FPM BASED PARTITIONING AND ASSIGNMENT ALGORITHM FOR DATA PARALLEL APPLICATIONS ON HETEROGENEOUS PLATFORMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MAHMOUD ALASMAR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

JULY 2022

Approval of the thesis:

FPM BASED PARTITIONING AND ASSIGNMENT ALGORITHM FOR DATA PARALLEL APPLICATIONS ON HETEROGENEOUS PLATFORMS

submitted by **MAHMOUD ALASMAR** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil KALIPÇILAR
Dean, Graduate School of **Natural and Applied Sciences** ____________

Prof. Dr. İlkay Ulusoy
Head of Department, **Electrical and Electronics Engineering** ____________

Prof. Dr. Gözde Akar
Supervisor, **Electrical and Electronics Engineering** ____________

Prof. Dr. Cüneyt F. Bazlamaçcı
Co-supervisor, **Computer Engineering** ____________

Examining Committee Members:

Prof. Dr. İlkay Ulusoy
Electrical and Electronics Engineering, METU ____________

Prof. Dr. Gözde B. Akar
Electrical and Electronics Engineering, METU ____________

Prof. Dr. Klaus Schmidt
Electrical and Electronics Engineering, METU ____________

Prof. Dr. Murat Manguoğlu
Computer Engineering, METU ____________

Prof. Dr. Cüneyt F. Bazlamaçcı
Computer Engineering, IZTECH ____________

Date: 01.07.2022

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Mahmoud Alasmar

Signature :

ABSTRACT

FPM BASED PARTITIONING AND ASSIGNMENT ALGORITHM FOR DATA PARALLEL APPLICATIONS ON HETEROGENEOUS PLATFORMS

Alasmar, Mahmoud
M.S., Department of Electrical and Electronics Engineering
Supervisor: Prof. Dr. Gözde Akar
Co-Supervisor: Prof. Dr. Cüneyt F. Bazlamaçcı

July 2022, 76 pages

Advances in modern computing devices and applications created the challenge of efficient utilization of resources in satisfying the requirements of running applications. The present work aims to find an efficient workload distribution algorithm for data parallel applications of type single program multiple data (SPMD) running on a heterogeneous computing platform. We first consider a discrete functional performance model (FPM) that integrates processing speed and capacity of processing elements with the size of the computational task. We then develop a mathematical model and propose an appropriate heuristic mapping algorithm for distributing a given total workload of size N on p processing elements such that the total computation time is minimized and resources are utilized efficiently. Results of our evaluation study show that the proposed method can speed up parallel applications significantly in comparison to classical approaches. The proposed method is able to generate better solutions than classical methods in a reasonable amount of time by using a limited amount of prior information.

Keywords: Heterogeneous Platforms, High Performance Computing, Workload Distribution, Task assignment, Functional Performance Model, Parallel Computing, Single Program Multiple Data

ÖZ

HETEROJEN PLATFORMLARDA VERİ PARALEL UYGULAMALARIİÇİN FPM TABANLI BÖLÜMLEME VE ATAMA ALGORİTMASI

Alasmar, Mahmoud
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Gözde Akar
Ortak Tez Yöneticisi: Prof. Dr. Cüneyt F. Bazlamaçcı

Temmuz 2022 , 76 sayfa

Modern bilgi işlem cihazları ve uygulamalardaki gelişmeler, uygulama gereksinimlerini karşılamada kaynakların verimli kullanılması sorununu ortaya çıkardı. Mevcut çalışma, heterojen bir bilgi işlem platformunda çalışan tek programlı çoklu verili (SPMD) tip paralel uygulamalar için verimli bir iş yükü dağıtım algoritması bulmayı amaçlamaktadır. İlk olarak, işlem hızı ve işlem öğelerinin kapasitesini hesaplama görevinin boyutuyla bütünleştiren ayrık bir işlevsel performans modelini (FPM) ele alınmaktadır. Daha sonra bir matematiksel model geliştirilip, toplam hesaplama süresinin en aza indirilmesi ve kaynakların verimli bir şekilde kullanılması için N boyutundaki toplam iş yükünü p işleme öğelerine dağıtmak için uygun bir buluşsal eşleme algoritması önerilmektedir. Değerlendirme çalışmamızın sonuçları, önerilen yöntemin klasik yaklaşımlara kıyasla paralel uygulamaları önemli ölçüde hızlandırabileceğini göstermektedir. Önerilen yöntem, sınırlı miktarda ön bilgi kullanarak makul bir sürede mevcut yöntemlerden daha iyi çözümler üretebilmektedir.

Anahtar Kelimeler: Heterojen Platformlar, Yüksek Başarımlı Hesaplama, İşyükü dağıtımı, Görev Ataması, İşlevsel Başarım Modeli, Paralel Hesaplama, Tek Programlı Çoklu Verili

ACKNOWLEDGMENTS

I want to thank all those who made this work possible by their support. In particular, I want to express my appreciation to my Master degree advisor, Prof. Cüneyt F. Bazlamaççı, for the source of instructive guidance and support.

At last, I would like to thank my parents, my sisters, who always encourage me with their belief and patience.

TABLE OF CONTENTS

ABSTRACT v

ÖZ vii

ACKNOWLEDGMENTS x

TABLE OF CONTENTS xi

LIST OF TABLES xiv

LIST OF FIGURES xv

LIST OF ABBREVIATIONS xvii

CHAPTERS

1 INTRODUCTION 1

1.1 Problem Definition 3

1.2 Thesis Contribution 5

1.3 Structure of the Thesis 5

2 BACKGROUND 7

2.1 Performance Modeling 7

2.1.1 Constant Performance Model 8

2.1.2 Functional Performance Model 8

2.2 Workload Distribution Methods 10

2.2.1 Workload Balancing 10

2.2.2 Static Workload Distribution 11

2.2.3 Dynamic Workload Distribution 12

2.3 Shared Memory Programming Model 12

2.4 Inter Process Communication 13

3 LITERATURE REVIEW 15

3.1 A Novel Data-Partitioning Algorithm for Performance Optimization of Data-Parallel Applications on Heterogeneous HPC Platforms . . . 15

3.2 A Proposed Data Partitioning Approach on Heterogeneous HPC Platforms: Data Locality Perspective 17

3.3 Sigmoid: An auto-tuned load balancing algorithm for heterogeneous systems 18

3.4 Data Partitioning with a Functional Performance Model of Heterogeneous Processors 20

4 WORKLOAD DISTRIBUTION ALGORITHM ON HETEROGENEOUS DEVICES (WDAH) 23

4.1 Problem Formulation 23

4.2 Generation of workload packages 25

4.3 Partitioning Correctness 27

4.4 Heterogeneity Requirements 28

4.5 Tree Search Algorithm 29

5 IMPLEMENTATION AND EXPERIMENTAL EVALUATION RESULTS . 39

5.1 Implementation Setup Architecture 39

5.2 Experimentation 46

5.2.1 Optimality Test (Solution quality vs. Problem size) 47

5.2.2 Speed Up 49

5.2.3 Backtrack and Cost Optimality 52
5.2.4 Running Time 53
5.2.5 Utilization of Computing Elements 56
6 CONCLUSIONS AND FUTURE WORK 59
6.1 Conclusion 59
REFERENCES 63
APPENDICES 66
A OBJECT ORIENTED IMPLEMENTATION 67
A.1 Solution Node 67
A.2 Solution Entity 68
A.3 Processor 71
A.4 Dispatcher 72
CURRICULUM VITAE 75

LIST OF TABLES

TABLES

Table 4.1 Processing speed subject to workload package 34

Table 5.1 Computing elements Specifications 42

LIST OF FIGURES

FIGURES

Figure 1.1 Hardware Accelerators (Brant, 2018) 2

Figure 2.1 Advanced performance model for 2D Convolution on ARM A15 and DSP C66. 9

Figure 2.2 Balanced workload distribution summary 10

Figure 2.3 Server/Client Stub (Zhang, 2010) 13

Figure 3.1 Load balancing technique proposed by (Lastovetsky & Manumachu, 2007) 21

Figure 3.2 Initialization of lines U and L proposed by (Lastovetsky & Manumachu, 2007) 22

Figure 4.1 Advanced performance model for Matrix Multiplication on ARM A15 and DSP C66. 26

Figure 4.2 Instances of possible partitions for an input matrix to compute parallel matrix multiplication 28

Figure 4.3 Instance of a partition for an input matrix 28

Figure 4.4 A snapshot of the tree search process for the example problem . 34

Figure 4.5 Search process of case 1. 35

Figure 4.6 Search process of case 2. 36

Figure 4.7 Search process of case 3. 37

Figure 5.1 Implementation Setup Architecture 40

Figure 5.2 RPC Message Layout 41

Figure 5.3 66AK2H12 Block Diagram (Texas Instruments, 2017) 43

Figure 5.4 Lookup Table Structure 45

Figure 5.5 Cost of Solution (seconds) in logarithmic scale vs. problem size . 48

Figure 5.6 Matrix Multiplication Speed Up $(4ARMCores + xDSPCores)$ 50

Figure 5.7 FFT Speed Up $(4ARMCores + xDSPCores)$ 50

Figure 5.8 FIR Speed Up $(4ARMCores + xDSPCores)$ 51

Figure 5.9 Effect of Backtrack value on solution quality 53

Figure 5.10 Algorithm Running Time and Solutions Cost 55

Figure 5.11 Utilization of computing elements in WDAH method 56

Figure 5.12 Utilization of computing elements in HPOPTA Method. 57

LIST OF ABBREVIATIONS

API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
CFD	Computational Fluid Dynamics
CPM	Constant Performance Model
FFT	Fast Fourier Transformation
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FPM	Functional Performance Model
GPU	Graphics Processing Unit
HPC	High Performance Computing
HPOPTA	Heterogenoues Performance Optimization Algorithm
IPC	Inter Process Communication
MAC	Multiplication-Accumulation
MFLOPS	Millions of Floating-point Operations per Second
MIPS	Millions of Instructions per Second
QoS	Quality of Service
RPC	Remote Procedure Call
RT Linux	Real Time Linux
RTOS	Real Time Operating System
SPMD	Single Program Multiple Data
SoC	System on Chip
WDAH	Workload Distribution Algorithm on Heterogeneous Platforms

CHAPTER 1

INTRODUCTION

The adaptation of heterogeneity to modern devices has become a favorable option to be used over uni-processors or homogeneous multi-processors in the last decade especially in large scale computation applications (Cardoso et al., 2017). Increasing processor speed by increasing clock frequency and by using efficient compiler optimization techniques was a viable solution to increase the throughput of such applications, however, due to heat dissipation limitation and gate delays, the processor speeds ceased to increase. Hence, a shift towards using more complex platforms became an appealing strategy. Such platforms are supported with (1) more processing units for parallel execution, (2) heterogeneous components to support specialization and customized computation and (3) distributed structures to support large scale applications such as Big Data and High Performance Computing (HPC).

The heterogeneity on such devices arises from having computing elements of different speed, memory size, memory access time, communication time and flexibility (Cierniak et al., 1997). Recent devices are supported with System-on-Chip circuits (SoC), which mainly consist of a General Purpose Processor and one or more of the following computing elements (1) Graphics Processing Unit (GPU), (2) Field Programming Gate Array (FPGA) and (3) Application Specific Integrated Circuit (ASIC) known as accelerators (Gupta & De Micheli, 1993; Nickolls & Dally, 2010; Rose et al., 1993). Figure 1.1 summarizes the main differences and properties of the accelerators. On the leftmost side, there is the CPU, which is the most flexible component as wide range of applications can run on it, unlike ASICs on rightmost side which are highly dedicated and inflexible in the sense that they are not programmable. However, in terms of efficiency ASICs are the most efficient processing resources, because of having hard

wired algorithms provisioning low latency, power consumption and high throughput (Brant, 2018).

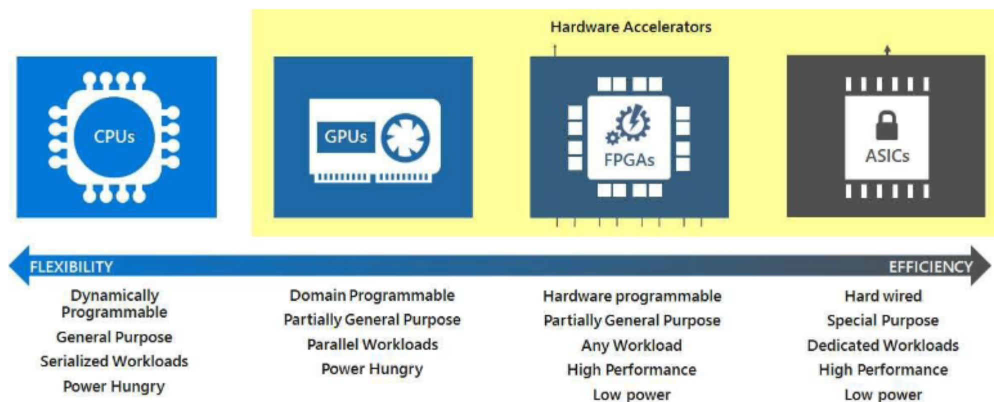


Figure 1.1: Hardware Accelerators (Brant, 2018)

As a special case of parallel applications, a parallel computing mode mainly consists of a number of processes performing the same type of computation but on different data sets utilizing multiple computing elements concurrently (Anthony, 2016). Computational Fluid Dynamics (CFD) used in weather forecasting, Convolution operations in Image Processing applications, Artificial Intelligence algorithms are all examples where parallel computing can be deployed. Pthread, OpenMP, OpenCL, CUDA and Open MPI are common libraries used in parallel computing which provides APIs to isolate low level details of the hardware. The mapping of threads or kernels onto computing elements can either be static (computed during compile time) or dynamic (computed during run time) based on predefined policies chosen by the user or as default options.

