation is only performed by GPU therefore, kernels in job pool takes a longer time to complete execution. Using this scheduling strategy, favorable throughput cannot be obtained from the system. Reason for this sub-optimal throughput is that CPU is not taking any part in Job Pool execution. Therefore, there is need for a scheduling strategy that can also schedule some kernels to CPU in order to reduce the overall execution time of a Job Pool. Kernels that are offloaded to CPU will be executed in parallel with kernels that are executed on GPU. In this way throughput of the system can be increased significantly.

3. **Load-balancing:** Severe load imbalance is observed between CPU load and GPU load due to CPU only management of the kernel execution on GPU and taking no part in the actual computation. The idle time that CPU spent while waiting for GPU to complete kernels execution is not desirable. Ideally, a scheduler is required that can schedule kernels to both CPU and GPU in such a way that both processors can complete processing at the same time. In this way energy consumption and heat dissipation due to idling processor are reduced but, more importantly, the execution time of Job Pool will also be reduced significantly [25, 26].

4. **Application device suitability:** Due to architectural differences between a CPU and GPU, some applications are better suited (i.e., have lower execution time) for CPU execution while others are better suited for GPU execution. Applications possessing a high level of task parallelism, high data transfer overhead, etc. perform better on CPU whereas applications having a high level of data-parallelism are better suited to GPU execution. If an application scheduler is unaware of the device suitability of an application, it may assign CPU suitable application to the GPU and GPU suitable application to the CPU. This will result in longer execution time of the job pool [27, 28].

Keeping in view above mentioned observations, it is clear that there is a need of a scheduler that can minimize execution time of job pool through load-balancing. If such a scheduler is developed, it will result in significant decrease in wastage of computational resources. Moreover, a load balancing scheduler will also lead to not only energy and power savings but also decrease in heat dissipation [29].

1.4 Thesis Objectives and Research Questions

While there is an abundance of scheduling schemes that have been discussed in the literature for assignment of tasks on heterogeneous CPU-GPU architecture, there is a shortage of scheduling schemes that considers job pool of kernels from different applications [19, 30]. Primary goal of scheduling schemes mentioned in literature is to split data of a single kernel between CPU and GPU in order to minimize execution time. To the best of my knowledge, only [27, 31] have discussed scheduling issues relating to job pool of OpenCL applications. Therefore, the first objective is to **develop a scheduling scheme for a job pool of OpenCL applications that can map job to heterogeneous devices in a load-balanced manner**.

Some applications are inherently suitable for CPU execution while others produce speedup when executed on a GPU due to architectural differences between a CPU

and a GPU. A smart job pool scheduler has to decide on the device suitability of an application from a job pool. Machine learning can be used to characterize applications into CPU suitable and GPU suitable using application code features. Therefore, the second objective is to **develop a scheduling scheme that can map jobs in a device suitable way while maintaining load balance across CPU and GPU considering speedup achieved by a job on the suitable device.**

Applications that are mapped to a GPU device assumes control of all the resources of the GPU. When a kernel is executed on a GPU with smaller data size, there is wastage of GPU resources because the kernel cannot keep all the GPU cores busy during its execution. Such resource wastage can be reduced if a GPU can be shared among two kernels. This will also result in decreasing the execution time of the job pool. However, some applications execution speedup when they share a GPU while others slowdown the application execution while sharing a GPU. An efficient job scheduler has to decide on which GPU mapped application can share the GPU resource to speed up their execution. Therefore, the third objective of the thesis is to **develop a resource sharing mechanism between two GPU mapped kernels while maintaining load-balance across heterogeneous devices.** The above-mentioned thesis objectives are used to achieve the goals of reducing job pool execution time, increasingload balance across CPU and GPU, and increasing system throughput. The research objectives of the thesis are summarized in a concise and coherent way below.

Research questions

1. How to develop a load-balancing job scheduler for a heterogeneous system?
2. How to reduce memory contention for two concurrently executing jobs?
3. How much performance gain can be achieved by device suitable mapping of jobs on the heterogeneous system?

4. How to fuse a pair of OpenCL kernels to increase GPU device utilization and reduce job pool execution time?

1.5 Contributions

In Chapter 3 the OSched scheduling heuristic that maps user jobs from a job pool in a load balanced manner to either a CPU or a GPU is presented. Load balance is achieved by contemplating the computation requirement of each job and computation power of each heterogeneous processor. The results show that the OSched reduces job pool execution time, increases system throughput while achieving load balance across CPU and GPU. Moreover, an enhancement to the OSched scheduling heuristic named as the E-OSched was also proposed in Chapter 3. The E-OSched further improves the results of the OSched by minimizing the memory contention in the main memory due to concurrently executing jobs. This contribution answers the research question 1 and research question 2 that were presented in section 1.4 of chapter 1.

Contribution (Chapter 3): *Khalid YN, Aleem M, Prodan R, Iqbal MA, Islam MA. E-OSched: a load balancing scheduler for heterogeneous multicores. The Journal of Supercomputing. 2018 Oct 1;74(10):5399-431.*

Chapter 4 presented the Troodon which is a machine learning based job scheduler, built on top of the E-OSched that considers device suitability of job during scheduling decisions. Device suitability of a job is determined using a machine learning based job classifier. The Troodon maps CPU suitable jobs to the CPU and GPU suitable jobs to the GPU while maintaining load balance across both the CPU and GPU. The results show that the Troodon reduces the execution time by 282% and 38% when compared to base-line scheduling heuristic and state-of-the-art scheduling heuristics respectively. This contribution answers the research question 3 that was presented in section 1.4 of chapter 1.

Contribution (Chapter 4): *Khalid YN, Aleem M, Ahmed U, Islam MA, Iqbal MA. Troodon: A machine-learning based load-balancing application scheduler for CPU–GPU system. Journal of Parallel and Distributed Computing. 2019 Jun 8.*

Chapter 5 presented the kernel fusing FusionCL scheduler that is built on top of the Troodon. The proposed scheduler reduces the execution time of the job pool by increasing GPU device occupancy through kernel fusing. A machine learning based kernel fusing classifier is used to decide on the pair of the kernel functions that, when fused together, will produce speedup over their serial execution.

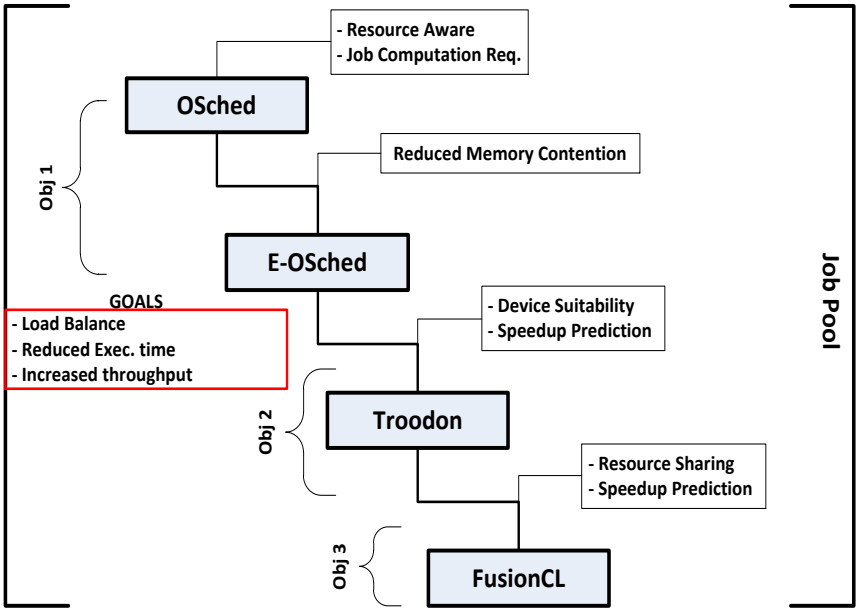

FIGURE 1.1: Research goals and research objectives

The experimental results show that the FusionCL reduces job pool execution time by 1.08X when compared to the state-of-the-art while maintaining load balance across both the CPU and the GPU. This contribution answers the research question 4 that was presented in section 1.4 of chapter 1. The research objectives, research goals and thesis contributions are presented in Figure 1.1.

1.6 Research Methodology

The research methodology of this thesis is similar to the one presented in [32]. This research was started with the goal of finding scheduling solutions for job placement in heterogeneous systems. In this regard, a systematic literature review (Chapter 3) was conducted that included research articles from top conferences and journals from the field of parallel computing. The reviewed conferences include IPDPS [33], SC [34], ISC [35], PACT [36], PPOPP [37], Euro-par [38], GPGPU [39], ICPP [40], ICPADS [41] whereas journal list include TACO [42], JPDC [43], TPDS [44], Super Computing [45], IJPP [46], Parallel Computing [47] etc. From literature review, it was observed that there is lack of scheduling schemes that deal with mapping of a job pool of heterogeneous jobs on a CPU-GPU systems (*problem awareness*).

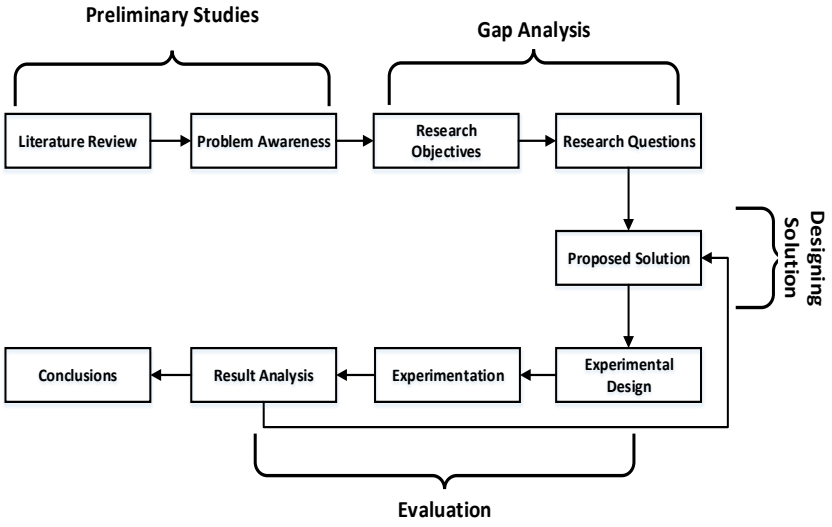

FIGURE 1.2: Proposed research methodology

This observation was used to design research objectives and formulate research questions (Section 1.4). Comprehensive solutions were designed to answer the research questions (Section 3.3, Section 4.3, Section 5.3). Experimental design was developed to test the validity of the proposed solution. In the end, experiments

were performed to prove the viability of the proposed solution (Section 3.4, Section 4.4, Section 5.4) to the research questions posed in Section 1.4. The overview of the adopted research methodology is presented in the Figure 1.2.

1.7 Thesis Organization

The thesis is divided into further five chapters. Chapter 2 introduces the necessary concepts used to accomplish the goals of this thesis. Moreover, a detailed scrutiny of the prior work related to job scheduling on heterogeneous systems is also part of this chapter.

In Chapter 3, the discussion is about the development of a load-balanced scheduling mechanism for a job pool of applications. Load-balance across a CPU and GPU is achieved by considering the processing requirement of the jobs and processing capabilities of heterogeneous devices.

Chapter 4 is focused on the use of machine learning in classifying jobs according to the device suitability. After jobs' classification, CPU suitable jobs are mapped to the CPU whereas GPU suitable jobs are mapped to the GPU. Load-balance is achieved by adapting the scheduling mechanism, presented in Chapter 3, to the device suitability problem that is tackled in Chapter 4.

In Chapter 5, a resource sharing mechanism is introduced that enable two applications to share a GPU for their execution. Machine learning is used to classify jobs as fusion suitable or fusion unsuitable. Thereafter, the best candidates, among all fusion suitable jobs are fused to increase the GPU device utilization. Furthermore, the scheduling mechanism, presented in Chapter 4, is adapted to the GPU device utilization problem that is presented in Chapter 5.

Chapter 6 provides concluding remarks about the research performed in this thesis. Moreover, a number of future research aspects are also provided for researchers to explore.

Chapter 2

Background and Related Work

This chapter presents the background knowledge that is necessary to familiarize the reader about the research conducted in this thesis. Moreover, the prior research work related to job scheduling on heterogeneous machines is also presented in this chapter.

2.1 Background

This section introduces the tools and techniques that are used by the research community to address the challenges related to scheduling in heterogeneous systems. Particular emphasis has been laid to introduce OpenCL, heterogeneous computing and an introduction to the classification of scheduling techniques that are used to map jobs in heterogeneous systems.

2.1.1 Heterogeneous Computing

Heterogeneous computing is a type of parallel computing can be defined as computing performed on systems consisting of more than one type of compute devices [31]. Heterogeneous systems can be further subdivided into single *Instruction Set Architecture* (ISA) and multiple ISA systems and are described below.

Single ISA Heterogeneous Systems

Heterogeneous systems that consist of a processor that has same ISA but operate at different clock frequencies can be labeled as single ISA heterogeneous systems. An example of such processors is Qualcomm Snapdragon 855 [48] that is an octa-core processor and consist of three types of ARM-based processing cores [49],[50].

1. 1 x 2.84GHz based on Cortex A76.
2. 3 x 2.42GHz based on Cortex A76.
3. 4 x 1.8GHz based on Cortex-A55.

As these processors are mostly used in mobile platforms, they are optimized to give better energy efficiency to increase battery life and provide energy efficiency. In these systems, cores are not optimized to perform specific kind of tasks i.e., compute-parallel or data-parallel tasks.

Multiple ISA Heterogeneous Systems

Heterogeneous systems that consist of processors with compute cores possessing different ISAs can be labeled as multiple ISA heterogeneous systems. The different ISA compute cores can be integrated on the same die (i.e., AMD Radeon E2-7110) [51] or can be connected through PCIe bus (i.e., systems consisting of a discrete GPU). Typically, heterogeneous ISA systems consist of a multi-core CPU and many-core GPU that is used as a co-processor to accelerate the execution of a data-parallel application. Mostly, CPUs are better suited to perform latency-sensitive tasks and incorporate architectural advances such as *branch-prediction*, *out-of-order execution*, and *super-scalar capabilities* [52]. Moreover, each CPU core is organized in a MIMD (Multiple Instruction Multiple Data) ways and are optimized to perform compute-parallel tasks.

Whereas, many-core GPUs are more suited to perform *data-parallel* and *throughput-sensitive* tasks [52] due to the inherent massive multi-threading capabilities [53].

The GPU cores are organized in a SIMD (Single Instruction Multiple Data) ways and are optimized to perform data-parallel tasks.

Among the two, heterogeneous ISA systems are widely used by the scientific community to accelerate the execution of applications [11, 54]. The focus of this research is on multi-ISA heterogeneous system that consists of a CPU and one or more GPUs.

2.1.2 OpenCL

Open Compute Language(OpenCL) [8] has emerged as an industry standard to develop data-parallel applications for heterogeneous multi-/many-core architectures. The OpenCL platform model consists of a host (which is always a CPU) and one or more compute devices (which may also be a CPU or a GPU or any other accelerator) as shown in Figure 2.1. Each compute device consists of one or more compute units which are further divided into one or more processing elements.

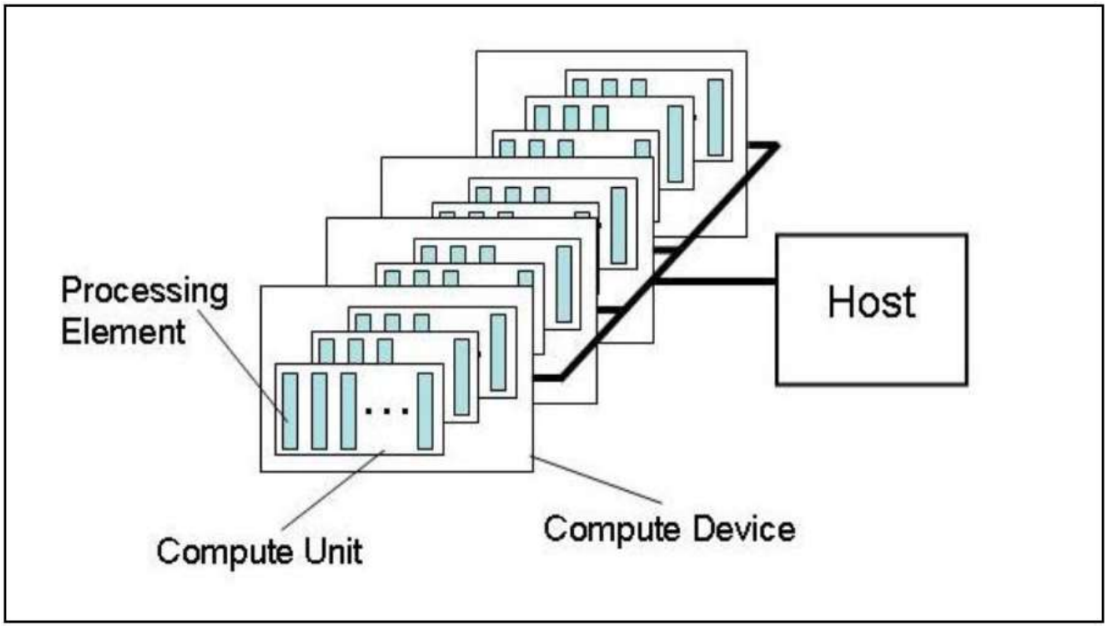

FIGURE 2.1: OpenCL platform model [11]

Major vendors in the computer hardware industry such as Intel [55], AMD [56], and NVIDIA [57], etc., support OpenCL platform. A CPU in the OpenCL execution

model as depicted in Figure 2.2. A CPU always executes the serial portion of an OpenCL program (known as the host). The host manages the overall execution of an OpenCL program. Its tasks include identifying possible devices for kernel execution, preparing data for transfer to OpenCL devices, creating and managing command queues etc. Moreover, creating and launching threads on a compute device is also performed by the host. The data-parallel portion (known as the kernel) is executed on a compute-device. The compute device can be on a CPU, a GPU, a FPGA, or any other supported accelerator device.

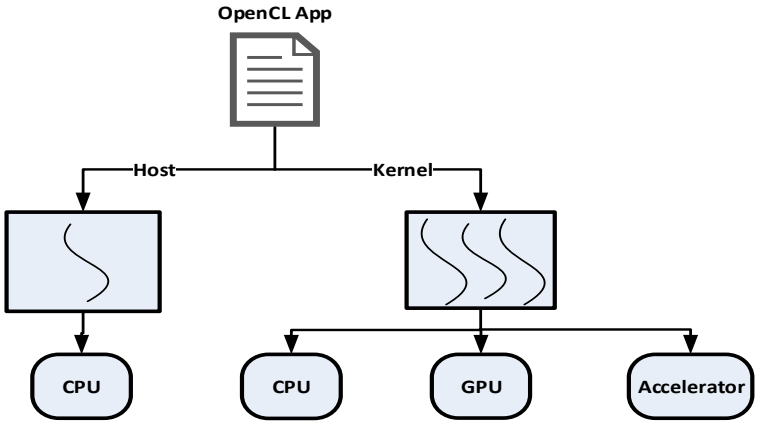

FIGURE 2.2: OpenCL execution environment

2.1.3 Scheduling techniques for heterogeneous systems

There are a plethora of techniques that are concerned with the scheduling of tasks on heterogeneous systems. These techniques can be broadly divided into three categories: *static*, *dynamic*, and *hybrid*. The above-mentioned three techniques along with their advantages and disadvantages are discussed in the subsequent subsections.

Static Scheduling

In static scheduling techniques, task mapping is performed before runtime and is fixed during the execution. Static scheduling is suitable when amount of work

to be performed or data to be processed is known apriori. Its advantages are simplicity and low scheduling-overhead. Major disadvantages of this technique are that optimal task partitioning and scheduling cannot be done before execution because of applications' characteristics and data characteristics (at compile time) are unknown.

Dynamic Scheduling

In dynamic scheduling, task mapping to a compute-unit is done at runtime. The major advantage of dynamic task scheduling is that decision to map the task is more favorable (considering the runtime attributes of the application and machine). Scheduling decisions can be adjusted during the execution of a program. Major cons of this scheduling are the increased complexity and higher scheduling overhead.

Hybrid Scheduling

Hybrid scheduling techniques incorporates both static and dynamic approaches for scheduling tasks. The hybrid technique takes advantage of both static and dynamic techniques. As the first step, generally, offline profiling (static) is used to extract program features, which are employed in scheduling decisions. Purpose of offline profiling is to reduce dynamic scheduling overhead. The major objective of this technique is to minimize load imbalance to improve the overall application performance.

2.2 Related Work

This section details the related work specific to task/job scheduling in heterogeneous systems. Section 2.2.1 presents scheduling techniques for heterogeneous

multi-core machines. Section 2.2.2 presents the use of machine learning for scheduling purpose in heterogeneous machines. finally, Section 2.2.3 presents kernel fusion techniques to improve GPU device utilization.

2.2.1 Scheduling Schemes for Heterogeneous Multi-cores

Currently there exists a plethora of scheduling techniques for heterogeneous multi-core machines [17, 58–64]. Some of these [59–62] have split either a single application kernel among CPUs and GPUs or have attempted to schedule a pool of applications [63–65] to minimize the overall execution time of the jobs and to improve the device utilization.

In [66], authors presented *Choose-between-Accelerate-the-fastest-and-Best-fit* (CAB) and *Greedy-Increase* (GrIn). The CAB is a mathematical model that consider performance and energy constraints for favorable task scheduling on a heterogeneous multi-core system. *GrIn* algorithm is an implementation of the general case of *CAB* mathematical model. According to the authors, each application program consists of a set of tasks (both serial and parallel). First, minimum and maximum throughput for each task type (for all the heterogeneous processors) is calculated using the *CAB* model. The tasks consuming minimum system energy and having minimum *Energy Delay Product* for a processor represents the highest throughput for that processor. The GrIn scheduling algorithm then assigns a task (using a greedy approach) to a processor in order to maximize system throughput. This research is analogous to our work for considering processing speed of processors; however, memory contention aspect has been ignored.

In [61], a system named *Qilin* for programming heterogeneous devices has been presented. At first, the *Qilin* system partitions the kernel data into two parts, which are further mapped onto CPU and GPU devices. The *Qilin* stores the execution performance of an application in a database. Afterwards, the *Qilin* uses the previously stored execution performance data of a certain application to project current execution performance to form scheduling decisions accordingly. In the

case of any hardware changes in the system, the *Qilin* initiates a new training session for altered hardware configuration. In contrast to *Qilin*, our scheduling algorithm does not require offline profiling and code change of the OpenCL applications.

In [67], a profiling-based kernel scheduling method is proposed in which, an off-line profiling has been performed to obtain execution time and data dependencies of an OpenCL kernel. Afterwards, a greedy algorithm has been employed to schedule the kernel on a computing device (i.e., CPU or GPU) that has resulted in the least executions time with no dependency violations. However, the proposed algorithm only schedules kernels of a single application. Whereas, our proposed scheduler does not require off-line code profiling and assigns pool of jobs to CPUs and GPUs.

StarPU [68] is a run time system that provides a unified execution environment for executing numerical kernels on heterogeneous architectures. The StarPU adopts simple scheduling policies for tasks distribution on heterogeneous architecture. The employed *Greedy policy* (priority based) assigns a task to a processor as soon as it becomes idle. The tasks get executed based on their priority preference (i.e., high priority task will be executed first). *No-prior* policy [68] is same as greedy; however, it does not consider task priority while assigning a task to the processor. The employed *Ws-policy* also assigns task greedily that rely on work-stealing principle. In *w-rand policy*, each worker-device (i.e., processor) is assigned an acceleration factor. A task is assigned to that worker-device whose probability is proportional to acceleration factor ratio. In *heft-tm* policy, a task is assigned to a computing unit which minimizes the task's execution time (considering the already assigned tasks on that compute unit). The StarPU's scheduling policies only consider *numeric kernels* (such as Matrix Multiplication, LU decomposition) while our proposed scheduling heuristic is capable to schedule different application kernels and employing multi-GPU configurations.

In [62], a run time system has been proposed that schedules legacy kernels (compute intensive part of applications) on a heterogeneous machine considering both the execution history and data-transfer overheads. Authors have proposed a run

time system that intercepts function calls of a kernel and schedules them on either a CPU or a GPU. Our proposed approach is distinguished as compared to the scheduler presented in [62] in terms of no overhead of offline profiling. Moreover, our proposed scheduler is capable of scheduling several data-parallel applications as compared to the single application scheduling approach adopted by [62].

HDSS [69] scheduling mechanism improves the execution time of a kernel via partitioning the workload between CPU and GPUs. Initially, profiling phase is utilized to learn the computing power of each processor by assigning a small number of loop iterations to it. In adaptive phase, remaining loop iterations are then assigned to each processor according to their processing speed. As a result, the HDSS mechanism ensures a load-balanced execution on heterogeneous machines. In contrast to the HDSS, our approach does not require job splitting and code change.

In [70], a runtime system has been proposed that allocates OpenCL based data-parallel tasks to a CPU or a GPU. In this system, a kernel is divided into several tasks. Afterwards, a profiler is employed to record the execution time of the scheduled tasks against each processor. Whenever a task arrives for execution, it is scheduled to either a CPU or a GPU device based on the stored performance profile. However, this system requires profiling and code changes which is not the case with our approach.

The algorithm presented in [26] divides a workload (based on a single kernel) into chunks and schedule the chunks to either a CPU or a GPU. The scheduler [26] starts with the assignment of uniform small-sized chunks for each device. The chunk sizes are increased/decreased exponentially according to the previous executions of those chunks. In such a way, the faster computing devices are loaded with larger sized chunks and slower devices are loaded with small sized chunks which ensure a good load balance. In contrast to this approach, our scheduler is memory-contention aware that ensures lower execution time along the load balanced execution.

In [64], *Estimated Execution* Time of an application is used to decide allocation to a CPU or a GPU. This technique requires a training period in which the execution

history of an application is observed. When an application arrives for execution, it is mapped to a device which is capable to complete the job earlier. Application completion time is estimated by considering the total execution time of the application and the execution time of the prior scheduled application on that device. In contrast, our proposed schedule does not require offline profiling. Moreover, our proposed technique supports multi-device configuration which is not provided by this heuristic [64].

In [17], historical run time data is used to schedule an application to a suitable device that can execute the job earlier. If that suitable device is busy executing other applications then the application is scheduled on a slower device. In contrast to this approach, our proposed scheduler is capable of employing multi-GPU configurations.

In [71], authors proposed an algorithm that schedules applications on heterogeneous multi-cores by considering the *system specifications*, the historical *run time data*, and *current system state*. In contrast to this approach, our scheduler considers both the computational requirements of an application and computing capabilities of devices. Moreover, our proposed approach schedules pool of applications ensuring a load-balanced, memory contention free execution.

In [72], three different scheduling algorithms have been proposed. Two algorithms are based on different variants of *First Free* and *First Come First Serve* heuristics. The third algorithm assigns tasks to a processor via considering the execution history. The performance history is then harnessed to predict waiting time for a task on a certain processor. On the other hand, our proposed scheduler minimizes the memory contention issue and does not require code changes.

In [73], an off-line prediction model is proposed to dynamically partition tasks between a CPU and a GPU. Machine learning techniques, based on *Artificial Neural Network* (ANN), are used to derive prediction model for task partitioning using the *Insieme* [74] run time system. This model depends on static code features (e.g., *OpenCL built-in functions*) and dynamic input sensitive features (e.g., *data-transfer* size of the split-able buffer) for training phase. The *Principal Component*

Analysis procedure is then used to further optimize the task partitioning. After that, the partitioned tasks are assigned to CPU and GPU for execution. In contrast to this approach, our proposed scheduler does not require data-splitting and off-line training.

In [75], authors presented a *Single Kernel Multiple Data* scheduling heuristics that employs splitting of a kernel across all available devices. For partitioning, the data represented by the *ND Range* is flattened and its subset is assigned to each computing device. The partial subset of results are obtained and merged in a seamless manner to produce the output. To ensure load-balanced distribution across multiple heterogeneous devices, the execution speed and data transfer cost to each device are considered. However, our proposed scheduler does not require data-splitting and is capable of scheduling a pool of jobs considering both the application and device's computing requirements.

In [76], authors presented an OpenCL run time system called *FluidiCL* that is capable of distributing and executing an OpenCL program using both CPU and GPU devices. This scheme does not require any prior offline training. It automatically handles data-transfers and results aggregation without involving the programmer. For data-distribution, the n-dimensional *ND Range* of a work group is flattened and used as a unit of allocation for execution. A kernel mapped on a GPU device starts executing the flattened work group from one end while on a CPU device a sub-kernel starts executing another part of the work group. In contrast to this approach, our proposed scheduler is capable of executing multiple jobs simultaneously (using CPUs and GPUs) providing a better application response time.

In [60], authors presented a two-phase scheduling algorithm named *Multi Kernel on Multi Devices* (MKMD) to schedule multiple kernels of an application. In the first phase, a kernel is assigned to that device which minimizes its execution time and data-transfer cost. In the second phase, a kernel is split into sub-kernels and rescheduled to heterogeneous processors to improve device utilization. The MKMD heuristic builds a regression model for each kernel considering different

input sizes and device mappings. The regression model is then used to decide whether a kernel should be split or mapped fully on a certain compute device. However, our proposed scheduler does not split kernels across heterogeneous devices and does not require offline profiling.

In [77], two scheduling strategies are proposed to partition the kernel workload between a CPU and a GPU. In naive profiling step, a small portion of work is assigned to both CPU and GPU device and the execution performance of both devices is analyzed. Afterwards, the collected profiling data is utilized for further jobs assignments. On the other hand, our scheme does not require kernel splitting and can efficiently schedule pool of kernel jobs.

In [27], authors presented a machine learning based task scheduling scheme to schedule multiple kernels from different programs. The focal aspect of this technique is to enhance the system throughput and decrease the average turnaround time. The distinguishing factor of this scheme is that authors have contemplated scheduling of multiple OpenCL applications on a heterogeneous platform. It considers (a) *static code features* such as a number of instructions, load/store operations etc., and (b) *run time features* such as input size, etc. Using both the static/dynamic features and a predictive model the data-parallel programs are categorized into high and low speedup. The high speed-up programs are scheduled on a GPU device and the low speed-up programs are scheduled on a CPU. In contrast to this technique, our proposed scheduler assigns OpenCL kernels considering both the application's computing requirements and the device's computing capabilities.

In [78], a dynamic scheduling scheme is presented for applications that are characterized by (a) *generalized loop reduction* and (b) *structured grid computation*. These applications consist of data-parallel loops, which are divided into chunks. These chunks are further divided into *chunk-lets*, which is a basic unit of assignment to a CPU or a GPU. Using the FCFS assignment, a CPU is assigned a single chunk-let at a time while several chunk-lets are combined and assigned to a GPU. Employing this methodology, the faster computing devices (such as GPUs) are allocated with more tasks as compared to the slower computing devices (i.e.,

CPUs). This task-mapping scheme ensures a good load-balance across computing devices. In contrast to the proposed scheduling technique [78], our scheme does not require kernel splitting and is used to schedule pool of jobs.

In [79], a two-phase scheduling scheme called *CAP* is proposed for heterogeneous parallel machines. In the first phase, a static partitioning method is used to distribute a small portion of workload equally to both a CPU and a GPU. The execution time of the assigned workload is analyzed. Considering the execution time of the previous assignment, the amount of work is increased twofold on the faster device. The scheduling using increased workload is performed until the variance between current and previous executions becomes less than a predefined threshold. In the second phase, the remaining workload is divided among compute-devices according to the sampling performed in the first phase. In contrast to this scheme, our proposed scheduler does not require prior analysis phase.