Utilizing a heterogeneous system at its maximum performance has always been a concern in the field of parallel computing and finding a workload offloading scheme or kernel mapping strategy such that certain objectives are satisfied is a challenging task. Indeed, finding an optimum solution that minimizes/maximizes an objective function for such problems is proven to be NP-hard (Garey & Johnson, 1990). Hence, modern approaches sought to use optimization techniques in order to find an optimum or near optimum solution using suitably formulated mathematical models.

Various optimization methods can be applied for solving the workload distribution

problem, which can be classified as exact or approximate methods (Puchinger & Raidl, 2005). Examples of the exact methods are branch-and-bound, Lagrangian relaxation based methods, linear and integer programming based methods and dynamic programming. On the other hand, approximate methods can either be heuristics, such as Greedy and Local Search Algorithms, or meta-heuristics such as Simulated Annealing (Kirkpatrick et al., 1983), Tabu search (Glover & Laguna, 1998) and Genetic algorithm (Deb & Jain, 2014). The difference between heuristics and meta-heuristics is that heuristics has a deterministic behaviour while meta-heuristics adapt probabilistic behaviour in order not to get trapped to local minimum solutions. Exact methods are guaranteed to find optimum solutions unlike approximate ones, however, their running time increases dramatically as the size of the problem increases (Puchinger & Raidl, 2005). Hence, heuristics are usually used for large scale problems in order to trade optimality for running time. Recently, methods involving machine learning techniques have also been investigated (Ahmed et al., 2021) in solving the workload mapping problem.

This work aims to propose an efficient mapping algorithm for parallel applications that will run on a heterogeneous computing platform such that the total execution time of the application is minimized. The success of the algorithm is measured in terms of the speed-up, cost reduction, running time and utilization of computing elements.

1.1 Problem Definition

Workload distribution on heterogeneous platforms is one of the important issues that has been targeted in the research field of data parallelism and parallel computing. Efficient mapping of workload over accelerators and CPUs according to their capabilities is essential in order to attain maximum gain and better resource utilization. The aim of the present work is to find an optimum or near optimum workload assignment on heterogeneous devices such that the computation time of an application is minimized and to simultaneously use the heterogeneity of these elements effectively.

We divide the problem into three main sub-problems:

- Characterizing the capabilities of computing elements.

- Designing workload assignment algorithm.
- Constructing the parallel programming model.

Characterizing the capabilities of computing elements means to predict the performance of the application on available computing resources. In order to find a minimum solution for the mapping problem, an accurate model that represents the relation between kernel's workload and existing computing elements is needed. Different models can be adapted to exhibit such information, the simplest is to assign a positive constant number for each computing element as an indication for its processing capabilities, such a model is known as Constant Performance Model (CPM) (Kalinov & Lastovetsky, 2001), which is independent of the application's computations and only applicable for certain cases. Another approach that can be used to characterize the capability of a computing element is Functional Performance Model (FPM) (Khaleghzadeh et al., 2018). The latter is more accurate in showing the dependency between the workload and the speed of processing element. The process of building FPM is known as profiling.

Designing workload assignment algorithm can be split into two sub-tasks: first, form an accurate mathematical model that truly mirror the problem, second, find a solver for the problem. Creating a mathematical model is a challenging task in the sense that the model should cover all aspects of the problem including assumptions, objective or cost function, constraints and conditions. The next challenge is to find a suitably designed algorithm to solve the problem. The outcome of the designed algorithm should be a feasible solution that can be found in a reasonable amount of time.

In a parallel computing environment, having a **parallel programming model** is essential in understanding the factors and limits that compromise the running application. There exist two main models, namely: shared memory and message passing (Calciu et al., 2013). In shared memory model, data is shared among the processors in a global address space. In this model, consistency of data needs to be ensured all the time via certain techniques, such as cache coherence protocols, and access to critical sections of the shared address space has to be managed using synchronization primitives, such as locks. On the other hand, message passing model, also known

as distributed memory model, is based on communication between processors using send/receive communication commands. Unlike shared memory, message passing is free of data race while it can suffer from deadlock during communication. Message passing model is less complex in terms of hardware requirement, however it suffers from communication overhead.

1.2 Thesis Contribution

This dissertation proposes a recursive tree search algorithm for finding a solution to the workload partitioning problem on heterogeneous platforms using a created pool of workload types. The contributions of the thesis are as follows:

- Introduces the concept of using a Functional Performance Model (FPM) for the purpose of profiling applications on different processing elements.
- A method for generating a set of workload types based on the profiling step.
- A technique for assigning a priority value to each computing element, where this priority is an indication for its capability in terms of speed.
- A priority based recursive tree search algorithm that distributes the existing total workload among processing elements using the generated pool of workload types.

1.3 Structure of the Thesis

The rest of this dissertation is organized as follows: Chapter 2 and 3 discuss the background and literature review. Chapter 4 presents our proposed methodology and its implementation. Chapter 5 illustrates our framework used for evaluation and the results obtained in our experiments. Finally, Chapter 6 discusses the potential improvements and presents final concluding remarks.

CHAPTER 2

BACKGROUND

This chapter covers the background related to the work discussed in this thesis.

2.1 Performance Modeling

Performance modeling is used to benchmark the performance of a computing element under certain circumstances. The factors that affect the performance of a computing element can be listed as i) computation of the running kernel, ii) memory hierarchy, iii) clock, iv) external interrupts and v) topology of the network in which the computing element is connected to. The running kernel degrades the performance of the computing resource depending on i) the complexity of the computation ii) how often the running kernel references the memory and iii) the number of branches. The time required to access a certain level within the memory hierarchy in order to load or store data can bring the ideal capability of the computing element highly down. Moreover, if the frequency of the computing resource changes dynamically during run time or in case the computing resource is interrupted by external events then its performance will be negatively affected considerably. Finally, the topology which connects the computing element to other resources can highly influence its performance especially in distributed memory architectures or message passing models.

Aiming to estimate the execution time of a certain task or kernel and approximate the performance of a computing element theoretically is a complicated task due to the fact that it requires an analysis at the instruction level or due to randomness of some factors. Different statistical approaches have been studied to find an alternative for estimating the performance of a computing element. In the present section two differ-

ent models will be discussed: Constant Performance Model (Kalinov & Lastovetsky, 2001) and Functional Performance Model (Khaleghzadeh et al., 2018).

2.1.1 Constant Performance Model

This is the simplest approach for modeling a computing machine in which a positive constant is used to indicate the computing capability of the machine. This constant could be MIPS (Millions of Instructions per Second), MFLOPS (Millions of Floating-point Operations per Second), normalized processor speed (NPS) as used in (Cierniak et al., 1997) or average execution time of an application. In general, such an approach is independent of the problem size, hence, the model lacks accuracy in most cases and is applicable only in some limited application types.

2.1.2 Functional Performance Model

A more accurate and advanced model for identifying the capability of a computing resource is known as Functional Performance Model (FPM), in which the performance of the processor is measured as a continuous function of the problem size. The performance metric in such a model is defined as the number of computation operations performed by the processor per unit time (Lastovetsky & Manumachu, 2007). A requirement for FPM is that the computational operations should not change during the execution of an application, otherwise, the information provided can be misleading. Figure 2.1 shows an example of (FPM) for two different computing elements running a single kernel of parallel 2D Convolution application. X-axis defines the size of the workload assigned to the kernel as a partition of input data A and Y-axis shows the speed of the computing element as measured in terms of Multiplication Accumulation Operations (MAC) per second. It is worth noting here that FPM should be smooth enough to be useful in finding a good solution.

The following metric can be used to measure the performance of the computing resource as a function of problem size:

$$S_i(w) = \frac{w}{T_i(w)} \tag{2.1}$$

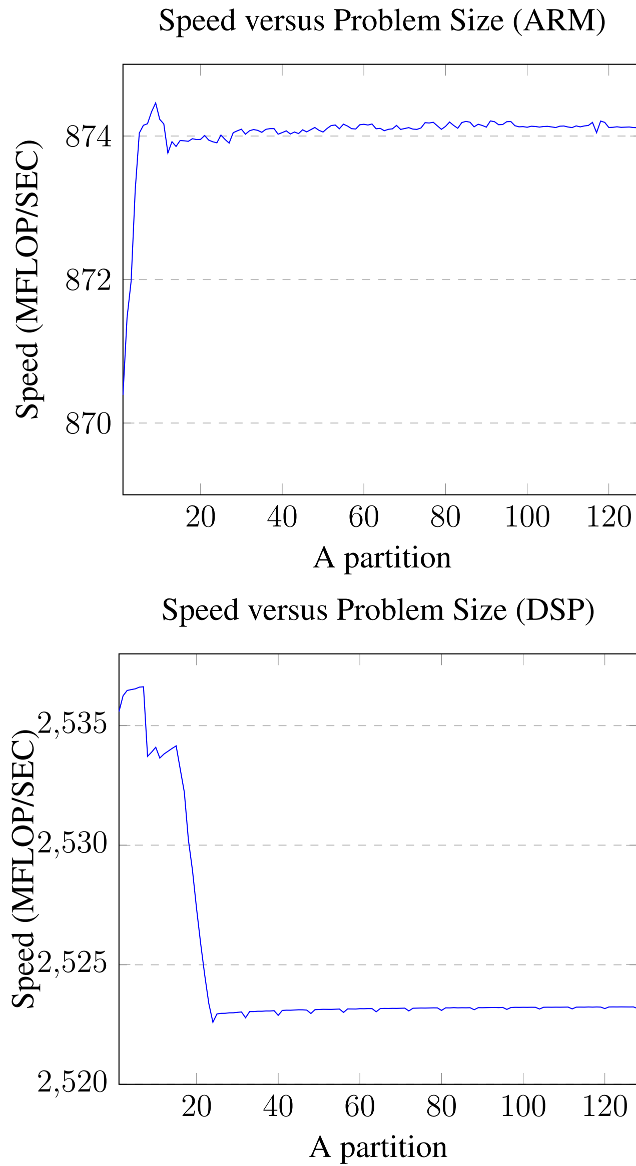


Figure 2.1: Advanced performance model for 2D Convolution on ARM A15 and DSP C66.

where $S_i(w)$ is the speed or computing performance of the i_{th} processor when workload of size w is processed, w is the size of the workload (problem size), which can be determined based on the number of operations to be applied on an assigned dataset. For this particular example, the size of the workload w is measured as number of (MAC) operations on a partition. $T_i(w)$ is the average execution time of computing workload w on the i_{th} computing element.

2.2 Workload Distribution Methods

Approaches that have been proposed to solve the problem of workload distribution can be classified as Static or Dynamic (Mittal & Vetter, 2015). The criteria for choosing a static or a dynamic approach depends on the application and the deployment requirement.

2.2.1 Workload Balancing

Load balancing is introduced in an attempt to improve the response time to a submitted workload by maximizing the utilization of computing resources. The aim is to avoid a situation in which there are overloaded computing elements while others being under-loaded or lightly-loaded. There are four basic steps in balancing the workload:

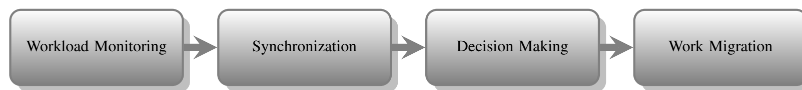


Figure 2.2: Balanced workload distribution summary

- **Workload Monitoring** : Monitoring the states of workloads, a workload can be waiting to be processed, in progress, suspended for dependency or priority purposes or finished.
- **Synchronization**: Exchanging workload information among computing resources.
- **Decision Making** : Evaluating the cost of a specific workload distribution or migration.
- **Work Migration** : The actual workload movement.

Load balancing can be achieved either statically or dynamically. Static approach is a mapping problem, which needs precise information about tasks and computing resources before the mapping. Static mapping can be performed at compile time.

However, for parallel applications in which the load varies over time the dynamic approach becomes more favourable because of the unpredictable behaviour of the tasks and computing elements during run time. Unlike static solutions in dynamic ones decisions are made during the execution time.

In the following subsections, some of the static and dynamic workload distribution algorithms are discussed.

2.2.2 Static Workload Distribution

Static distribution techniques are the ones in which the size of the workload is known in advance and an estimated processing time of the application on a specific computing environment is used to efficiently maximize the utilization of the computing resources. Such approaches are preferable, where data locality has a significant effect on the running application, due to the fact that static methods do not require data migration or redistribution. However, static approaches are inapplicable for cases where the size of the workload changes over time.

A. Lastovetsky and Manumachu (Lastovetsky & Manumachu, 2007) have proposed a data partitioning algorithm on heterogeneous systems based on static approach. They used a functional performance model (FPM) satisfying certain conditions in order to predict the execution time of the application. The approach is based on mathematically finding the points on the functional performance model in which all computing elements have the same amount of running time for the corresponding size. Downside of the approach is that it is highly dependent on the assumptions and criteria imposed on the functional performance model.

Another research conducted on Axel cluster (Tsoi & Luk, 2010) targeting the problem of workload distribution among CPU and FPGA. In this model, the execution and communication time is estimated offline based on the available bandwidth, speed and running frequency of the computing element. The drawback is that it predicts the performance of the application theoretically, which is unlikely to observe during the actual performance.

In general, static approaches are good for finding optimum solutions in case work-

load size is fixed, however, they are highly dependent on the application and existing computing elements.

2.2.3 Dynamic Workload Distribution

Dynamic distribution methods are more efficient in distributing the workload and in utilizing the computational units during run time. Work stealing (Augonnet et al., 2009) scheme is one of the dynamic approaches in which the work can be migrated from one computing element to another in order to achieve balancing, however, such a methodology can involve some redundant data migrations which affects the execution time negatively.

Another strategy is used in (Tse et al., 2010), where the workload assigned to a processing element increases linearly or exponentially. Each time a computing element requests a workload to process, the scheduler increases its associated workload size. Nevertheless, such an approach has the potential to suffer from overloaded or under loaded resources.

To sum up, dynamic solutions are more efficient for mapping dynamically changing workload but because of frequent migration of data, it is possible that a large overhead degrades the performance.

2.3 Shared Memory Programming Model

In shared memory programming model, all computing elements have the ability to access the global address space (Kaeli et al., 2015). This model is free of data movement management, however, a coherent agreement concerning updating the data, synchronization and mutual exclusion has to be defined in order to ensure memory consistency. Memory consistency can be achieved in software using high level primitives such as mutexes, semaphores and bridges. As the number of computing resources competing to gain a shared resource increases, the performance of shared memory model decreases due to contention.

2.4 Inter Process Communication

Inter process communication (IPC) is a mechanism, which allows different processes or computing resources to communicate with each other. IPC can be used to exchange messages between different computing resources. Typical structure of an IPC message consists of *Header + Payload*. The header contains information about the type of communication, source, destination and size of message. IPC uses socket as end point communication where each computing element has its own queue to store the received messages. In general, IPC is limited by the amount of data, transmission speed, latency, and data transmission route (Hossain & Tokhi, 2002).

Another form of inter process communication used is known as **Remote Procedure Call (RPC)**. RPC is typically used for one computing resource to cause a procedure or subroutine run on another computing resource in a different address space. Such form of IPC is suitable for *Server-Client* model of distributed computing.

In a distributed computing system, the entity responsible for handling communications between client and server is known as **Stub**. Each time the client issues a remote procedure request, the stub is locally called. Its job is to send the request and arguments passed by the client to the server and then to receive the result (Zhang, 2010).

One of the main advantages of using a stub is to provision transparency, in other words, the user should be unaware of how an object is being represented, accessed, located, migrated and shared, as all these functionalities are handled by the stub.

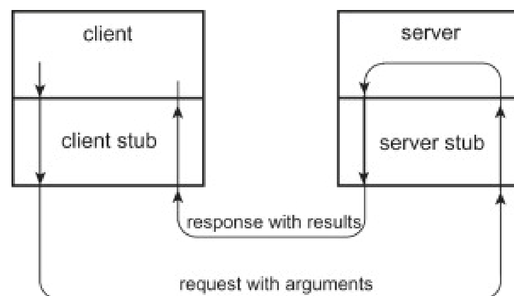


Figure 2.3: Server/Client Stub (Zhang, 2010)

CHAPTER 3

LITERATURE REVIEW

Workload distribution and parallel computing on HPC systems are vital for some applications. Thus, various approaches have been studied and proposed in order to optimize certain objectives. This chapter discusses four different approaches found in the literature for solving the problem of data partitioning and workload distribution.

3.1 A Novel Data-Partitioning Algorithm for Performance Optimization of Data-Parallel Applications on Heterogeneous HPC Platforms

(Khaleghzadeh et al., 2018) proposed a new state of the art workload partitioning algorithm on heterogeneous system. Results show an improvement in terms of speed up compared to other load balancing techniques.

The work is dedicated for data parallel applications in which all computing resources perform the same computation on different data, data set differing size. According to the proposed model, each computing element is to be assigned on a single partition of the total workload.

Their formulation of the problem is as follows:

- Let workload size be n
- Define $S = \{s_0(x), \dots, s_{p-1}(x)\}$, where $x \in \{1, 2, \dots, m\}$.
- Define $D = \{x_0, x_1, \dots, x_{p-1}\}$.

where set S is a set of speed functions of each processing element as a function of

workload size x where number of processing elements is p. The speed can be found by using equation 2.1. D is a set of assigned partitions to each processing element. The objective function is to minimize total computation time t_{opt}:

$$t_{opt} = \min_{D} \max_{i=0}^{p-1} \frac{x_i}{s_i(x_i)} \tag{3.1}$$

subject to

$$x_0 + x_1 + \cdots + x_{p-1} = n \tag{3.2}$$

where

- $0 \leq x \leq m$ and $i = 0, \ldots, p-1$
- $p, m, n \in \mathbb{Z}_{>0}$, $x_i \in \mathbb{Z}_{\geq 0}$ and $s_i(x) \in \mathbb{R}_{>0}$

The aim is to find a workload distribution set D such that objective function t_{opt} is minimized. According to this model, each computing element will be assigned a single partition of the total workload, hence the solution might produce an imbalanced workload distribution.

Their algorithm to find the minimum distribution is a modified recursive branch and bound algorithm, where the upper bound cost is defined initially as the cost of an equal distribution case, in which the workload is distributed equally among the processing elements. Cost is updated each time a minimum solution is found. The algorithm is capable of detecting a potential worst case solution so that the corresponding branch can be pruned early to improve the run time and memory usage of the proposed algorithm.

Although their algorithm is proven to improve the performance, there are still some drawbacks for further investigation. First, for the algorithm to be able to find an optimal workload distribution, the resolution of the FPM has to be large, i.e. FPM has to be smooth enough. Second, the algorithm may fail to find a feasible solution in case the size of the workload to be distributed is larger than possible workload sizes defined in the corresponding FPM.

3.2 A Proposed Data Partitioning Approach on Heterogeneous HPC Platforms: Data Locality Perspective

(Al-Hashimi & Basuhail, 2021) proposed a data partitioning algorithm which seeks to minimize the execution time of parallel applications in heterogeneous HPC environment based on two models:

- Fine-grained computational model
- Communication model

The fine-grained computational model is a performance model similar to the FPM proposed by (Khaleghzadeh et al., 2018), the performance model has to be adequate and accurate in order to obtain an optimal result. The communication model, on the other hand, is used in order to estimate the data transfer time between two processors. Their assumption is to have P number of processors modeled as a completely connected virtual network. A single link connects two processors p_i and p_j. Assuming a start-up time T_{ij} and bandwidth rate B_{ij}, the required time to send a message of size m between p_i and p_j is:

$$Cmm_{pi-pj} = T_{ij} + \frac{m}{B_{ij}} \tag{3.3}$$

in case the targeted processor is an accelerator, then to include the time can be estimated as:

$$Cmm_{pi-pj} = T_{ij} + \frac{m}{B_{ij}} + t_{AccH} \tag{3.4}$$

where t_{AccH} is the time to transfer data between host core and accelerator.

The problem is formulated as having a data of size n to be distributed on heterogeneous processors p, and a time function $t_i(x)$ where $i \in \{1, 2, \ldots, p\}$, and $x \in \{1, 2, \ldots, m\}$, for simplicity m is assumed to be equal to n and the aim is to find partition size d_i for each processor. The authors generalized the total execution time of a processor as:

$$t_i = t_{comp} + t_{comm} + t_{idle} \tag{3.5}$$

where t_{comp} is the computational time, t_{comm} is the communication time and t_{idle} is the idle time which is assumed to be small and negligible.

In order to solve the problem, the authors proposed a greedy algorithm which works iteratively to find appropriate partition size for each processor.

The steps of their algorithm is as follows:

- Find the processor with highest speed (minimum execution time) at workload size $d_i = \frac{n}{p}$ where n is the total size of the workload and p is the number of available processors.
- Iterate over the processors to find the one that has the highest speed in the range $\{\frac{n}{p}, \frac{n}{p} + 1, \ldots, n\}$. The workload of the highest speed will be assigned to the chosen processor and the execution time of that processor will be used as an initial upper bound for the next step.
- Iterate over the processors to assign workloads until the following criteria is satisfied:

$$\sum_{i=1}^{u} d_i = n \tag{3.6}$$

where $u \leq p$. The assignment of a workload x to a specific processor i is decided according to the following inequality:

$$t_i(x) + comm_i < high \tag{3.7}$$

where $t_i(x)$ is the computational time of workload x on i_{th} processor, $comm_i$ is the communication time between processor i and its predecessor and $high$ is the time upper bound found so far. If this inequality is satisfied then $high$ is updated to be $t_i(x)$ otherwise the upper bound is relaxed by incrementing $high$. In this way, the algorithm would find a workload assignment where the communication overhead is hidden by the computations.

3.3 Sigmoid: An auto-tuned load balancing algorithm for heterogeneous systems

The work of (Pérez et al., 2021) proposed a dynamic, adaptive and guided load balancing algorithm to reduce the response time and energy consumption, their aim is to efficiently execute a single OpenCL data-parallel kernel on all heterogeneous devices.

They proposed an algorithm based on a sigmoid function which splits the workload proportionally to the capabilities of the devices.

The flow of their algorithm is as follows: First an initial package or workload is assigned to each of the available devices and then wait for its completion. The decision on the size of the package is based on the initial computing speed or GFLOPs stated by the specification of the device. Whenever a device completes its job, the algorithm assigns a new package to the idle device, at the same time, the size and response time of the completed packages are analyzed in order to tune internal parameters of the algorithm throughout the execution. A good load balancing algorithm should keep package sizes as high as possible to exploit devices efficiently while not compromising adaptiveness. The core function of their algorithm which chooses a proper package size based on the current state of the system and internal parameters is:

$$Pack_size(i, G_r) = \frac{1 - e^{-k(6\frac{G_r}{G})}}{1 + e^{-k(6\frac{G_r}{G})}} \frac{G}{2N} \frac{S_i}{S_T} \tag{3.8}$$

where

- G_r : is the remaining work groups
- G : total number of work groups
- k : rate at which package size decreases
- N : number of available devices
- S_i : number of work groups that device i can compute per second
- S_T : Aggregated computing speed of the system

Some of these parameters are known beforehand, such as G_r, G and N, while the others are to be computed and updated during run time. These parameters are tuned carefully to efficiently adapt to system behaviour and avoid overhead (host-device interaction). The computing speed of each device S_i is used for the purpose of adjusting to the capabilities of the devices. The speed is determined by monitoring and finding the average speed of the last three executed packages (found experimentally). k is a parameter which determines the rate at which the size of a package decreases

throughout the execution. If k is chosen to be large, the function produces fewer and large packages which compromises the adaptiveness at the end of the execution, while choosing a small k value produces many small packages which improves adaptiveness but causes a significant amount of overhead. The authors found that for regular kernels (work groups that have the same amount of computing operations) it is better to choose large k, while for irregular kernels (having different computing loads hence unpredictable execution time) a small value for k is used, in this case a lower bound for the size of a package is set, in order to avoid excessive overhead.

3.4 Data Partitioning with a Functional Performance Model of Heterogeneous Processors

(Lastovetsky & Manumachu, 2007) proposed a methodology in which a continuous functional performance model is used in order to balance workload across a number of processors. The functional performance model represents the speed of a processor. To be able to find an optimum balanced workload distribution, the following criteria have to be satisfied by the FPM:

- On the interval $[0, x]$ the function must be:
  - monotonically increasing
  - concave
  - any straight line passing through the origin intersects the graph of the function in no more than one point.
- On the interval $[x, \infty]$ the function is monotonically decreasing.

The problem is formulated as follows: given a total workload of size n and p number of processors, the aim is to find a distribution where:

$$\frac{x_1}{S_1(x_1)} = \frac{x_2}{S_2(x_2)} = \cdots = \frac{x_p}{S_p(x_p)} \tag{3.9}$$

such that

$$x_1 + x_2 + \cdots + x_p = n \tag{3.10}$$

where x_i is the size of load on the i_{th} processor and $S_i(x_i)$ is the speed of the i_{th} processor at load x. If 3.9 is satisfied, then all points $(x_1, S_1(x_1))(x_2, S_2(x_2)) \ldots (x_p, S_p(x_p))$ lie on a straight line passing through the origin as in figure 3.1.

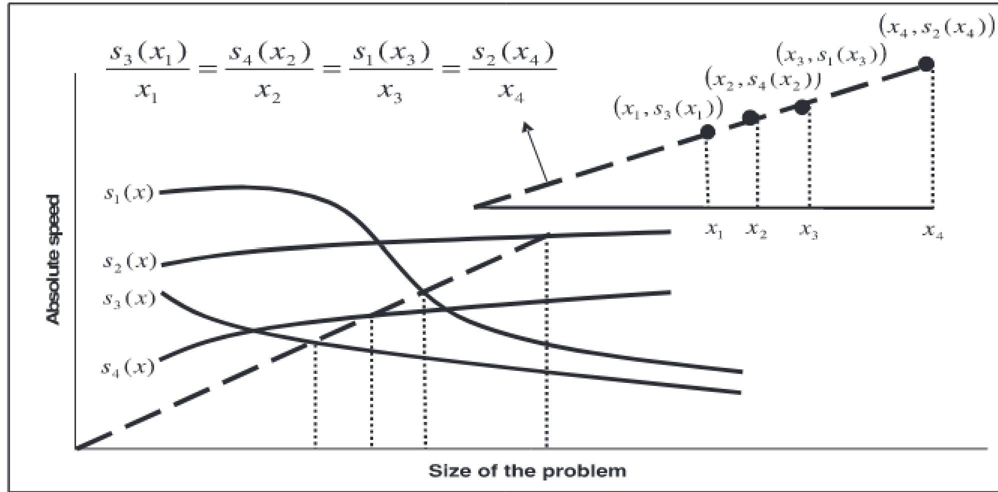


Figure 3.1: Load balancing technique proposed by (Lastovetsky & Manumachu, 2007)

The actual load size n_i assigned to i_{th} processor has to be an integer, hence n_i is approximated as either $n_i = \lfloor x_i \rfloor$ or $n_i = \lfloor x_i \rfloor - 1$.

Their proposed technique first defines two initial boundary lines U and L, both passing through the origin, and then iteratively finds the line M that bisects the angle between lines U and L and decides whether M is a new U or L based on its coordinates. This procedure stops once the approximation $n_1 + n_2 + \cdots + n_p = n$ is satisfied.