In [31], a machine learning based heuristic is proposed that employs OpenCL code-features (such as *instructions, blocks, math functions*) to determine device suitability. Moreover, certain code-features (e.g., *branch ratio, data size*, etc.) are utilized to determine whether to schedule a kernel in isolation (to a GPU) or to combine it with other kernels to improve execution performance. In contrast, our proposed scheduler considers devices' computing capabilities to balance the load of job pool and a memory contention free execution of multiple data-parallel applications. *Maat* [22] library distributes a single kernel workload across heterogeneous devices in a load balanced manner. *Maat* includes a set of load balancing techniques: (a) *static* where each device is assigned workload according to its computing power; (b) *dynamic* where the workload is divided into equal sized packages and assigned to the free devices; (c) *guided* method is used to reduce the synchronization points (required during *dynamic* method); and (d) *adaptive* method is based on [26] that divides workload in small chunks and assign them to devices to determine execution time of data on each device. The adaptive method assigns the rest of the data to devices statically according to the ratio of the obtained execution time. The proposed technique differs from our technique as it is oblivious

to device-suitability and is not applicable to multiple OpenCL workloads mapped on a different device (executing concurrently).

In [80], a hybrid scheduling approach is presented that schedules a job in a load balanced manner on a heterogeneous CPU-GPU system. In the two-phase scheduling technique, first (i.e., *static phase*) partition ready OpenCL kernels are developed from the original kernel for mapping on different heterogeneous computing devices. The execution time of partition ready kernels is noted and then in the *dynamic* phase, the granularity of partitioned kernels is adjusted in order to achieve load balance (for iterative kernel execution). In contrast to our proposed scheduling scheme (i.e., *Troodon*) it does not consider device suitability and speedup prediction for the scheduling decisions. In [81], a scheduling mechanism named *E-OSched* is presented as a run time system that achieves near optimal device utilization through load balanced execution of the job pool. Using the *E-OSched*, first, the processing requirements of all jobs in a job pool are determined and jobs are sorted in ascending order (shortest sized jobs first) of the processing requirements. Jobs requiring low processing are scheduled to CPU and the high processing requirement jobs are mapped to the GPU device. Jobs mapping to CPU and GPU is performed according to the ratio of the device's computational power. In contrast to our proposed scheduling mechanism, the *E-OSched* does not consider the device suitability and relative speedup for scheduling which may result in important performance losses.

In [82] and [83], the authors suggest that a user-defined command queue mapping to the computing device is not efficient and requires a thorough understanding of the heterogeneous architecture and the kernel characteristics. To relieve the programmer from the burden of mapping command queues to devices, the authors proposed an extension (named *MultiCL*) to OpenCL API that decouple command queue from devices. Using *MultiCL*, the programmer is only concerned with developing parallel applications and scheduling of command queues to devices is handled by *MultiCL*. *MultiCL* supports two global scheduling policies for command queues—device mapping: 1) *Round Robin*, and 2) *Autofit*. In *Round Robin*-based command queue scheduling, the tasks are mapped on the next available device.

In *Autofit*, an optimal device for a command queue is decided using static device profiling, dynamic kernel profiling, and dynamic device mapping. The approaches presented in [82] and [83] are more suitable for task-parallel workload whereas our proposed approach is appropriate for data-parallel workloads too.

Choi et al. in [64] highlight the importance of the device selection for attaining a good performance in a heterogeneous multicore machine. The authors presented an execution estimation-based model that requires execution history of applications for training. A new task is scheduled for a device that could complete the job earlier. The total execution time of the application (on that device) and the execution time of the currently executing application are used to estimate finish time of the new application. In contrast, our proposed model use device suitability and relative speedup for mapping jobs to most appropriate devices.

In summary, most of the contemporary state-of-the-art techniques are either concerned with the single kernel based scheduling or limited to the scheduling of certain kinds of applications. Most of them require an off-line training or profiling along with some code change. To the best of our knowledge, there does not exist any technique that emphasizes on the load-balanced scheduling of job pool without requiring profiling and off-line training. Considering all aforementioned deficiencies, our proposed scheduling heuristic contemplates both the computational requirements of a job, computing capabilities of the device, and memory contention-free mapping of OpenCL jobs.

2.2.2 Heterogeneous Scheduling Based on Prediction Models

Machine Learning (ML) based application scheduling is considered one of the powerful tool for optimizing parallel programs execution [27, 58, 63, 73]. The ML prediction model is first trained using past experiences and then the trained ML model exhibits an adaptive behavior (appropriate mapping) for different computing platforms and applications.

In [58], *int operations, float operations, barriers, work items* etc., based code features have been extracted from OpenCL kernels during the compilation time. These extracted features are passed to machine learning based predictive model SVM (*Support Vector Machine*) to form scheduling decisions i.e., whether to map a kernel to a CPU, GPU or to partition the kernel among available computing devices. In contrast to our proposed work, the authors in this study presented a scheduling algorithm for scheduling a single OpenCL kernel across heterogeneous devices; whereas, our proposed approach is employed to schedule job pool of applications. This work has further been extended by [84], wherein a machine learning based scheduling approach is proposed. It works on the basis of branch divergence (i.e., *if else statements*) for accurate job scheduling. Authors have emphasized that the branch divergence plays a critical role in application execution; therefore, this code feature should be employed while building a machine learning model for the scheduler. They proposed the scheduler based on offline profiling and training phase with the input of the extracted code-features. In contrast to their proposed scheduler, our technique does not require offline profiling and training. However, it has not ensured load-balanced scheduling of tasks.

Grewe et al. in [58], proposed an ML-based partitioning scheme for OpenCL programs. At the compile time, the model extracts the static code features (i.e., *int operations*, *float operations*, *barriers* etc.). After that, pre-trained ML model (based on Support Vector Machine) is utilized to predict whether to map a kernel to a CPU, to GPU or to partition the kernel among available computing devices. The trained model for GPU predicted with an accuracy of up to 91% and up to 95% accuracy for CPU related predictions. In contrast, our proposed ML scheduling model not only predicts device suitability but also relative speedup gain for the available computing devices.

In [73], Kofler et al. proposed an *Artificial Neural Network* (ANN) based prediction model. The main concept implemented in this model is to dynamically partition the given task between a CPU and a GPU. Authors use *Insieme* [74] source-to-source compiler to translate a single-device kernel code into multi-device kernel code. The feature set includes static code features (i.e., *scalar int operations*,

vector float operations, etc.) and dynamic input based sensitive features (such as *Data size of the split-able buffer*, etc.). The partitioning of the task is further improved using Principal Component Analysis. In contrast to this approach [73], our proposed scheduling mechanism maps job pool of heterogeneous applications considering the optimal or most suited device for mapping the compute kernel.

Troodon [65] is a machine learning based multi-job scheduler for heterogeneous systems. The *Troodon* [65] is built on top of the *E-OSched* [81] and include device suitability (i.e., CPU suitable or GPU suitable) classification of jobs and speedup prediction in the scheduling decisions. The *Troodon* [65] first classifies each submitted job as CPU suitable or GPU suitable. At the same time, the *Troodon* [65] also predict the expected speedup gain when a job is executed on a suitable device. All the CPU suitable and GPU suitable jobs are then separately sorted according to speedup. Thereafter, the *E-OSched* [81] scheduling mechanism is used to map jobs to suitable device for execution.

In [27], Wen et al. argue that in order to get increased system throughput and decreased turnaround time, there is a need to determine device-suitability of different applications for scheduling. The authors trained an ML-based model using code feature (i.e., *number of control instructions*, *float operations*, *int operations* etc.) and run time features (i.e., output size, global work size etc.). The trained ML prediction model maps high-speedup tasks (compute kernels) to GPU device whereas low-speedup tasks are mapped to a CPU device. The support vector machine (employing a *radial base kernel*) is used for developing the prediction modeling. In addition to the device-suitability, our proposed scheduling scheme also considers predicted speedup of an application when executed on a suitable device.

Gregg et al. in [15] emphasized that the mapping of applications to a faster device (i.e., GPU only) mostly leads to a load imbalance problem. Moreover, slower computing resources (i.e., CPUs) remain mostly underutilized which cause excessive contention on the faster devices. Therefore, the authors suggest that by mapping

some tasks to slower device results in increased system throughput. Task mapping to the slower device is decided based on historical run time data. Whenever a faster device is busy executing a task, the slower device is utilized by mapping an arriving task. In contrast to this scheduling policy, our proposed technique considers both the devices' computing power and applications' processing speed for task mapping.

In [63], Wen et al. argue that improving GPU utilization by executing OpenCL kernels concurrently can lead to improved performance. Authors in [63], proposed a decision tree based ML prediction model to determine whether an application kernel required to be scheduled separately on a GPU or it should be combined with other compute kernels to improve GPU utilization. The feature set used for the predictive model includes features such as *compute instruction ratio*, *memory instruction ratio*, etc. in addition to the static (i.e., number of load/store operations, blocks, control instructions) and dynamic (i.e., input and output size) features of [27]. In the first step, the prediction model separates OpenCL kernels into CPU and GPU suitable classes. Next, for the GPU suitable kernels, it is predicted whether or not to merge two GPU kernels to improve GPU utilization. In contrast, our proposed scheduler considers the device's computing capabilities and applications' processing requirements to schedule tasks in a load balanced manner. Table 2.1 demonstrates the summary of all the scrutinized state-of-the-art techniques presented in Section 2.2.1 and Section 2.2.2.

Please note that in Table 2.1 **RF** is used as an abbreviation for *Random Forest*, **GB** for *Random Forest*, **SVM** for *Support Vector Machine*, **DT** for *Decision Tree*, **RS** for *Runtime System*, **API** for *Application Programming Interface*, **MW** for *Middle Ware*, **NA** for *Information Not Available* respectively. These abbreviations are used for summarization in Table 2.1.

TABLE 2.1: Summary of literature review techniques

[Reference]year \ Attributes	Scheduling Type	Job Allocation	Load Balancing	Resource-Aware	Implementation	Data Size Consideration	Multi-GPU Support	Code Feature Extraction	Machine Learning Usage (technique)	Device Affinity	Speedup Prediction
[65]2019	Static	Job Pool	Yes	Yes	RS	Yes	Yes	Yes	Yes (RF & GB)	Yes	Yes
[81]2018	Static	Job Pool	Yes	Yes	RS	Yes	Yes	No	No	No	No
[66]2017	Static	Job Pool	Yes	Yes	RS	No	No	No	No	No	No
[84]2016	Static	Single job	No	No	RS	Yes	No	No	No	No	No
[61]2009	Dynamic	Single job	Yes	Yes	API	Yes	No	No	No	No	No
[67]2012	Dynamic	Single job	No	Yes	RS	No	No	No	No	No	No
[68]2011	Dynamic	Single job	Yes	Yes	API	No	NA	No	No	No	No
[62]2010	Dynamic	Single job	No	No	RS	Yes	No	No	No	No	No
[69]2013	Dynamic	Single job	Yes	Yes	API	No	No	No	No	No	No
[70]2011	Dynamic	Single job	Yes	Yes	Lib	Yes	Yes	No	No	No	No
[26]2013	Dynamic	Single job	Yes	Yes	RS	No	Yes	No	No	No	No

Table 2.1 Summary of literature review techniques

[Reference]year / Attributes	Scheduling Type	Job Allocation	Load Balancing	Resource-Aware	Implementation	Data Size Consideration	Multi-GPU Support	Code Feature Extraction	Machine Learning Usage (technique)	Device Affinity	Speedup Prediction
[64]2013	Dynamic	Single job	No	Yes	RS	No	No	No	No	No	No
[71]2010	Dynamic	Single job	No	Yes	RS	No	No	No	No	No	No
[72]2009	Dynamic	Job Pool	Yes	No	RS	No	No	No	No	No	No
[73]2013	Dynamic	Single job	No	No	RS	Yes	Yes	No	No	No	No
[75]2013	Dynamic	Single job	Yes	Yes	MW	Yes	Yes	No	No	No	No
[22]2016	Hybrid	Single job	Yes	Yes	Lib	No	Yes	No	No	No	No
[80]2016	Hybrid	Single job	Yes	No	RS	No	No	No	No	No	No
[58]2011	Static	Single job	No	No	RS	Yes	No	Yes	Yes (SVM)	No	No
[82]2015	Hybrid	Single job	No	Yes	RS	No	Yes	No	No	No	No
[83]2016	Hybrid	Single job	No	Yes	RS	No	Yes	No	No	No	No
[17]2011	Dynamic	Job Pool	Yes	Yes	RS	Yes	No	No	No	No	No
[60]2015	Dynamic	Single job	Yes	No	Lib	Yes	No	No	No	No	No

Table 2.1 Summary of literature review techniques

[Reference]year / Attributes	Scheduling Type	Job Allocation	Load Balancing	Resource-Aware	Implementation	Data Size Consideration	Multi-GPU Support	Code Feature Extraction	Machine Learning Usage (technique)	Device Affinity	Speedup Prediction
[77]2014	Dynamic	Single job	Yes	Yes	RS	No	No	No	No	No	No
[27]2014	Hybrid	Job Pool	No	No	RS	Yes	No	Yes	Yes (SVM)	Yes	No
[63]2017	Hybrid	Job Pool	No	No	RS	Yes	No	Yes	Yes (DT)	Yes	No

In summary, most of the related work is limited to a specific type of application or perform scheduling of a single OpenCL kernel. Moreover, several other research efforts require code splitting and profiling of the applications. Techniques pertaining to the scheduling of job pool of applications are either oblivious to processing speed of computing resources (i.e., CPU and GPU) or do not consider device-suitability or predicted speedup in scheduling decisions.

2.2.3 Kernel Fusion Techniques

There is extensive work that has been conducted regarding task scheduling on heterogeneous multi-cores. Some of these schemes [59, 69] employ data-splitting mechanisms (between CPU and GPU devices) while the other focus on a shared usage of CPU and GPU device in multi-job scenario [63, 64, 81]. Due to the lack of operating system's support for GPUs, in the recent years, some researchers [85,

86] have focused on sharing GPU device among multiple concurrently executing applications.

In [87], it is observed that the concurrent execution of multiple kernels often does not produce significant speed-up. due to contention-based execution. Some of the reasons are device contention, serialization (i.e., kernels are launched concurrently, however, in reality, are executed sequentially) due to lack of enough GPU resources for the kernel execution, serialization due to memory transfer, serialization due to long executing kernels, etc. To overcome the above-mentioned issues, the authors proposed a mechanism named *Elastic Kernels*. First, the kernels which are to be executed concurrently are transformed into a single (named as *Elastic*) kernel using source-to-source transformation. The transformed kernel also decouples physical resource-allocation from thread blocks of the original kernels. Afterward, the authors proposed scheduling techniques such as *Median* and *Equal* that decide on the number of GPU resources which can be assigned to each thread block of the original kernels.

In [63], Wen et al. argue that improving GPU utilization by executing OpenCL kernels concurrently can lead to improved performance. Authors in [63] proposed a decision tree based ML prediction model to determine whether an application kernel required to be scheduled separately on a GPU or combined with other compute kernels to improve GPU utilization. The feature set used for the predictive model includes features such as *compute instruction ratio*, *memory instruction ratio*, etc. in addition to the static (i.e., number of load/store operations, blocks, control instructions) and dynamic (i.e., input and output size) features of [27]. In the first step, the prediction model separates OpenCL kernels into CPU and GPU suitable classes. Next, for the GPU suitable kernels, it is predicted whether or not to merge two GPU kernels to improve GPU utilization. In contrast, our proposed scheduler uses expected to speed up achievable due to merged execution of the kernels to achieve load balance between heterogeneous processors.

MaxPair [85] is a kernel merging technique that uses graph based method to decide whether to merge two kernels or execute them separately. Distinct kernels

represent nodes whereas the edges between kernels represent the speed-up that can be achieved by kernel merging. The proposed technique is different from other kernel merging techniques which generally uses a greedy approach to find a suitable candidate for merging. Whereas, MaxPair adopts maximum matching pair algorithm to find the best candidate for merging that can produce the highest speed-up.

In [88], the authors present the accelOS, a portable and transparent runtime and Just-In-Time compiler that share GPU resources among two kernels without kernel merging. This is done by determining the workgroup requirements of multiple executing kernels that need to share GPU resources and then dynamically reduces work of each concurrently executing kernel so that all kernels can be executed concurrently executed on GPU and each kernel have an equal share of GPU resources.

To improve GPU resource utilization through concurrent kernel execution, a runtime system named as *kernelet* is presented in [86]. The *kernelet* dynamically divides a kernel into sub-kernels (named as *slices*). Afterward, a markov chain based performance model is used to co-schedule kernel slices which complement one another (i.e., co-scheduling slices from a compute intensive and memory intensive kernels). Afterward, a greedy algorithm is used to co-schedule those slices of the selected kernels that produce the highest performance gain according to the developed system model (Markov chain based).

In [89], a concurrent kernel execution method that is focused on improving energy efficiency through concurrent kernel execution is proposed. Similarly to [86], the kernel is divided into multiple slices. Now for every possible pair of concurrent kernels and their slices ratio GOPS (Giga Operations Per Second)/watt, and the corresponding frequency of GPU, is estimated using Neural Network. This process is also repeated for sequential executions of the kernels. Then, the kernel pair with highest energy improvement is scheduled to GPU. If there is no pair of kernels that provide energy improvement then kernels are executed using FIFO policy.

In [90], the authors argue that data transfer between CPU and GPU should also be considered when using GPU as a shared resource for executing multiple kernels.

For this purpose, authors present *CUDA Multi-Application Scheduling* (CuMAS) which considers PCI-e uplink and PCI-e downlink along with GPU as a shared resource. In contrast to GPU space sharing (for concurrently executing kernels), the proposed scheduling scheme allows only use of one of the three shared resources i.e., PCI-e uplink, GPU, or PCI-e downlink to overlap execution with data transfers. CuMAS uses profiling and timing models to estimate kernel execution time and data transfer time of a kernel. On the basis of execution time and data transfer time information, the CuMAS scheduler decides how to reorder the execution of CUDA applications in task queue so that their data transfer and execution time can be overlapped.

In summary, there is a lack of scheduling heuristics that use predictive modeling to determine fusion suitability of a pair of OpenCL kernels while ensuring load balance across heterogeneous devices. To the best of our knowledge, this research is the first effort that uses machine learning based predictive modeling to determine fusion suitability of jobs and the predicted speedup attainable due to fused execution. Moreover, the proposed approach is also able to a high degree of load balance during the job pool's execution on heterogeneous devices.

Chapter 3

Resource-Aware Load-Balanced Scheduling for Heterogeneous Architectures

3.1 Introduction

To execute compute-intensive applications, the dependence on a single processor does not play a consequential role in maintaining high speedup execution. Over the past few years, the paradigm of the single-core era is gradually being shifted towards multi-/many-core devices such as heterogeneous processing devices ranging from smartphones [91] to super-computers [92]. Heterogeneous processing devices are generally equipped with a general-purpose multi-core CPU and a many-core GPU. Recent advances in computer architecture have modeled the GPUs immensely programmable and proficient to conduct general-purpose computing. GPUs are basically deemed to be extremely efficient in terms of high speedup execution of those scientific applications that involve an excessive amount of parallel computations [93], [94]. To program GPUs, OpenCL [8] has emerged as an industry standard. Due to its portable nature, OpenCL applications can be executed on many types of accelerators including GPUs, CPUs etc. [9], [10].

An OpenCL program comprises of two parts, host program and kernel function. Supervision of kernel function execution on accelerator devices (i.e., CPU, GPU etc.) is the responsibility of a host program that executes on a CPU device [11]. The kernel function deals with the compute-intensive data-parallel code. Within an OpenCL program, buffer creation, data transfer, and kernel mapping to the devices are manually managed by the application developer [12].

The conventional scheduling mechanisms adhere to employ GPUs for the execution of compute kernel whereas the host execution is performed by the CPU device [63]. Such scheduling strategies originate certain problems, for instance, CPU remains idle and underutilized during the kernel execution phase that increases the program execution time and further cause the load imbalance. Over the past few years, the research community is eagerly focused to overcome the aforementioned issues via splitting a compute-kernel to utilize the CPU and GPU simultaneously within a heterogeneous machine [58], [68], [67], [84], [62], [69]. The simultaneous utilization of both CPU and GPU devices for execution often results in load-balanced execution, better device utilization, and reduced execution time.

The mainstream usage of multi-core heterogeneous machines stipulates to employ a scheduling support to efficiently utilize the computing resources and to reduce the overall execution time of the compute-intensive applications [27]. Moreover, in a data-center or Cloud computing environment, dependence on central scheduler rather than application developer plays a more significant role in improving the resource utilization and reducing the execution time for the job-pool [95]. Single job-based scheduling strategies are not appropriate for the scenarios where several jobs are being submitted by different users. The single-job scheduling techniques are often specific job-oriented and employ kernel profiling and splitting to utilize both CPUs and GPUs. In addition to the application code change, devising a generic single job-based scheduling technique is a difficult task. Moreover, the results returned from the split kernels must be accurately combined to produce the correct results. Machine learning based scheduling strategies devised for a

[1]In this research job terminology is used to define an OpenCL application that consists of a host program and kernel functions.

job pool require kernel profiling to assign a kernel to that device on which a job will be executed faster [27], [63]. Moreover, other scheduling schemes [17], [64] require the provenance of execution of the applications to decide the appropriate device for particular application to enhance overall throughput of job-pool. Most of the job pool scheduling schemes [17, 27, 63, 72] require extensive offline profiling to schedule kernels. Furthermore, these scheduling schemes do not consider load balancing as a major factor to decrease the execution time of the job pool [17, 63, 64]. Therefore, there should be a job scheduler for data-parallel compute-intensive applications that does not require application code change, balances the load across the employed heterogeneous computing devices to reduce the execution time of the job-pool, to increase throughput, and to improve device utilization.

Therefore, an *OpenCL Scheduler* (OSched) that considers jobs' computing requirement, processing capabilities of the devices, and jobs' data size to balance load across the heterogeneous computing devices is presented in this chapter. The load balanced execution by the OSched results in reduced execution time of the job pool, maximized throughput, and increased resource utilization. In general, the applications with lower computation requirements are scheduled to slower processing devices and vice versa. Moreover, a certain compute device (a CPU or a GPU) is assigned with computations according to its computing capability while maintaining a load-balance within the heterogeneous multi-core machine. The load-balance assignment of jobs ensures the reduced execution time for the job pool, higher throughput, and better device utilization. Moreover, an enhancement of the proposed OSched scheduler named *Enhanced OSched* (E-OSched) is also designed to further reduce the execution time (of the job pool) and increase the throughput via mitigation of memory contention.

In a more coherent way, the contributions of the proposed scheme are as follow:

1. Two novel scheduling schemes *OSched* and *E-OSched*, performing resource-aware assignment of data-parallel jobs on CPUs and GPUs to reduce make-span time for the job pool, to increase throughput, to increase device utilization, and to reduce the memory contention.

2. Mathematical modeling of the proposed *OSched* and *E-OSched* algorithms.

3. Experimental evaluation to justify the concept of the proposed *OSched* and *E-OSched* algorithms in terms of load-balanced execution with lower execution time, higher throughput, and improved utilization of the computing resources.

4. Implementation and performance comparison of *OSched* and *E-OSched* with the base-line and the state-of-the-art scheduling schemes, which certainly indicates the addressed challenges associated with forthcoming scheduling issues.

5. Significant scrutinization of contemporary state-of-the-art that provides a comprehensive outline to understand the shortcomings of the existing scheduling heuristics.

The rest of the chapter is organized as follows. Section 3.2 presents a multi-job execution scenario on a heterogeneous system to highlight the motivation for this research. The proposed methodology is detailed in Section 3.3. Section 3.4 presents the experimental setup, the experimented scheduling policies, the evaluation metrics, and the experimental results. Section 3.5 concludes the chapter.

3.2 Motivation

CPU executes data-parallel application much slower than GPU due to low level of parallelism. On the other hand, applications with the small data foot-prints get efficiently executed on a CPU device as compared to a GPU device due to the higher data-transfer overhead, under utilization of GPU resources, and small computation to communication ratio.To highlight the discussed scenario, Figure 3.1 shows the execution of Bitonic sort application from AMD [56]. Bitonic sort application has been executed using 09 different input data-sizes and application is mapped to both CPU and GPU separately.

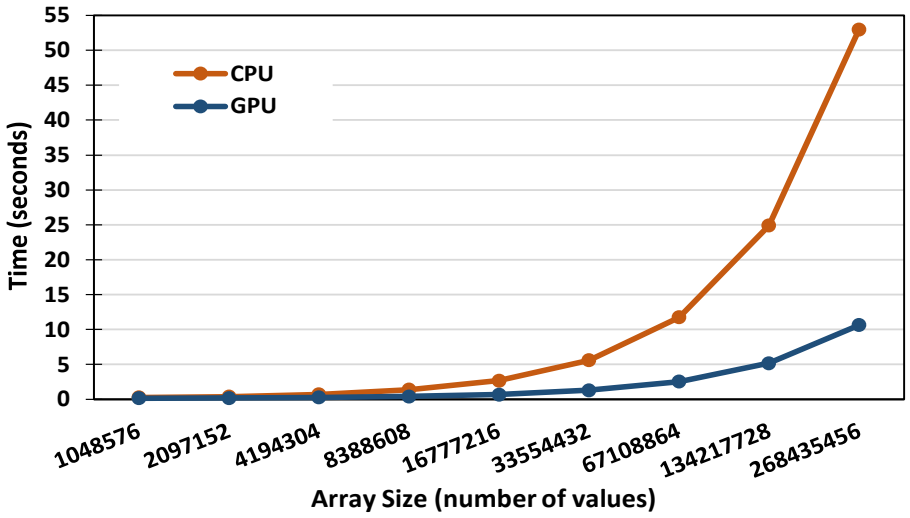

FIGURE 3.1: Execution of Bitonicsort - CPU vs. GPU

Figure 3.1 shows that with small data-size (less number of values to be sorted) the CPU and the GPU based executions have exhibited the similar performances. Whereas with the increased data-size, the execution time of the CPU based application has increased exponentially, while the GPU based execution has shown a linear increase in the execution time. As shown in Figure 3.1, for the largest data-size (i.e., 268435456 values) the GPU based execution has resulted in 05× reduced execution time as compared to the CPU based execution of the application. Therefore, such heterogeneous scheduler should be designed that maps the small data-sized jobs on CPUs while the jobs having large data-size should be mapped on GPUs. In addition to that, the load-balance factor should not be ignored at all so that all the employed devices (i.e., CPUs and GPUs) accomplish the execution of assigned job within the approximately same time duration. A load-balanced and application suited mapping (small jobs on CPUs and large on GPUs) often results in reduced execution time for the job pool, improved resource utilization, and higher throughput.

3.3 OpenCL Scheduler (OSched)

The *OSched* assigns jobs to CPUs and GPUs in such a way that load is balanced across the employed computing devices. It maps jobs in a load-balanced manner by contemplating the computational requirement of a job and processing capabilities of a device. All the submitted jobs are arranged in the job pool according (smaller size first) to their computational requirements, which splits the pool into two segments, first half contains less computational intensive jobs and second half comprises of jobs requiring high computation power. In OSched, the jobs involving low computational requirements (i.e., first pool segment) tends to be mapped to a CPU (having low computing power) while the jobs having high computational requirements (second pool segment) are scheduled to a GPU device. Each device (either a CPU or any GPU) is assigned the pool segment considering its computational share or capability. This load balanced mapping of jobs ensures least execution time of the job pool, higher throughput, and improved device utilization.

3.3.1 System Architecture

The *OSched* based scheduling system comprises of three layers: 1) Hardware layer, 2) System Software layer, and 3) the *OSched* scheduler. This hierarchical structure is presented in 3.2. The hardware layer is the bottom-most layer that contains a heterogeneous multi-core machine based on CPUs and GPUs. At top of the hardware layer is the System Software layer that consists of *Operating System* and *OpenCL Run time* [11]. The OSched scheduler is at the topmost layer of the heterogeneous system. The user of a system submits varying sizes jobs (depicted by the different sized circles in 3.2). Afterwards, the computational requirements of each submitted job is calculated by harnessing the job's computational complexity, for instance, a matrix multiplication job completion will require $2N^3$ operations to complete, where N is a dimension of the square matrix. All the submitted jobs are arranged in the job pool in ascending order (smaller size first) of their computational requirements. The Resource Manager (shown in Figure 3.2) is

responsible for extraction of hardware resource information (i.e., processors detail) of the machine. The Processing Speed of a processor (which may consist of multiple computing cores), measured in FLOPS[2], is calculated using following equation [96]:

$$Processing\_speed = number\_of\_cores \times \frac{cycles}{seconds} \times \frac{flops}{cycles}$$

where, $number\_of\_cores$ represents total cores of a processor, $cycles/second$ represents clock frequency of a core (in Hz), and $flops$ represent total number of floating point operations performed.

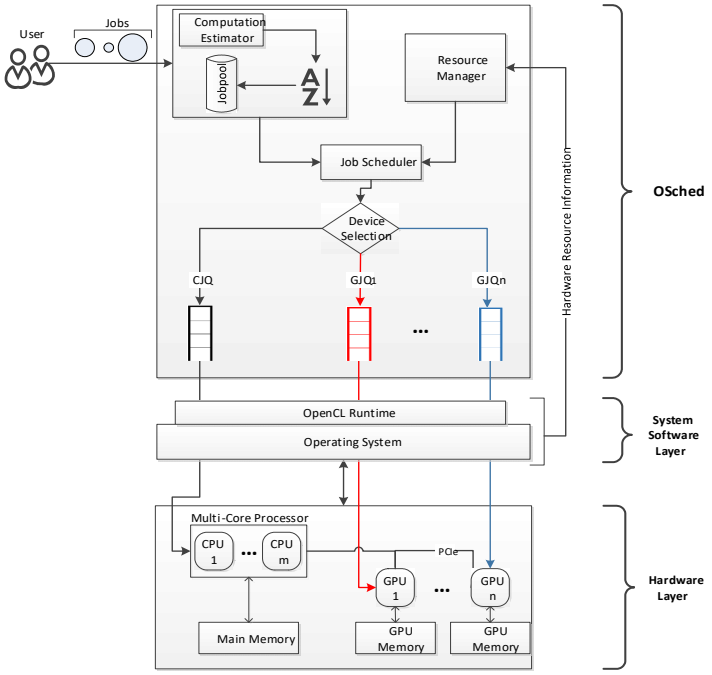

FIGURE 3.2: System Architecture of the OSched

The *Job scheduler* divides the job-pool into *CPU* job queue (represented as CJQ in 3.2) and GPU job queues (represented as $GJQ_{1\_to\_n}$). The reason for single CJQ is that OpenCL, by default, considers a CPU (even if there are multiple CPUs) as a single device. The decision of the job-pool division (into CJQ and $GJQ_{1\_to\_n}$) and device selection is determined pursuant to the computational requirements of jobs and the computing power of each processor. After that, the jobs from

[2]FLOPS = Floating Point Operations Per Second

CJQ and $\text{GJQ}_{1\_to\_n}$ are assigned to the respective processor for execution (based on *First In First Out* (FIFO)).

The complete mechanism of *OSched* is illustrated in Figure 3.3. The first step involves calculation of computational requirement (of jobs) and computing power of devices. Next, the *CPU computational share* (depicted as CS_{cpu} in Figure 3.3) is calculated. The first segment of the pool (consisting of smaller jobs) is allocated to a CPU and the second segment (larger jobs) is allocated to GPUs. First, the CPU based scheduling is performed by mapping $m-1$ jobs (where m represents the last job that can be assigned to the CPU) such that all the mapped jobs' computing requirement is less than or equal to the CPU's computational share. The m^{th} job is mapped to the CPU only if the CPU's computational share is higher than the mapped $m-1$ jobs' computing requirements.