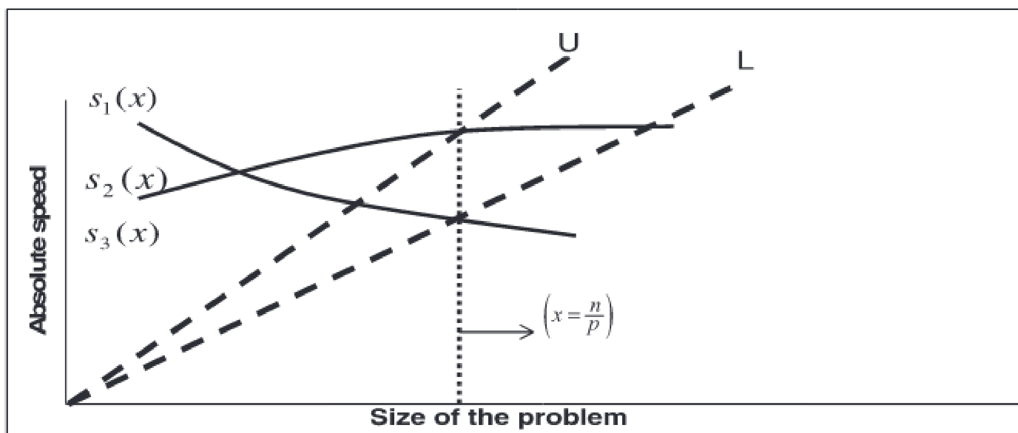


Figure 3.2: Initialization of lines U and L proposed by (Lastovetsky & Manumachu, 2007)

CHAPTER 4

WORKLOAD DISTRIBUTION ALGORITHM ON HETEROGENEOUS DEVICES (WDAH)

In this chapter, we present our proposed workload distribution algorithm. Our methodology is targeted for data parallel applications where processing kernel or threads are of the same functionality working on different partitions of the workload. Our method is initially proposed as a compile-time mapping algorithm. The present section is organized as follows: we first introduce our mathematical formulation and the concept of workload types and and then present our proposed recursive tree search algorithm. Within the scope of this chapter, *tasks* and *kernels* are used interchangeably.

4.1 Problem Formulation

Given a total workload of size N, our aim is to distribute this workload among a set of processing elements such that the total execution time is minimized. We assume that the total workload can be partitioned into smaller sub-partitions, which we call *workload packages workload types* of different sizes. We create a number of tasks (kernels) that can run in parallel and each task will be assigned a single sub-partition. The following are the definitions of problem variables:

- N : total workload size.
- $P = \{p_1, p_2, \ldots, p_k\}$: set of processing elements, $k \in \mathbb{Z}_{>0}$.
- $W_j = \{w_{1j}, w_{2j}, \ldots, w_{mj}\}$: set of workload packages (sub-problems or partitions) associated with processing element j, $m \in \mathbb{Z}_{>0}$ and $j \in \{1, 2, \ldots, k\}$.

- $S_j(w)$: function that returns the execution speed of load w on processing element j, $S_j(w) \in \mathbb{R}_{>0}$.
- $T = \{t_1, t_2, \ldots, t_n\}$: Generated set of tasks (sub-problems), $n \in \mathbb{Z}_{>0}$, initially empty.
- (w^i_{rj}, D_i): 2-tuple associated with task t_i, $i \in \{1, 2, \ldots, n\}$.
  - w^i_{rj}: workload package (partition) where $w^i_{rj} \in W_j$ and $w^i_{rj} \in \mathbb{Z}_{>0}$ and $r \in \{1, 2, \ldots, m\}$.
  - D_i: partition of the workload assigned to t_i.

$$u_{ij} = \begin{cases} 1 & \text{if } t_i \text{ is assigned to } p_j \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

Then the problem is to minimize f

$$f = \min\,(\max_{j=1}^{k} \sum_{i=1}^{n} \frac{w^i_{rj}}{S_j(w^i_{rj})} \cdot u_{ij}) \tag{4.2}$$

subject to

$$\sum_{j=1}^{k} u_{ij} = 1, i \in \{1, 2, \ldots, n\} \tag{4.3}$$

$$\sum_{i=1}^{n} \sum_{j=1}^{k} w^i_{rj} \cdot u_{ij} = N \tag{4.4}$$

$$\bigcap_{i=1}^{n} D_i = \emptyset \tag{4.5}$$

u_{ij} is the binary utilization variable that shows whether task t_i is assigned to processor p_j. Objective function f aims to map n number of tasks, where each task is responsible for a specific sub-partition of the total workload, to k number of available computing resources such that the total computation time is minimized.

Constraint 4.3 states that each task should be assigned to a single processing element. Constraint 4.4 ensures that the total workload of all assigned tasks sum up to the

original total workload size while constraint 4.5 ensures that partitions assigned to tasks do not overlap. The number of tasks created is not defined in the model but is part of the solution hence differs for each feasible solution.

4.2 Generation of workload packages

Functional Performance Model represents a profile for the performance of the computing resource when a sequential routine or a kernel processes a workload. Looking at the graphical representation of a functional performance model, a certain behaviour can be observed. Consider figures 4.1 and 2.1, which show FPM for two different applications on two different processing elements. The noticeable behaviour is that FPM initially increases monotonically, reaches a maximum and then starts decreasing again. Such behaviour can be reasoned as for small workloads the computing resource performs computations most of the time but as workload size increases, number of computations performed in unit time increases so overall speed increases while memory access time becomes dominant after some point. Our proposed hypothesis is that using only the monotonically increasing region of FPM, dashed and coloured in red in figure 4.1, one can find a minimum solution for the workload distribution problem, this hypothesis is verified experimentally.

We propose to create workload packages set W_j by using the monotonically increasing region of FPM. The entities of set W_j represents possible workload sizes that can be offloaded to a particular task. Range of workload sizes contains the smallest possible workload size up to and including the workload size of maximum speed for the targeted computing resource.

In order to ensure the validity of the proposed method, we impose the following requirement: FPMs should have its smallest workload size w_{min} such that $N \mod w_{min} \equiv 0$, where N is the problem size or total workload size. Moreover, FPMs are highly recommended to be fine-grained for better optimization.

Our approach is aimed to use only the generated 'workload packages set' in order to find a near optimum workload distribution that is independent of the problem size. In other words, our methodology tries to use the minimal amount of prior information

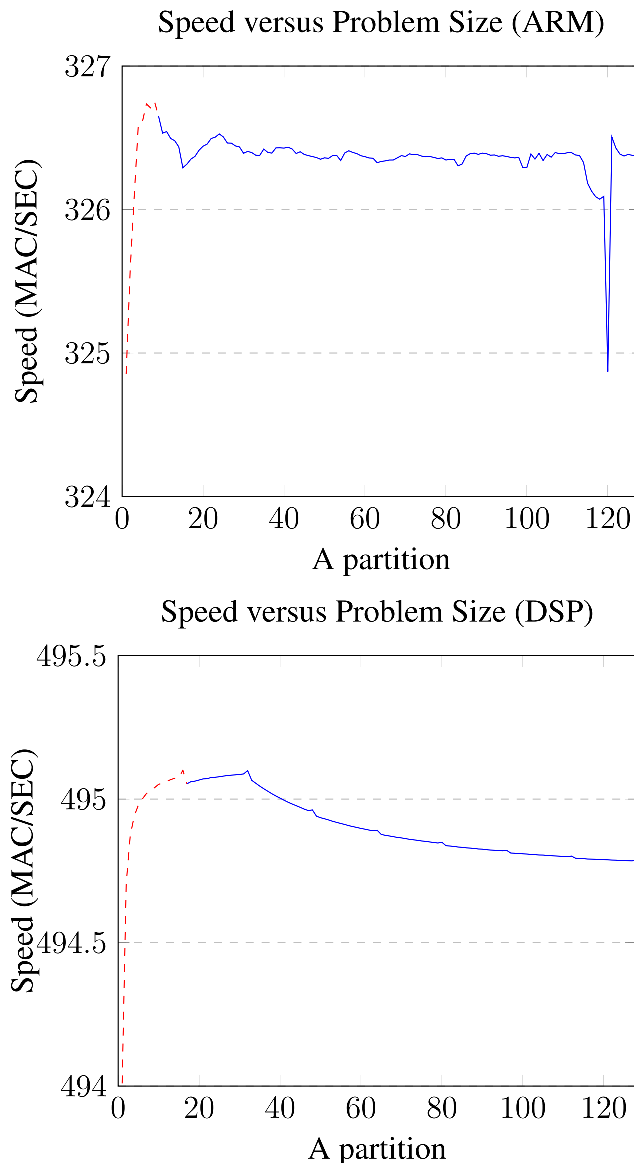


Figure 4.1: Advanced performance model for Matrix Multiplication on ARM A15 and DSP C66.

about the computing resource. Using a small amount of prior information is of vital importance from efficiency and feasibility point of view.

Using only a small amount of profiling information results in reducing the solution space, hence reducing the complexity and consequently the running time of the algorithm and the memory usage required by the algorithm. Reducing the complexity of the mapping algorithm and its memory usage is significant for low-cost and real

time systems. Low cost systems are not typically meant for huge computations nor to store large amounts of information, hence, using a small amount of prior information makes our methodology feasible to run on low cost devices also. Work dispatchers of real time systems needs to have dispatching overhead as low as possible. This is essential in order to maintain the *Quality of Service (QoS)* required by the applications or clients.

Moreover, it is infeasible to create an FPM profile for large size problems from both time and memory point of view. For creating a profile that covers large problem sizes, a large number of samples might be needed. Consider a problem that needs 512×512 MAC operations as total workload. In order to create an FPM for such a problem, we need to start with a single MAC operation as the possible minimum workload and for having a smooth FPM, the stride between possible workload sizes has to be small. For this example, if the stride is set to be 1, there are 512×512 number of possible loads that need to be profiled, which requires a considerable amount of time to complete. Furthermore, the available memory may not be sufficient hence making minimization need for prior information to be critical.

4.3 Partitioning Correctness

Merging generated partitions and verifying the correctness of merge operation is left as a high level responsibility within the scope of this study. In other words, when subproblems (partitions) are simply combined, it is not guaranteed that the final output is equivalent to the output if the input is solved sequentially because further steps might be needed for correct merging. Nevertheless, it is advised, but not mandatory, to build the FPM based on a defined minimal independent amount of workload package or a partition, called as λ, so that solving λ as a part of the overall input will result in correct final output as a result of a trivial merging, and then define other possible partitions of the FPM to be multiples of λ, i.e. $2\lambda, 3\lambda$...etc.

Figure 4.2 shows an example of partitioning for parallel matrix multiplication. First, one row of input matrix is chosen as λ, applying multiplication on one row results in a correct one row at the output. Another possible partitioning is three consecutive

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \end{pmatrix} \qquad \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \end{pmatrix}$$

Figure 4.2: Instances of possible partitions for an input matrix to compute parallel matrix multiplication

row partition, i.e. 3λ.

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \end{pmatrix}$$

Figure 4.3: Instance of a partition for an input matrix

The partition in figure 4.3 is also possible, however, the processing steps of such partition has to be different than the one for cases in figure 4.2, as more processing operations might be needed for generating an output.

4.4 Heterogeneity Requirements

The method proposed in the present work is highly effective in case of strong heterogeneity between computing elements, i.e. the difference in computing performance among existing computing elements is significant and noticeable. In case of weak heterogeneity where computing elements are similar in terms of performance, then our method might not provide any advantage in comparison to classical methods.

4.5 Tree Search Algorithm

The algorithm presented in this subsection is designed to solve the model formulated in 4.1.

Assume that we are given a total workload of size N and K number of processing elements. After the profiling step, for each processing element j, there is an associated set of workload packages W_j, and a speed function $S_j(w)$ which returns the computation speed if load w is assigned on processor j. The workload packages in W_j are sorted in descending order based on their speed. The following variables are also initialized:

- $RemSize$: remaining workload size, initially N.
- P: set of processing elements.
- pe: a processing element with three attributes; ID, set of workload packages, initial priority.
- sol: initially empty solution implemented as a linked list of nodes, each containing information about creating a task and assigning it to a certain processing element.
- $fsol$: initially empty feasible solution corresponding to minimum cost found so far.
- $Cost$: global cost that is updated every time a feasible solution is reached, initially a large value.
- $BackTrack$: variable to control and limit the number of times the search tree is pruned.

Each processing element pe is initially given a priority value calculated using 4.6.

$$priority_j = \frac{\max_{w \in W_j} S_j(w)}{\sum_{i \in P} \max_{w \in W_i} S_i(w)} \times N \tag{4.6}$$

where $S_{ij} = S_j(w_i)$. The priority value is an indication of the computing resource's speed in comparison to others. The higher the priority, the better the computational

capability. In order to avoid starvation, the priority assigned to each computing element is dynamically changed throughout the offloading process. Each time a task with a certain workload is assigned to a processing element, its priority will be decremented by an amount equal to the size of the assigned workload. Thus, there will be no over-loaded or under-loaded computing elements after the offloading process workloads.

The pseudo-code of our proposed recursive tree search based workload distribution algorithm (WDAH) is presented below in the form of three algorithm components, nemely Algorithm 1 (initialization),2 Branch-and-Cut(tree search) and 3 (get pocessor). The steps are analyzed and the behaviour of the overall algorithm is described in detail next.

Algorithm 1 Workload distribution algorithm (WDAH) - initialization

```
1:  procedure main(void)
2:      RemSize = N
3:      P = ∅
4:      for i = 0 → k − 1 do
5:          pe(i, WorkloadPack, Priority)
6:          P.InsertProc(pe)
7:      sol = null
8:      fsol = null
9:      Cost = ∞
10:     BackTrack = LargeValue
11:     WDAH-Branch-and-Cut(RemSize)
```

The proposed solution is described in detail below:

1. First check the remaining size $RemSize$ of the workload:
    (a) If it is not larger than zero, a feasible solution is reached hence update the global cost $Cost$ and final solution list $fsol$.
    (b) Otherwise proceed to algorithm 2 step 3.
2. Choose the processing element pe with the highest priority value from set P

Algorithm 2 Workload distribution algorithm (WDAH) - Branch and Cut (tree search)

```
1: procedure WDAH − Branch − and − Cut(RemSize)
2:     if RemSize > 0 then
3:         pe = GetProcessor()
4:         wt = pe.GetWorkloadPack()
5:         for w ∈ wt do
6:             if w ≤ RemSize then
7:                 sol.InsertNode(w, pe)
8:                 if sol.GetCost() > Cost then
9:                     sol.RemoveTopNode()
10:                else
11:                    if BackTrack > 0 then
12:                        WDAH-Branch-and-Cut(RemSize − w)
13:                        BackTrack − −
14:                    sol.RemoveTopNode()
15:    else
16:        Cost = sol.GetCost()
17:        fsol = sol
```

Algorithm 3 Workload distribution algorithm (WDAH) - get processing element)

```
1: procedure GetProcessor(void)
2:     x, PE = max(P)
3:     if x < 0 then
4:         P.ResetPriority()
5:         x, PE = max(P)
6:     return PE;
```

3. Get the associated set of workload packages.

4. Iterate over the set of workload packages W_j, where j is the ID of the chosen pe, starting with the workload package w corresponding to highest speed.

5. Check if the remaining workload $RemSize$ fits to the size of the chosen workload package w :

   (a) If no then go back and continue at algorithm 2 step 5.

   (b) Otherwise algorithm 2 step 7.

6. Create a task t_i responsible for computing the chosen workload package w and assign it to the chosen processing element. Update the current cost after each task creation and the priority of the utilized computing element pe.

7. Compare current cost with the global one:

   (a) If current cost is larger than global, remove the newly added and evaluated assignment, restore pe priority and continue with algorithm 2 step 5.

   (b) Otherwise proceed to algorithm 2 step 11.

8. Check the number of backtracks:

   (a) If larger than zero, update the remaining workload size and go back to algorithm 2 step 1.

   (b) Otherwise remove added assignment and proceed to continue with algorithm 2 step 5.

In algorithm 2, the function $GetProcessor()$ picks the processing element with the highest priority from P. If highest priority is negative, priority values of all processing elements in P gets reset using $ResetPriority()$. $GetProcessor()$ returns a processing element object pe. The workload packages associated with a processing element can be accessed using $GetWorkloadPack()$, which returns a W_j. The algorithm iterates over workload packages starting with the highest speed. Once a workload package is chosen and the remaining workload fits in the size of the workload type chosen, $InsertNode(w, pe)$ is invoked so that a new task is created with load w and assigned to pe.

When a task assignment is done, an intermediate data structure node carrying this information is added to the list of sol object to be used in realizing the tree search. When a new node is added to the list, the sol object automatically updates its cost, pe's priority and compares the current cost with the global one and if the former is larger, then the algorithm removes the newly added node from the list using $RemoveTopNode()$, meaning that the current branch of the solution tree is pruned. If the current cost is less than the global one then the algorithm proceeds to expand the current branch of the search tree recursively after updating the remaining workload size as long as $BackTrack$ is larger than zero.

Upon invoking $InsertNode(w, pe)$ the priority of pe is updated according to the following equation:

$$priority(pe) = priority(pe) - w \tag{4.7}$$

while calling $RemoveTopNode()$ updates the priority of pe according to:

$$priority(pe) = priority(pe) + w \tag{4.8}$$

where w is the size of the workload added or removed.

The following simple example is constructed to demonstrate and clarify the steps of algorithm. Consider $N = 16$, $P = \{1, 2\}$, $W_0 = 2, 4, 8$, $W_1 = 2, 4, 8$ and

$$A = \begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 6 \end{bmatrix} \tag{4.9}$$

where rows and columns of matrix A correspond to workload packages and processing elements, respectively. Each entry is the average execution time of workload package w on p_j.

First step is to order the workload packages for each processing element based on their speed using equation 2.1. Table 4.1 shows the speed of each workload package of this example. It is worth noting that each column in the table corresponds to FPM of the corresponding processing element.

For this particular example, using equation 4.6, one can find the priority of p_0 as 9.6 and p_1 as 6.4.

Table 4.1: Processing speed subject to workload package

workload package	P0 Speed	P1 Speed
2	1.00	0.67
4	1.33	1.00
8	2.00	1.33

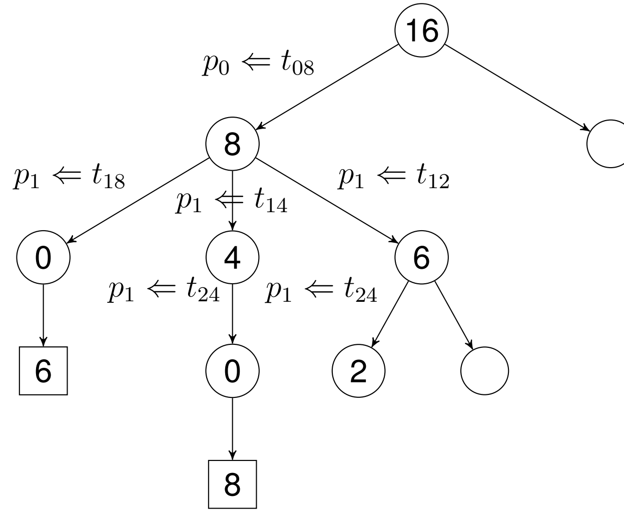


Figure 4.4: A snapshot of the tree search process for the example problem

Figure 4.4 shows a snapshot of the tree search process. The value at each node represents the remaining size of the yet unassigned workload and each edge represents the creation and an assignment of a task to a processing element. The terminals of the search tree are feasible solutions, the path from the root to a terminal constructs the assignments associated with the solution. The value in each rectangular box is the cost of the corresponding feasible solution. For this particular example, the root node has a value 16, which is equal to the total size of the workload. In figure 4.4, the notation $p_j \Leftarrow t_{ik}$ means that task number i and of workload package having size k is created and assigned to j_{th} processing element.

Step (2) checks the remaining size of workload firs. Since we are at the root of the tree, the remaining size is 16. Then the processing element with the highest priority is selected, in this case P_0. At step (3), (4) and (5) the workload package having the highest speed is selected, in this case $w = 8$, since $8 \leq 16$ the condition at

step (6) is true and we continue at step (7). At step (7), a task t_0 is created with load w being 8 and is assigned to run on P_0. After this assignment, at step (8), the current cost is compared with the global cost. Since the condition is not satisfied, the algorithm proceeds to step (10). At step (11), the number of backtracks is checked. The backtrack value is larger than 0 and the remaining workload size ($RemSize$) is now 8. The priority of P_0 is updated to be 1.6 and the accumulated cost is $\frac{8}{S_0(8)} = 4$. The highlighted branch in Case 1 in figure 4.5 illustrates the first search path evaluated by the algorithm which corresponds to the feasible solution with total cost 6 at the associated terminal node and assignment.

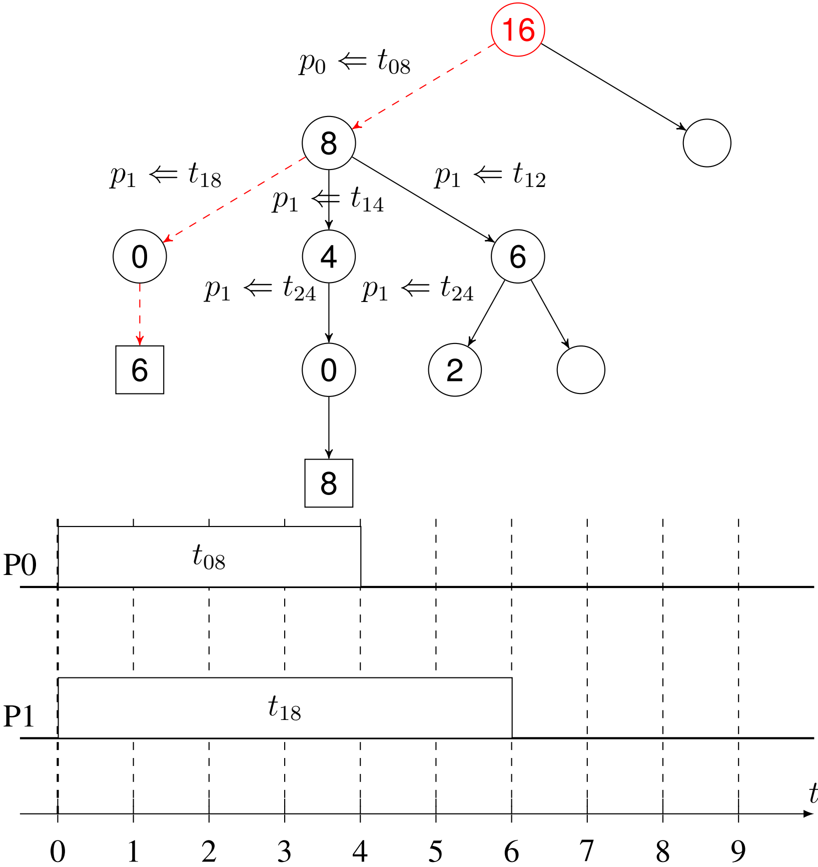


Figure 4.5: Search process of case 1.

When a terminal is encountered, the algorithm backtracks and starts investigating other possible solution paths in the search space. The case depicted in case 2 in figure 4.6 shows a scenario, where the algorithm performs a backtrack step back to node having remaining workload size 8. After each backtrack step, the assignment on the edge of the backtracked tree edge is removed. For example, assignment of t_{18} to p_1

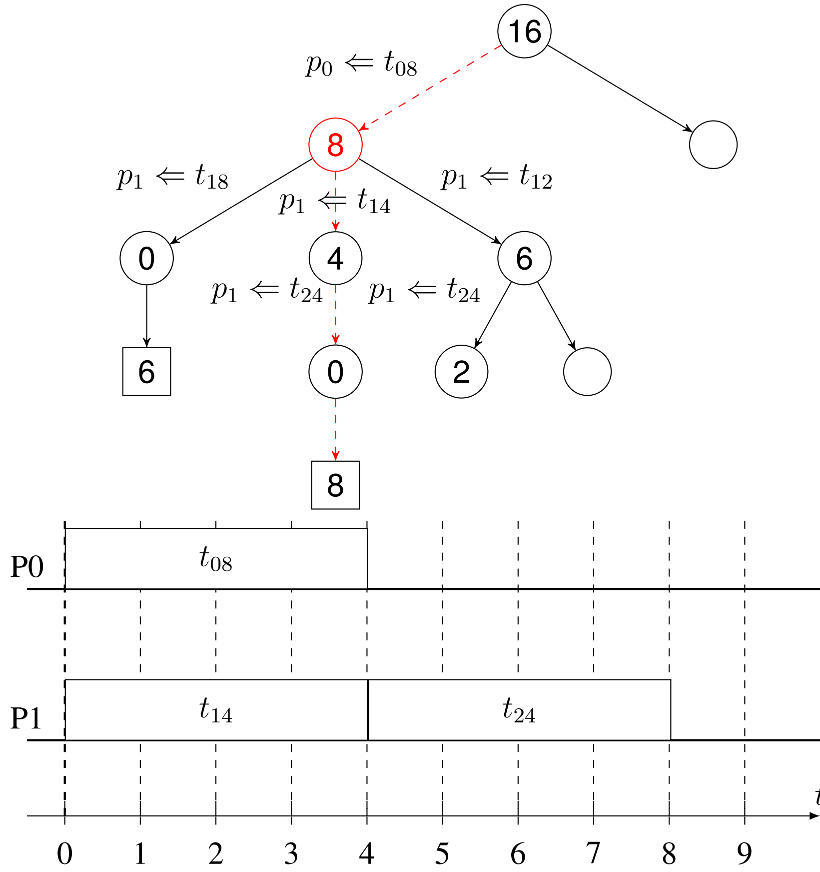


Figure 4.6: Search process of case 2.

is removed and a new assignment $p_1 \Leftarrow t_{14}$ is performed, meaning a new workload package is assigned to t_1. Following this path, the algorithm reaches another feasible solution, however, its cost (8) is larger than the current minimum cost (6) found up to this point. As a result, this solution is ignored.

Eliminating or cutting a branch in the search tree is also considered in our approach in order to reduce the search space. The case where the algorithm performs the following assignment, $p_0 \Leftarrow t_{08}$, $p_1 \Leftarrow t_{12}$, $p_1 \Leftarrow t_{24}$ as shown in Case 3 in figure 4.7, makes the remaining workload size 2, however, the current cost becomes 7, which is larger than the current global cost, hence the algorithm may stop expanding that branch.

The algorithm keeps expanding and searching branches looking for a global minimum solution, nevertheless, the number of branches to be expanded is limited by the number of backtrack operations the algorithm is permitted to perform, the larger the number of backtracks, the higher the chance to find a global minimum. With lim-

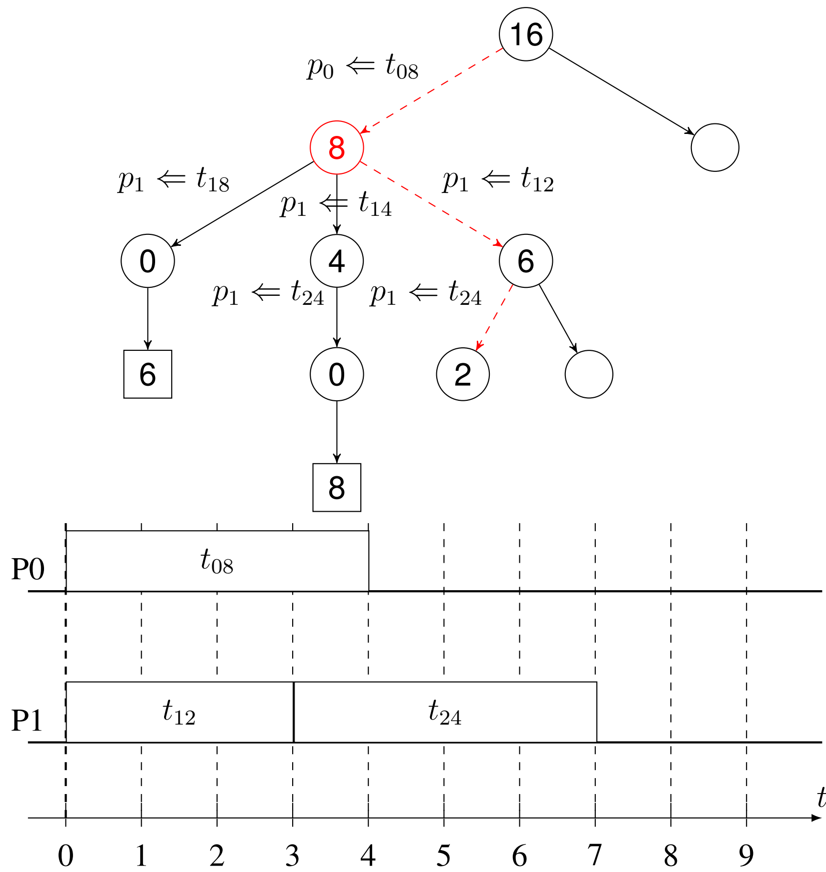


Figure 4.7: Search process of case 3.

ited number of backtracks, the algorithm behaves as a heuristic approach, the found solution being possibly non-optimal.