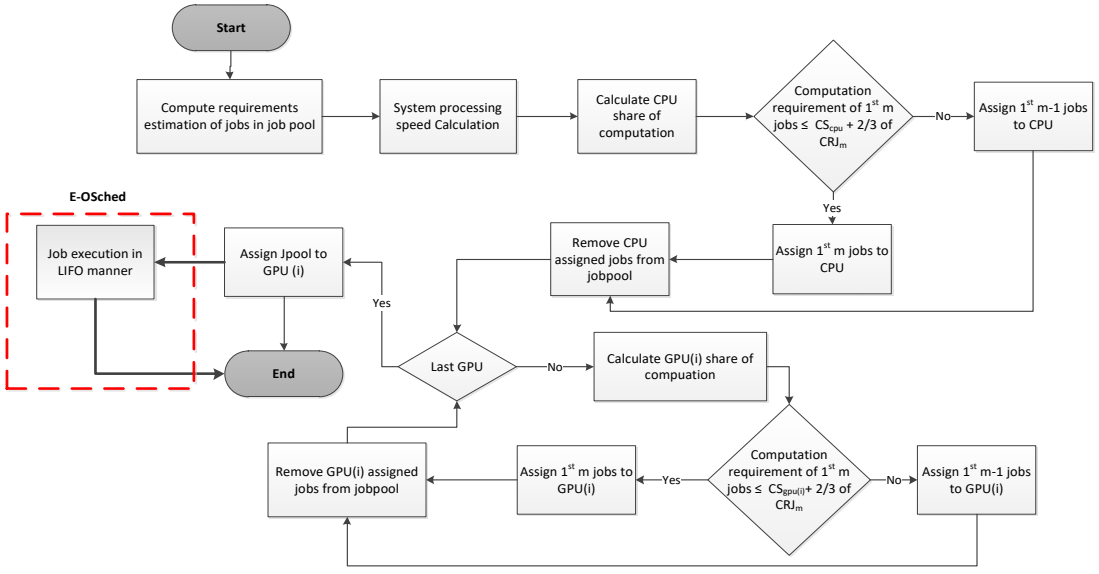

FIGURE 3.3: OSched job scheduling Flowchart

In addition to that, the CPU's share must not exceed CPU's total computational requirement after adding $2/3^{rd}$ computational requirements of m^{th} job (to the

CPU's share). Once all the jobs of the first segment are mapped to CPU, these jobs are removed from the job pool. Subsequently, the GPUs job assignment phase commences in a similar fashion. The exception is for last GPU (GPU_n or only GPU device on the machine) in a heterogeneous machine wherein all the remaining jobs of the pool are assigned to it. The dotted rectangle (depicted in Figure 3.3) represents an optimization (discussed in Section 3.3.4) to the basic *OSched* heuristic.

3.3.2 OSched System Model

This section presents the mathematical model of the *OSched* system. Concerning jobs execution in job pool J, where $J = \{J_1, J_2, \ldots, J_k | CRJ_i \leq CRJ_{i+1}\}$ is a set of k jobs, in a sorted order, with respect to the computation requirement of a job (represented as CRJ). To assign jobs to processors, let $P = \{P_1, P_2, \ldots, P_m \mid PS_i \leq PS_{i+1}\}$ be a set of m processors, in sorted order with respect to the *processing speed* (PS).

In order to determine job assignment to a CPU and GPUs, total computation requirement (J_{CR}) of all the jobs in J and *total processing speed of all processors* (T_{CP}) are required, which are listed in the following equations:

$$J_{CR} = \sum_{i=1}^{n} CRJ_i \tag{3.1}$$

$$T_{CP} = \sum_{i=1}^{m} PS_i \tag{3.2}$$

where, m represents the processing speed of m computing devices (a multi-core CPU and GPUs). From Eq. 3.1 and Eq. 3.2, the computational share of CPU (CS_{cpu}), which is the portion of computation that is assigned to a CPU from J_{CR}, can be calculated by using the formula given in Eq. 3.3.

$$CS_{cpu} = \left(\frac{P_c}{T_{CP}} \times J_{CR} \right) - \alpha \tag{3.3}$$

where P_c represents *processing speed of a CPU* and α is the *adjustment factor* to balance the load across CPU and GPUs. The value of P_C is calculated using the processing speed formula given in Section 3.3.1. The value of α is obtained empirically and is explained in Section 3.4.2.1.

From Eq. 3.3, to determine jobs that are assigned to CJQ, let $J_{cpu} = \{J_1, J_2, \ldots, J_q\}$ be a subset of 1^{st} q jobs from job pool J. Here $Ceiling_{cpu}$ is required to represent the upper boundary for number of jobs in J_{cpu}, which are assigned to CPU. The value of $Ceiling_{cpu}$ is calculated as the sum of CS_{cpu} and $2/3^{rd}$ of *computation requirement of* q^{th} *job* (CRJ_q):

$$Ceiling_{cpu} = CS_{cpu} + \frac{2}{3}CRJ_q \tag{3.4}$$

where, the ratio 2/3 in Eq. 3.4 ensures that q^{th} job should be mapped to CPU only if most of its computation (i.e., 2/3) falls within the limit of CS_{cpu}. Mapping the q^{th} job to CPU (whose $2/3_{rd}$ part can be accommodated within the CPU share) will cause a minor load imbalance as compared to mapping the job to a GPU. This value is also computed empirically and is best suited to balance load across CPU and GPU(s).

Using Eq. 3.4, the number of jobs that are assigned to CJQ is given by:

$$CJQ = \begin{cases} J_{cpu} & \sum_{i=1}^{q} CRJ_i \leq Ceiling_{cpu} \\ J_{cpu} - J_q & \sum_{i=1}^{q} CRJ_i > Ceiling_{cpu} \end{cases} \tag{3.5}$$

After job assignment to CJQ, another set J_{update} is formed by subtracting CJQ from J:

$$J_{update} = J \setminus CJQ = \{job \mid job \in J \wedge job \notin CJQ\} \tag{3.6}$$

Similarly, for job assignment to *GJQ(s)*, Eq. 3.1 and Eq. 3.2 are used to calculate $CS_{gpu}(i)$, which represents the *computational share of* i^{th} *GPU*. $CS_{gpu}(i)$ is calculated by the relationship given below:

$$CS_{gpu}(i) = \left(\frac{P_G(i)}{T_{CP}} \times J_{CR}\right) + \frac{\alpha}{g} \tag{3.7}$$

where $P_G(i)$ represents the *processing speed of* i^{th} *GPU* and g is the number of GPUs in the system. The $P_G(i)$ is calculated using the *processing speed* formula given in Section 3.3.1. From Eq. 3.7, to determine jobs that are assigned to *GJQ(i)*, let $J_{gpu} = \{J_1, J_2, \ldots, J_r\}$ be a subset of 1^{st} r jobs from J_{update}. Here $Ceiling_{gpu}(i)$ represents the upper boundary for number of jobs from J_{gpu}, which are assigned to i^{th} GPU. It is equal to the sum of $CS_{gpu}(i)$ and $2/3^{rd}$ of *computation requirement of* r^{th} *job* (CRJ_r) and is represented mathematically as:

$$Ceiling_{gpu}(i) = CS_{gpu}(i) + \frac{2}{3}CRJ_r \tag{3.8}$$

where, the $2/3$ CRJ_r represents a majority computing part of the r^{th} job (similar to the explanation related to q^{th} job in Eq. 3.4).

Using Eq. 3.8, the number of jobs that are assigned to *GJQ(i)*, for i^{th} GPU given by:

$$GJQ(i) = \begin{cases} J_{gpu} & \sum_{i=1}^{r} CRJ_i \leq Ceiling_{gpu}(i) \\ J_{gpu} - J_r & \sum_{i=1}^{r} CRJ_i > Ceiling_{gpu}(i) \\ J_{update} & i == g \end{cases} \tag{3.9}$$

After job assignment to *GJQ(i)*, the set J_{update} will be updated by subtracting *GJQ(i)* from J_{update}.

$$J_{update} = J_{update} \setminus GJQ(i) = \{job \mid job \in J_{update} \wedge job \notin GJQ(i)\} \tag{3.10}$$

After the completion of jobs assignment to CJQ and GJQ(s), a family of sets $Q = \{CJQ, GJQ_1, GJQ_2, ..., GJQ_g\}$ over J is obtained. The output set Q is governed by the constraints given in Eq. 3.11, Eq. 3.12, and Eq. 3.13.

$$\emptyset \notin Q \tag{3.11}$$

Eq. 3.11 ensures that CPU job-set and GPU(s) job-sets in Q cannot be an empty set $\emptyset$.

$$\cup_{A \in Q} A = J \tag{3.12}$$

Eq. 3.12 specifies that union of all member set (of Q) is equal to job pool J.

$$(\forall A, B \in Q) A \neq B \Rightarrow A \cap B = \emptyset \tag{3.13}$$

Eq. 3.13 stipulates that intersection for all non-equivalent member sets (A and B) of Q would be an empty set $\emptyset$.

3.3.3 OSched Algorithm

Algorithm 3.1 presents the detailed steps for the proposed scheduling mechanism OSched. First of all, J_{CR} and T_{CP} are initialized. Next step is to calculate the values of J_{CR} and T_{CP}. The value of α is calculated in order to compute CP_{CPU}. Afterwards, CJA_{CPU} the computation requirement of jobs assigned to CPU is initialized. Subsequently, the number of jobs which are assigned to CJQ are determined. The value of J_{update} is calculated by subtracting CJQ from J. For job assignment to i^{th} GPU, first $CS_{gpu}(i)$ is calculated CJA_{gpu}(i), which is the computation requirement of jobs assigned to i^{th} GPU. Next, CJA_{gpu} is initialized. Afterwards, the number of jobs that are assigned to $GJQ(i)$ is determined and J_{update} is updated by subtracting $GJQ(i)$ from J_{update}.

For the execution of an OpenCL application, the required data to be computed is first stored in main memory buffers. After that, these data buffers are transferred to the device memory (CPU or GPU memory), where all the computations have to be performed. The multiple large data buffers lead to contention in main memory which slows down the data transfer (from main memory to the device memory); moreover, this also slows down the execution of the OpenCL job on CPU device. Let's consider an example, where two OpenCL jobs e.g., *Matrix Multiplication* (MM) and *Discrete Cosine Transform* (DCT) (taken from the AMD benchmark[12] suit [56]) are concurrently being executed as shown in Figure 3.4. Total available main memory on the machine is 08 GBs represented with memory

[12]Benchmarks are used here to evaluate the scheduling policies. The hardware setup that is used to evaluate scheduling policies has remained constant for all the evaluated scheduling policies.

Algorithm 3.1: OSched job scheduling algorithm

Input :

i A job pool J in which jobs are sorted in increasing order of computation requirement

ii List of CPU and GPU(s) in the system that is sorted in increasing order of processing speed

Output:

i All jobs in job pool are assigned to CPU and GPU(s) for execution

1 **INITIALIZE** $J_{CR} \leftarrow 0$
2 **INITIALIZE** $T_{CP} \leftarrow 0$
3 **for** *i=1 to Total Jobs* **do**
4 | **ADD** $J_{CR} += CRJ_i$
5 **end**
6 **for** *i=1 to Total Processors* **do**
7 | **ADD** $T_{CP} += PS_i$
8 **end**
9 **CALCULATE** $\alpha = (4.12/100) \times J_{CR}$
10 **CALCULATE** $CS_{cpu} = ((P_C/T_{CP}) \times J_{CR}) - \alpha$
11 **INITIALIZE** $CJA_{CPU} \leftarrow 0$
12 /* CJA_{CPU} = Computation Requirements of CPU Jobs */
13 **for** *i=1 to* q^{th} *job* ***from*** *job pool* **do**
14 | **ADD** $CJA_{CPU} += CRJ_i$
15 | **if** $CJA_{CPU} \leq CS_{CPU} + 2/3(CRJ_i)$ **then**
16 | | **ASSIGN** CJQ $\leftarrow J_i$
17 | **end**
18 **end**
19 **CALCULATE** J_{update} = J - CJQ

Algorithm 3.1: OSched job scheduling algorithm (Continued...)

```
20 for i=1 to Total GPUs do
21     CALCULATE CS_gpu(i) = ((P_G(i)/T_CP) × J_CR) + (α/TotalGPUs)
22     INITIALIZE CJA_gpu(i) ← 0
23     // CJA_gpu(i) = Computation Requirements of i^th GPU Jobs
24     for i=1 to r^th job from J_update do
25         CALCULATE CJA_gpu(i) += CRJ_i
26         if CJA_gpu(i) ≤ CS_gpu(i) + 2/3 (CRJ_i) then
27             ASSIGN GJQ(i) ← J_i
28         end
29         if i = Total GPUs then
30             ASSIGN GJQ(i) ← J_update
31         end
32         SUBTRACT J_update -= GJQ(i)
33     end
34 end
```

footprint rectangle, where each sub-part of the rectangle represents 100MBs of capacity. For MM application, each square matrix is of size 12512 × 12512 float elements (requiring approximately 600 MBs of storage). Therefore, the storage space required for data is 1.8 GBs with additional 1.8 GBs for data-buffer. The memory requirements for the three matrices are represented as red-colored area.

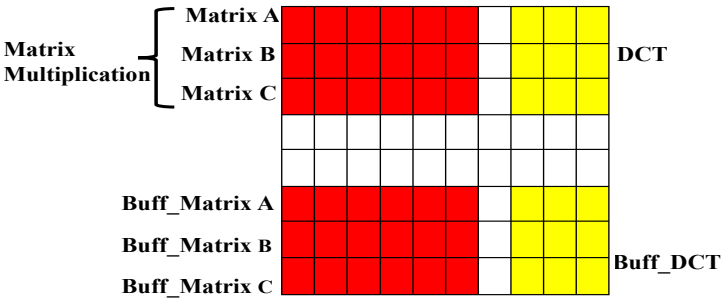

FIGURE 3.4: Memory snapshot for concurrently executing jobs

The yellow-colored area shows the memory requirement for the DCT application (for an image size of 15000 × 15000) wherein each image buffer will require 900 MBs of memory (total 1.8 GBs of memory). Additional to the execution of a large OpenCL application, the main memory also contains other operating systems services etc., which originates the issue of memory contention and degrade the execution performance of the application. To mitigate the memory contention problem, an extension to OSched named as *Enhanced-OSched* (E-OSched) is proposed for scheduling data-parallel applications.

3.3.4 Enhanced-OSched

Enhanced OSched (E-OSched) further refines the performance of the OSched by overcoming memory contention problem (highlighted in the previous section). The E-OSched commences the execution of CPU job queue (CJQ) with the smaller job first and the GPU job queue (GJQ) with the larger job first.

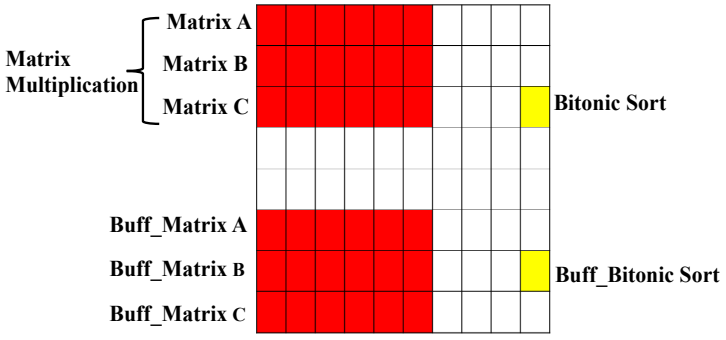

FIGURE 3.5: Memory snapshot after optimization by the E-OSched

The induced job selection mechanism results in mitigating the memory contention problem. Figure 3.5 shows an example schedule (by the proposed E-OSched) wherein largest size job (e.g., Matrix Multiplication) from GJQ is scheduled along

the smallest size job (e.g., Bitonic sort) from the CJQ. The employed scheduling heuristic is depicted in Figure 3.3 (as a dotted rectangle labeled *E-OSched*).

3.3.5 Scheduling Overhead

The scheduling overhead of both OSched and E-OSched scheduling heuristics is $O(N^2)$ as they have to sort N jobs according to computation requirements and then map the jobs to CPU and GPU in a load-balanced manner. The scheduling overhead is negligible for the proposed scheduling schemes and amounts to few milliseconds. Moreover, as there is no data transfer involved during scheduling, both OSched and E-OSched heuristics do not incur communication cost.

3.4 Experiments and Results

The employed experimental setup consists of a heterogeneous multi-core machine. For experimentations, 18 data-parallel applications from five different benchmark suits are used such as AMD [56], Parboil [97], Polybench [98], and Rodinia [99]. The benchmark applications (shown in Table 3.2 are executed using several problem sizes resulting in a job pool of 182 jobs. The problem sizes for applications in the job pools were generated using Monte Carlo simulation by following the methodology presented in [100, 101].

Moreover, some applications (i.e., Matrix Transpose from [56]) have problem size constraints which were preserved during problem size generation e.g., Matrix Transpose can only be executed for the problem size that is equal to power of 2. All jobs in the job pool are independent and are non-preemptive. Moreover, processing cores within a processor (i.e., CPU and GPU) are considered homogeneous in this study. All the experiments are conducted 05 times and results are reported in the form of their average. The specifications of the employed machine are presented in Table 3.1.

TABLE 3.1: Experimental setup

Device	CPU	GPU
Model	Intel Core i5-4460	Nvidia GeForce GTX 760
Base Clock	3.2 GHz	0.980 GHz
Boost Clock	3.4 GHz	1.033 GHz
Total Cores	4	1152 (CUDA cores)
Memory	8 GB	2 GB
Processing Speed (Single Precision)	409.6 GFLOPS[3]	2257.9 GFLOPS
Operating System	Ubuntu 16.04 LTS	
OpenCLSDK	Intel SDK for OpenCL2016	CUDA 8.0
Compiler	GCC 5.4.0	Nvcc

3.4.1 Scheduling Policies and Evaluation Metrics

For evaluation, five scheduling techniques (listed below) have been employed:

1. ***All_On_CPU:*** all jobs are assigned to a CPU device for execution. It is a naive heuristic act as a baseline for performance comparison.

2. ***All_On_GPU:*** all jobs are scheduled on a GPU device. This scheduling scheme indicates that programmers generally prefer GPUs only for execution of all the compute-intensive applications that lead to poorer utilization of the other computing devices such as CPU.

3. ***Work item guided:*** this heuristic [27] first sorts jobs according to kernel's size of global work-items. Next, CPU commences execution from job with

[3]The processor contains 04 cores. Each core ticks 3.2×10^9 times per second and is capable of performing 32 floating point operations per cycle. Therefore, $4 \times 3.2 \times 10^9 \times 32 = 409.6$ GFLOPS

the lowest number of global work items whereas GPU starts execution with the highest number of global work items.

4. ***Input size guided:*** In this scheme [27], jobs are sorted in ascending order in the task-queue according to data (number of bytes) required to be transferred among a CPU and a GPUs. CPU starts execution of the task queue from the top of the sorted queue while GPU computes tasks from end of the sorted task queue (largest jobs first).

5. ***Machine Learning based Task (MLT) scheduling:*** The heuristic presented by [27], is based on machine learning based classification that employs the static code features (such as instructions, math functions, barriers etc.) and dynamic run time features (such as local_work_size, global_work_size, input data size etc.) to determine device suitability of OpenCL kernels.

TABLE 3.2: Benchmarks along with input data sizes

Benchmark Suits	**Data Parallel Applications**	**Input Data Size**	**Total Versions**
AMD	Matrix Multiplication	786,432—7,077,888	3
	Binomial Options	32,768—131,072	3
	Bitonic Sort	32,768—16,777,216	9
	Fast Walsh Transform	8,192—204,800	14
	Matrix Transpose	32,768—67,108,864	7
	Discrete Cosine Transformation	2,097,152—943,718,400	17
	Floyd Warshall	262,144—6,553,600	3
Polybench	3MM (3 Matrix Multiplications)	7,000,000—10,080,000	2
	GEMM (Matrix-multiply C=alpha.A.B+beta.C)	750,000—12,000,000	5
	GESUMMV (Scalar, Vector and Matrix Multiplication)	8,012,000—1,800,180,000	17

Benchmark Suits	**Data Parallel Applications**	**Input Data Size**	**Total Versions**
	MVT (Matrix Vector Product and Transpose)	4,016,000—900,240,000	17
	ATAX (Matrix Transpose and Vector Multiplication)	4,012,000—900,180,000	17
	2MM (2 Matrix Multiplications)	5,000,000—12,800,000	2
	2DCONV (2D Convolution kernel)	2000000—1,568,000,000	17
	3DCONV (3D Convolution kernel)	1,000,000—1,728,000,000	17
Parboil	3D Stencil Operations	2,097,152—67,108,864	2
Rodinia	BFS (Breadth First Search)	43,963—592,428,022	14
Own Developed	Matrix-Vector Multiplication	4,202,496—1,514,299,392	16

For performance evaluation, the following performance metrics were considered:

1. ***Execution time:*** The execution time depicts the time consumed in the execution of all jobs of the job pool. The smaller value of the execution time is an indication of better results.

2. ***Throughput:*** represents the number of jobs completed per unit time. The higher value of throughput manifest the better results.

3. ***Average time (of a job):*** is defined as an average amount of time taken by a job (of a job pool) to complete its execution. The lower average time exhibits the improved performance.

4. ***Load balance:*** Load balance is a distribution of workload among CPU and GPUs in the form that both computing devices accomplish the execution of

the assigned workload within the approximately same time duration. Load balance has been calculated as a percentage of the difference between execution times of all the jobs mapped on a CPU and a GPU device. The lower value of this metric shows a more load-balanced execution.

3.4.2 Execution Performance

In this section, the execution time of all the scheduling heuristics is presented. The execution time of each scheduling heuristic was recorded with varying number of CPU cores (i.e., 1, 2 and 4 cores) to analyze its impact on the execution time. Moreover, analysis of adjustment factor α is also presented in this section.

Figure 3.6 presents execution time-based results (using 4 CPU cores and the GPU device) of the proposed and the other scheduling heuristics. These results specify that the E-OSched and the OSched have outperformed all other scheduling heuristics in terms of the execution time of job pool (mentioned in Section 3.4.1). As compared to the baseline scheduling (i.e., *All_On_CPU*), the *OSched* and the E-OSched consumes 2.03× and 2.01×s lower execution time respectively. For the GPU based execution of the job pool (i.e., *All_On_GPU*), the *OSched*, and *E-OSched* consume 1.70× and 1.71× times reduced execution time. The E-OSched consumes 6.25% reduced execution time as compared to the input size guided scheduling heuristic [27] (as shown in Figure 3.6). As compared to the *input size guided* heuristic [27], the proposed *E-OSched* consumes 7.35% reduced execution time. As compared to the work-item guided scheduling heuristic [27], the *OSched* and *E-OSched* consume 5.16% and 6.25% reduced execution time respectively. The *OSched* consumes 2.54% less execution time for the job pool execution as compared to the scheduling heuristic of *MLT* [27]. Moreover, the *E-OSched* further improves the execution time and results in 3.51% reduced execution time as compared to the heuristic of *MLT* [27]. Figure 3.7 presents the execution time based results (using 02 CPU cores and the GPU device) for the proposed and the other scheduling heuristics. As compared to the *All_On_CPU* and *All_On_GPU*

scheduling schemes, the *OSched* reduce execution time by 2.05× and 1.6× respectively whereas the *E-OSched* consumes 2.1× and 1.7× lower execution time respectively.

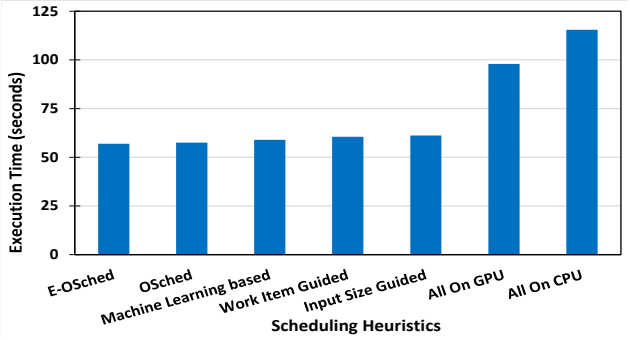

FIGURE 3.6: Execution time of scheduling heuristics for job pool using 04 CPU cores and a GPU

The *OSched* and the *E-OSched* consume 1.07× and 1.09× reduced execution time respectively as compared to the *input size guided* scheduling heuristic [27]. As compared to the *work-item guided* scheduling heuristic [27], the *OSched* and *E-OSched* consume 1.07× and 1.05× reduced execution time respectively.

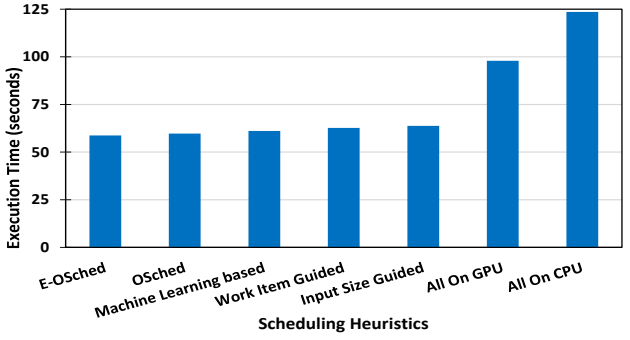

FIGURE 3.7: Execution time of scheduling heuristics for job pool using 02 CPU cores and a GPU

When compared to the state-of-the-art scheduling scheme MLT [27], the reduction in execution time by the *OSched* and the *E-OSched* is 1.04× and 1.02× respectively.

Next, the execution time based results (using 01 CPU cores and the GPU device) for the proposed and the other scheduling heuristics (as shown in Figure 3.8). The results show that *All_On_CPU* and *All_On_GPU* scheduling schemes are outperformed by the *OSched* by respectively consuming 1.6× and 1.4× less execution time. Similarly, the *E-OSched* reduces execution time by 1.64× and 1.17× respectively when compared to *All_On_CPU* and *All_On_GPU* scheduling schemes. The *OSched* and the *E-OSched* reduce execution time by 1.1× and 1.13× respectively in comparison to the input size guided scheduling heuristic [27]. Reduction in execution time is by 1.11× and 1.13× respectively when compared to *work-item guided* scheduling heuristic [27]. When compared to the state-of-the-art scheduling scheme *MLT* [27], the reduction in execution time by the *OSched* and the *E-OSched* is 1.04× and 1.02× respectively. The *OSched* and the *E-OSched*, when compared to the *MLT* [27] scheduling scheme, respectively consumes 1.05× and 1.08× less execution time.

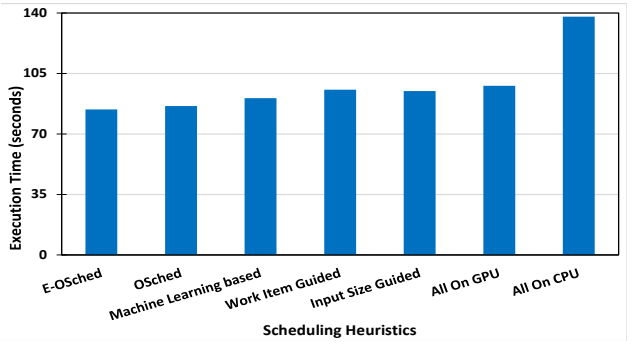

FIGURE 3.8: Execution time of scheduling heuristics for job pool using 01 CPU cores and a GPU

The execution time based results (Figure 3.6 — Figure 3.8) show that the *OSched* and the *E-OSched* consistently produce better result (in terms of the execution time) as compared to the other employed scheduling heuristics. The reduced execution time consumed by the proposed *OSched* and *E-OSched* scheduling heuristics are due to the load balanced execution of the job pool (as evident in the further results in this section). An interesting phenomenon observed during the experimentation is that when the number of CPU cores is reduced from 04 to 02, the

execution time of all the employed scheduling schemes increases slightly (3%–7% higher). However, when CPU cores are further reduced to 01, a significant increase in terms of the execution time is noticed (up to 53%). The increased execution time (for 02 CPU cores) results due to the less number of available CPU cores and the majority of the execution load is mapped on the GPU device. However, when the CPU cores are reduced to only 01 core, the execution time of the job pool increases significantly as the only CPU core is now responsible to execute the assigned workload (according to its processing capability) in addition to the execution of the host programs (of all the concurrently executing applications). As shown in Figure 3.6 — Figure 3.8, the execution time based results for *All_On_GPU* scheduling scheme remain almost similar with varying CPU cores. The reason for the similar attained execution time (by the *All_On_GPU* scheduling scheme) is that it only utilizes GPU device for execution and no job pool related load in mapped on CPU.

3.4.2.1 Analysis of Adjustment Factor "α"

The host program of an OpenCL application (mapped either on a CPU or a GPU) is always executed on a CPU device. The execution of the host programs on a CPU device causes a certain execution overhead during the concurrent executions of OpenCL applications. Due to this overhead, the CPU based execution of a kernel often suffers from increased execution time. Figure 3.9 shows an execution profile of different OpenCL programs mapped on a GPU device. It is evident from Figure 3.9 that the CPU has to spend a non-negligible time in the execution of the host program. An *adjustment factor* α overcomes this overhead and the induced load imbalance via off-loading a fraction of computations from CPU to GPU device. The value of α depends on the time spent by the CPU only for the execution of the host programs of a job pool; therefore, it is difficult to specify the exact value of α to achieve 100% load balance. The value of α is calculated by considering the mean execution time (of host programs only) of different OpenCL applications in the job pool.

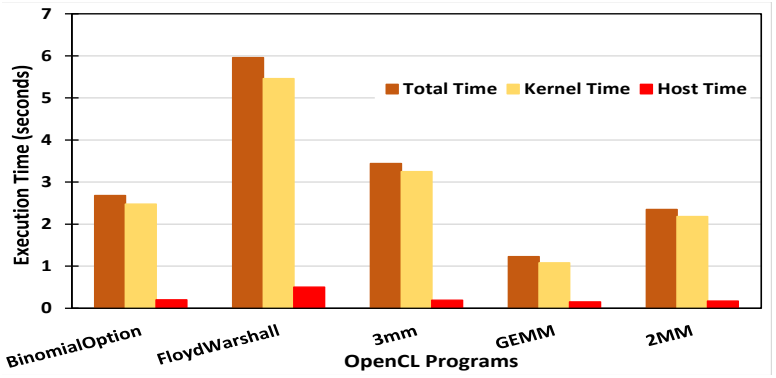

FIGURE 3.9: Application Execution profile

Therefore, a factor α with 4.12% of J_{CR} is introduced for the proposed scheduling heuristic (as shown in Algorithm 3.1). To adjust the host program execution overhead (on a CPU device), α percentage of the computational load is deducted from the CPU's share and added to the GPU that results in very favorable load-balance.

3.4.3 Average Execution Time

In this section, average execution time of a job (using 04 CPU cores and a GPU) are reported. Figure 3.10 shows the average execution time of a job against seven scheduling heuristics. As compared to the *All On CPU* scheduling, the proposed *OSched* and the *E-OSched* result in 101.27% and 103.88% reduced execution time (for a single job) on average respectively. The *OSched* and the *E-OSched* consume on average 70.92% and 73.13% reduced execution time respectively as compared to the *All on GPU* scheduling. As compared to the *input size* guided heuristic [27], the *OSched* and the *E-OSched* consume on average 6.7% and 8.09% reduced execution time respectively. The *OSched* and the *E-OSched* consume on average 5.75% and 7.11% reduced execution time respectively as compared to the *work-item guided* [27] scheduling. As compared to the MLT [27] scheduling, the *OSched*, and the *E-OSched* consume on average 3.19% and 4.53% reduced execution time respectively.