A special case of our algorithm is when $Backtrack = 1$, in this case the algorithm reduces to a simple heuristic approach, the pseudocode of this case is depicted in 4.

Theoretically, the solutions generated by WDAH-Heuristic should be as good as the solutions generated by WDAH-Branch-and-Cut, the reason for that is the supervised search process using the priority assignment scheme and the generated workload packages. Indeed, the algorithm is designed as such it explores better quality solutions first. Hence, using WDAH-Heuristic is more reasonable to use in most of the cases instead of WDAH-Branch-and-Cut since it involves less computations and generates solutions having the same quality of WDAH-Branch-and-Cut.

Algorithm 4 Workload distribution algorithm (WDAH-Heuristic)

```
procedure WDAH − Heuristic(RemSize)
    if RemSize > 0 then
        pe = GetProcessor()
        wt = pe.GetWorkloadPack()
        for w ∈ wt do
            if w.Size ≤ RemSize then
                sol.InsertNode(w.Size, pe)
                if sol.GetCost() > Cost then
                    sol.RemoveTopNode()
                else
                    WDAH-Heuristic(RemSize − w.Size)
    else
        Cost = sol.GetCost()
        fsol = sol
```

The full implementation of 1, 2, 3 and 4 can be found in A.

CHAPTER 5

IMPLEMENTATION AND EXPERIMENTAL EVALUATION RESULTS

The present chapter discusses our implementation and the experiments conducted to verify and evaluate our proposal. First, the set up and the hardware architecture used in our experimental study is presented. Then the conducted experiments are described and then obtained results are discusses

5.1 Implementation Setup Architecture

For an accurate verification, a well defined parallel computing model is needed. As was mentioned earlier, this work assumes shared memory computing model. Furthermore, for application and computing elements management, the *Server-Client* model of distributed computing systems is inherited. Finally, the Remote Procedure Call (RPC) form of inter process communication is deployed for resource communication and task dispatching purposes.

The architecture of our implementation is depicted in figure 5.1. There are four main components:

- Dispatcher.
- Lookup Table.
- Computing elements.
- Shared Memory.

The mapping solution generated by our proposed tree search algorithm is stored in a *Lookup Table*. Based on the application type and size of the workload, the *Dispatcher*

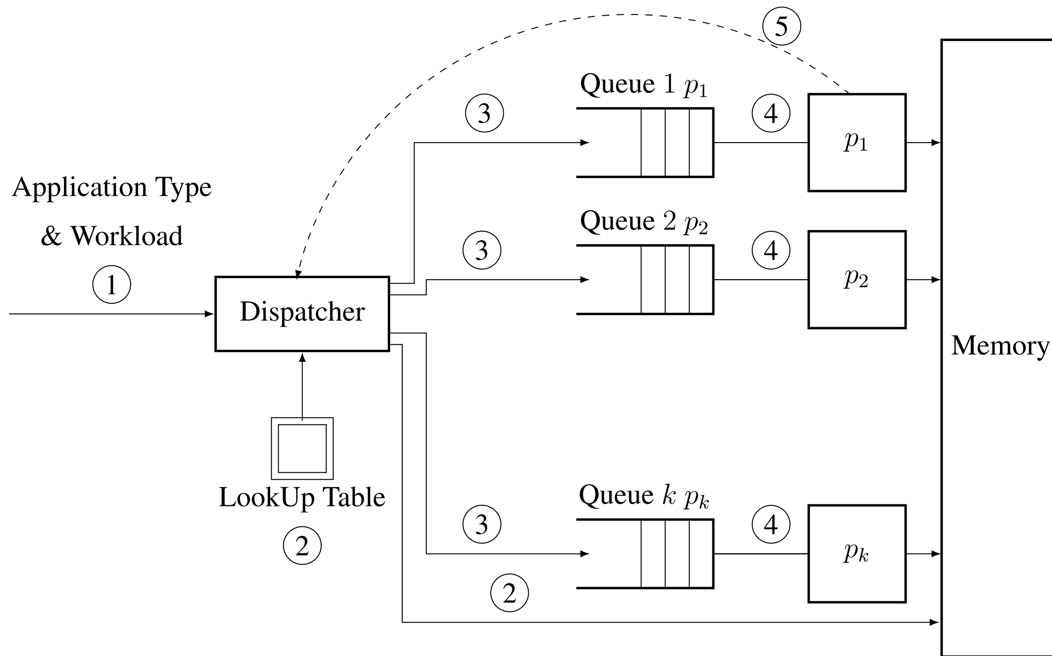


Figure 5.1: Implementation Setup Architecture

fetches the associated mapping and offloads the tasks to the queues of the computing elements. The *Computing Element* reads and executes the tasks pushed into its queue as long as the queue is not empty. The workload associated with each task is stored in *Shared Memory*. The steps shown in figure 5.1 are summarized as follows:

1. The dispatcher receives an application and its workload as an input.
2. The dispatcher uses the lookup table in order to get the stored mapping solution and at the same time it writes the workload to the shared memory.
3. The dispatcher pushes RPC messages into the queue of each computing element using the given mapping. RPC messages contain information about the task to be executed including its workload.
4. Each computing element checks the status of its queue, if there is a message within the queue, the computing element starts handling message immediately.
5. Finally, after handling each message, the computing element sends a message to inform the dispatcher that the task has been processed.

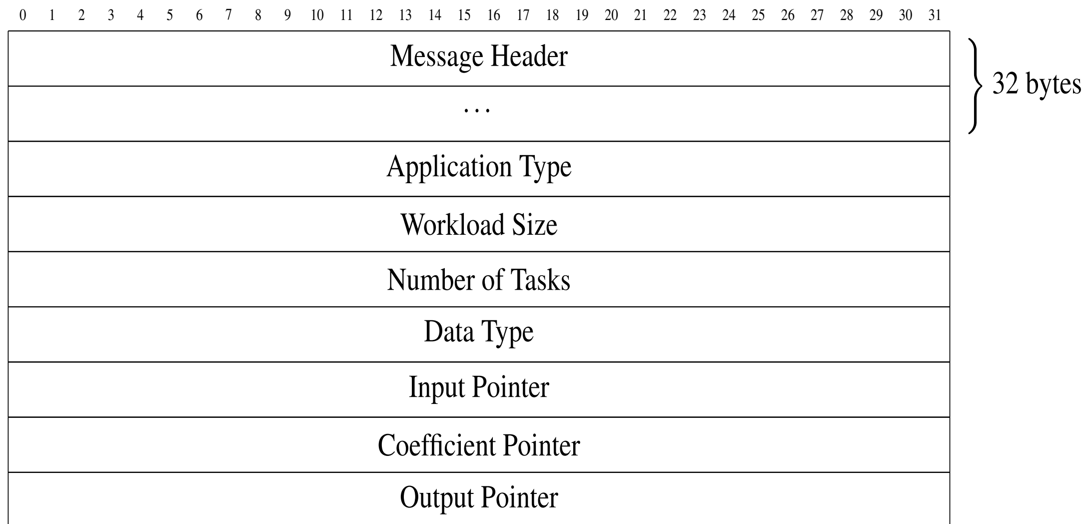


Figure 5.2: RPC Message Layout

The structure shown in figure 5.2 illustrates the fields of an RPC message to be exchanged between the dispatcher and a computing element. There are eight different fields:

- **Message Header**: Field of 32 bytes containing information about the source and destination computing elements, total size of the message, message ID, . . . etc.
- **Application Type**: Indicates type of computation to be performed. The targeted computing elements decides which routine to execute based on this field.
- **Workload Size**: Indicates the size of the assigned workload.
- **Number of Tasks**: Indicates the number of tasks having the same type to be executed.
- **Data Type**: Determines type of data (e.g. integers, floating points values, complex . . . etc)
- **Input, Coefficient, Output Pointers**: Pointers to the memory region in which the computations should take place.

The field *Number of Tasks* in the message structure is mainly used to reduce the number of messages to be sent so that communication overhead can be reduced. In

such a framework, there could be a possibility to create a number of tasks having the same workload size but working on different regions of the dataset, instead of sending multiple messages for each task, a single message can be sent with the field *Number of Tasks* containing number of tasks of this type to be handled. The memory region of each task can be implicitly found using the *Workload Size*, *Number of Tasks* and *Data Type* as an offset from the initial *Input, Coefficient and Output Pointers*.

In order to verify our methodology, two types of experiments are conducted: i) Simulation and ii) Physical Test. Our simulation experiment is based on collecting already used benchmark functions on different devices as targeted applications. The benchmark works as a functional performance model of each application on each computing element. The benchmark is fed as an input to our proposed algorithm along with a total workload size. The output of the algorithm is then a workload distribution solution with its estimated execution time. These simulation results are used to compare our proposed solution with existing methods.

On the other hand, physical experiments are also conducted using 66AK2H12 TI platform. Broad specification of this device is given in figure 5.3. The device consists of 8 DSP C66 cores and 4 ARM cores Cortex-A15. The board resembles our defined architecture in figure 5.1. Each core is considered as a computing resource. Although HPC systems should have a large degree of diversity, this board can still be classified as a small scale heterogeneous device.

Table 5.1: Computing elements Specifications

Specification	ARM A15	DSP C66
Maximum Operating Frequency (GHz)	1.4	1.2
Maximum Theoretical Speed (GMACs)	Not Applicable	38.4
L1D Memory (KBytes)	32	32
L2 Memory (KBytes)	4096 (Shared by 4 Cores)	1024 (per core)

During run time, RT Linux version 06.03.00.106 is set to run on ARM CorePac while TI-RTOS (also known as SYS/BIOS) is running on C66 Cores. The experimental framework is developed in C programming with the help of PThread library and IPC Library. PThread Library is used to create multiple working threads, one for each

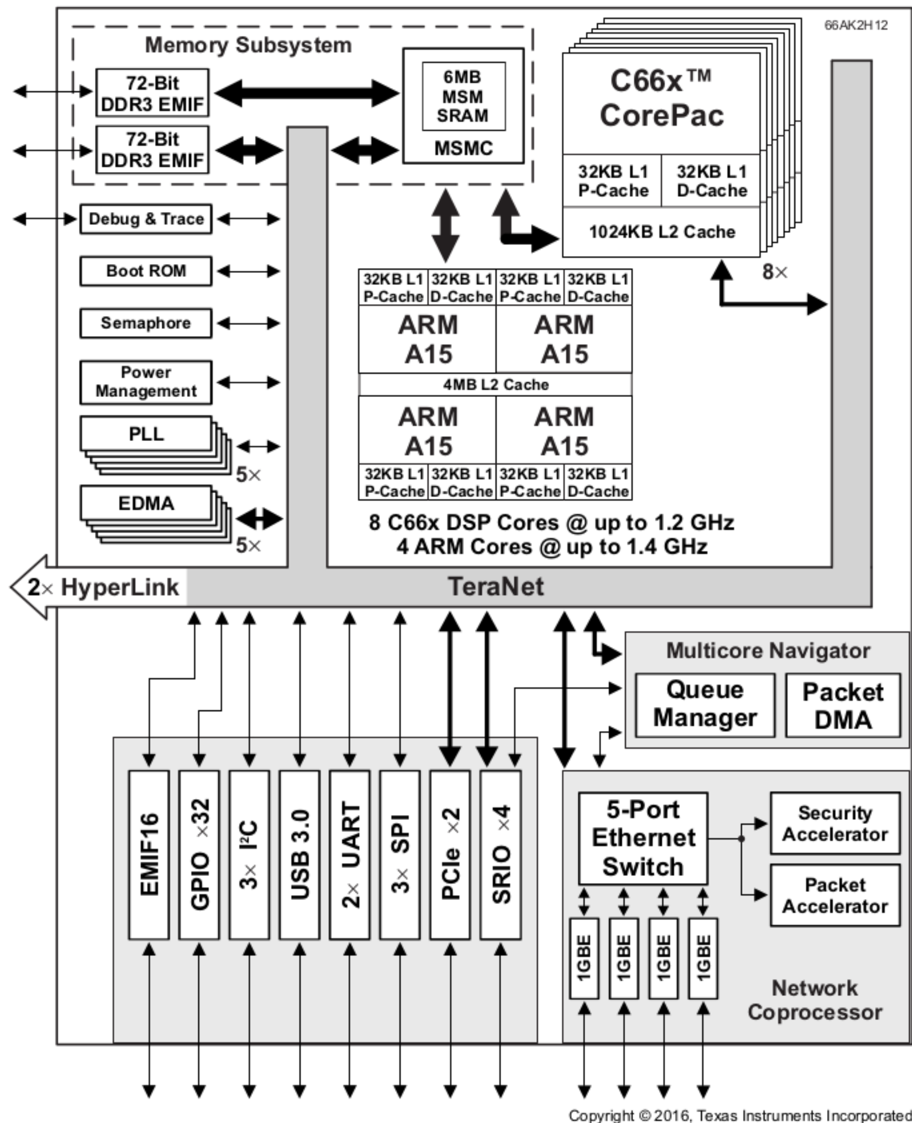


Figure 5.3: 66AK2H12 Block Diagram (Texas Instruments, 2017)

ARM Core, and the Dispatcher. The Dispatcher here works as a coordinator in order to manage the flow of the work. On the DSP side, TI-RTOS application programming interfaces (APIs) are used to create working threads. The worker threads are blocked on working queues as long as these queues are empty. IPC library is used in order to create RPC messages and append these messages to targeted queues. IPC library also works as a stub so that information exchanged between ARM and C66 cores are correctly interpreted by both sides, in particular, memory address translation and mutual exclusion are handled. IPC also handles the access process to both reader and writer process queues using mutexes and semaphore primitives. The dispatcher

thread working on the ARM side, reads a stored workload mapping for a particular application from the lookup table, then starts to append RPC messages to each core's queue in accordance to the given mapping solution. Algorithms 5 and 6 summarises the working policy of our experimental framework.