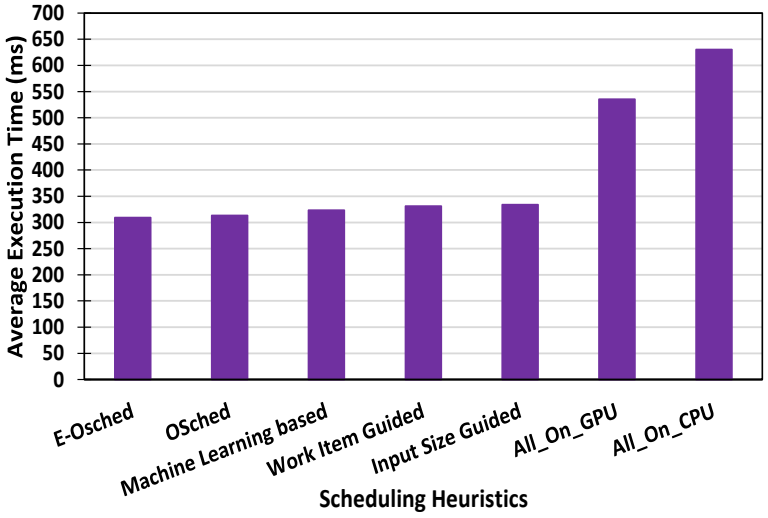

FIGURE 3.10: Average Execution Time

3.4.4 Throughput Analysis

For throughput analysis, the scheduling heuristic of *MLT* proposed by [27] is considered as a baseline and all other scheduling heuristics were evaluated against the achieved throughput of *MLT*. Figure 3.11 presents the attained throughput of the schedulers (using 04 CPU cores and a GPU device) as compared to the baseline heuristic [27]. The *E-OSched* and *OSched* achieve the highest throughput as compared to the baseline (2.45% and 3.51% improved throughput respectively) and the other scheduling heuristics. The proposed schedulers *OSched* and *E-OSched* attain 5.02% and 6.08% higher throughput respectively as compared to the *work-item guided* heuristic [27]. As compared to the *input size guided* scheduling [27], the *OSched* and the *E-OSched* attain 6.03% and 7.07% improved throughput respectively. The *OSched* and the *E-OSched* attain 42.25% and 43.31% higher throughput respectively as compared to the *All On GPU* scheduling. As compared to the *All On CPU* scheduling, the *OSched* and the *E-OSched* attain 51.4% and 52.46% improved throughput respectively.

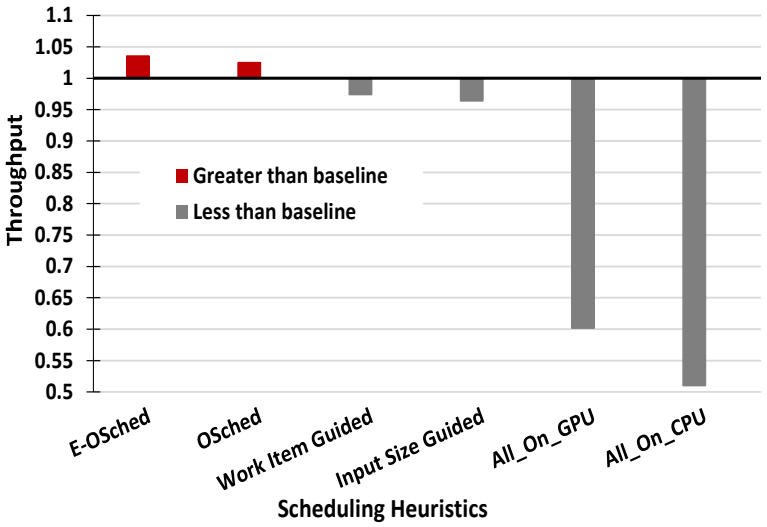

FIGURE 3.11: Throughput analysis

3.4.5 Load Balance Analysis

To demonstrate the effectiveness of factor α, this section presents load balance analysis achieved by the OSched and the E-OSched scheduling heuristics with and without adjusting α. Figure 3.12 presents the load balance achieved by the proposed schedulers OSched and E-OSched without adjusting factor α.

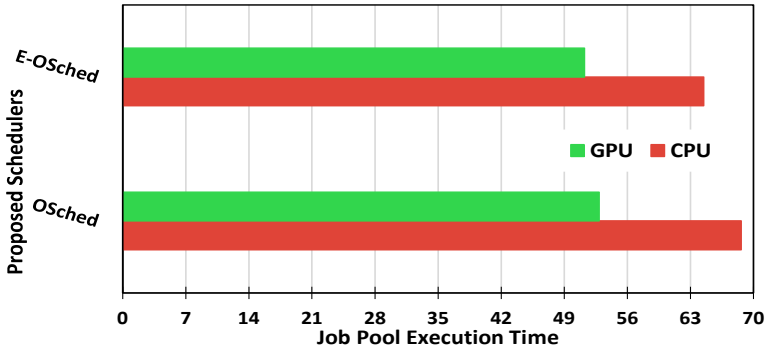

FIGURE 3.12: Load balance without factor α

With the adjusted α, the load balance attained by the OSched and E-OSched is presented in Figure 3.13. The results presented in Figure 3.12 and Figure 3.13 are obtained using 04 CPU cores and a GPU device. Figure 3.12 shows that the load imbalance is 23% and 20.5% for the *OSched* and the *E-OSched* respectively (without adjusting overhead factor α).

After the introduction of α (Figure 3.13), load imbalance decreases to 2.7% and 1.9% for the *OSched* and the *E-OSched* respectively and results in a significant decrease in execution time of job pool. When compared to the load imbalance induced by *work-item guided* [27] scheduling heuristic, the *OSched* and the *E-OSched* reduce load imbalance by 2.3% and 3.1% respectively. The *OSched* and the *E-OSched* reduce load imbalance by 1.97% and 2.68% respectively when compared to load imbalance induced by *input size guided* [27] scheduling heuristic. When compared to MLT [27], reduction in load imbalance is up to 0.15% and 0.95% respectively.

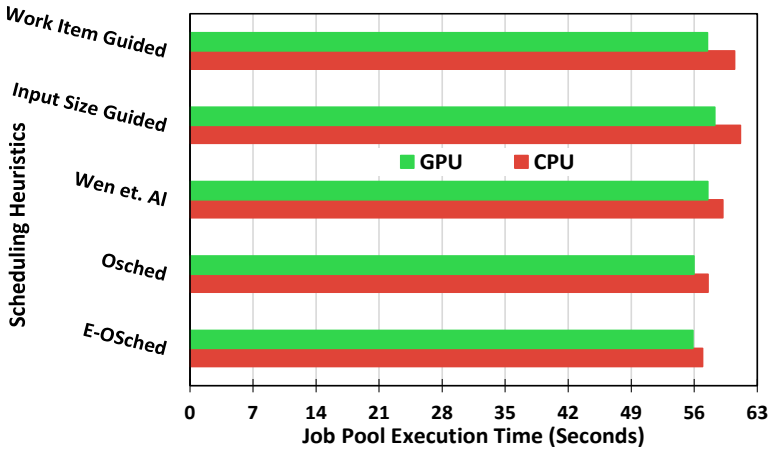

FIGURE 3.13: Load balance after adjusting factor α (in OSched and E-OSched)

3.5 Conclusions

This work focuses on two novel scheduling schemes named OSched and E-OSched to ensure the load balancing of compute-intensive applications on heterogeneous

multi-cores. These schedulers perform the resource-aware assignment of jobs while contemplating the job requirements and processing capabilities of the employed computing devices (i.e., CPUs and GPUs). The performance evaluation of experiments has revealed that OSched has significantly maintained a balanced load on the employed computing devices, minimized the job-pool make-span, maximized throughput, and maximized the resource utilization than the base-line (CPU or GPU only) and the state-of-the-art scheduling heuristics. Moreover, the E-OSched has further enhanced the execution performance via incorporation of main memory contention factor. The experimental evaluation shows that the execution time was reduced by 70.92% and 73.13% respectively as compared to the base-line scheduling heuristics. As compared to the state-of-the-art scheduling heuristics, the *OSched* and *E-OSched* have reduced the execution time by 6.7% and 8.09%, respectively. The OSched and E-Osched both have improved the throughput by 51.4% and 52.46%, respectively than the base-line heuristics. The OSched and E-OSched have achieved 6.03% and 7.07% higher throughput, respectively. As most of the today's large compute-clusters and supercomputers are based on heterogeneous computing nodes[4] that requires a load balanced scheduling mechanism to map jobs across a variety of the computing devices considering the device capabilities. The OSched and E-OSched will aid greatly to schedule compute-intensive data-parallel jobs in a load balanced manner to reduce execution time, to improve system throughput, and to increase device utilization. The future intention is to extend the E-OSched scheduler for CPU/GPU cluster based on multi-machine configurations.

[4]https://www.top500.org/lists/2017/11/

Chapter 4

Machine Learning based Multi-Job Scheduling

4.1 Introduction

Heterogeneous systems based on CPUs and GPUs are becoming mainstream due to disparate processing and performance capabilities of these multi-core architectures [102]. Mostly, CPUs are better suited to perform *latency-sensitive* tasks and incorporate architectural advances such as *branch-prediction*, *out-of-order execution*, and *super-scalar* capabilities [51]. Whereas, many-core GPUs are more suited to perform data-parallel and throughput-sensitive tasks due to the inherent massive multi-threading capabilities [52]. In order to harness the parallelism and computing capabilities of CPUs and GPUs, application developers must focus on exploiting the computing capabilities and architectural characteristics of these devices such as executing applications' serial code on a CPU device, while the parallel and compute-intensive code on GPUs.

Mostly, the application developers schedule compute-intensive part of the applications on GPUs and serial or less compute-intensive code on a CPU device. Using GPUs only for compute-intensive code and CPU for serial execution leaves CPU mostly idle and a GPU device overloaded [103]. This results in underutilization of

CPU and a longer execution time for the executing application. Optimal utilization of CPU-GPU heterogeneous systems is, in itself, a challenging task. There are several research efforts in recent years that address the issue of device underutilization in a heterogeneous system. The research techniques in [22, 60, 67, 80] split the OpenCL kernel between a CPU and a GPU resulting in improved execution time and a balanced workload across the heterogeneous devices. However, these techniques [22, 60, 67, 80] consider splitting of the OpenCL kernel(s) of a single application on a CPU and a GPU device.

Due to the increased programmability of heterogeneous CPU-GPU systems, there is a paradigm shift from mapping a single OpenCL application to a computing-device towards scheduling of diverse OpenCL applications on heterogeneous systems [27]. Furthermore, user application scheduling on heterogeneous devices in a Cloud environment should be managed by a central scheduling module instead of a user-controlled mapping [95]. The reason is that the application programmers are mostly inclined to map application kernels to faster devices (i.e., GPUs) that cause imbalanced work-distribution. In addition to that, the *device suitability*[5] of an application should also be considered in scheduling decisions. While one application may be suitable for GPU-based execution, another application may be suitable for CPU-based execution. Therefore, a heterogeneous application scheduler should ensure efficient utilization of both CPU and GPU devices and also consider the device-suitability in the scheduling decisions. Furthermore, in order to induce load balance across CPUs and GPUs, speedup prediction-based mapping could result in better performance for the scheduled applications. In fact, most of the applications developed using OpenCL are suited to a GPU-based execution and scheduling only using device-suitability aspect will result in an imbalance mapping of the jobs to CPUs and GPUs. To mitigate this issue, the low speedup jobs (suitable for a GPU device) can be shifted to a CPU (incurring minimal performance penalty) to improve load balance in a heterogeneous multi-core machine. This scheduling strategy will lead to reduced execution time for the applications and load-balanced distribution of workload or job pool. There are several research

[5]In this research device suitability of an application means that the application executes faster on the device to which it is suitable. Device may be a CPU or a GPU.

efforts (i.e., [17, 27, 58, 81]) that target scheduling of job pool on heterogeneous multi-core machines. However, these techniques are either oblivious to device-suitability of an application [17, 64] or do not consider speedup prediction for the scheduling decisions [27, 63].

In the previous chapter, the *E-OSched* application scheduler was introduced that maps the job pool of OpenCL applications to the heterogeneous computing devices in a load-balanced manner. The *E-OSched* attains a load-balanced execution by considering jobs' processing requirements when assigning it to either a CPU or a GPU device. Jobs with low computation requirements are assigned to a CPU whereas the higher computation requirement-based jobs are mapped to a GPU device. The jobs are mapped according to the computing capabilities of the devices (such as CPUs and GPUs) that results in a near optimal balanced schedule. A limitation of the E-OSched is that it does not consider the device-suitability and speedup-factor (of jobs) aspects while making the scheduling decisions [27, 63, 104, 105].

To fill this gap, this chapter presents *Troodon*[6] a novel machine learning-based scheduling mechanism build on top of the *E-OSched* that assigns jobs to computing devices (CPUs and GPUs) considering both the device suitability and speedup factor while maintaining a load balanced execution across the devices. Initially, the Jobs exhibiting *device suitability* to CPU are mapped to a CPU device while the jobs exhibiting *device suitability* to GPU are mapped to the GPU device. Afterward, the *E-OSched* scheduling heuristics is adapted to include predicted speedup[7] during job mapping decisions i.e., low speedup jobs from the potentially over-utilized devices are transferred towards the under-utilized devices. Moreover, the resource-aware scheduling according to the computing capabilities of the devices is induced by the E-OSched to provide a load balanced mapping of jobs. Multiple benchmark suits [56, 97–99] have been used to train the prediction model.

[6]Troodon is proclaimed as world's most intelligent dinosaur specie possessing considerably large brain size in proportion to its body size.

[7]Speedup of a job on a suitable device = Execution time of a job on a slower device/execution time of a job on faster device

The aim of the diversified training (based on the several types of benchmark applications) of the model is to attain more accurate speedup prediction. Experimental results (presented in this chapter) show that the Troodon improves the execution time up to 20% when compared to the state-of-the-art scheduling heuristics.

In particular, following are the contributions that have been made in this chapter:

- In-depth analysis of the state-of-the-art scheduling mechanisms for heterogeneous machines to identify the merits and demerits of several existing heuristics.
- Development of a machine learning -based classifier to decide a job's suitability for a particular multi-core processor.
- Development of a machine learning-based relative speedup predictor of a job on heterogeneous computing devices.
- A novel and accurate (due to usage of diverse benchmarks to increase prediction accuracy) machine-learning based scheduling heuristic for heterogeneous multi-cores that considers device-suitability, speedup-factor, and computing capabilities of the devices for scheduling and execution of a job pool in a load-balanced manner.
- Empirical investigation and comparison of the proposed scheduling technique with state-of-the-art scheduling heuristics from literature, demonstrating significant improvement in execution time, throughput, and load balance.

The rest of the chapter is organized as follows. Section 4.2 presents a motivation case-study to highlight the significance of this research. The detailed methodology of the Troodon scheduling heuristic is presented in Section 4.3. The experimental setup, evaluation parameters, obtained results, and discussions are presented in Section 4.4 along with the limitations of the proposed scheduling heuristic. In the end, Section 4.5 concludes this chapter.

4.2 Case-Study: Incorporating Speedup Prediction in Scheduling

Most of the data-parallel applications attain a higher speedup for the GPU-based execution while there are some applications (such as *Breadth First Search*, *Dot product*, etc.) that perform inadequately on GPUs [27, 63]. Additionally, the same application may attain varying performance (on the faster device) for different input data-sizes. Consider the obtained speedup in Figure 4.1 when Matrix Multiplication [56] is executed with different data sizes on a CPU and a GPU device. The employed experimental setup for this experiment is mentioned in Section 4.4. The *Y-axis* displays the obtained relative speedup of GPU over CPU for Matrix Multiplication whereas the *X-axis* shows the number of elements in a matrix. It can be observed that when the data size for an application is varied, the speedup likewise changes too. The variation in the speedup is mainly due to the memory alignment issue, a number of threads launched on each core.

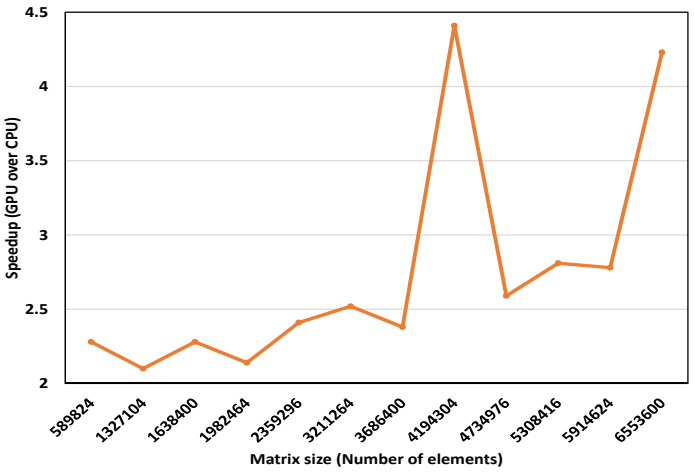

FIGURE 4.1: Obtained speedup of GPU over CPU for different data sizes of Matrix Multiplication

Therefore, job scheduling policy should not only consider jobs' device suitability but also the potential speedup that a job (with a particular data-size) attains when executed on a suitable device. The reason for speedup consideration is that if all jobs are assigned on the basis of device suitability, then the vast majority of

applications are allocated to GPU (as stated in Section 4.1) that results in load imbalance and less favorable execution time for a job pool. To highlight the impact of device suitability and load balance, Figure 4.2 shows an execution example using three scheduling scenarios (i.e., *device-suitability* based, *GPU* only, and an *Oracle heuristic*) of a job pool (based on 06 OpenCL applications, where 04 i.e., *Matrix Multiplication*, *Bitonic Sort*, *Monte Carlo Asian DP*, *Black Scholes* are GPU-suited while the 02 applications such as *GEMM* and *Matrix-Vector Multiplication* are CPU suited). In a *Device suitability-based* scheduling, a job is assigned to a device on which it could execute faster. The *GPU-Only* scheduler represents the execution of jobs by employing only GPU devices. The third scheduling (mentioned in Figure 4.2) labeled as *Oracle* represents a mapping of jobs on a suitable device. In addition to that, for a load balanced execution lower speedup jobs are transferred from an over-loaded device (i.e., GPU) to an under-utilized computing device (such as CPU). Figure 4.2 shows that the Oracle-based scheduling attains approximately 2× and 3.5× performance improvements in execution time as compared to Device Suitability and GPU-Only scheduling schemes, respectively.

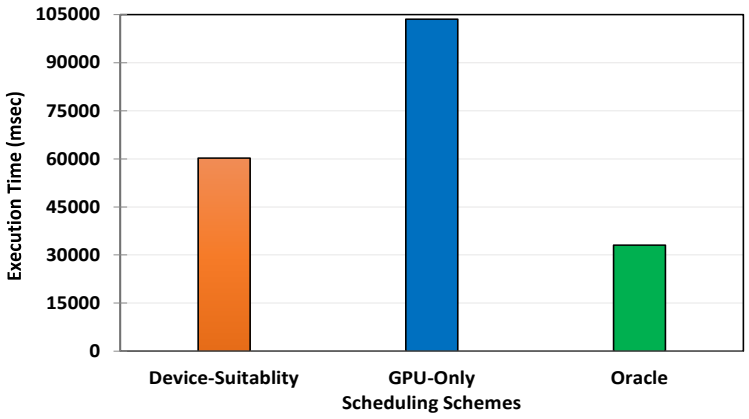

FIGURE 4.2: Execution time of different scheduling schemes displaying the impact of speedup prediction

From Figure 4.1 and Figure 4.2, it can be concluded that an unprecedented scheduling policy can be designed if the device suitability, speedup factor, and a load balanced execution is considered. The scheduling mechanism based on the

key aspects will result in higher device utilization, lower execution time (for the job pool), and a higher speedup.

4.3 Troodon

A scheduler named Troodon is presented that is capable to assign jobs to CPUs and GPUs in a load balanced manner while taken into consideration the job processing requirement, device suitability, and predicted performance on a certain computing device. Jobs suited to CPU device are classified and labeled as *CPU-Suitable* while those suited to GPU based execution are classified as *GPU-Suitable*. All jobs are then organized in a job-pool according to the device suitability. The jobs in the job-pool are then sorted according to the predicted speedup. A load balance scheduling is achieved by considering the jobs' processing requirements and devices' computation capabilities. The device computational capabilities are incorporated by calculating the computing share for each device (i.e., CPUs and GPUs) with respect to the available workload to be scheduled. The job pool of the OpenCL applications is then scheduled strictly adhering to the limits of device computing share that ensures a load balanced mapping of OpenCL jobs or compute kernels to the suitable devices. The device suitability and load balanced mapping of jobs produces lower execution time of the job pool, maximized throughput, and higher device utilization.

4.3.1 System Architecture

The overall scheme of the Troodon scheduling heuristic is presented in Figure 4.3. Light gray colored shapes represent the major component of the current study. Users submit OpenCL applications and the *Computational Assessment* module assess computational requirements of each submitted job (using the computational complexity and the input data sizes provided by the user e.g., for an N×N matrix, a matrix multiplication job completion will require $2N^3$ operations). At the core

of the Troodon system lies the *kernel feature extractor*, *device suitability classifier*, *speedup predictor*, and *CPU-GPU job pools*. The *Kernel Features Extractor* component extracts 23 OpenCL kernel code-features (details of the feature extraction process and the extracted code features are provided in Section 4.3.5) of each submitted OpenCL job. Extracted code features along with the *input data-size* of a job are then provided to the *Device Suitability Classifier* module which classifies and labels the submitted jobs according to the device suitability (a potential best performing device for that job i.e., a CPU or a GPU). After that, the *Speedup Predictor* component takes the input of the labeled OpenCL jobs (classified by the *Device Suitability Classifier*), extracted code features, and input data size to predict the jobs' speedup relative to the other device (a non-suitable device for that application classified by the Device Suitability Classifier). Section 4.3.7 presents a detailed discussion of the *Device Suitability Classifier* and the *Speedup Predictor* modules. Each application is then stored according to the *Device suitability* in a separate CPU or GPU job pools. Next, the CPU job pool is sorted in the descending order (based on the predicted relative speedup) and the GPU job pool is sorted in the ascending order of the predicted jobs' speedups. These two job pools (i.e., for *CPU* and *GPU*) are then combined to form a collective job pool for the scheduling. To map jobs in a load balanced manner, the E-OSched [81] scheduling mechanism has been employed. E-OSched starts mapping jobs of the job pool to both the CPU and GPU devices. Jobs at the top of the job pool (i.e., CPU suited with a high predicted speedup) are mapped to the CPU device while the jobs at the bottom of the job pool (Jobs suited to a GPU device having a high predicted Speedup) are mapped to GPU device. The E-OSched mechanism continues assigning jobs in this manner until a device load reaches the ratio of the load calculated for that device by the E-OSched. After that, all remaining jobs that are suited to a particular device are mapped to the non-suitable device. CPU and GPU jobs sorting in descending and ascending order respectively guarantee that jobs, after a device has reached its assigned computation load, mapped to a non-suitable device are the ones possessing lowest predicted speedup on the suitable device. In this way, the minimum penalty is incurred by mapping jobs to a

non-suitable device while ensuring load-balanced execution of the job pool.

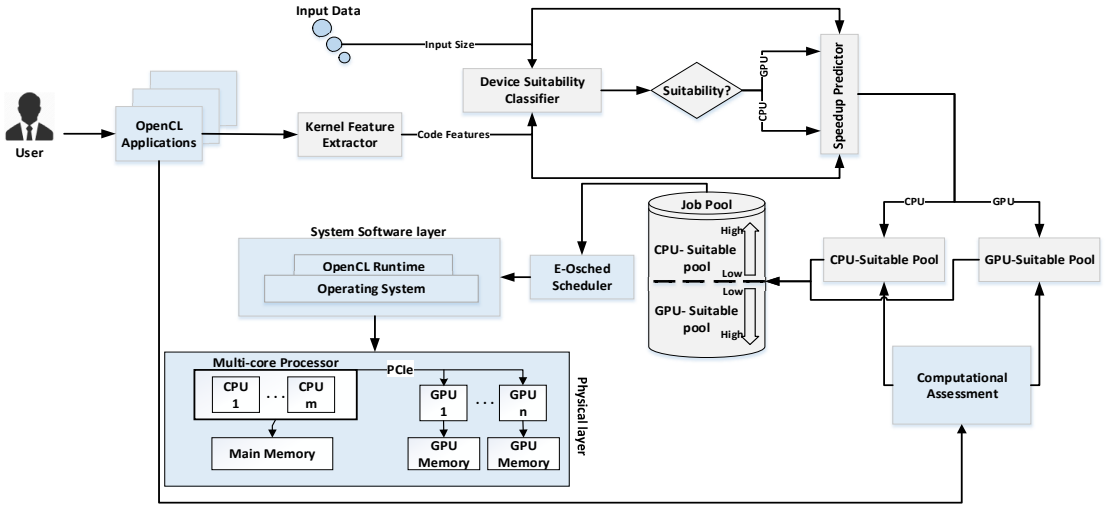

FIGURE 4.3: System Architecture of the Troodon

4.3.2 System Model

In this section, a mathematical model for the proposed Troodon scheduling mechanism is presented. The system model of the E-OSched [81] has been adapted to the scenario represented in this study where jobs are mapped to a heterogeneous system using machine learning-based device-suitability classification and speedup prediction model.

Consider $\mathbb{C} = \{c_1, c_2, ..., c_m\}$ be a set of m submitted jobs that are required to be scheduled on a heterogeneous system (consisting of a CPU and multiple GPUs). In order to schedule a job, it is required to extract code features $F = \{F_0, F_1, \ldots, F_{22}\}$ (mentioned in Table 4.3) from the kernel code of each job c_i in $\mathbb{C}$. The job c_i in $\mathbb{C}$ is provided as an input to the *Kernel Feature Extractor* (KFE) to determine *Count of each code feature* (CF_i) (i.e., the number of times a code feature occur

in the kernel code of a job) in the kernel code of the job. This relationship is represented by Eq. 4.1:

$$C_i \xrightarrow{KFE} \aleph \tag{4.1}$$

where $\aleph_i = \{CF_0, CF_1, \ldots, CF_{22}\}$ represents the count of each code-feature in c_i. Next step is to determine the device suitability of c_i using *Machine Learning based Device Suitability* (ML_{DS}) model. For this purpose, $\aleph_i$ for c_i is provided as an input to the ML_{DS} which classify c_i as either suitable to CPU execution or suitable to i^{th} GPU execution. This relationship is represented in Eq. 4.2:

$$\aleph_i \xrightarrow{ML_{DS}} DS(c_i)|(DS(c_i) = CPU) \vee (DS(c_i) = GPU(i)) \tag{4.2}$$

where $DS(c_i)$ represents *Device Suitability* of c_i. Next, the speedup prediction of c_i on the suitable device is obtained using *Machine Learning-based Speedup Predictor* (ML_{SP}). In this regard, $\aleph_i$, $DS(c_i)$ and N (where N represents input size of c_i) are provided as an input to ML_{SP}. The output of ML_{SP} is the *Speedup of* c_i on a suitable device (SPC_i) where the suitable device may be either a CPU or GPU(i). This relationship is represented in Eq. 4.3:

$$\aleph_i \wedge DS(c_i) \wedge N \xrightarrow{ML_{SP}} SPC_i \tag{4.3}$$

Moreover, to balance the load across a CPU and GPU(s), computation requirement of each job c_i (in terms of Flops) is estimated using *Computational Assessment* (CA) module. The job c_i is provided as an input to the CA and it estimates Computational Requirement of c_i (CRC_i). This relationship is represented by Eq. 4.4.

$$C_i \xrightarrow{CA} CRC_i \tag{4.4}$$

After determining $DS(c_i)$, SPC_i, and CRC_i of all jobs in $\mathbb{C}$, mapping of jobs to compute devices is performed. For this purpose, consider A_{cpu} be a subset of $\mathbb{C}$ jobs from $\mathbb{C}$ displaying device suitability to CPU, such that $A_{cpu} = A_{cpu}(1), A_{cpu}(2), \ldots, A_{cpu}(m) \mid SPA_{cpu}(i) \leq SPA_{cpu}(i+1)$ in sorted order with respect to predicted speedup (higher speedup first) of i^{th} job ($SPA_{cpu}(i)$). Moreover,

let $A_{gpu(i)} = \{A_{gpu(i)}(1), A_{gpu(i)}(2), \ldots, A_{gpu(i)}(n) \mid SPA_{gpu(i)}(j) \leq SPA_{gpu(i)}(j+1)\}$ be a subset of n jobs from $\mathbb{C}$ displaying device suitability to the i^{th} GPU, in sorted order with respect to speed up (lower speed up first) of the j^{th} job ($SPA_{gpu(i)}(j)$). A_{cpu} and A_{gpu} are concatenated together to create the final job pool J of $m+n$ jobs which is given by the following equation:

$$J = A_{cpu}||A_{gpu} = \{A_{cpu}(1), A_{cpu}(2), ..., A_{cpu}(m)||A_{gpu}(1), A_{gpu}(2), ..., A_{gpu}(n)\} \tag{4.5}$$

From Eq. 4.5

$$J = \{A_{cpu}(1), A_{cpu}(2), ..., A_{cpu}(m), A_{gpu}(1), A_{gpu}(2), ..., A_{gpu}(n)\} \tag{4.6}$$

To adhere to E-OSched terminology, members of set J in Eq. 4.6 are renamed as:

$$J = \{J_1, J_2, ..., J_p, J_{p+1}, J_{p+2}, ..., J_q\} \tag{4.7}$$

where

$J_1 \Rightarrow A_{cpu}(1)$

$J_p \Rightarrow A_{cpu}(m)$

$J_{p+1} \Rightarrow A_{gpu}(1)$

and

$J_q \Rightarrow A_{gpu}(n)$

In order to determine job assignment to a CPU and GPUs, *total computation requirement (J_{CR})* of all the jobs in J and *total processing speed of all processors (T_{CP})* are required, which are listed in following equations:

$$J_{CR} = \sum_{i=1}^{n} CRJ_i \tag{4.8}$$

$$T_{CP} = \sum_{i=1}^{m} PS_i \tag{4.9}$$

Where there are m processors in the heterogeneous system (a multi-core CPU and GPUs). From Eq. 4.8 and Eq. 4.9, the *computational share of CPU (CS_{cpu})*,

representing the portion of computation that is assigned to a CPU from J_{CR}, can be calculated by using the formula given in Eq. 4.10.

$$CS_{cpu} = \left(\frac{P_c}{T_{CP}} \times J_{CR}\right) - \alpha \tag{4.10}$$

Where P_c represents the *processing speed of a CPU* and α is the *adjustment factor* to balance the load across CPU and GPUs. The value of α is obtained empirically and is explained in the Section 3.4.2.1 of the previous chapter.

From Eq. 4.10, to determine jobs that are assigned to *CPU Job Queue (CJQ)*, let $J_{cpu} = J_1, J_2, \ldots, J_q$ be a subset of 1^{st} q jobs from job pool J. Here $Ceiling_{cpu}$ is required to represent the upper boundary for a number of jobs in J_{cpu}, which are assigned to CPU. The value of $Ceiling_{cpu}$ is calculated as the sum of CS_{cpu} and $1/3^{rd}$ of *computation requirement of* q^{th} *job* (CRJ_q):

$$Ceiling_{cpu} = CS_{cpu} + \frac{1}{3}CRJ_q \tag{4.11}$$

Where the ratio 1/3 in Eq. 4.11 ensures that q^{th} job should be mapped to CPU only if some part of its computation (i.e., 1/3) falls within the limit of CS_{cpu}. Mapping the q^{th} job to CPU (whose $1/3^{rd}$ part can be accommodated within the CPU share) will cause a minor load imbalance as compared to mapping the job to a GPU.

Using Eq. 4.11, the number of jobs that are assigned to CJQ is given by:

$$CJQ = \begin{cases} J_{cpu} & \sum_{i=1}^{q} CRJ_i \leq Ceiling_{cpu} \\ J_{cpu} - J_q & \sum_{i=1}^{q} CRJ_i > Ceiling_{cpu} \end{cases} \tag{4.12}$$

After job assignment to CJQ, another set J_{update} is formed by subtracting CJQ from J:

$$J_{update} = J \setminus CJQ = \{job \mid job \in J \wedge job \notin CJQ\} \tag{4.13}$$

Similarly, for job assignment to GJQ(s), Eq. 4.8 and Eq. 4.9 are used to calculate CS_{gpu}(i), which represents the computational share of ith GPU. CS_{gpu}(i) is

calculated by the relationship given below:

$$CS_{gpu}(i) = \left(\frac{P_G(i)}{T_{CP}} \times J_{CR}\right) + \frac{\alpha}{g} \tag{4.14}$$

Where $P_G(i)$ represents the *processing speed* of ith *GPU* and g is the *number of GPUs* in the system. From Eq. 4.14, to determine jobs that are assigned to *GJQ(i)*, let $J_{gpu} = \{J_1, J_2, \ldots, J_r\}$ be a subset of 1^{st} r jobs from J_{update}. Here, $Ceiling_{gpu}$(i) represents the upper boundary for a number of jobs from J_{gpu}, which are assigned to i^{th} GPU. It is equal to the sum of $CS_{gpu}(i)$ and $1/3^{rd}$ of computation requirement of the r^{th} job (CRJ_r) and is represented mathematically as:

$$Ceiling_{gpu}(i) = CS_{gpu}(i) + \frac{1}{3}CRH_r \tag{4.15}$$

where the $1/3$ CRJ_r represents a majority computing part of the r^{th} job (similar to the explanation related to q^{th} job in Eq. 4.11). Using Eq. 4.15, the number of jobs that are assigned to GJQ(i), for i_{th} GPU given by:

$$GJQ(i) = \begin{cases} J_{gpu} & \sum_{i=1}^{r} CRJ_i \leq Ceiling_{gpu}(i) \\ J_{gpu} - J_q & \sum_{i=1}^{r} CRJ_i > Ceiling_{gpu}(i) \\ J_{update} & i == g \end{cases} \tag{4.16}$$

After job assignment to GJQ(i), the set J_{update} will be updated by subtracting GJQ(i) from J_{update}.

$$J_{update} = J_{update} \setminus CJQ(i) = job \mid job \in J_{update} \wedge job \notin GJQ(i) \tag{4.17}$$

After the completion of jobs assignment to CJQ and GJQ(s), a family of sets $Q = \{CJQ, GJQ_1, GJQ_2, ..., GJQ_g\}$ over J is obtained. The output set Q is governed by the constraints given in Eq. 4.18, Eq. 4.19, and Eq. 4.20.

$$\emptyset \notin Q \tag{4.18}$$

Eq. 4.18 ensures that CPU job-set and GPU(s) job-sets in Q cannot be an empty set $\emptyset$.

$$\cup_{A \in Q} A = J \tag{4.19}$$

Eq. 4.19 specifies that union of all member set (of Q) is equal to job pool J.

$$(\forall A, B \in Q) A \neq B \Rightarrow A \cap B = \emptyset \tag{4.20}$$

Eq. 4.20 stipulates that intersection for all non-equivalent member sets A and B of Q would be an empty set $\emptyset$.

4.3.3 Troodon Algorithm

Pseudo-code for the Troodon scheduling mechanism is presented in Algorithm 4.1. The kernel-code of each job is provided as an input to the KFE module to extract code-features. The extracted code features are then provided to the ML_{DS} module to determine device-suitability of the job. The output of KFE, ML_{DS} along with N (input size of the job c_i) are used to determine the speedup of the job (on the suitable device) using ML_{SP} module. Moreover, CRJ of the job is also determined using the CA module. Thereafter, $A_{cpu}[\,]$ and $A_{gpu}[\,]$ are sorted and concatenated together to form J which is used for job mapping to the computing devices (i.e., CPU or GPU).

Next, the E-OSched scheduling mechanism has been adapted to map jobs to CPU or GPU(s). First of all, J_{CR} and T_{CP} are initialized. Next, the values of J_{CR} and T_{CP} are calculated for the purpose of determining total computation-requirement of all jobs in the job pool and total processing speed of all processors in the system, respectively. The value of α is calculated in order to compute CS_{cpu} that is followed by the initialization of CJA_{cpu} (the computation requirement of jobs assigned to CPU). Subsequently, the number of jobs which are assigned to CJQ are determined. The value of J_{update} is calculated by subtracting CJQ from J. For job assignment to i^{th} GPU, first $CS_{gpu}(i)$ is calculated and $CJA_{gpu}(i)$, which is the computation requirement of jobs assigned to i^{th} GPU is initialized. The

number of jobs that are assigned to GJQ(i) is determined and J_{update} is updated by subtracting GJQ(i) from J_{update}.

Algorithm 4.1: Troodon job scheduling algorithm

Input :

i A job pool $\mathbb{C}$

ii List of CPU and GPU(s) sorted in increasing order of processing speed

Output:

i All jobs in the job pool are assigned to CPU and GPU(s) for execution

for *i=1 to Total Jobs* **do**
  $\aleph[CF_0, CF_2, ..., CF_{22}] \leftarrow KFF(c_i)$
  $DS(c_i) \leftarrow ML_{DS}(\aleph_i[CF_0, CF_2, ..., CF_{22}])$
  if $DS(c_i) = CPU$ **then**
    $A_{cpu}[\,] \leftarrow c_i$
  end
  else
    $A_{gpu}[\,] \leftarrow c_i$
  end
end
$SPC_i \leftarrow ML_{SP}(\aleph_i[CF_0, CF_2, ..., CF_{22}], DS(c_i), N_i)$
$CRC_i \leftarrow CA(c_i)$
SORT $A_{cpu}[\,]$ in Descending Order according to SPC_i // $SPC_i \geq SPC_{i+1}$
SORT $A_{gpu}[\,]$ in Ascending Order according to SPC_i // $SPC_i \leq SPC_{i+1}$
$J[J_1, J_2, ..., J_n] = A_{cpu}[\,]$ **CONCATENATE** $A_{gpu}[\,]$
$J_{CR} \leftarrow 0$
$T_{CP} \leftarrow 0$
for *i=1 to Total Jobs* **do**
  $J_{CR} = J_{CR} + CRJ_i$
end
for *i=1 to Total Processors* **do**
  $T_{CP} = T_{CP} + PS_i$
end
$\alpha = (4.223/100) \times J_{CR}$ // *α's value is obtained empirically*
$CS_{gpu} = ((P_c/T_{CP}) \times J_{CR}) - \alpha$

Algorithm 4.1: Troodon job scheduling algorithm (Continued...)

INITIATE $CJA_{cpu} \leftarrow 0$

/* CJA_{cpu} = Computation Requirements of CPU Jobs */

for *i=1 to* q_{th} *job from job pool* **do**
    $CJA_{cpu} = CJA_{cpu} + CRJ_i$
    if $CJA_{cpu} \leq CS_{cpu} + 1/3(CRJ_i)$ **then**
        $CJQ \leftarrow J_i$
    end
end

$J_{update} = J - CJQ$

for *i=1 to Total GPUs* **do**
    $CS_{gpu}(i) = ((P_G(i)/T_{CP}) \times J_{CR}) + (\alpha/TotalGPUs)$
    INITIATE $CJA_{gpu}(i) \leftarrow 0$ /* CJA_{gpu} = Computation Requirements of Jobs that are assigned to ith GPU */
    for *j=1 to* r^{th} *job from* J_{update} **do**
        $CJA_{gpu} = CJA_{gpu} + CRJ_i$
        if $CJA_{gpu} \leq CS_{gpu}(i) + 1/3(CRJ_i)$ **then**
            $GJQ(i) \leftarrow J_i$
        end
        if i = *Total GPUs* **then**
            $GJQ(i) \leftarrow J_{update}$
        end
    end
    $J_{update} = J_{update} - CJQ(i)$
end

4.3.4 Scheduling overhead

The scheduling overhead of the Troodon scheduling heuristic is O(N^2) as it uses the E-OSched scheduling scheme for job scheduling. However, the Troodon also performs additional steps of feature extraction, device suitability classification, and speedup prediction before job scheduling. The training phase of the device suitability classifier and the speedup predictor took 10–12 hours. During deployment, the overhead of feature extraction, device suitability classification, and speedup prediction, in our case, amounts to 15–20 milliseconds which is negligible when compared to the overall execution time of the job pool. Similar to the E-OSched

heuristic, as there is no data transfer involved during scheduling, Troodon does not incur communication cost.

4.3.5 Feature Extraction

To extract the OpenCL kernel code-features, a *static code analyzer* has been developed[8]. The extracted code-features are then used to train the device-suitability machine learning classifier. However, all the available code-features (i.e., 23 in total and mentioned in Table 4.1) are not of equal importance in classifying device suitability of a kernel and speedup prediction. Therefore, selected features were identified to be used in device suitability classification and speedup prediction. The feature selection process is detailed in section 4.3.6. The extracted code-features are similar to the feature set of [27, 63] but are greater in number. The activities performed for the code-feature extraction are depicted in Figure 4.4 that comprises of Clang [106] front compiler, Intermediate Representation formation, LLVM [106] Passes, and Static Analyzer. The description of each phase is provided below.

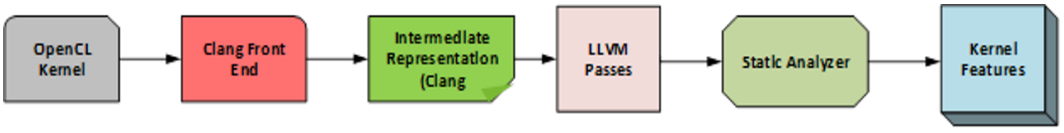

FIGURE 4.4: Static code feature extraction process

An OpenCL kernel is compiled (using Clang frontend compiler) to make sure that the kernel code is error-free. The Clang LLVM parser is used to extract code-features based on LLVM Intermediate Representation (IR). For extraction of the undetected code-features (by Clang LLVM frontend compiler), a python-based script has been developed and used to extract the remaining code features (such as loop-count and iterations of each loop, etc. as shown in Table 4.1). The ML-based Device suitability classifier and Speedup predictor were trained (offline). In

[8]The static code analyzer is publicly available at the following link:
https://github.com/Troodon-Scheduler/StaticCodeAnalyzer

addition to the OpenCL kernel code features, the data-size feature is also provided for the training purpose to the Speedup predictor.

TABLE 4.1: Code Features Extracted for Training of the Device Suitability and Speedup Prediction Models

No.	**Feature Name**	**No.**	**Feature Name**
1	Data Size	13	Total number of Addition (Float Datatype) instruction
2	Total number of Return statement	14	Total number of Addition (Integer Datatype) instruction
3	Total number of Control Statement	15	Total number of Subtraction (Float Datatype)
4	Total number of Allocation instruction	16	Total number of Subtraction (Integer Datatype)
5	Total number of Load Instructions	17	Total number of Function Call instruction
6	Total number of Store Instructions	18	Total number of Functions
7	Total number of Multiplication (Float Datatype) Operation	19	Total number of Blocks
8	Total number of Addition (Integer Datatype) Instruction	20	Total number of Instructions
9	Total number of Multiplication (Integer Datatype) Instruction	21	Total number of Float Operation
10	Total number of Division (Float Datatype) instruction	22	Total number of Integer Operation
11	Total number of Division (Integer Datatype) instruction	23	Total number of Loop Operation
12	Total number of Condition Check instruction		

4.3.6 Feature Selection

The motivation for using a smaller number of features (for predictive models) is to reduced overfitting issues, improved accuracy, and decreased training time of the algorithms. In this chapter, two feature-selection techniques i.e., *correlation analysis* and *tree-based* feature selection has been employed. The correlation analysis provides an indication of the relative change between the two features. If a change in a feature's value causes a change in other feature's value too then the features are referred to as co-related [107]. The features are referred to as *positively co-related* if the change of values is in the same direction (i.e., *increased or decreased*) for both features. The features are denoted as negatively co-related if the change of one feature's value causes an inverse change in another feature's value. The correlation analysis is very crucial for the selection of useful training features because employing highly correlated features do not help the predictive model to achieve good prediction accuracy. The tree-based estimators i.e., Random forest can be used to compute feature importance, which in turn can be used to discard irrelevant features based on information gain [108]. Figure 4.5 shows that the features "1:Datasize", "7:Total number of Multiplication (float datatype) operations", "16-Total number of subtraction(Integer datatype) instruction", "13-Total number of addition (Float datatype) instructions", "17-Total no of function call instructions", "9-Total number of multiplication (Integer datatype) Instructions", "23-Total number of loop operations", and "21-Total number of float operation" has the smaller positive or a negative correlation with all the other employed features which are ranked as top features by the tree-based feature selection method (shown in Figure 4.6). With the analysis of the correlation matrix and ranking of features, it is concluded that the features "6-Total number of Store Instructions", "2-Total number of Return statement", "18-Total number of Functions", "4- Total number of Allocation instruction", and "11-Total number of Division (Integer Datatype) instruction" exhibit low or no contribution to the output class (of the two ML models); therefore, these features were omitted from the training of the models.

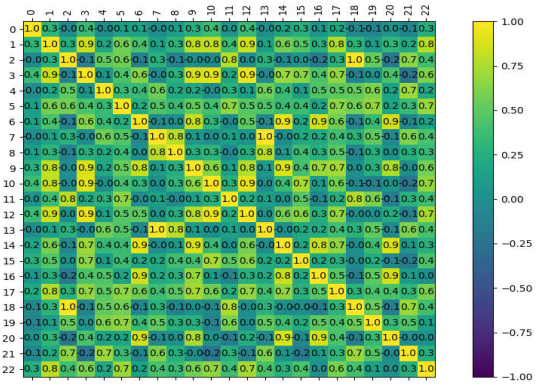

FIGURE 4.5: Correlation analysis of employed code features

The selected code-features cover most of the computation and communication events of OpenCL kernels and are very useful in accurately classifying jobs according to device suitability and predicting obtainable speedup on suitable device. However, if jobs that are not suitable for data-parallelism are mapped to OpenCL then the device suitability classifier and speedup predictor may suffer from degradation in prediction accuracy.

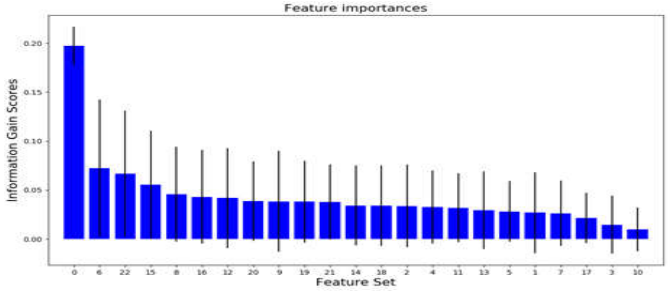

FIGURE 4.6: Feature set ranking based on information gain

4.3.7 Model Description

To build device suitability classifier and speedup predictor models, a 3-step mechanism based on a supervised machine learning approach was performed. (i) *Generation of diverse training data*, (ii) *Prediction models' training*, and (iii) *Prediction model's deployment*. 42 OpenCL applications from mainstream benchmark suits (i.e., AMD [56], Polybench [98], Rodinia [99], and Parboil [97]) are used to build

a diverse training data. Kernel-codes from these benchmarks are provided to a static code analyzer which extracts the code-features in a first step. The reduced set of features are provided to the ML models for training purpose. Figure 4.6 depicts the complete feature extraction process used for training (shown in Figure 4.7) of the ML models.

To train device suitability classifier and speedup predictor, it is required to label training OpenCL kernels and the extracted code-features. Therefore, the output class for device suitability model was found by executing each application on a CPU and a GPU device. The execution time for each OpenCL kernel is noted for both the CPU and GPU devices. The device that produces the lower execution time is labeled as a suitable device for that kernel. Similarly, all the applications in the training set are labeled as CPU and GPU suited considering the lower execution time for a certain device.

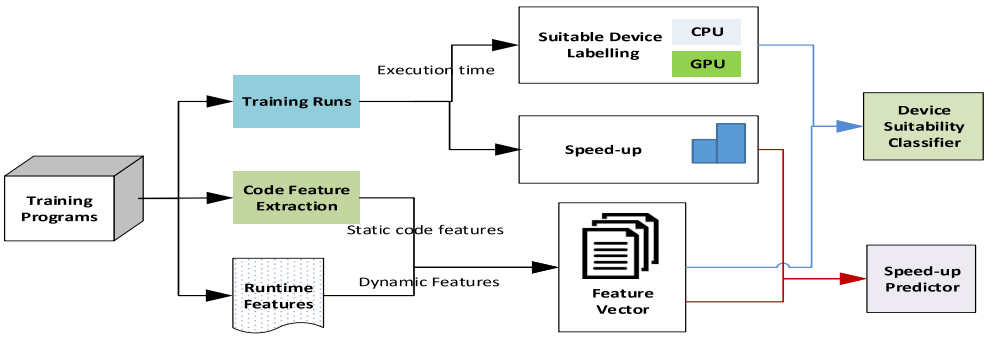

FIGURE 4.7: Training process of prediction models

For the relative speedup predictive model. The execution time of a kernel on a slower device is divided by the execution time of the same kernel on a faster device to obtain the relative speedup of the faster device (selected device) as compared to the slower device (non-selected device). It is also observed that a large number of the high-performance computing OpenCL applications are more suitable for a GPU based execution as compared to a CPU based execution. This

observation leads us to the issue of class *imbalance* [109] (due to a significantly smaller number of the training samples (OpenCL kernels suitable for CPU based execution). To mitigate the data-imbalance problem, *SMOTE* [109] was employed which is a python toolbox to handle imbalanced data. Moreover, in this research, a *Tree-based Pipeline Optimization* Tool (TPOT) [110] (based on an evolutionary algorithm) is used to design and optimize the machine-learning pipelines. TPOT was employed to boost the classification accuracy of the machine-learning models.

TABLE 4.2: Tune parameters for device-suitability ML model

Methods	**Hyper Tuning Parameters**					
Feature Selection	1- Correlation Analysis	2- Tree Based Feature Selection				
Nystroem	Gamma	Kernel	n_components			
	0.65	Laplacian	8			
Zero Count	Count the number of features having zero value					
MaxAbs Scaler	Scale each feature by its maximum absolute value.					
Random Forest Calssifier	bootstrap	criterion	max features	min samples leaf	min samples split	n estimators
	False	entropy	0.6	2	15	100

The labeled data (i.e., *CPU* or *GPU*) is provided to the TPOT classification class to determine the device suitability. Furthermore, the speedup category labeled data is provided to the TPOT regression class to predict the potential relative speedup on a suitable device as compared to the unsuitable computing device. Both the TPOT classes return hyper-tuned models for both types of data (as shown in Table 4.2 and Table 4.3).

TABLE 4.3: Tune parameters for relative speedup ML model

Methods	**Hyper Tuning Parameters**			
Polynomial Features	Degree	Include bias		Interaction only
	2	False		False
Standard Scaler	Standardize features by removing the mean and scaling to unit variance.			
Gradient Boosting Regressor	n estimator	Learning rate	Max depth	loss
	100	0.1	1	ls

4.3.8 Troodon Usage Scenario

Troodon can be used in a data center and Cloud environment where multiple user jobs are mapped to a single node. The service provider's requirement is to execute user jobs using a minimum number of resources while providing acceptable service quality. The user will provide a job pool of applications to the system along with OpenCL source code, input data sizes and computational requirements of each job. The Troodon will produce a schedule that will execute user submitted job pool in the least amount of time. Furthermore, the user can also analyze multiple aspects of each submitted job i.e., device suitability of a job and predicted speedup of a job when executed on a suitable device. In the case of a homogeneous cluster in a data center, the machine learning modules of the Troodon will be trained once and then deployed on all machines. In the case of a heterogeneous data center environment, the machine learning modules of the Troodon have to be trained on each architecturally different type of machine. Moreover, the Troodon can also be used in user systems to schedule general-purpose applications that are developed in OpenCL to take advantage of the acceleration provided by GPUs [111].

4.4 Experiments and Results

We evaluated our approach on 02 separate CPU-GPU heterogeneous systems names as *System1* and *System2* respectively. Both the systems host Ubuntu 16.04 Linux operating system.

TABLE 4.4: Experimental setup

	System1		**System2**		
Device	**CPU (i5 4550)**	**GPU1 (GTX 760)**	**CPU (i5 4550)**	**GPU1 (GTX 760)**	**GPU2 (GT 740)**
Architecture	Haswell	Kepler	Haswell	Kepler	Kepler
Base Clock	3.2 GHz	0.980 GHz	3.2 GHz	0.980 GHz	0.993 GHz
Boost Clock	3.4 GHz	1.033 GHz	3.4 GHz	1.033 GHz	NA
Total Cores	4	1152 (CUDA cores)	4	1152 (CUDA cores)	384 (CUDA cores)
Memory	8 GB	2 GB	8 GB	2 GB	2 GB
Bandwidth	25.6 GB/s	192.2 GB/s	25.6 GB/s	192.2 GB/s	192.2 GB/s
Performance (Single Precision)	409.6 GFLOPS	2257.9 GFLOPS	409.6 GFLOPS	2257.9 GFLOPS	762.6 GFLOPS
OpenCL SDK	Intel SDK for OpenCL 2016	CUDA 8.0	Intel SDK for OpenCL 2016	CUDA 8.0	CUDA 8.0
Compiler	GCC 5.4.0	Nvcc	GCC 5.4.0	Nvcc	Nvcc

Moreover, Intel SDK for OpenCL with GCC 5.4.0 was used for CPU compilation

and CUDA 8.0 with Nvcc was used for GPU compilation. The specifications of the employed machines are presented in Table 4.4. For experimentations, 17 benchmark applications that belong to 4 mainstream benchmark suits are used. The benchmarks are executed using several problem sizes resulting in a job pool of total of 199 jobs (as shown in Table 4.5. All the experiments are conducted 10 times and mean values are reported.

4.4.1 Scheduling Policies and Evaluation Metrics

We compared our approach against the following seven scheduling techniques:

1. ***CPU-Only:*** All jobs are dispatched to a CPU device for execution. It is a naive heuristic that forms a baseline for performance comparison [112].

2. ***GPU-Only:*** All jobs are dispatched to a GPU device for execution. This scheduling scheme indicates that programmers generally tend to map compute-intensive applications on GPUs that mostly leads to under-utilization of the other available computing devices such as CPU [112].

3. ***Alternate Assignment (AA):*** In AA based scheduling [64], all jobs are added in a random order in the job pool. The jobs are assigned alternately to a CPU and a GPU (or any other available device). The AA scheduling policy is used to show that assigning each processor almost equal number of jobs without considering jobs' device suitability results in less favorable execution and lower throughput of the job pool;

4. ***Input size guided (ISG):*** In this scheduling scheme [27], jobs are sorted in ascending order of the data size. CPU starts execution of the task queue from the top of the sorted queue (*smallest job first*) while the GPU computes tasks from the end of the sorted task queue (executing the largest job first). Moreover, the ISG scheduling heuristic is unable to utilize more than one GPU in a multi-GPU setup. Therefore, the negative effect of not utilizing all the GPUs in the system is highlighted.

5. ***Device Suitability (DS):*** All jobs are assigned to a device on which a job has shorter execution time i.e., CPU suitable jobs are assigned to a CPU and GPU suitable jobs are assigned to a GPU. As the vast majority of OpenCL applications are well suited for GPU based execution, the DS scheduling scheme is used to show that mapping jobs using device suitability only will result in severe load-imbalance between a CPU and a GPU;

6. ***Wen et al.:*** The scheduling heuristic of [27] is based on machine learning based classification that employs both the static code features (such as *instructions*, *math functions*, *barriers*, etc.) and dynamic runtime features (such as *local_work_size*, *global_work_size*, *input data size*, etc.) to determine device suitability of OpenCL kernels;

7. ***E-OSched:*** This scheduling algorithm [81] first sorts job pool in ascending order (smaller sized jobs first) considering the jobs' computational requirements. The jobs from one end of the job-pool (smaller size jobs having lower computational requirement) are assigned to a CPU device whereas the jobs from the other end of the job pool (jobs having higher computation requirements) are assigned to a GPU device. Jobs assignment to a CPU and GPUs are carried out in a resource-aware manner considering the proportion of the computing power provided by the devices.

TABLE 4.5: Benchmark applications and the employed input data sizes

Benchmark Suits	**Applications**	**Input Data Size**	**Total Versions**
AMD	Matrix Multiplication	1,769,472—12,582,912	3
	Binomial Options	32,768—294,912	9
	Bitonic Sort	32,768—268,435,456	13

Benchmark Suits	**Applications**	**Input Data Size**	**Total Versions**
	Fast Walsh Transform	8,192—221,184	17
	Matrix Transpose	131,072—536,870,912	6
	Discrete Cosine Transformation	2,097,152—1,887,436,800	17
	Floyd Warshall	524,288—25,690,112	6
Polybench	3MM (3 Matrix Multiplications)	7,000,000—17,920,000	3
	GEMM (Matrix-multiply C=alpha.A.B+beta.C)	3,000,000—27,000,000	5
	GESUMMV (Scalar, Vector and Matrix Multiplication)	8,012,000—1,800,180,000	17
	MVT (Matrix Vector Product and Transpose)	4,016,000—900,240,000	17
	ATAX (Matrix Transpose and Vector Multiplication)	4,012,000—900,180,000	17
	2MM (2 Matrix Multiplications)	5,000,000—45,000,000	5
	2DCONV (2D Convolution kernel)	2000000—1,568,000,000	17
	3DCONV (3D Convolution kernel)	1,000,000—1,728,000,000	17
Rodinia	BFS (Breadth First Search)	39,360—327,141,488	14
Own Developed	Matrix-Vector Multiplication	4,202,496—1,514,299,392	16

For the performance evaluation, the following performance metrics were considered:

1. ***Execution time:*** execution time exhibits the time consumed in the computation of all jobs. The results have been reported with the confidence level of 95% while in the discussion of the results and geometric mean (Geo Mean) have been compared.

2. ***Throughput:*** represents the number of jobs completed per unit time. During the discussion of the result, Geo Mean has been used to compare the employed scheduling heuristics.

3. ***Average time (of a job):*** is defined as an average amount of time taken by a job (of a job pool) to complete its execution. Average time results have been compared using the Geo Mean. The reported results have a confidence level of 95%.

4. ***Load balance:*** Load balance is a distribution of job pool among the computing devices (i.e., CPU and GPUs) such that all the processing devices accomplish the execution of the assigned jobs within the approximately same time duration. Load balance is calculated using the formula in Eq. 4.21. The higher value of this metric shows a more load-balanced execution.

$$\%age\ load\ balance = 100 - \left(\left(1 - \frac{Execution\ time\ of\ fastestprocessor\ assigned\ jobs}{Execution\ time\ of\ slowest\ processor\ assigned\ jobs}\right) \times 100\right) \tag{4.21}$$

Where *Execution time of the slowest processor assigned jobs* represents Execution time of the job pool.

4.4.2 Scheduling Results

Figure 4.8 presents the execution time-based results of the proposed and the other scheduling schemes. These results show that the proposed scheduling mechanism

named Troodon outperforms all the other scheduling schemes in terms of the execution time of the job pool (presented in Section 4.4.1). As compared to the baseline scheduling (i.e., *CPU-Only*), the Troodon scheduling scheme consumes 182% lower execution time. The *Troodon* scheduling heuristic consumes 108% reduced execution time as compared to the GPU-based execution of the job pool (i.e., *GPU-Only*). The *Troodon* scheduling heuristic consumes 78%, 65%, and 41% reduced execution time as compared to the DS, ISG [27], and AA scheduling schemes, respectively as shown in Figure 4.8. As compared to the state-of-the-art scheduling heuristics such as Wen et al. [27] and E-OSched [81], the proposed scheduling scheme further improves the execution time by consuming 38% and 10% reduced execution time, respectively.

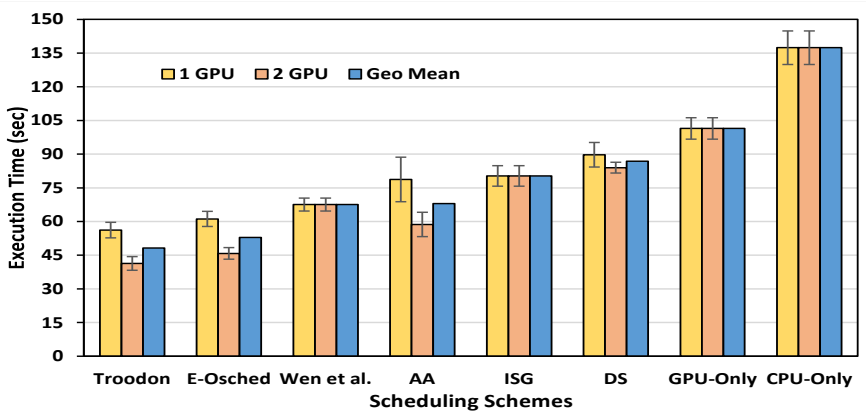

FIGURE 4.8: Execution time of scheduling schemes for job pool

4.4.2.1 Average Execution Time

Figure 4.9 shows the average execution time-based results for the complete execution of the job pool. As compared to the *CPU-Only* and *GPU-Only* scheduling heuristics, the proposed heuristic *Troodon* results in 2.85 times and 2.1 times reduced average execution time (for a job), respectively. As compared to the DS, ISG [27], and AA scheduling schemes the proposed scheme *Troodon* consume on average 1.8, 1.67, and 1.41 times reduced execution time, respectively.

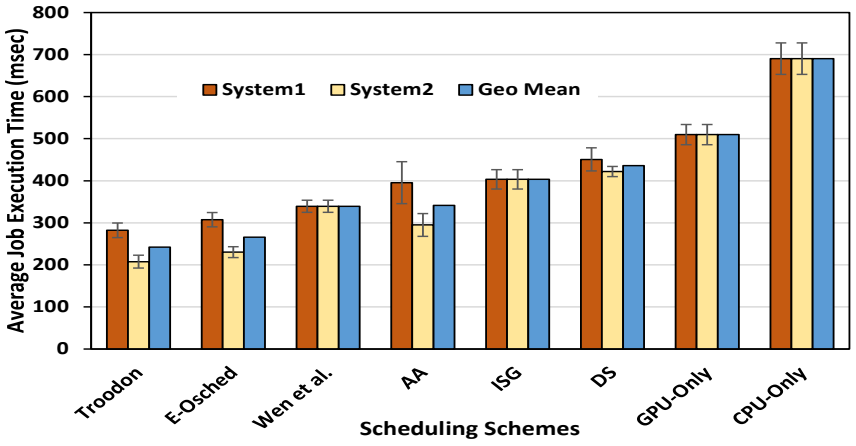

FIGURE 4.9: Average execution time of a job in the job pool

As compared to *Wen et al.* [27], and *E-OSched* [81] scheduling, the proposed scheduling scheme *Troodon* consumes on average 1.40 and 1.10 times reduced execution time, respectively.

4.4.2.2 Throughput Analysis

For throughput analysis, a state-of-the-art scheduling heuristic proposed by Wen et al. [27] was considered as a baseline. Figure 4.10 presents the attained throughput of all the other schedulers as compared to the base-line scheduling heuristic [27].