Algorithm 5 Dispatcher

1: $Map = GetMapping(AppType, N)$
2: $NumMsg = 0$
3: **for** $Slot \in Map$ **do**
4:     **for** $Item \in Slot$ **do**
5:         $Msg = CreateRPC(Slot.PE, Item.W, Item.numTasks, AppType)$
6:         $AppendRCP(Msg)$
7:         $NumMsg++$
8: **while** $NumMsg! = 0$ **do**
9:     $ReadQueue()$
10:     $NumMsg--$

Algorithm 6 Worker

1: **while** $True$ **do**
2:     $Msg = ReadQueue()$
3:     **for** $tsk \in Msg.numTasks$ **do**
4:         $ExecRoutine(Msg, tsk)$
5:     $ReplyBack(Msg)$

Our setup is an asynchronous and event-driven system, which means it is triggered based on occurring events. Each new application and workload triggers the dispatcher to start an off-loading process for the given workload.

$GetMapping(AppType, N)$ uses the lookup table structure given in figure 5.4 to get the appropriate mapping based on application's type and workload size. The returned mapping information is in the form of a table containing slots, each slot corresponds to a processing element and items within each slot being tasks to be created and assigned.

In order to fetch a mapping from the lookup table, first an application needs to be

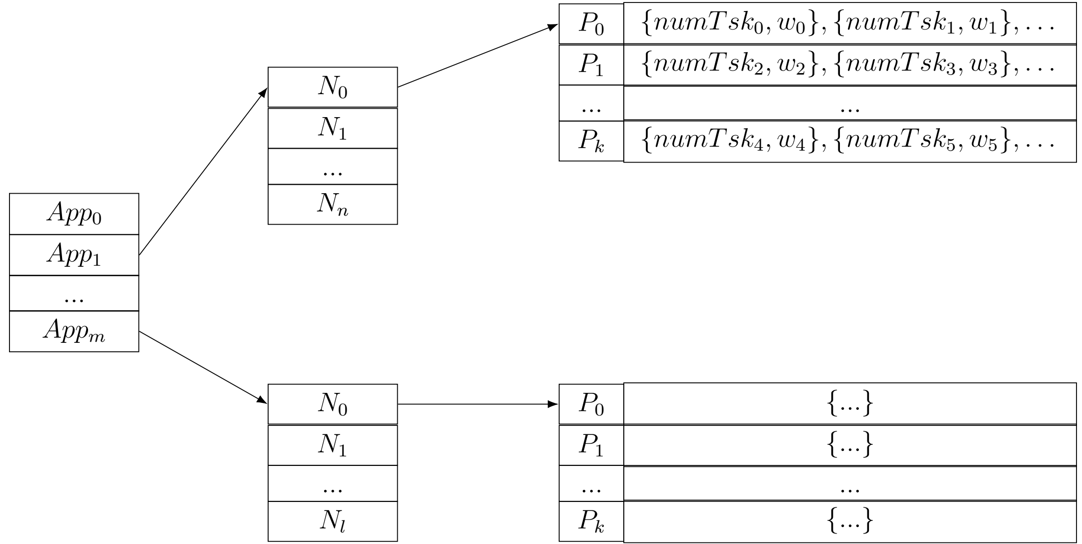


Figure 5.4: Lookup Table Structure

matched. Second, after resolving the application, required workload size needs to be found, each workload size pointing to a bundle of slots in which the actual mapping is contained.

For each slot within the mapping, the dispatcher parses the suppressed items, each item consisting of a workload type and the number of tasks with the same workload type to be dispatched.

The function $CreateRPC(Slot.PE, Item.W, Item.numTasks, AppType)$ creates a packet with the format shown in figure 5.2. The entries of the message are filled using the input arguments as well as globally declared variables, in particular

- $Current_Input_Pointer$
- $Current_Coefficient_Pointer$
- $Current_Output_Pointer$

Each time a message is created, these variables are updated to ensure consistency. The equations used for updating these variables are given below:

- $Current_Input_Pointer = Current_Input_pointer + NumTsk \times w$
- $Current_Output_Pointer = Current_Output_Pointer + NumTsk \times w$

where $NumTsk$ and w represent number of tasks and workload size, respectively. $Current_Coefficient_Pointer$ is updated based on whether the coefficients are shared or private.

Subsequently, the messages are appended to the queue of the targeted processing element using the function $AppendRCP(Msg)$. Target processing element is found using the destination processor field of the message header. $NumMsg$ is a local variable used to accumulate the number of messages sent. Tthe dispatcher should receive the same number of sent messages as an indication that all computing elements have handled all assigned tasks. $ReadQueue()$ function is called to return the message at the front of the queue. In case queue is empty, the caller blocks waiting for an event to happen, in this case for an insertion of a message.

On the other hand, the computing element or $Worker$ starts by calling the $ReadQueue()$ function. As long as there are no messages, the $Worker$ blocks waiting for a new task. For each task, the $Worker$ calls $ExecRoutine(Msg, tsk)$ that resolves type and size of computation to process and the address range i.e. input and output data regions to be used for the computation. The $Worker$ finds the memory addresses of the input and output data using the following equations:

- $InAddr = Msg.Input_Pointer + Msg.WorloadSize \times tsk$
- $OutAddr = Msg.Output_Pointer + Msg.WorloadSize \times tsk$

where $InAddr$ and $OutAddr$ are input and output addresses and tsk is the task ID number. Upon completion, the $Worker$ sends a reply back message to the dispatcher using $ReplyBack(Msg)$ and either goes to block state if there is no job to handle or starts processing another task.

5.2 Experimentation

Four different tests are conducted in order to evaluate the performance of our proposed method:

- Optimality test with respect to problem size.

- Speed up test with respect to number of computing elements.
- Optimality test with respect to backtrack parameter.
- Running time of the algorithm (time complexity).
- Utilization test.

The first test is conducted in order to observe the ability of the algorithm to generate a minimum cost solution as the problem size varies. The speed up test is performed to observe the obtained improvement when the number of computing elements changes. The third test is meant to observe the performance of the algorithm as we vary the backtrack parameter. Then, we study timing performance the algorithm by measuring the running time with respect to problem size and finally we test it to observe the utilization of the computing elements.

5.2.1 Optimality Test (Solution quality vs. Problem size)

This test aims to observe the change in the cost of a generated solution, i.e. total computational time for an application, when problem size varies. This test is performed as a simulation of a benchmark Fast Fourier Transformation (FFT) application on three different computing elements; more specifically Intel Haswell multicore CPU, Nvidia K40c GPU, and Intel Xeon Phi 3120P. FFTW is a library used for computing the discrete Fourier transform (DFT), which also supports a multi-threaded implementation of the FFT routine.

We obviously did not have these computing resources physically. We collected already reported benchmark FFTW [1] results on these computing elements and used these as the Functional Performance Model, provided as an input to our algorithm and used the cost of the created mapping solution obtained. Throughout this test, the comparison of the created solutions is made with the following alternative approaches: (i) classical equal distribution, (ii) balanced workload distribution [1] and (iii) Heterogeneous Performance Optimization Algorithm (HPOPTA) which is discussed in section 3.1 [1].

[1] The benchmark data-set and the source code for the HPOPTA and balanced workload distribu-

Two types of FFT FPMs are studied: real and smooth. The real case exhibits a realistic behaviour of FFT routine where the speed difference between two consecutive samples in the FPM is abrupt (i.e. discontinuous curve). While for the smooth case the speed difference between two consecutive samples in FPM is smooth (continuous).

Problem size is measured in terms of number of multiply-accumulate operations (MAC) needed. In case of FFT, the number of MAC operations can be found using the following formula:

$$MAC = N \times \log_2(N) \tag{5.1}$$

where N is the size of the FFT.

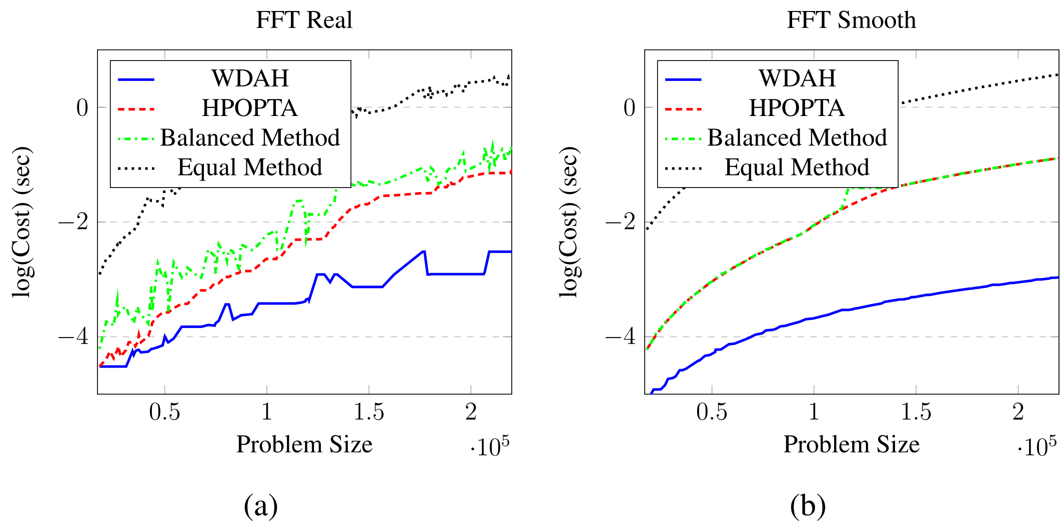


Figure 5.5: Cost of Solution (seconds) in logarithmic scale vs. problem size

The cost of each generated solution by each methodology is reported in logarithmic scale. Results in figure 5.5 illustrates that our proposed algorithm WDAH is capable of creating a solution to our original mapping problem providing the smallest execution time in comparison to other methods (the lowest the better). Equal distribution method has the worst performance while the solutions generated by balanced, HPOPTA and proposed WDAH are almost the same small size for problems and as

tion are associated with work in 3.1, the source code and the data can be accessed here: git@csgitlab.ucd.ie:HKhaleghzadeh/hpopt.git

problem size increases, the performance of the methods deviate from each other.

5.2.2 Speed Up

This test aims to measure the speed up as the number of computing elements changes, conducted on real device. The formula used to find the speed up is

$$SpeedUp = \frac{T_{parallel}}{T_{Sequential}} \tag{5.2}$$

where $T_{parallel}$ and $T_{Sequential}$ are parallel and sequential execution times of an application, respectively. In our case, the sequential execution time is chosen as the running time of the method on general purpose processor (GPP), of our 66AK2H12 board, where we have chosen ARM processor to be the GPP.

This test consists of four cases. In each case, the number of involved computing elements is $4ARMCores + xDSPCores$, where x is varied in each case. Initially, x is chosen as 2 and is incremented by 2 for each case. Each case is repeated 100 times and the average parallel computational time is reported.

Three different applications are chosen as candidates for our evaluation. These are: i) Matrix Multiplication, ii) 2D Fast Fourier Transformation (FFT) and iii) 2D Convolution or (FIR). Size of the problem is fixed throughout the test.

For Matrix Multiplication, two matrices of different sizes are considered and $A \times B$ is computed, where A is the input matrix to be partitioned and B is the shared matrix, i.e. accessible by all computing elements. Size of B is fixed and chosen to be 256×256 and size of matrix A is chosen to be 1024×256. The data type for the matrices is $FLOAT$ (each element being 4 bytes).

Fast Fourier Transformation is applied on 2D data. 1D FFT is applied on each row first, and 1D FFT is applied on each column next. 2D input data is to be partitioned among computing elements. The size of the data is chosen as 512×512. The data type is $Complex$, i.e. each data is 8 bytes (4 bytes for real and 4 bytes for imaginary parts).

Finally, in the convolution application we applied a finite impulse response filter on 2D input data, which is to be partitioned among computing elements and Total size

of data is chosen as 512×512 and data type is now $FLOAT$ (each data is 4 bytes).

Throughout our test, NEON, Ne10 and DSPLIB libraries are deployed for performing the computations. For our proposal algorithm the backtrack value is set to 100 throughout this test.