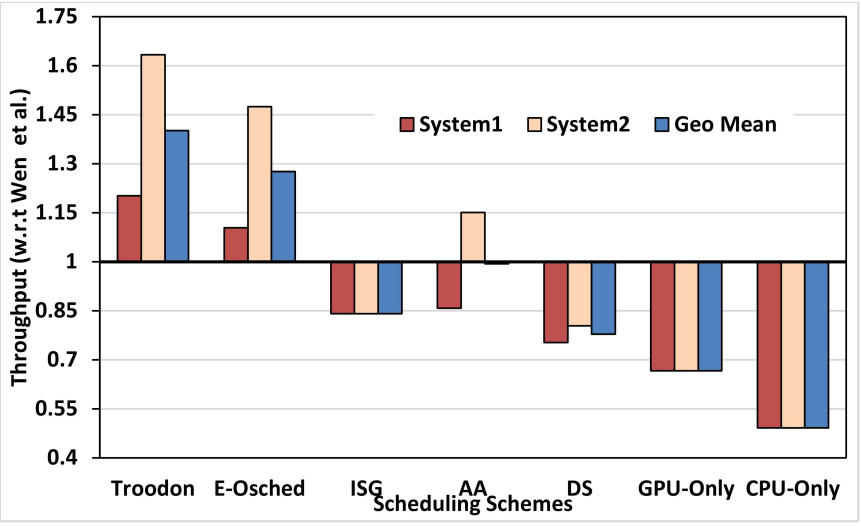

FIGURE 4.10: Throughput analysis

The proposed Troodon scheduler achieves the highest throughput as compared to the baseline (i.e., 40% higher throughput). The Troodon scheduler attains 10% higher throughput as compared to the *E-OSched* [81]. As compared to the DS, ISG [27], and AA, the *Troodon* scheduler attains 80%, 66%, and 41% higher throughput, respectively. As compared to the *CPU-Only* and *GPU-Only* scheduling mechanisms, the Troodon attains 185% and 110% improved throughput, respectively.

4.4.2.3 Load Balance Analysis

System1: Figure 4.11 presents the load balance analysis of the proposed and the other scheduling heuristics for *System1*. As shown in Figure 4.11, the Troodon heuristic achieves a load balance of 98.4%. When compared to the load balance achieved by DS and AA, the Troodon attains more load balance up to 79.7% and 27.8%, respectively. As compared to the ISG [27] heuristic, the *Troodon* scheduler induces a slight load imbalance of 0.8%.

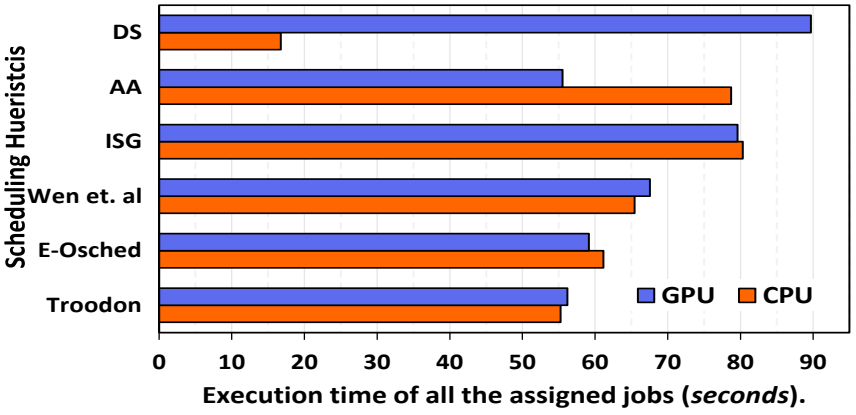

FIGURE 4.11: Achieved load balance by scheduling schemes on *System1*

However, the ISG [27] scheduling heuristic results in longer execution time as compared to the Troodon because it does not considers device suitability for scheduling. As compared to the *E-OSched* [81] and Wen et al. [27], the proposed scheduler Troodon attains up to 1.7% and 1.5% improved load balance, respectively.

System2: The load balance analysis of the Troodon and other scheduling schemes for *System2* is presented in Figure 4.12. The ISG [27] and Wen et al. [27] have not been used for load balance analysis on *System2* as these scheduling heuristics are designed to use only one GPU during job pool execution. The proposed Troodon scheduling heuristic achieves a load balance of 94.33% which is approximately equal (1% improved load balance) to the load balance achieved by the *E-OSched* [81]. The Troodon achieves 373% and 13% more load balance when compared to the DS and AA scheduling heuristics.

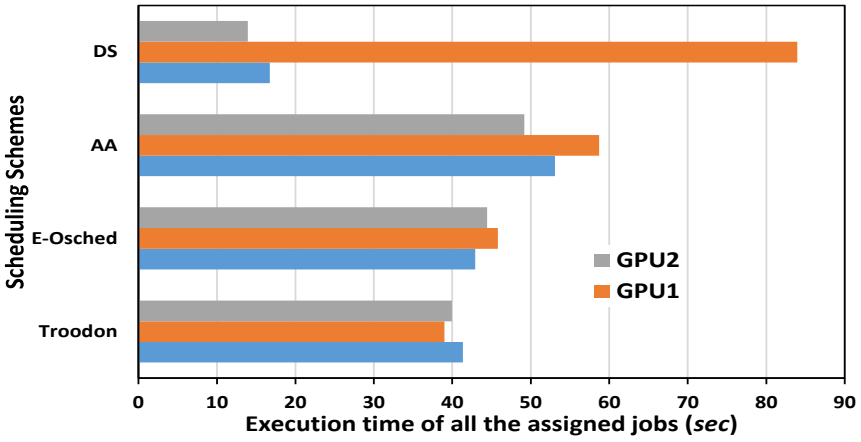

FIGURE 4.12: Achieved load balance by scheduling schemes on *System2*

4.4.2.4 Results Discussion

Experimental results (presented in Figures 4.8 — 4.12) show that the proposed scheduling mechanism Troodon outperforms all the other scheduling schemes with respect to execution time (for complete job pool), average job execution, throughput, and load balance performance.

Metrics on employed *System1* and *System2*. The improved performance of the proposed Troodon scheduling heuristic is due to the inherent 2-tier scheduling approach. First, all the jobs (in the job pool) are categorized according to the device suitability. In order to improve load balance, jobs with low potential speedup are

transferred from the overloaded job queues of the devices to the job queues of the device having a smaller number of jobs (as compared to the potential computing capability of the devices). The load balanced and a device suited application mapping by the proposed *Troodon* scheduling heuristic results in reduced execution time (for the job pool), higher system throughput, and higher device utilization as compared to state-of-the-art heuristics. The state-of-the-art scheduling schemes such as ISG [27] and E-OSched [81] are unaware of the device suitability of the applications resulting in decreased performances as compared to the Troodon. The AA scheduling mechanism [64] is oblivious to device-suitability of the jobs and because of the inherent *fair-scheduling* mechanism causes slower devices (such as CPUs) overloaded with more computational load and faster devices (such as GPUs) with less amount of computational work. The scheduling heuristic proposed by Wen et al. [27] considers the device suitability to map jobs on devices, however, it does not employ a load balancing mechanism to evenly distribute the workload across the computing devices.

4.4.3 Prediction Models Discussion

To train *Device Suitability* and *Speedup* predictor classifiers, a tool named *Scikit-learn* [108] was used. To train the *Device Suitability* classifier, 10-fold cross-validation technique was used. The *split ratio* approach [113] is employed to train the *Speedup* predictor machine learning model. To build the Speedup prediction model, 80% of the data is used for the training purpose and 20% of the data for testing. The evaluation parameters such as *average precision*, *average recall*, and *mean Receiver Optimization Characteristics* (ROC) are calculated to evaluate the accuracy of the trained classifiers (i.e., device suitability and speedup predictor). ROC depicts the ratio of the *True Positive Rate* (TPR) against the *False Positive Rate* (FPR) for different thresholds of the data employed in classification. The mean ROC curve for *Device Suitability Classifier* is shown in Figure 4.12. Mean ROC curve for the device suitability classifier is 0.92, however, the precision-recall curve of class 0 (i.e., CPU device) is 0.98 whereas for the class 1 (i.e., for GPU

device) is 0.97. The F-measure score of the device suitability model is 0.94. The curve (black for Class 1 (GPU) and green for Class 0 (CPU)) in Figure 4.13 is closer to the Y-axis and the top border that signifies the higher correct detection decisions by the ML models for device suitability.

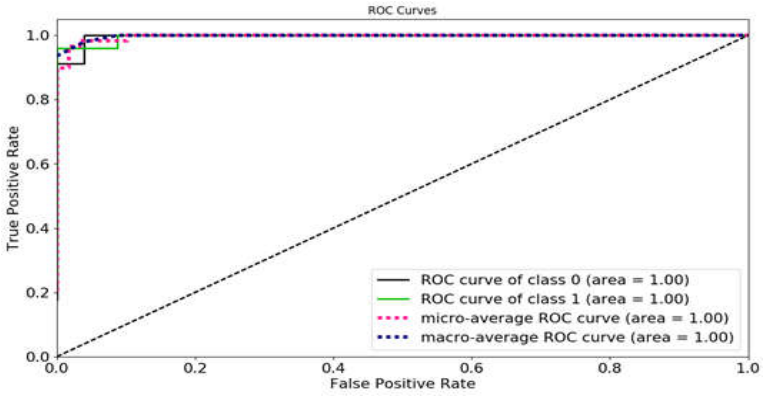

FIGURE 4.13: ROC of Device suitability classifier

This result also signifies that a higher TPR is observed with very low FPR. The high value of F-measure exhibits the fact that the machine learning model is capable of making fine distinctions. The high recall value represents that the correctly predicted class occurrences are very high.

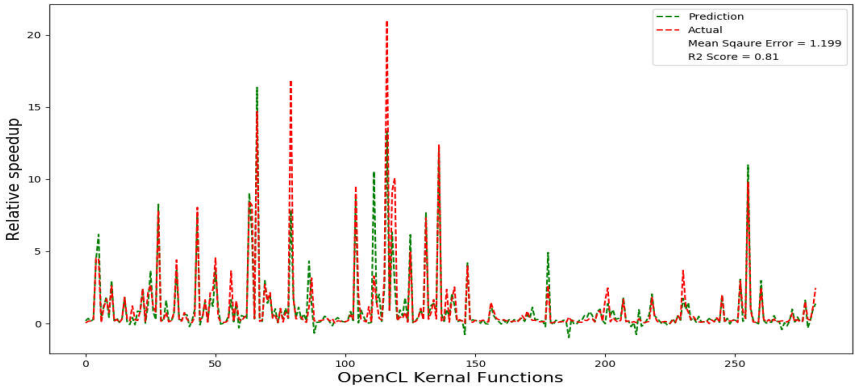

FIGURE 4.14: OpenCL kernel forecasting of speedup predictor

Figure 4.14 presents the relative speedup results obtained using the ML-based Speedup predictor model and the actual attained speedup value in the experiments. It is evident (from Figure 4.14) that the speedup predictor model attained impressive results with 2.52 mean square error (for 25 randomly selected OpenCL kernels). These results highlight that the trained ML models for the device suitability and speedup prediction produce accurate prediction results.

4.4.4 Analysis of Selected Features Impact on Device Suitability Model

After analyzing the statistics for each extracted code feature, some interesting facts have been highlighted about the role of a code feature in the Device Suitability model. It has been observed that the Device Suitability model tends to classify applications having low value for "Data-size" feature as CPU Suitable whereas the high value for "Data-size" feature are mostly classified as GPU Suitable. The reason is that a higher amount of data parallelism leads to better execution time performance on a GPU. Furthermore, when the input data size of an application's kernel is small, the speedup achieved due to GPU execution may be degraded due to data transfer overhead to/from GPU. The occurrence of features "Total no of Addition Integer instruction", "Total no of Addition Integer instruction" and "Total no of Multiplication Integer Instruction" tends to be low in case of GPU Suitable classified applications. Kernels in which there is a higher occurrence of features "Total no of Control Statement", "Total no of Loop", and "Total no of Condition Check instruction" are classified as CPU Suitable by the Device Suitability classifier. This leads to the conclusion that the application's kernel in which there is more branch divergence tends to be suitable for CPU execution.

4.4.5 Limitations

Scheduling with the Troodon heuristic requires that either computation requirement of all the submitted jobs are known beforehand or can be estimated using the

computational complexity formula. The Troodon also requires source code of the kernel/s of the submitted job in order to extract static code features. Moreover, the Troodon has been developed for a single node heterogeneous system (consisting of a CPU and one or more GPUs) and currently not applicable to a cluster of heterogeneous compute nodes. However, the Troodon can be easily extended for scheduling on multiple compute nodes by incorporating features such as data transfer delay from one node to another, hardware features, bandwidth, latency, etc.

4.5 Conclusions

This chapter presents Troodon, a novel application scheduler for CPU-GPU heterogeneous system to map data-parallel applications in a load balanced manner. In contrast to the state-of-the-art, the Troodon scheduler is based on a machine-learning model to gauge the applications' device-suitability, potential speedup (of applications on certain computing devices), and incorporate these in the scheduling decisions to map application in load balanced manner. The Troodon scheduling heuristic results in reduced execution time of the applications and higher system throughput. It has been evaluated that larger input data size plays an influential role in declaring a job as GPU-suitable by the Troodon. However, the high occurrence of arithmetic operations (e.g., addition instruction and multiplication instruction) and control statements in source code are influential in classifying a job as CPU-suitable. Moreover, the load balanced scheduling mechanism further improves the device utilization and system throughput. The Troodon scheduler was evaluated using several benchmarks and a large pool of OpenCL application (i.e., 199). As compared to the baseline scheduling schemes (i.e., CPU or GPU only), the Troodon scheduler consumed up to 2.82 times less time to execute the applications on average. As compared to the state-of-the-art scheduling heuristics, the Troodon consumes up to 38% reduced execution time on average. Moreover, in terms of system throughput, device utilization, and load balance the Troodon has

outperformed the other scheduling heuristics. The overall improvement in the system's performance due to the *Troodon* scheduling heuristic indicates that accurate machine learning based classification and prediction can play an important role in scheduling decisions. Furthermore, there are certain code features that play a significant role in device suitability of an application. In the future, it is intend to extend the scheduler to incorporate remote devices (over the network) to employ a cluster of heterogeneous devices for the execution of data-parallel applications in heterogeneous computing machines.

Chapter 5

Role of Kernel Fusion to Improve Device Utilization

5.1 Introduction

An OpenCL application consists of a *host* and *kernel* code. The host-code is always executed on the CPU and manages overall execution of an OpenCL application whereas the device-code consists of data-parallel part of the application and can be executed on any supported processor. Programmers tend to map their OpenCL applications to GPUs to obtain speed-up due to huge parallelism available in GPUs. With an increasing number of ported OpenCL applications, there is a need for a scheduler that can map compute kernels of multiple OpenCL applications (i.e., a job pool of submitted applications) on heterogeneous CPU-GPU systems in a load balanced manner. Such a job scheduler should take kernel mapping decisions considering the appropriateness of devices and applications needs to attain improved execution performance and higher job execution throughput (resulting in better device utilization too). Moreover, due to contrasting architectural differences between a CPU and a GPU, some jobs are better suited to GPU-based execution whereas others are better suited for CPU-based execution [114, 115].

In our previous work [65], a heterogeneous job scheduler named Troodon was developed that maps jobs to a CPU or GPU devices considering the device suitability and expected speed-up gain attainable on a certain computing device. Moreover, load balance is achieved between CPU and GPUs by considering jobs processing requirements and computing capabilities of a processor (e.g., CPU or GPU). The results indicated that the Troodon is able to reduce the execution time of the job pool by 38% while at the same time increasing the system throughput when compared to the state-of-the-art scheduling heuristics.

With an increasing number of applications being ported to GPUs, accelerators (i.e., GPUs) should be utilized as a shared compute-resources possibly executing multiple small compute kernels to improve device utilization and device occupancy [116, 117]. However, the proposed OpenCL scheduler in Chapter 4 (i.e., *Troodon*) lacks such an ability. To address this issue (which the Troodon lacks), a novel machine learning based scheduling technique named as *FusionCL* is presented in this chapter. FusionCL fuses kernels of two different OpenCL applications to increase GPU occupancy during job pool execution. Similar to Troodon, the proposed scheduler first predicts the device suitability for jobs in the job pool and the potential speed-up the job may attain on a certain device. Moreover, a machine learning based classifier has been developed that predicts whether merging two specific kernels or executing them separately will produce better speed-up. For all the candidate kernels, the classifier chooses the best candidate kernel that may result in greater speed-up. The experimental results show that the proposed scheduling heuristic reduces execution time and increase throughput by 8% and 9%, respectively when compared to the state-of-the-art scheduling mechanisms.

In particular, the following contributions are made in this work:

1. A machine learning based kernel merging classifier that predicts the fusion suitability of OpenCL kernels.

2. Selection of best pair of kernels for fusion among all the candidate kernels

3. Development of speedup predictor that predicts the relative speed up that can be attained due to fused execution.

4. Experimental evaluation of the FusionCL scheduling heuristics with the results showing that the FusionCL outperforms all other scheduling heuristics in terms of employed performance parameters.

The rest of the chapter is organized as follows. Section 5.2 presents a motivation case-study to highlight the significance of this research. The detailed methodology is presented in Section 5.3. The experimental setup used to verify the methodology, results and discussions are presented in Section 5.4 whereas this chapter concludes with Section 5.5.

5.2 A Case-Study For Fusing Or Executing Kernels Separately

Some kernels achieve speed-up when they are fused while some of the OpenCL kernels may perform inadequately after fusion. The poor performance of the fused kernels can be attributed to the contention for the shared resources i.e., streaming cores, GPU memory bandwidth, etc. This fact has been highlighted in Figure 5.1 which presents the achieved relative speed-up of the fused kernels (over their sequential executions, represented as baseline). The experimented applications (in Figure 5.1) are taken from the *Polybench* benchmark suite. The X-axis depicts the kernels that were fused together whereas the Y-axis presents the relative speed-up of the fused kernel execution. The results show that some of the kernels attain speed-up due to kernel fusion whereas some suffer significant slow-down. This motivates us to device a machine learning model capable to predict potential speedup (or slow-downs) for the fused kernels. Moreover, this experiment reveals that the fusion decision is also affected by the input data size of the two kernels. For some data sizes, the fused kernels can attain speed-up while for other data sizes the performance can suffer causing slow-downs.

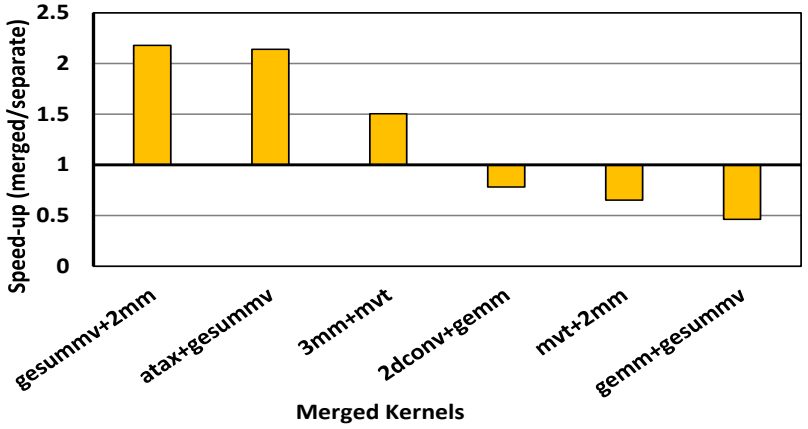

FIGURE 5.1: Speed-up achieved due to fused kernel execution over sequential kernel executions

Figure 5.2 presents the speed up performance of the two kernels *2dconv* and *atax* employing different data-sizes. The Y-axis value greater than one shows the attained speed-up whereas the value less than one shows slow-down. Figure 5-2 shows that speed-up of 2.17X is attained for a data size of 135360 whereas, for data size 1053696, a slow-down of 6X is deteriorated. This highlights the fact that a scheduler should be able to decide, on the basis of data size, when to fuse two kernels and when to execute them separately.

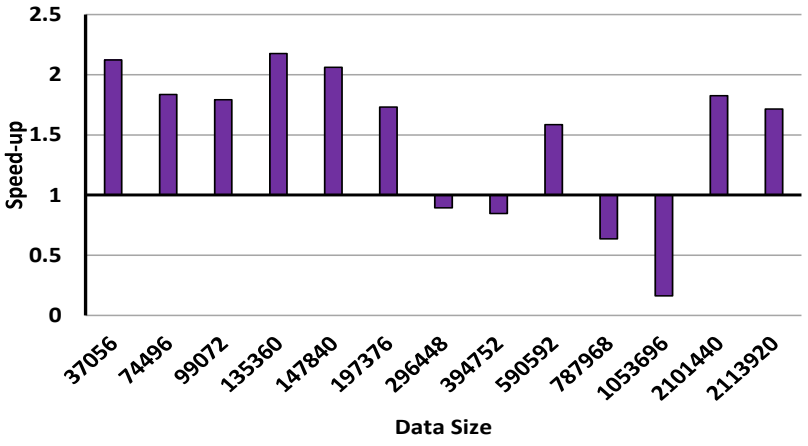

FIGURE 5.2: Speed-up achieved for multiple data sizes when 2dconv and atax kernels are fused

5.3 FusionCL

FusionCL scheduler that is built on top of the Troodon scheduler is presented in this section. The FusionCL decides whether to fuse kernels of a pair of OpenCL applications or execute them separately to increase the occupancy of the GPU device. The overall goal is to reduce the execution time of the job pool through load balancing and increasing GPU occupancy. The FusionCL scheduler uses Troodon [65] scheduling scheme to determine device suitability of a job and relative speedup that is attainable when the job is executed on the suitable device. The GPU suitable jobs are then passed to the *fusion classifier* to decide whether fusing two GPU suitable jobs (to increase device occupancy) or executing them separately will produce speedup. Both the CPU and GPU suitable jobs are then combined in the job pool. Subsequently, the jobs are mapped to heterogeneous devices (i.e., CPU and GPUs) on the basis of jobs' computing requirement and devices' processing speed. The E-OSched scheduling heuristics is used to ensure the load balanced execution of jobs on the heterogeneous devices. The load-balanced execution of jobs results in increasing throughput of the system.

5.3.1 System Architecture

The FusionCL scheduler's working is presented in Figure 5.3. Multiple users submit their job to the system. The *Feature Extractor* tool extracts code features (labeled as $F_1, F_2, \ldots, F_n$ in Figure 5.3) from each of the submitted user jobs. The extracted code features along with the data size of the user job are submitted to the *Device Suitability Classifier* which decides on the job's suitability to either CPU or GPU. Moreover, the *Device Suitability Classifier* also determines the speedup that is obtained by the user job when it is executed on the suitable device (as compared to the non-suitable device). The job is then inserted into either *CPU Suitable* or *GPU Suitable* Jobs queues according to the predicted device suitability. At the heart of the FusionCL system are the *Fusion Classifier* and the *Fusion Speedup Predictor*. The *Fusion Classifier* determines whether the

GPU suitable jobs will benefit (in terms of execution time) from fusing kernels of a pair of jobs or will be better off when executed separately. The GPU suitable jobs that are better suited to be executed separately are inserted into *Separate Kernel GPU* jobs whereas the jobs that are candidates for fusion are passed to the *Kernel Fusion Tool*. After kernels fusion, the fused kernel jobs are inserted into the *Fused Kernel GPU* jobs. Jobs in the *Separate Kernel GPU* jobs list and *Fused Kernel GPU* jobs list are sorted according to the predicted speedup in their respective job lists (smallest speedup first). The two job lists are concatenated together in the *GPU Job Queue*. Thereafter, jobs from the *CPU Suitable Jobs* are inserted at one end of the *Job Pool* whereas jobs from the *GPU Job Queue* are inserted at the other end of *Job Pool*. The *E-OSched* [81] scheduling mechanism is then employed to map jobs from the Job Pool to either CPU or GPU device in a load balanced manner.

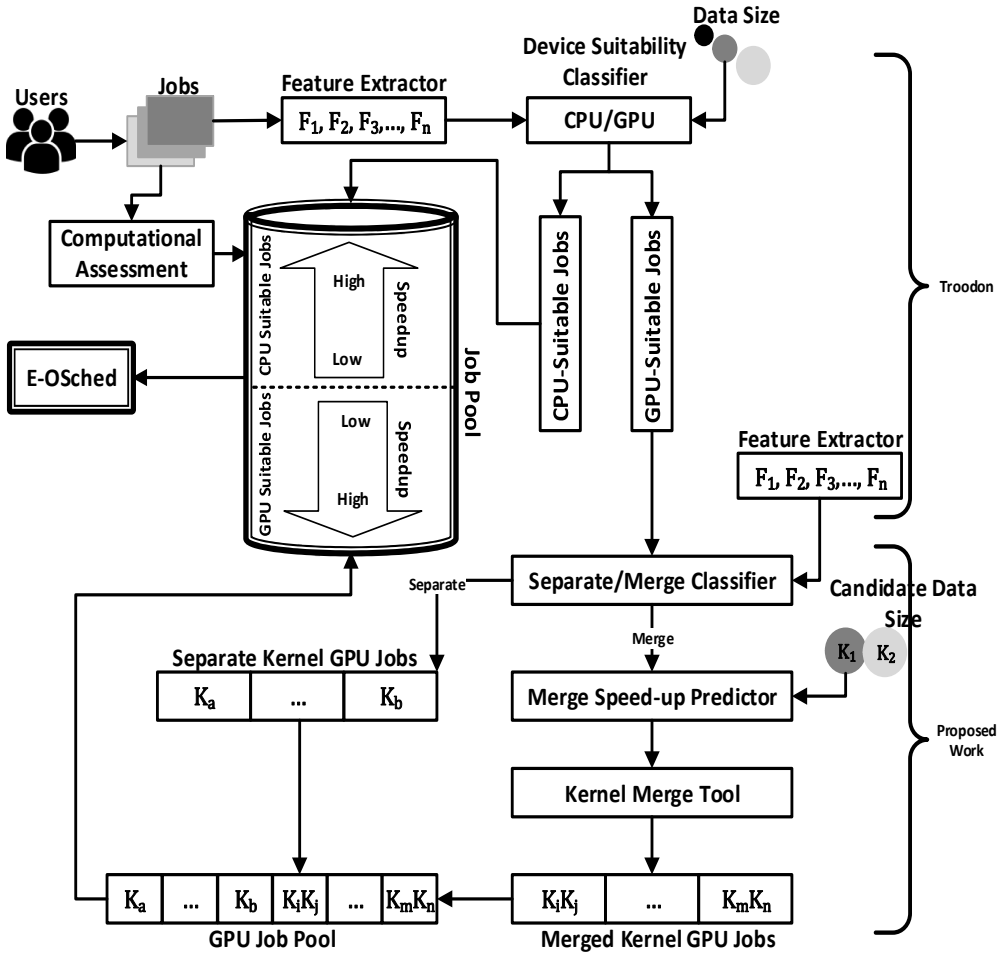

FIGURE 5.3: System Architecture of the proposed kernel fusion based scheduling scheme

The flowchart of the FusionCL scheduler is presented in Figure 5.4. First of all, the computation requirement of each user job is estimated and its code features are extracted. Next step is to classify jobs according to their device suitability. CPU suitable jobs are inserted into CPU suitable list and GPU suitable jobs are inserted into GPU suitable list. The extracted code features and data size of each job from CPU job list are used to determine the expected speedup that each job will obtain when executed on CPU in comparison to job's execution on the GPU. Subsequently, all the jobs in CPU job list are sorted in descending order of expected speedup (highest speedup first). The same process is also repeated for all the jobs in the GPU job list.

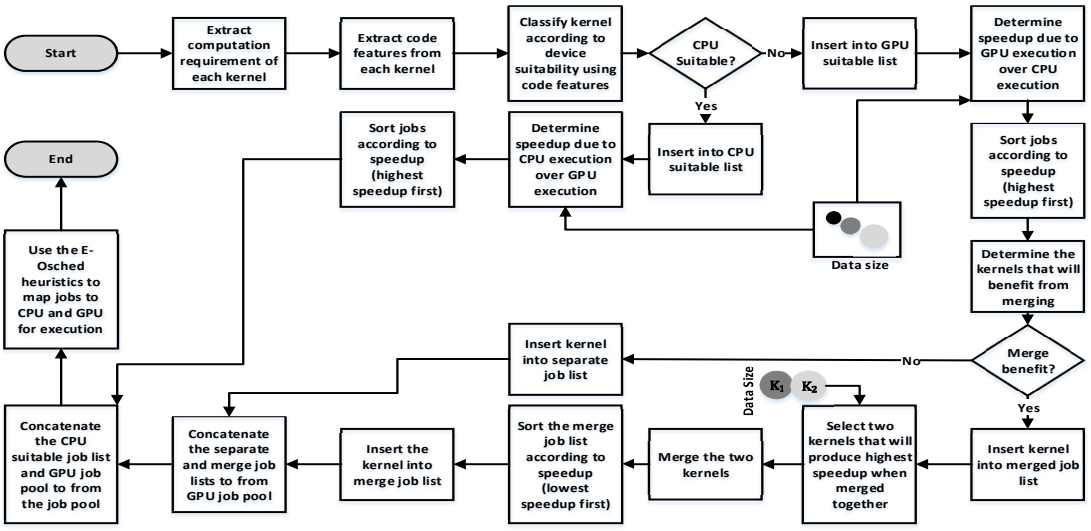

FIGURE 5.4: Flowchart of the FusionCL scheduling scheme

In addition to the steps mentioned above, some additional steps are performed on the jobs in the GPU job list. Extracted code features are used to determine the kernels that will produce speedup when fused with another kernel. The kernels that would benefit from fused execution are inserted into the fused job list while kernels that would not benefit from fused execution are inserted into separate job list. Using extracted code features and input data size, speed up for all kernel pairs

is determined. Thereafter, the kernel pair providing highest speedup are fused and the process is repeated until all the candidate kernels are fused. Afterward, the fused kernel pairs are sorted in ascending order of speed up (lowest speedup first) and concatenated with the separate job list to produce GPU job pool. Finally, CPU suitable list and GPU job pool are concatenated together to for job pool. The E-OSched scheduling mechanism is then used to map all the jobs in the job pool to a heterogeneous CPU-GPU system in a load balanced manner.

5.3.2 FusionCL System Model

This section details the system model for the FusionCL scheduling scheme. Troodon system model [65] was adapted to represent the scenario where machine learning based kernel fusion classifier has been employed to identify each pair of GPU suitable kernels that produce the highest speedup (over FIFO based kernels execution) when fused together.

Consider $\mathbb{C} = \{c_1, c_2, \ldots, c_m\}$ be a set of m submitted jobs that are required to be scheduled on a heterogeneous system (consisting of a CPU and a GPU). In order to schedule a job, it is required to extract code features (23 employed features, presented in Table 5.2) $F = \{F_0, F_1, \ldots, F_{22}\}$ from the kernel code of each job c_i in $\mathbb{C}$. The job c_i in $\mathbb{C}$ is provided as an input to the *Kernel Feature Extractor* (KFE) to extract code features and determine Count of each code feature (CF_i) (i.e., the number of times a code feature occur in the kernel code of a job). This relationship is represented by Eq. 5.1:

$$C_i \xrightarrow{KFE} \aleph \tag{5.1}$$

Where $\aleph_i = \{CF_0, CF_1, \ldots, CF_{22}\}$ represents the count of each code-feature in c_i. Next step is to determine the device suitability of c_i using *Machine Learning based Device Suitability* (ML_{DS}) model. For this purpose, $\aleph_i$ for c_i is provided as an input to the ML_{DS} which classify c_i as either suitable to CPU execution or

suitable to GPU execution. This relationship is represented in Eq. 5.2:

$$\aleph_i \xrightarrow{ML_{DS}} DS(c_i) \mid (DS(c_i) = CPU) \vee (DS(c_i) = GPU) \tag{5.2}$$

where $DS(c_i)$ represents *Device Suitability* of c_i. Next, the speedup prediction of c_i on the suitable device is obtained using *Machine Learning-based Speedup Predictor* (ML_{SP}). In this regard, $\aleph_i$, $DS(c_i)$ and N (where N represents input size of c_i) is provided as an input to ML_{SP}. The output of ML_{SP} is the *Speedup of* c_i on a suitable device (SPC_i) where the suitable device may be either a CPU or a GPU. This relationship is represented in Eq. 5.3:

$$\aleph_i \wedge DS(c_i) \wedge N \xrightarrow{ML_{SP}} SPC_i \tag{5.3}$$

Moreover, to balance the load across a CPU and a GPU, computation requirement of each job c_i (in terms of Flops) is estimated using *Computational Assessment* (CA) module. The job c_i is provided as an input to the CA and it estimates *Computational Requirement of* c_i (CRC_i). This relationship is represented by Eq. 5.4.

$$C_i \xrightarrow{CA} CRC_i \tag{5.4}$$

Once $DS(c_i)$, SPC_i, and CRC_i of all jobs in $\mathbb{C}$ is determined, all the jobs are gathered into two subsets of $\mathbb{C}$ i.e., A_{cpu} and A_{gpu}. A_{cpu} is a subset of m jobs from $\mathbb{C}$ displaying device suitability to CPU, such that $A_{cpu} = \{A_{cpu}(1), A_{cpu}(2), \ldots, A_{cpu}(m) \mid SPA_{cpu}(i) \leq SPA_{cpu}(i+1)\}$ in sorted order with respect to predicted speedup (higher speedup first) of i^{th} job ($SPA_{cpu}(i)$). Similarly, $A_{gpu} = \{A_{gpu}(1), A_{gpu}(2), \ldots, A_{gpu}(n) \mid SPA_{gpu}(j) \leq SPA_{gpu}(j+1)\}$ be a subset of n jobs from $\mathbb{C}$ displaying device suitability to the GPU, in sorted order with respect to speed up (lower speed up first) of the j^{th} job ($SPA_{gpu}(j)$).

Next step is to determine the fusion suitability of the jobs in A_{gpu}. For this purpose $\aleph_i$ and N of all the jobs in A_{gpu} are passed to the *Machine Learning based Fusion Suitability* (ML_{FS}) classifier that classifies the jobs in A_{gpu} as either *Fusion*

suitable $FS(A_{gpu})$ or *Fusion unsuitable*. This classification is presented in Eq. 5.5:

$$\forall_i \in A_{gpu}(\aleph \wedge N) \xrightarrow{ML_{FS}} (A_{gpu})(FS(A_{gpu}) = TRUE \vee FS(A_{gpu}) = FALSE) \quad (5.5)$$

Thenceforth, all the jobs with fusion suitability equal to FALSE are accumulated in the *Fusion Unsuitable Job List* (A) while all the jobs with fusion suitability equal to TRUE are accumulated in the *Fusion Suitable Job List* (B) as presented in Eq. 5.6 and Eq. 5.7:

$$A = \forall_i \in A_{gpu} \wedge MS(A_{gpu}) = FALSE \quad (5.6)$$

$$B = \forall_i \in A_{gpu} \wedge MS(A_{gpu}) = TRUE \quad (5.7)$$

All the jobs in A are sorted according to speedup (lowest speedup job first) whereas jobs in B are passed on to the *Machine Learning* based *Fusion Speedup Predictor* (ML_{FSP}) to predict the *speedup when each pair of jobs (x and y) in B are fused (SP(x×y))* for data sizes N_x and N_y respectively. This relationship is presented in Eq. 5.8:

$$C = \forall_{x,y} \in B, (x \wedge N_x) \wedge (y \wedge N_y) \xrightarrow{ML_{FSP}} SP(x \times y) \quad (5.8)$$

Where, $x = 1, 2, 3, \ldots, n$ and $y = 1, 2, 3, \ldots, n$. Now C contains a combination of all jobs from B with every other job from B along with their predicted speedup. All the elements (i.e., fused candidate jobs) in C are then sorted according to predicted speedup (highest speedup first).

Now, the first entry in C consists of two jobs (i.e., job j and job k) that produce the highest speedup when they are fused together. Therefore, the two jobs are fused together using *Kernel Fuser* and are removed from the sorted C. This relationship is presented by Eq. 5.9:

$$\forall_i \in C, C(i) \xrightarrow{KernelFuser} (\hat{C(i)}) \quad (5.9)$$

where, $i = 1, 2, 3, \ldots, n$ and *C(i)=SP(x×y)*. Moreover, all the entries in C that

contain either of the two jobs x or y are removed from the list. This process is repeatedly performed until all the jobs in C have been fused. All the fused jobs in C are sorted according to speedup (lowest speedup first). A and C are concatenated together to create Final GPU job pool (JP_{gpu}) as presented in Eq. 5.10:

$$JP_{gpu} = A \| C = \{A(1), A(2), ..., A(e) \| C(1), C(2), ..., C(f)\} \tag{5.10}$$

To adhere to the Troodon terminologies, all the members of set JP_{gpu} in Eq. 5.10 are renamed as:

$$JP_{gpu} = \{A_{gpu}(1), A_{gpu}(2), ..., A_{gpu}(m), A_{gpu}(m+1), A_{gpu}(m+2), ..., A_{gpu}(n)\} \tag{5.11}$$

where
$A_{gpu}(1) \Rightarrow A(1)$,
$A_{gpu}(m) \Rightarrow A(e)$
$A_{gpu}(m+1) \Rightarrow C(1)$
and
$C(f) \Rightarrow A_{gpu}(n)$

Finally, A_{cpu} and JP_{gpu} are also concatenated together to create a job pool J of $m+n$ jobs which is given by the following equation:

$$J = A_{cpu} \| JP_{gpu} = A_{cpu}(1), A_{cpu}(2), ..., A_{cpu}(m) \| A_{gpu}(1), A_{gpu}(2), ..., A_{gpu}(n) \tag{5.12}$$

from Eq. 5.12

$$J = \{A_{cpu}(1), A_{cpu}(2), ..., A_{cpu}(m), A_{gpu}(1), A_{gpu}(2), ..., A_{gpu}(n)\} \tag{5.13}$$

To adhere to *E-OSched* terminologies, all members of set J in Eq. 5.13 are renamed as:

$$J = \{J_1, J_2, ..., J_p, J_{p+1}, J_{p+2}, ..., J_q\} \tag{5.14}$$

where

$J_1 \Rightarrow A_{cpu}(1)$,

$J_p \Rightarrow A_{cpu}(m)$,

$J_{p+1} \Rightarrow A_{gpu}(1)$,

and

$J_q \Rightarrow A_{gpu}(n)$

In order to determine job assignment to the CPU and the GPU, total computation requirement (J_{CR}) of all the jobs in J and total processing speed of all processors (T_{CP}) are required, which are listed in following equations:

$$J_{CR} = \sum_{i=1}^{n} CRJ_i \tag{5.15}$$

$$J_{CR} = \sum_{i=1}^{m} PS_i \tag{5.16}$$

where there are m processors in the heterogeneous system (a multi-core CPU and a GPU). From Eq. 5.15 and Eq. 5.16, the computational share of CPU (CS_{cpu}), representing the portion of computation that is assigned to a CPU from J_{CR}, can be calculated by using the formula given in Eq. (5-17)

$$CS_{cpu} = \left(\frac{P_c}{T_{CP}} \times J_{CR}\right) - \alpha \tag{5.17}$$

where, P_C represents the *processing speed of a CPU* and α is the adjustment factor to balance the load across CPU and GPUs. The value of α is obtained empirically and is explained in section 3.4.2.1.

From Eq. 5.17, to determine jobs that are assigned to *CPU Job Queue (CJQ)*, let $J_{cpu} = J_1, J_2, \ldots, J_q$ be a subset of 1^{st} q jobs from job pool J. Here $Ceiling_{cpu}$ is required to represent the upper boundary for a number of jobs in J_{cpu}, which are assigned to CPU. The value of $Ceiling_{cpu}$ is calculated as the sum of CS_{cpu} and $1/3^{rd}$ of computation requirement of q^{th} job (CRJ_q):

$$Ceiling_{cpu} = CS_{cpu} + \frac{1}{3}CRJ_q \tag{5.18}$$

where, the ratio 1/3 in Eq. 5.18 ensures that q^{th} job should be mapped to CPU only if some part of its computation (i.e., 1/3) falls within the limit of CS_{cpu}. Mapping the q^{th} job to CPU (whose $1/3^{rd}$ part can be accommodated within the CPU share) will cause a minor load imbalance as compared to mapping the job to a GPU.

Using Eq. 5.18, the number of jobs that are assigned to CJQ is given by:

$$CJQ = \begin{cases} J_{cpu} & \sum_{i=1}^{q} CRJ_i \leq Ceiling_{cpu} \\ J_{cpu} - J_q & \sum_{i=1}^{q} CRJ_i > Ceiling_{cpu} \end{cases} \tag{5.19}$$

After job assignment to CJQ, another set J_{update}e is formed by subtracting CJQ from J:

$$J_{update} = J \setminus CJQ = job \mid\mid job \in J \ \wedge \ job \notin CJQ \tag{5.20}$$

After job assignment to CJQ is performed, all the remaining jobs in J_{update} are assigned to GJQ for execution on the GPU.

After the completion of jobs assignment to CJQ and GJQ, a family of sets $Q = CJQ, GJQ$ over J is obtained. The output set Q is governed by the constraints given in Eq. 5.21, Eq. 5.22, and Eq. 5.23.

$$\emptyset \notin Q \tag{5.21}$$

Eq. 5.21 ensures that CPU job-set and GPU job-set in Q cannot be an empty set $\emptyset$

$$\cup_{A \in Q} A = J \tag{5.22}$$

Eq. 5.22 specifies that union of all member set (of Q) is equal to job pool J.

$$(\forall A, B \in Q) A \neq B \Rightarrow A \cap B = \emptyset \tag{5.23}$$

Eq. 5.23 stipulates that intersection for all non-equivalent member sets (A and B) of Q would be an empty set $\emptyset$.

5.3.3 Kernel Fusion Mechanism

The kernel fusion process is depicted in Figure 5.5. The standard process as described by [63, 87] was used for kernel fusion. As shown in Figure 5.5, there are two kernels i.e., *Kernel 1* and *Kernel 2* that belong to two different applications i.e., *Application 1* and *Application 2* respectively. The fused kernel was developed by merging the arguments of *Kernel 1* and *Kernel 2*. Furthermore, the code of both kernals has been combined in the *Fused Kernel*. However, code of Kernel 1 and Kernel 2 are separated by a condition inside the Fused Kernel. If the condition is true *Kernel 1* code will be executed otherwise *Kernel 2* code's execution will be performed. In this way device utilization of the GPU is improved during the execution of data parallel applications with small input data size by sharing resources between kernels of two different applications.

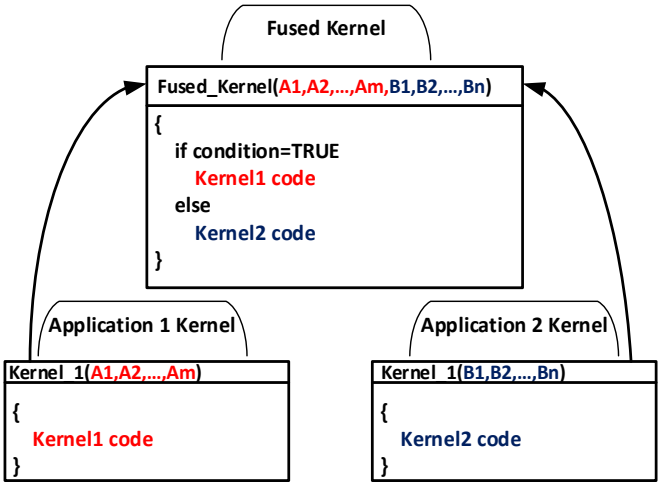

FIGURE 5.5: Kernel fusion process

5.3.4 Prediction Models

The classification model developed to classify jobs as fusion suitable or fusion unsuitable is an extreme gradient boosting [118] whereas the prediction model developed to predict the relative speedup of a pair of fused kernels over their serial execution is an extreme gradient boosting regressor. The training phase of prediction models is presented in Figure 5.6. For training, each pair of *fusion candidates*

jobs (i.e., the kernel codes of OpenCL programs) is fused and executed to obtain the fusion-based execution time. Meanwhile, the pair of Fusion Candidate jobs are also executed serially. On the basis of fusion and serial execution time, it is decided whether the pair of the fusion candidates are Fusion-suitable or Fusion-unsuitable jobs. The relative speedup of Fusion-suitable jobs is obtained using the formula presented in Eq. 5.24.

$$FusionSpeedup(A,B) = \frac{Execution\,Time\,of\,A\,and\,B\,after\,fusion}{Exection\,time\,of\,A\,+\,Exection\,time\,of\,B} \tag{5.24}$$

where A and B are a pair of kernels that are candidates for fusion.

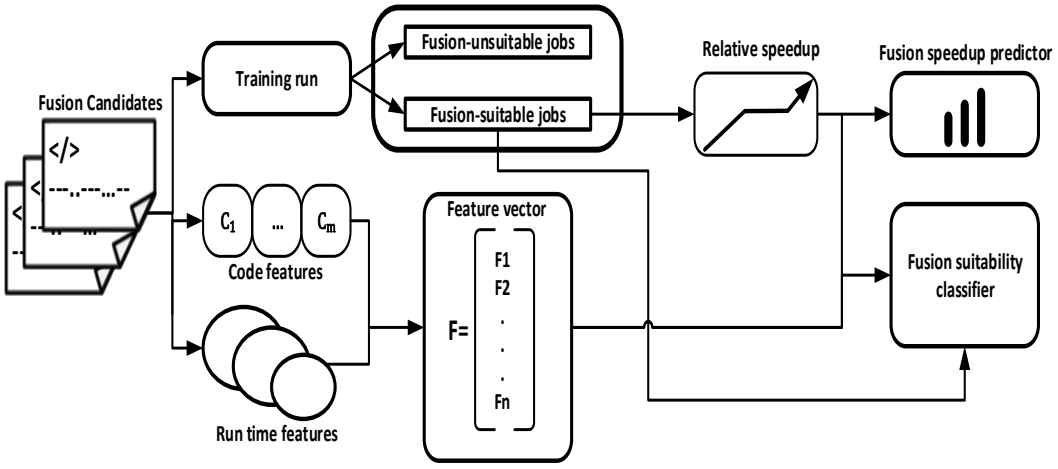

FIGURE 5.6: Training phase of Fusion speedup predictor and Fusion suitability classifier

Concurrent to the training runs of programs, their *Code features* and *Runtime features* are extracted. The feature vector along with the relative speedup information is used to develop the *Fusion speedup predictor*. Similarly, Feature vector together with the Fusion-suitable class/label is used to develop the *Fusion suitability classifier*.

5.3.4.1 Feature Extraction and Selection

We have used a similar feature extraction process as proposed by the Troodon [65]. LLVM compiler tool chain [106] and its front-end Clang are used to extract code

features from OpenCL kernel code. A list of all the extracted features is presented in Table 5.1.

TABLE 5.1: List of extracted features

No.	**Features Name**	**No.**	**Features Name**
1	Data Size	13	Total number of Addition (Float Datatype) instruction
2	Total number of Return statement	14	Total number of Addition (Integer Datatype) instruction
3	Total number of Control Statement	15	Total number of Subtraction (Float Datatype)
4	Total number of Allocation instruction	16	Total number of Subtraction (Integer Datatype)
5	Total number of Load Instructions	17	Total number of Function Call instruction
6	Total number of Store Instructions	18	Total number of Functions
7	Total number of Multiplication (Float Datatype) Operation	19	Total number of Blocks
8	Total number of Addition (Integer Datatype) Instruction	20	Total number of Instructions
9	Total number of Multiplication (Integer Datatype) Instruction	21	Total number of Float Operation
10	Total number of Division (Float Datatype) instruction	22	Total number of Integer Operation
11	Total number of Division (Integer Datatype) instruction	23	Total number of Loop Operation
12	Total number of Condition Check instruction		

From the list of extracted features, only those features were used that have high information gain. The motivation behind using a selected features (for predictive

models) is that it helps to reduced overfitting issues, improved accuracy, and decreased training time of the algorithm. The list of selected features in addition to their information gain is provided in Table 5.2.

TABLE 5.2: Top Ranked Selected Features

Rank	**No.**	**Feature Name**	**Information Gain**
1	1	Data Size	0.447328
2	9	Total no of Multiplication (Integer) Instruction	0.082088
3	19	Total no of Blocks	0.074392
4	17	Total no of Function Call instruction	0.067998
5	20	Total no of Instructions	0.0524
6	3	Total no of Control Statement	0.041888

5.4 Experiments and Results

We evaluated the FusionCL scheduling approach on a heterogeneous system that consists of an Intel Core i5-4460 CPU and an Nvidia GeForce GTX 760 GPU.

TABLE 5.3: Experimental setup

Device	**CPU**	**GPU**
Architecture	Haswell	Kepler
Base Clock	3.2 GHz	0.980 GHz
Boost Clock	3.4 GHz	1.033 GHz
Total Cores	4	1152 (CUDA cores)
Memory	8 GB	2 GB
Memory bandwidth	25.6 GB/s	192.2 GB/s
Performance (Single Precision)	409.6 GFLOPS	2257.9 GFLOPS
OpenCL SDK	Intel SDK for OpenCL 2016	CUDA 8.0

The specifications of the employed experimental setup are presented in Table 5.3. The operating system of the host machine for experimental setup is Ubuntu 16.04. All the programs that are used for experimentation were compiled using gcc 5.4.0. The experiments are conducted using 15 benchmarks from mainstream benchmark suits and other scientific applications. All the applications are executed with several input sizes to create a job pool of 236 program instances. Table 5.4 presents the benchmark programs along with their benchmark suits.

TABLE 5.4: Benchmark applications and the employed input data sizes

Benchmark Suits	**Applications**	**Input Data Size**	**Total Versions**
AMD	Matrix Multiplication	1,769,472—12,582,912	3
	Binomial Options	32,768—294,912	9
	Bitonic Sort	32,768—268,435,456	13
	Fast Walsh Transform	8,192—221,184	17
	Matrix Transpose	131,072—536,870,912	6
	Discrete Cosine Transformation	2,097,152—1,887,436,800	17
	Floyd Warshall	524,288—25,690,112	6
Polybench	3MM (3 Matrix Multiplications)	7,000,000—17,920,000	3
	GEMM (Matrix-multiply C=alpha.A.B+beta.C)	3,000,000—27,000,000	5
	GESUMMV (Scalar, Vector and Matrix Multiplication)	8,012,000—1,800,180,000	17

Benchmark Suits	**Applications**	**Input Data Size**	**Total Versions**
	MVT (Matrix Vector Product and Transpose)	4,016,000—900,240,000	17
	ATAX (Matrix Transpose and Vector Multiplication)	4,012,000—900,180,000	17
	2MM (2 Matrix Multiplications)	5,000,000—45,000,000	5
	2DCONV (2D Convolution kernel)	2000000—1,568,000,000	17
	3DCONV (3D Convolution kernel)	1,000,000—1,728,000,000	17
Own Developed	Matrix-Vector Multiplication	4,202,496—1,514,299,392	16

5.4.1 Scheduling Policies And Evaluation Metrics

This section details the scheduling schemes that were used for comparison with the FusionCL scheduler.

1. ***CPU-Only:*** As the name suggests, all incoming jobs are mapped to a CPU for execution. This scheduling scheme is used to provide a baseline which must be improved by a scheduling scheme to be considered for evaluation [112].

2. ***GPU-Only:*** Similarly to CPU-Only scheduling heuristics, all jobs are mapped to a GPU for execution. Use of this scheduling heuristics presents the general tendency of the programmers to map all their jobs to the faster device (i.e., GPU) resulting in severe load imbalance across a CPU-GPU heterogeneous system.

3. ***Alternate Assignment (AA):*** Jobs are assigned alternately to a CPU and a GPU [64] in the order in which the jobs arrive. The AA scheduling policy is used to show that naive assignment of jobs to a CPU and a GPU results in under-utilization of heterogeneous devices.

4. ***First Come First Serve (FCFS):*** In this scheduling scheme, jobs are assigned to the device in the order of their arrival. When one device finish execution of its job, the next non-assigned job in the arrival queue is assigned to that device.

5. ***E-OSched***: This scheduling algorithm [81] first sorts job pool in ascending order (smaller sized jobs first) considering the jobs' computational requirements. The jobs from one end of the job-pool (smaller size jobs having lower computational requirement) are assigned to a CPU device whereas the jobs from the other end of the job pool (jobs having higher computation requirements) are assigned to a GPU device. Jobs assignment to a CPU and GPUs are carried out in a resource-aware manner considering the proportion of the computing power provided by the devices.

For the performance evaluation, the following performance metrics were considered:

1. ***Execution time:*** execution time exhibits the time consumed in the computation of all jobs.

2. ***Throughput:*** represents the number of jobs completed per unit time.

3. ***Average time (of a job):*** is defined as an average amount of time taken by a job (of a job pool) to complete its execution.

4. ***Load balance:*** Load balance is a distribution of job pool among the computing devices (i.e., CPU and GPUs) such that all the processing devices accomplish the execution of the assigned jobs within the approximately same time duration. Load balance is calculated by using the following formula:

$$100 - \left(\left| \frac{Executiontimeof CPU assignedjobs - Executiontimeof GPU assignedjobs}{Executiontimeof theJobpool} \right| \times 100 \right) \quad (5.25)$$

The higher value of this metric shows a more load-balanced execution.

5.4.2 Scheduling Results

The execution time-based results of the FusionCL and alternative scheduling schemes are depicted in Figure 5.7. It can be observed from Figure 5.7 that the FusionCL scheduling scheme outperforms the baseline scheduling schemes (i.e., *GPU-Only and CPU-Only*) by consuming 73% and 183% less execution time, respectively. When the execution time of the FusionCL approach is compared with alternative scheduling schemes that use both CPU and GPU for the execution of jobs (i.e., *E-OSched*, *FCFS*, and *Alternate Assignment*) the reduction in execution time is 8%, 17%, and 41%, respectively.

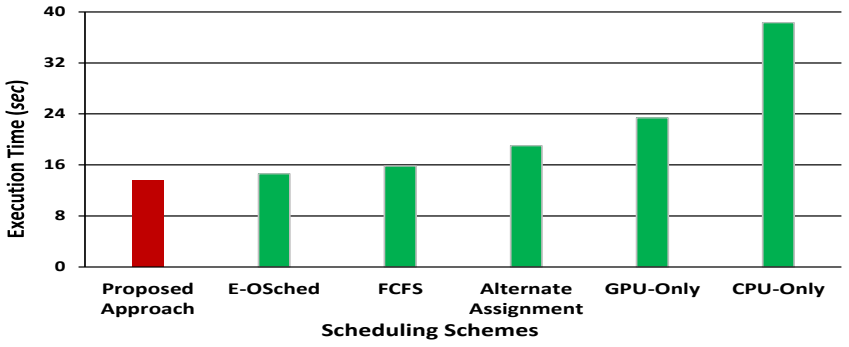

FIGURE 5.7: Execution time of scheduling schemes for job pool

5.4.3 Average Execution Time

The experimental results based on the average execution time of a job in the job pool are depicted in Figure 5.8. The FusionCL scheduling scheme consumes 1.73X and 2.83X less average execution time than the GPU-Only and CPU-Only scheduling heuristics, respectively (see Figure 5.8). As compared to the E-OSched,

FCFS, and AA scheduling heuristics, the FusionCL scheme reduces average execution time by 1.08X, 1.17X, and 1.41X, respectively (as illustrated in Figure 5.8).

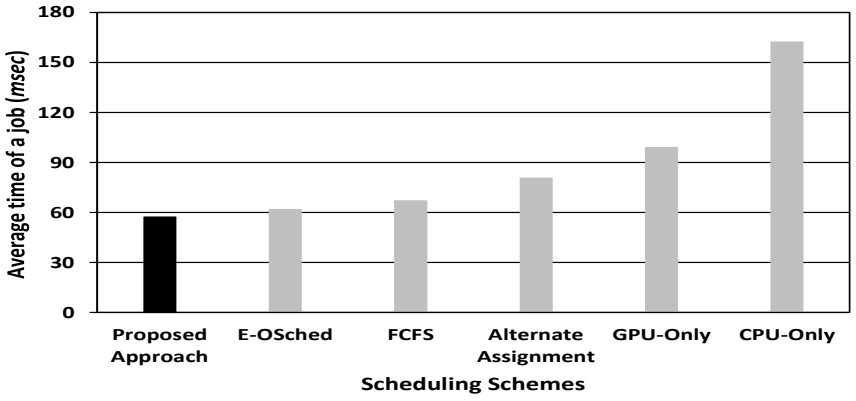

FIGURE 5.8: Average execution time of a job in the job pool

5.4.4 Throughput Analysis

The throughput results of the FusionCL approach and other scheduling heuristics are presented in Figure 5.9. The achieved throughput by *FCFS* was considered as a baseline of 1 for throughput analysis.

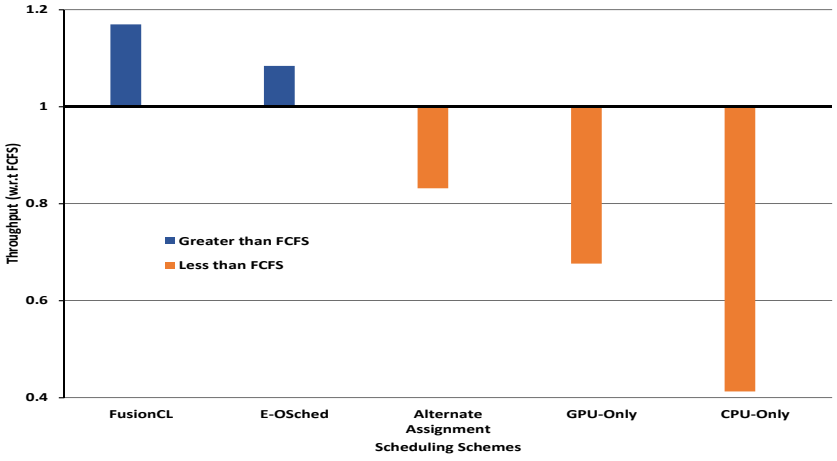

FIGURE 5.9: Throughput analysis

The bars above the baseline illustrates better results than the baseline whereas the bars below the baseline illustrates degraded results (as compared to the baseline).

The FusionCL scheduling heuristics and the *E-OSched* are the only scheduling heuristics that produce better results than the baseline. The FusionCL scheme and the *E-OSched* improve the throughput by 1.17X and 1.08X, respectively when compared to the baseline. The AA, GPU-Only, and CPU-Only only produce 0.83X, 0.68X and 0.41X of the throughput produced by the baseline.

5.4.5 Load Balance Analysis

Figure 5.10 illustrates the results of load balance analysis of the FusionCL scheme and other scheduling heuristics. The FusionCL scheduling heuristics achieves a load balance of 92.1% as can be noticed from Figure 5.10. The attained load balance of the FusionCL scheduler is 4% and 8% less than E-OSched and FCFS, respectively. However, the attained load balance of the FusionCL scheduler is 36% more than the load balance attained by AA scheduling mechanism.

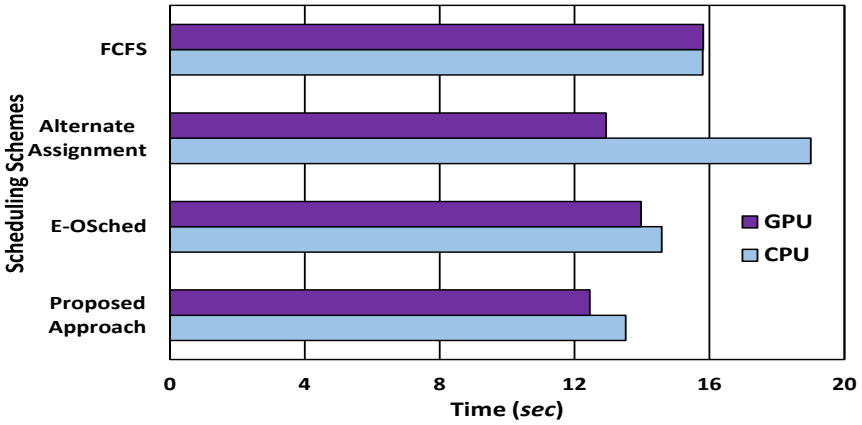

FIGURE 5.10: Achieved load balance by scheduling schemes

5.4.6 Results Discussion

Experimental results (presented in Figures 5.7—5.10) highlights the FusionCL approach's superior performance in comparison to all other scheduling schemes with

respect to the performance metric presented in Section 5.4.1. Accurate classification and fusion of fusion suitable kernels, to increase GPU occupancy, is the main reason for the improved performance by the FusionCL approach. Furthermore, identification of the best pair of kernels for fusion results in a speedup of fused pair of kernels of over their serial execution. This results in a decreased execution time of the job pool as is evident from the analysis performed in the subsequent section.

The state-of-the-art approach E-OSched [81] is oblivious to the device suitability of each job resulting in less favorable job mapping to heterogeneous devices. Moreover, the E-OSched executes GPU mapped jobs serially whereas the FusionCL approach fuse a pair of kernels to increase GPU occupancy. This results in a further reduction in the job pool's execution time and an increase in system throughput.

5.4.7 Impact of Proposed Kernel Fusion Approach on GPU Execution Time

The impact of kernel fusion on GPU execution time is visualized in Figure 5.11 (A). The X-axis presents the scheduling schemes whereas the Y-axis represents the execution time (in seconds) of GPU mapped jobs that have been identified as Fusion suitable. The FusionCL approach represents the scheduling scheme for this research. Each pair of kernels that have been identified by the machine learning-based predictor as fusion suitable and will produce the highest speedup among all the Fusion candidates are fused together. Afterward, the fused kernels are sorted according to speed up (highest speedup first) and are mapped to the GPU device. The FCFS scheme represents the kernels that are identified as fusion suitable but are executed serially in the order of their arrival. The results show that the FusionCL approach consumes 2.23× times reduced execution time when compared to the separate execution of the Fusion suitable kernels.

The impact of the FusionCL approach on selecting the Fusion suitable kernels and best fusion suitable kernel pair (i.e., the kernel pair that will produce the highest

speedup over their serial execution) is presented in Figure 5.11 (B). The X-axis presents the scheduling schemes whereas the Y-axis represents the execution time (in seconds) of all GPU mapped jobs. Random fusion scheduling scheme presents the scenario in which all the GPU assigned jobs are randomly fused. The results depicted in Figure 5.11 (B) indicate that the FusionCL approach outperforms the random fusion technique by consuming 2.23× reduced execution time.

The results in Figure 5.11 shows that kernel fusion is a promising technique for reducing execution time of a job pool by improving GPU occupancy. However, true identification of fusion suitable kernels is vital for achieving reduced execution time of GPU mapped jobs.

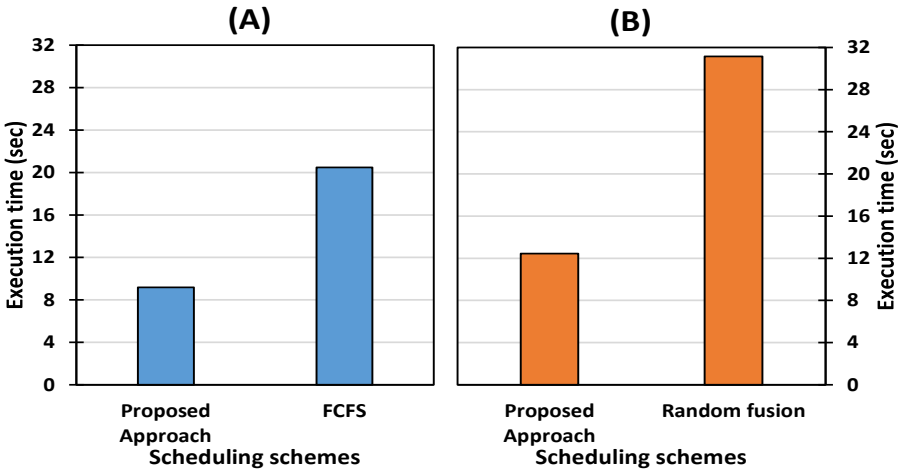

FIGURE 5.11: Execution time of fused and separate kernels in (A). The execution time of the fused kernel using the FusionCL scheme and random fusion in (B)

Moreover, it is also important to identify the best pair of kernels for fusion among fusion suitable kernels. The FusionCL approach is able to accurately classify kernels as fusion suitable and fusion unsuitable kernels while also identifying the best pair for fusion among fusion suitable kernels. This results in a considerable reduction in execution time while using the FusionCL approach.

5.4.8 Predictive Modeling Results

It is pertinent to describe the prediction accuracy of our machine learning models (i.e., kernel fusion classifier and fusion speedup predictor) in order to gain knowledge of the effect of the developed prediction models on the achieved experimental results. An accuracy of 0.98 was achieved for the kernel fusion classifier whereas fusion speedup predictor achieves an R square of 0.96. The accuracy measures of the prediction models are presented in Table 5.5. The use of TPOT [110] for the selection of machine learning algorithm for prediction was vital in achieving better results. Moreover, an extensive dataset of 2100 programs from multiple benchmarks, was prepared to train the machine learning model.

TABLE 5.5: Predictive modeling results

Model	**Precision**	**Recall**	**F-measure**	**R Square**	**Mean square**
Kernel fusion classifier	0.97	0.99	0.98	-	-
Fusion speedup predictor	-	-	-	0.966	0.011

Precision and Recall Impact on Scheduling Accuracy

Precision measures the relevant results (exactness of the prediction) rather than irrelevant results and recall measures the sensitivity of the most relevant results (completeness of the prediction). Precision and recall are convenient to be combined into a single metric called F1 score [119]. The F1 score is the harmonic mean of precision and recall. The harmonic means gives more weight to lower values as compared to the regular mean. The F1 measure is widely known metrics that estimate the entire prediction system performance by combining the precision and recall [120]. The scale of the F1 measure is 1.0: perfect prediction, 0.9: excellent prediction, 0.8: good prediction, 0.7: mediocre prediction, 0.6: poor prediction,

0.5: random prediction, and less than 0.5: poor prediction [120]. If the percentage value of precision and recall were lower for fusion suitability classifier, then applications that are not most suitable for fusion would have been fused. Such a fusion would have resulted in longer job pool execution time and lower system throughput. The fusion suitability classifier will only get a high F1 score if both precision and recall are high. As mentioned in the 5.4, the recall is 0.99 and precision is 0.97 that leads to F1 score is 0.98. The achieved F1 score suggests that the fusion suitability is able to predict best fusion candidate kernels with very high accuracy which leads to lower execution time and higher system throughput for the FusionCL scheduling heuristic.

5.4.9 Scheduling overhead

The scheduling overhead of the FusionCL scheduling heuristic is O(N^2) as it is built on top of the Troodon. However, the FusionCL also performs additional steps of fusion suitability classification, and fusion speedup prediction, and kernel fusion before using the Troodon for job scheduling. The training phase of the fusion suitability classifier and the speedup predictor took 10–12 hours. During deployment, the overhead of fusion suitability classification, and fusion speedup prediction, and kernel fusion, in our case, amounts to 35–50 milliseconds which is negligible when compared to the overall execution time of the job pool.

5.5 Conclusions

The increasing integration of GPUs in computing systems have made GPU a viable alternative to CPU for the execution of data-parallel applications. However, concurrent execution of applications (GPU sharing between kernels) is not supported by today's GPUs. This may result in wastage of GPU resources when an application with a small data size is executed on a GPU. To solve this problem, a kernel fusion mechanism is presented in this chapter that fuses a pair of kernels

to increase GPU occupancy and reduce GPU resource wastage. Furthermore, a machine learning based model has been developed that identifies a pair of kernel such that fusing them will produce maximum speed up over their serial execution. The experimental evaluation shows that the FusionCL kernel fusion mechanism reduces execution time by 2.83X when compared to a baseline scheduling scheme. When compared to state-of-the-art, the reduction in execution time is 1.08X. In the future, it is intended to develop a machine learning based mechanism that can also predict the percentage of GPU resources (i.e., compute-cores) that should be assigned to each fused pair of the kernel for efficient utilization of GPU resources.

Chapter 6

Conclusions

The contemporary era in the field of computing belongs to the heterogeneous architectures. Devices ranging from smart phones to super computers are using co-processors for the execution of user applications. GPU is the most widely used co-processor due to architectural advancements that enables it to accelerate the execution of data-parallel applications. Moreover, GPU also offer better performance-per-watt than the general purpose CPU. OpenCL has become an industry standard to program heterogeneous applications that can be executed on either a CPU or a GPU. Scheduling of OpenCL application, to either a CPU or a GPU in a load balanced manner is a challenging task which is the focus of this thesis.

This thesis tackles the problem of how to schedule a job pool of applications on a heterogeneous system in a load balanced manner. The E-OSched solves this problem by considering the computation power of jobs and processing power of heterogeneous processors in the scheduling decisions. However, it was observed that the execution time of the job pool can be further reduced if jobs are mapped to devices according to jobs' device suitability. The Troodon solves this problem by including job's device suitability in the scheduling decisions. The results obtained by the Troodon shows that a job/task scheduler for heterogeneous systems should include device suitability prediction in scheduling decisions. Furthermore, device utilization of the GPU can be increased if a GPU can be shared between two kernels

that cannot occupy all GPU resources when executed separately. GPU sharing among a pair of the kernel functions will not only increase the GPU occupancy but also reduce the job pool's execution time. The FusionCL scheduler uses a novel kernel fusion mechanism enabling a pair of the kernel to use a GPU concurrently. FusionCL shows that there is an urgent need for support of operating system services on the GPU.

6.1 Limitations

Following are some of the limitations of the work proposed in this thesis.

1. A limitation of the scheduling heuristics proposed in this thesis is that the computation requirement of all the jobs in the job pool should be known before jobs mapping to the heterogeneous devices is performed.
2. There shouldn't be any data dependence between kernels that belong to different programs.
3. In this research, it is assumed that all the jobs arrive at the same time.
4. Kernel fusion can only be performed on kernels that belongs to two different jobs.

6.2 Future Research Directions

This section discusses a list of possible research directions that can be used to further the work presented in this thesis.

1. **Job scheduling using execution time predictions for dynamic job arrival:** As stated previously, for research in this thesis, it is assumed that all the job arrives at the same time. However, the schedulers presented in the work can be modified to include dynamic job arrival time in scheduling

decisions. Machine learning can be used to predict the job completion time for all devices considering its current load. Thereafter, the newly arrived job can be assigned to the device that will execute the job in lest amount of time.

2. **Job priority and preemption:** Due to lack of operating system support on the GPU, jobs are executed on the GPU in the order in which they arrive (i.e.,FIFO). This can lead to unnecessary delay in the execution of prioritized jobs. Therefore, the proposed work of this thesis can be enhanced to include a mechanism that allows for assigning priorities to the incoming jobs.

3. **Using integrated GPU job execution:** In the scheduling heuristics, presented in Chapter 4 and Chapter 5, it was assumed that the heterogeneous system consist of only discreet GPU i.e., GPUs that are connected to the CPU through PCIe bus. However, nowadays all heterogeneous systems also consist of an integrated GPU (i.e., GPU exist with the CPU on the same die). The integrated GPU use portion of main memory as GPU memory and do not involve communication cost that is inherent in the use of discrete GPU. Applications that are better suited GPU execution but suffer from communication cost due to data transfer to/from the discrete GPU can be executed on the integrated GPU to further reduce the execution of the job pool.

4. **Transformation of OpenCL kernel code to make it better suitable for CPU execution:** OpenCL applications are functionally portable but are not performance portable i.e., an OpenCL application developed for GPU will execute seamlessly on a CPU (functionally portable) but will not produce similar results to an OpenMP implementation of the application. Therefore, CPU suitable OpenCL applications can be transformed for CPU execution so that it executes faster on CPU when compared to the default implementation. For this purpose, the use of vector instructions that are provided in OpenCL can be used to take advantage of SIMD lines present in a CPU core.

5. **Job data size impact on the performance of proposed scheduling heuristics:** The job pools used in this study consist of multiple types of job sizes (i.e., relative small, medium and large data sizes). Therefore, in the future it is intended to analyze the performance of proposed scheduling heuristics on job pools that consist of jobs of one type of data sizes e.g., only small data size jobs and also to design working principles about how job could be labeled as small, medium, and large data size job. Moreover, it is also planned to perform the scalability study on the performance of proposed scheduling heuristics w.r.t job pool types.

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff." *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, Sep. 2006.

[2] J. Parkhurst, J. Darringer, and B. Grundmann, "From single core to multi-core: Preparing for a new exponential," in *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD '06. New York, NY, USA: ACM, 2006, pp. 67–72. [Online]. Available: http://doi.acm.org/10.1145/1233501.1233516

[3] D. R. Butenhof, *Programming with POSIX Threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[4] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, Jan 1998.

[5] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*, ser. O'Reilly Series. O'Reilly Media, 2007. [Online]. Available: https://books.google.ad/books?id=tR2cAgAAQBAJ

[6] "Top 500 super computer list," https://www.top500.org/lists/2018/11/, accessed: 2019-01-12.

[7] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "Gpus and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, Sep. 2011.

[8] "Opencl - the open standard for parallel programming of heterogeneous systems," https://www.khronos.org/opencl/, accessed: 2017-01-03.

[9] S. Rul, H. Vandierendonck, J. D'Haene, and K. De Bosschere, "An experimental study on performance portability of opencl kernels," in *Application Accelerators in High Performance Computing, 2010 Symposium, Papers*, 2010, p. 3. [Online]. Available: http://saahpc.ncsa.illinois.edu/papers/paper_2.pdf

[10] X. Yan, X. Shi, L. Wang, and H. Yang, "An opencl micro-benchmark suite for gpus and cpus," *The Journal of Supercomputing*, vol. 69, no. 2, pp. 693–713, Aug 2014. [Online]. Available: https://doi.org/10.1007/s11227-014-1112-2

[11] A. Munshi, "The opencl specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*, Aug 2009, pp. 1–314.

[12] E. Sun, D. Schaa, R. Bagley, N. Rubin, and D. Kaeli, "Enabling task-level scheduling on heterogeneous platforms," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, ser. GPGPU-5. New York, NY, USA: ACM, 2012, pp. 84–93. [Online]. Available: http://doi.acm.org/10.1145/2159430.2159440

[13] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: Survey of current and emerging trends," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: ACM, 2013, pp. 1:1–1:10. [Online]. Available: http://doi.acm.org/10.1145/2463209.2488734

[14] V. T. Ravi, W. Ma, D. Chiu, and G. Agrawal, "Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 137–146. [Online]. Available: http://doi.acm.org/10.1145/1810085.1810106

[15] H. Topcuoglu, S. Hariri, and Min-You Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, March 2002.

[16] I. Galindo, F. Almeida, and J. M. Badía-Contelles, "Dynamic load balancing on dedicated heterogeneous systems," in *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 64–74. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87475-1_14

[17] C. Gregg, M. Boyer, K. Hazelwood, and K. Skadron, "Dynamic heterogeneous scheduling decisions using historical runtime data," in *Workshop on Applications for Multi-and Many-Core Processors (A4MMC)*, 2011.

[18] A. Gamatié, X. An, Y. Zhang, A. Kang, and G. Sassatelli, "Empirical model-based performance prediction for application mapping on multicore architectures," *Journal of Systems Architecture*, vol. 98, pp. 1 – 16, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762118306465

[19] S. Mittal and J. S. Vetter, "A survey of cpu-gpu heterogeneous computing techniques," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 69:1–69:35, Jul. 2015. [Online]. Available: http://doi.acm.org/10.1145/2788396

[20] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang, "Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures," in *2012 41st International Conference on Parallel Processing*, Sep. 2012, pp. 48–57.

[21] S. Mittal, "A survey of architectural techniques for DRAM power management," *International Journal of High Performance Systems Architecture (IJHPSA)*, vol. 4, no. 2, pp. 110 – 119, 2012. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01104430

[22] B. Pérez, J. L. Bosque, and R. Beivide, "Simplifying programming and load balancing of data parallel applications on heterogeneous systems," in *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, ser. GPGPU '16. New York, NY, USA: ACM, 2016, pp. 42–51. [Online]. Available: http://doi.acm.org/10.1145/2884045.2884051

[23] M. P. Robson, R. Buch, and L. V. Kale, "Runtime coordinated heterogeneous tasks in charm++," in *2016 Second International Workshop on Extreme Scale Programming Models and Middlewar (ESPM2)*, Nov 2016, pp. 40–43.

[24] D. Gerzhoy, X. Sun, M. Zuzak, and D. Yeung, "Nested mimd-simd parallelization for heterogeneous microprocessors," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3368304

[25] Z. Wang, L. Zheng, Q. Chen, and M. Guo, "Cpu+gpu scheduling with asymptotic profiling," *Parallel Computing*, vol. 40, no. 2, pp. 107 – 115, 2014, special issue on programming models and applications for multicores and manycores. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819113001415

[26] M. Boyer, K. Skadron, S. Che, and N. Jayasena, "Load balancing in a changing world: Dealing with heterogeneity and performance variability," in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '13. New York, NY, USA: ACM, 2013, pp. 21:1–21:10. [Online]. Available: http://doi.acm.org/10.1145/2482767.2482794

[27] Y. Wen, Z. Wang, and M. F. P. O'Boyle, "Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms," in *2014 21st International Conference on High Performance Computing (HiPC)*, Dec 2014, pp. 1–10.

[28] D. Lustig and M. Martonosi, "Reducing gpu offload latency via fine-grained cpu-gpu synchronization," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2013, pp. 354–365.

[29] S. Mittal and J. S. Vetter, "A survey of methods for analyzing and improving gpu energy efficiency," *ACM Comput. Surv.*, vol. 47, no. 2, pp. 19:1–19:23, Aug. 2014. [Online]. Available: http://doi.acm.org/10.1145/2636342

[30] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 62:1–62:33, Mar. 2016. [Online]. Available: http://doi.acm.org/10.1145/2893356

[31] Y. Wen, "Multi-tasking scheduling for heterogeneous systems," *The University of Edinburgh*, 2017.

[32] V. K. Vaishnavi and W. Kuechler, *Design Science Research Methods and Patterns: Innovating Information and Communication Technology, 2nd Edition*, 2nd ed. USA: CRC Press, Inc., 2015.

[33] "International parallel and distributed processing symposium," http://www.ipdps.org/, accessed: 2020-01-05.

[34] "Super computing," http://supercomputing.org/, accessed: 2020-01-05.

[35] "International conference on supercomputing," https://www.ics-conference.org/, accessed: 2020-01-05.

[36] "International conference on parallel architectures and compilation techniques," https://dl.acm.org/conference/pact, accessed: 2020-01-05.

[37] "Symposium on principles and practice of parallel programming," https://dl.acm.org/conference/ppopp, accessed: 2020-01-05.

[38] "International european conference on parallel and distributed computing," https://2020.euro-par.org/, accessed: 2020-01-05.

[39] "General purpose processing using gpus," https://dl.acm.org/conference/gpgpu, accessed: 2020-01-05.

[40] "International conference parallel processing," https://dl.acm.org/conference/icpp, accessed: 2020-01-05.

[41] "International conference on parallel and distributed systems," https://ieeexplore.ieee.org/xpl/conhome/1000534/all-proceedings, accessed: 2020-01-05.

[42] "Transactions on architecture and code optimization," https://dl.acm.org/journal/taco, accessed: 2020-01-05.

[43] "Journal of parallel and distributed computing," https://www.journals.elsevier.com/journal-of-parallel-and-distributed-computing, accessed: 2020-01-05.

[44] "Transactions on parallel and distributed systems," https://www.computer.org/csdl/journal/td, accessed: 2020-01-05.

[45] "The journal of super computing," https://link.springer.com/journal/11227, accessed: 2020-01-05.

[46] "International journal of parallel programming," https://www.springer.com/journal/10766, accessed: 2020-01-05.

[47] "Parallel computing," https://www.journals.elsevier.com/parallel-computing, accessed: 2020-01-05.

[48] "Snapdragon 855 mobile platform — qualcomm," https://www.qualcomm.com/products/snapdragon-855-mobile-platform, accessed: 2019-06-27.

[49] "Qualcomm snapdragon 855 specs deep dive," https://www.androidauthority.com/qualcomm-snapdragon-855-912505/, accessed: 2019-06-29.

[50] "E2-7110 with radeontm r2 graphics — amd," https://www.amd.com/en/product/5951, accessed: 2019-06-27.

[51] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal,

and P. Dubey, "Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 451–460. [Online]. Available: http://doi.acm.org/10.1145/1815961.1816021

[52] B. A. Hechtman and D. J. Sorin, "Exploring memory consistency for massively-threaded throughput-oriented processors," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 201–212. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485940

[53] S. Chilingaryan, E. Ametova, A. Kopmann, and A. Mirone, "Balancing load of gpu subsystems to accelerate image reconstruction in parallel beam tomography," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Sep. 2018, pp. 158–166.

[54] M. Shen and G. Luo, "Corolla: Gpu-accelerated fpga routing based on subgraph dynamic expansion," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 105–114. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021732

[55] "Intel sdk for opencltm applications," https://software.intel.com/en-us/intel-opencl, accessed: 2017-10-31.

[56] "App sdk," http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/, accessed: 2017-01-05.

[57] "Cuda toolkit," https://developer.nvidia.com/cuda-toolkit, accessed: 2017-01-05.

[58] D. Grewe and M. F. P. O'Boyle, "A static task partitioning approach for heterogeneous systems using opencl," in *Compiler Construction*, J. Knoop, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 286–305.

[59] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "Skmd: Single kernel on multiple devices for transparent cpu-gpu collaboration," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 9:1–9:27, Aug. 2015. [Online]. Available: http://doi.acm.org/10.1145/2798725

[60] J. Lee, M. Samadi, and S. Mahlke, "Orchestrating multiple data-parallel kernels on multiple devices," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, Oct 2015, pp. 355–366.

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

[62] M. Becchi, S. Byna, S. Cadambi, and S. Chakradhar, "Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010, pp. 82–91. [Online]. Available: http://doi.acm.org/10.1145/1810479.1810498

[63] Y. Wen and M. F. O'Boyle, "Merge or separate?: Multi-job scheduling for opencl kernels on cpu/gpu platforms," in *Proceedings of the General Purpose GPUs*, ser. GPGPU-10. New York, NY, USA: ACM, 2017, pp. 22–31. [Online]. Available: http://doi.acm.org/10.1145/3038228.3038235

[64] H. J. Choi, D. O. Son, S. G. Kang, J. M. Kim, H.-H. Lee, and C. H. Kim, "An efficient scheduling scheme using estimated execution time for heterogeneous computing systems," *The Journal of Supercomputing*, vol. 65, no. 2, pp. 886–902, Aug 2013. [Online]. Available: https://doi.org/10.1007/s11227-013-0870-6

[65] Y. N. Khalid, M. Aleem, U. Ahmed, M. A. Islam, and M. A. Iqbal, "Troodon: A machine-learning based load-balancing application scheduler for cpu–gpu system," *Journal of Parallel and Distributed*

Computing, vol. 132, pp. 79 – 94, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731518306841

[66] Z. Chen and D. Marculescu, "Task scheduling for heterogeneous multicore systems," *ArXiv*, vol. abs/1712.03209, 2017.

[67] O. E. Albayrak, I. Akturk, and O. Ozturk, "Effective kernel mapping for opencl applications in heterogeneous platforms," in *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*, ser. ICPPW '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 81–88. [Online]. Available: http://dx.doi.org/10.1109/ICPPW.2012.14

[68] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, pp. 187–198, Feb. 2011. [Online]. Available: http://dx.doi.org/10.1002/cpe.1631

[69] M. E. Belviranli, L. N. Bhuyan, and R. Gupta, "A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 57:1–57:20, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400716

[70] A. P. D. Binotto, C. E. Pereira, A. Kuijper, A. Stork, and D. W. Fellner, "An effective dynamic scheduling runtime and tuning system for heterogeneous multi and many-core desktop platforms," in *Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications*, ser. HPCC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 78–85. [Online]. Available: http://dx.doi.org/10.1109/HPCC.2011.20

[71] C. Gregg, J. Brantley, and K. Hazelwood, "Contention-aware scheduling of parallel code for heterogeneous systems," in *2nd USENIX workshop on hot topics in parallelism, HotPar, Berkeley, CA*, 2010.

[72] V. J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro, "Predictive runtime code scheduling for heterogeneous

architectures," in *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 19–33. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-92990-1_4

[73] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer, "An automatic input-sensitive approach for heterogeneous task partitioning," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 149–160. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465007

[74] "Insieme compiler project," http://www.insieme-compiler.org/, accessed: 2017-09-07.

[75] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems," in *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 245–256. [Online]. Available: http://dl.acm.org/citation.cfm?id=2523721.2523756

[76] P. Pandit and R. Govindarajan, "Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014, pp. 273:273–273:283. [Online]. Available: http://doi.acm.org/10.1145/2581122.2544163

[77] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali, "Adaptive heterogeneous scheduling for integrated gpus," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 151–162. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628088

[78] V. T. Ravi and G. Agrawal, "A dynamic scheduling framework for emerging heterogeneous systems," in *Proceedings of the 2011 18th International*

Conference on High Performance Computing, ser. HIPC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1–10. [Online]. Available: https://doi.org/10.1109/HiPC.2011.6152724

[79] Z. Wang, L. Zheng, Q. Chen, and M. Guo, "Cap: Co-scheduling based on asymptotic profiling in cpu+gpu hybrid systems," in *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM '13. New York, NY, USA: ACM, 2013, pp. 107–114. [Online]. Available: http://doi.acm.org/10.1145/2442992.2443004

[80] P. Huchant, M.-C. Counilh, and D. Barthou, "Automatic opencl task adaptation for heterogeneous architectures," in *Euro-Par 2016: Parallel Processing*, P.-F. Dutot and D. Trystram, Eds. Cham: Springer International Publishing, 2016, pp. 684–696.

[81] Y. N. Khalid, M. Aleem, R. Prodan, M. A. Iqbal, and M. A. Islam, "E-osched: a load balancing scheduler for heterogeneous multicores," *The Journal of Supercomputing*, vol. 74, no. 10, pp. 5399–5431, Oct 2018. [Online]. Available: https://doi.org/10.1007/s11227-018-2435-1

[82] A. M. Aji, A. J. Peña, P. Balaji, and W.-c. Feng, "Automatic command queue scheduling for task-parallel workloads in opencl," in *Proceedings of the 2015 IEEE International Conference on Cluster Computing*, ser. CLUSTER '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 42–51. [Online]. Available: https://doi.org/10.1109/CLUSTER.2015.15

[83] A. M. Aji, A. J. Peña, P. Balaji, and W. chun Feng, "Multicl: Enabling automatic scheduling for task-parallel workloads in opencl," *Parallel Computing*, vol. 58, pp. 37 – 55, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819116300357

[84] A. Ghose, S. Dey, P. Mitra, and M. Chaudhuri, "Divergence aware automated partitioning of opencl workloads," in *Proceedings of the 9th India Software Engineering Conference*, ser. ISEC '16.

New York, NY, USA: ACM, 2016, pp. 131–135. [Online]. Available: http://doi.acm.org/10.1145/2856636.2856639

[85] Y. Wen, M. F. O'Boyle, and C. Fensch, "Maxpair: Enhance opencl concurrent kernel execution by weighted maximum matching," in *Proceedings of the 11th Workshop on General Purpose GPUs*, ser. GPGPU-11. New York, NY, USA: ACM, 2018, pp. 40–49. [Online]. Available: http://doi.acm.org/10.1145/3180270.3180272

[86] J. Zhong and B. He, "Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1522–1532, Jun. 2014. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2013.257

[87] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving gpgpu concurrency with elastic kernels," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 407–418. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451160

[88] C. Margiolas and M. F. P. O'Boyle, "Portable and transparent software managed scheduling on accelerators for fair resource sharing," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16. New York, NY, USA: ACM, 2016, pp. 82–93. [Online]. Available: http://doi.acm.org/10.1145/2854038.2854040

[89] Q. Jiao, M. Lu, H. P. Huynh, and T. Mitra, "Improving gpgpu energy-efficiency through concurrent kernel execution and dvfs," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 1–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2738600.2738602

[90] M. E. Belviranli, F. Khorasani, L. N. Bhuyan, and R. Gupta, "Cumas: Data transfer aware multi-application scheduling for shared gpus," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: ACM, 2016, pp. 31:1–31:12. [Online]. Available: http://doi.acm.org/10.1145/2925426.2926271

[91] "Samsung galaxy s8+ - full phone specifications," http://www.gsmarena.com/samsung_galaxy_s8+-8523.php, accessed: 2017-07-10.

[92] D. Rohr, S. Kalcher, M. Bach, A. A. Alaqeeliy, H. M. Alzaidy, D. Eschweiler, V. Lindenstruth, S. B. Alkhereyfy, A. Alharthiy, A. Almubaraky, I. Alqwaizy, and R. B. Suliman, "An energy-efficient multi-gpu supercomputer," in *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*, Aug 2014, pp. 42–45.

[93] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[94] M. Aleem, R. Prodan, and T. Fahringer, "Scheduling javasymphony applications on many-core parallel computers," in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 167–179. [Online]. Available: http://dl.acm.org/citation.cfm?id=2033345.2033365

[95] A. Lösch, T. Beisel, T. Kenter, C. Plessl, and M. Platzner, "Performance-centric scheduling with task migration for a heterogeneous compute node in the data center," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, ser. DATE '16. San Jose, CA, USA: EDA Consortium, 2016, pp. 912–917. [Online]. Available: http://dl.acm.org/citation.cfm?id=2971808.2972018

[96] R. Dolbeau, "Theoretical peak flops per instruction set: A tutorial," *J. Supercomput.*, vol. 74, no. 3, pp. 1341–1377, Mar. 2018. [Online]. Available: https://doi.org/10.1007/s11227-017-2177-5

[97] "Impact: Parboil benchmarks, 2007," http://impact.crhc.illinois.edu/parboil/parboil.aspx, accessed: 2017-01-05.

[98] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 Innovative Parallel Computing (InPar)*, May 2012, pp. 1–10.

[99] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54. [Online]. Available: https://doi.org/10.1109/IISWC.2009.5306797

[100] A. Hussain, M. Aleem, A. Khan, M. A. Iqbal, and M. A. Islam, "Ralba: a computation-aware load balancing scheduler for cloud computing," *Cluster Computing*, vol. 21, no. 3, pp. 1667–1680, 2018.

[101] A. Hussain and M. Aleem, "Gocj: Google cloud jobs dataset for distributed and cloud computing infrastructures," *Data*, vol. 3, no. 4, p. 38, 2018.

[102] M. A. Dávila Guzmán, R. Nozal, R. Gran Tejero, M. Villarroya-Gaudó, D. Suárez Gracia, and J. L. Bosque, "Cooperative cpu, gpu, and fpga heterogeneous execution with enginecl," *The Journal of Supercomputing*, vol. 75, no. 3, pp. 1732–1746, Mar 2019. [Online]. Available: https://doi.org/10.1007/s11227-019-02768-y

[103] B. Pérez, E. Stafford, J. L. Bosque, and R. Beivide, "Energy efficiency of load balancing for data-parallel applications in heterogeneous systems," *J. Supercomput.*, vol. 73, no. 1, pp. 330–342, Jan. 2017. [Online]. Available: https://doi.org/10.1007/s11227-016-1864-y

[104] Z. Wang, D. Grewe, and M. F. P. O'boyle, "Automatic and portable mapping of data parallel programs to opencl for gpu-based heterogeneous systems," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 42:1–42:26, Dec. 2014. [Online]. Available: http://doi.acm.org/10.1145/2677036

[105] H. Dreuning, R. Heirman, and A. L. Varbanescu, "A beginner's guide to estimating and improving performance portability," in *High Performance Computing*, R. Yokota, M. Weiland, J. Shalf, and S. Alam, Eds. Cham: Springer International Publishing, 2018, pp. 724–742.

[106] "The llvm compiler infrastructure project," https://llvm.org/, accessed: 2018-03-02.

[107] J. Brownlee, "Naive bayes classifier from scratch in python," *Machine Learning Mastery*, vol. 31, 2018.

[108] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Nov. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1953048.2078195

[109] G. Lemaître, F. Nogueira, and C. K. Aridas, "Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning," *J. Mach. Learn. Res.*, vol. 18, no. 1, pp. 559–563, Jan. 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3122009.3122026

[110] R. S. Olson, N. Bartley, R. J. Urbanowicz, and J. H. Moore, "Evaluation of a tree-based pipeline optimization tool for automating data science," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, ser. GECCO '16. New York, NY, USA: ACM, 2016, pp. 485–492. [Online]. Available: http://doi.acm.org/10.1145/2908812.2908918

[111] S.-K. Shekofteh, H. Noori, M. Naghibzadeh, H. S. Yazdi, and H. Fröning, "Metric selection for gpu kernel classification," *ACM Trans. Archit. Code*

Optim., vol. 15, no. 4, pp. 68:1–68:27, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3295690

[112] V. T. Ravi, M. Becchi, W. Jiang, G. Agrawal, and S. Chakradhar, "Scheduling concurrent applications on a cluster of cpu-gpu nodes," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, ser. CCGRID '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 140–147. [Online]. Available: https://doi.org/10.1109/CCGrid.2012.78

[113] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[114] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 308–317. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155656

[115] W.-M. Hwu, C. Rodrigues, S. Ryoo, and J. Stratton, "Compute unified device architecture application suitability," *Computing in Science and Engg.*, vol. 11, no. 3, pp. 16–26, May 2009. [Online]. Available: https://doi.org/10.1109/MCSE.2009.48

[116] H. Wang, F. Luo, M. Ibrahim, O. Kayiran, and A. Jog, "Efficient and fair multi-programming in gpus via effective bandwidth management," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 247–258.

[117] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of gpu memory system for multi-application execution," in *Proceedings of the 2015 International Symposium on Memory Systems*, ser. MEMSYS '15.

New York, NY, USA: ACM, 2015, pp. 223–234. [Online]. Available: http://doi.acm.org/10.1145/2818950.2818979

[118] G. Ridgeway, "The state of boosting," *Computing Science and Statistics*, pp. 172–181, 1999.

[119] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems.* O'Reilly Media, 2019.

[120] F. A. Narudin, A. Feizollah, N. B. Anuar, and A. Gani, "Evaluation of machine learning classifiers for mobile malware detection," *Soft Computing*, vol. 20, no. 1, pp. 343–357, 2016.