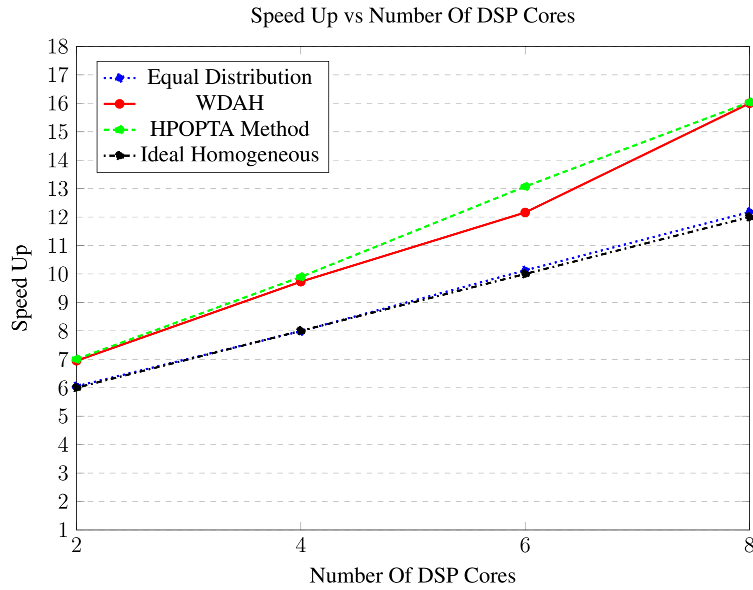


Figure 5.6: Matrix Multiplication Speed Up ($4ARMCores + xDSPCores$)

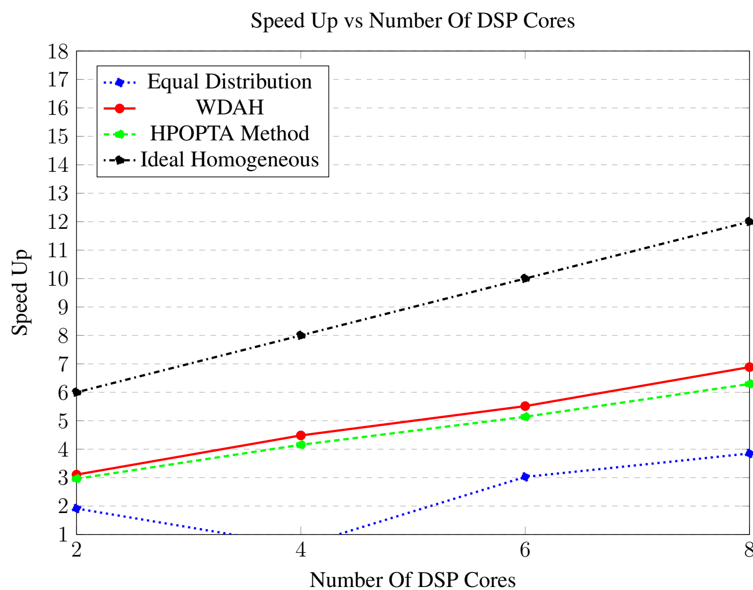


Figure 5.7: FFT Speed Up ($4ARMCores + xDSPCores$)

The results demonstrate that our proposed method can perform as good as the HPOPTA

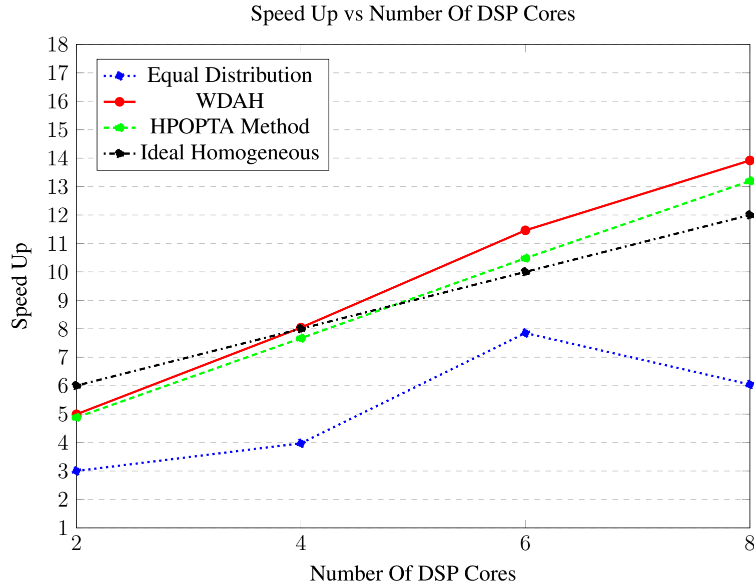


Figure 5.8: FIR Speed Up ($4ARMCores + xDSPCores$)

method and both performs much better than equal distribution. The significant observation here is that our proposed method and HPOPTA can exploit and utilize the heterogeneity of computing elements more efficiently. We observe figures 5.6, 5.7 and 5.8, where black line is the ideal speed up case if all computing elements were similar (homogeneous system). Noting that our platform is a heterogeneous one, we expect to have a better speed up than the black line but we observe that the speed up of our proposal and HPOPTA are higher than the ideal speed up for matrix multiplication and convolution while equal distribution is way lower than the ideal case, which indicates that equal distribution is not an efficient approach in case of heterogeneous devices. Unlike Matrix Multiplication and FIR, the speed up for the FFT is worse than the ideal speed up for all methods, which can be due to, for instance memory access operations, dominating the FFT performance, which may impose a bottleneck for speed up.

We observe that the difference between WDAH and HPOPTA results is small unlike the simulation results where it was relatively large. There may be two reasons for this: i) computing elements in simulation tests are different than the devices used in our real computing tests and ii) simulation environment lacks the hardware bottlenecks and overheads which probably exist in the real devices used.

5.2.3 Backtrack and Cost Optimality

Backtrack parameter allows our algorithm to investigate multiple branches of the tree, the higher the value of the Backtrack, the higher the possibility to hit more solutions. However, using a large value of Backtrack degrades the performance of the algorithm by increasing its running time considerable. Hence, finding a proper trade-off point is essential. Therefore, the proposed algorithm (WDAH-Branch-and-Cut) is designed such that the search process is supervised and thus it is expected that even with a unity backtrack value (WDAH-Heuristic) the generated solution should be a near optimum one.

This test aims to study the effect of changing the value of the Backtrack parameter on the cost of the generated solution and it is a simulated test. The test involves four different cases $\{WDAH1, WDAH10, WDAH100, WDAH1000\}$
where $Backtrack = \{1, 10, 100, 1000\}$.

Four different applications are studied: FFT real, FFT smooth, Semi-Random and Full-Random. The Semi-random case is randomly generated FPMs but the shape of the FPM satisfies the conditions stated in 3.4. The Full-Random case is a randomly generated FPMs where the behaviour is unpredictable and unconstrained.

HPOPTA method is used as a reference to study the quality of the generated solution. Figure 5.9 illustrates the obtained results.

Obtained results in figure 5.9 complies with our expectations and aims. For FFT real and FFT smooth cases, the cost of the generated solution by the WDAH algorithm is better than the HPOPTA. Moreover, increasing the backtrack value does not change the cost of the solution significantly, thus, (WDAH-Heuristic) which is equivalent to (WDAH1) is proven to be powerful enough to generate low cost solutions and increasing backtrack will not improve the quality remarkably. For the Semi-Random case, it is observed that the HPOPTA can generate better solutions at small problem sizes but not at large problem sizes. Finally, for the Full-random case, the generated solution by the proposed method is not as good as the HPOPTA method and this is expected since workload packages generation will not be efficient in this case.

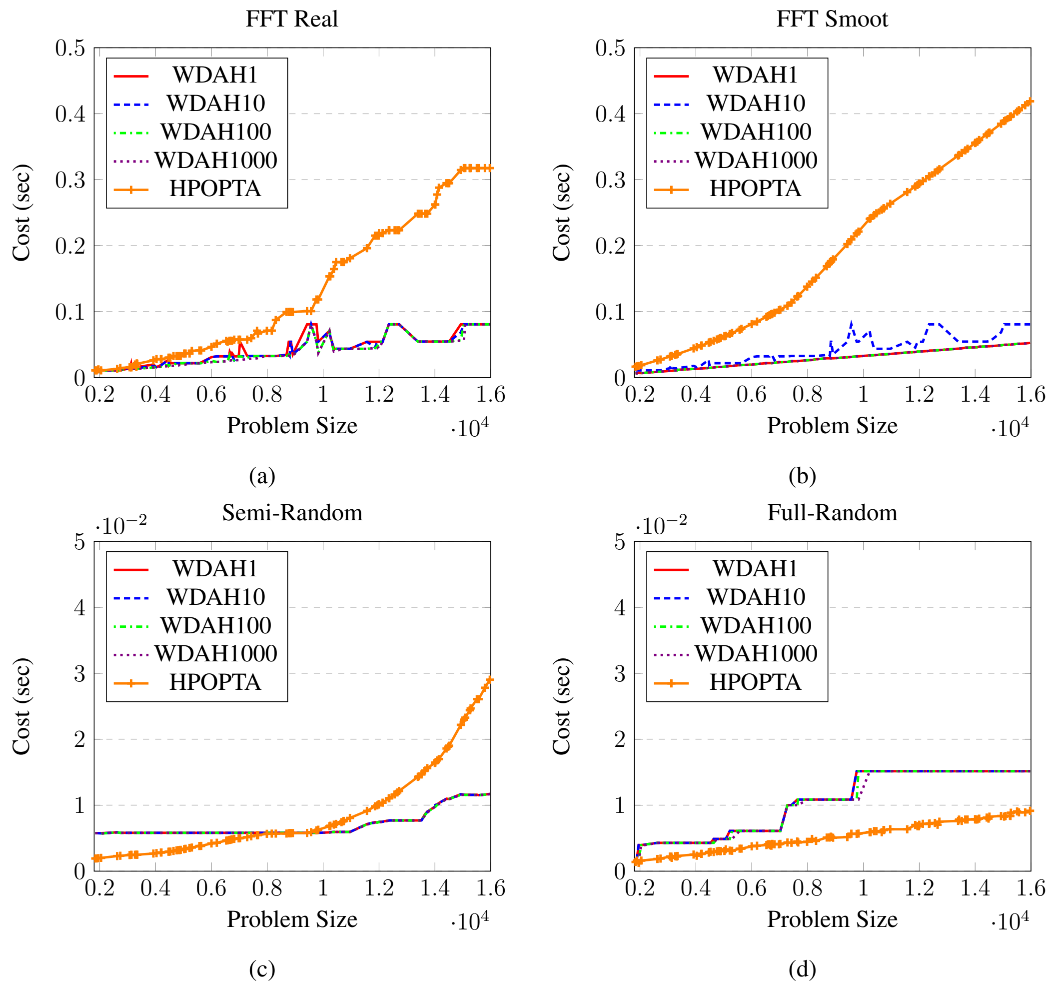
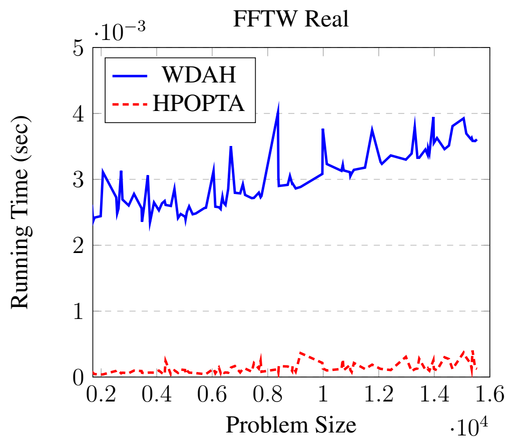


Figure 5.9: Effect of Backtrack value on solution quality

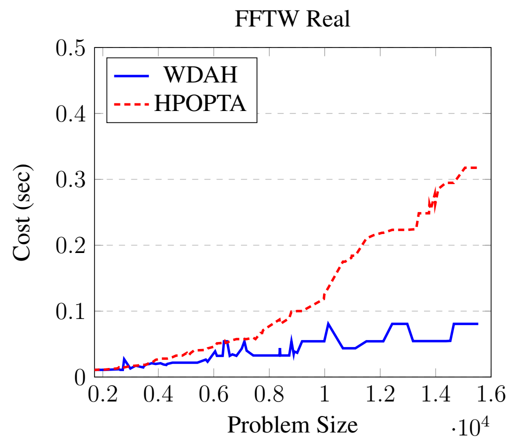
5.2.4 Running Time

The efficiency of the algorithm is not only measured in terms of created solution, but also in terms of its own running time. It is important for algorithms, specially real time ones, to find a solution in a reasonable amount of time. For instance, finding an optimum solution for our problem with an exhaustive search causes the complexity

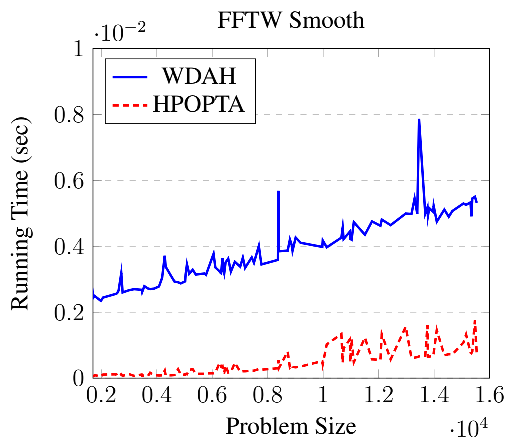
of the algorithm to grow exponentially as $O(p^n)$, where p is the number of computing elements and n is the number of possible partitions or workload packages.This test is conducted in order to observe the running time of the algorithm as problem size varies. Four different cases are studied: FFT real, FFT smooth, Semi-Random and Full-Random. For FFT real and smooth the number of computing elements is set to 3 and for the Random cases it is set to 4. Backtrack value of WDAH-Branch-and-Cut is set to 100. The running time of the algorithm is reported. The performance of the algorithm is compared with the performance of the HPOPTA. This test is conducted on AMD Ryzen 7 4800H processor running Ubuntu 20.04.



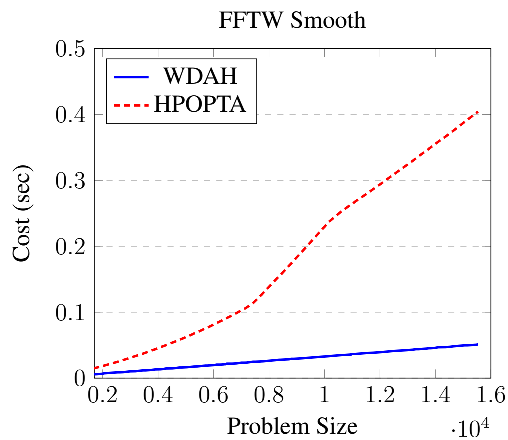
(a)



(b)



(c)



(d)

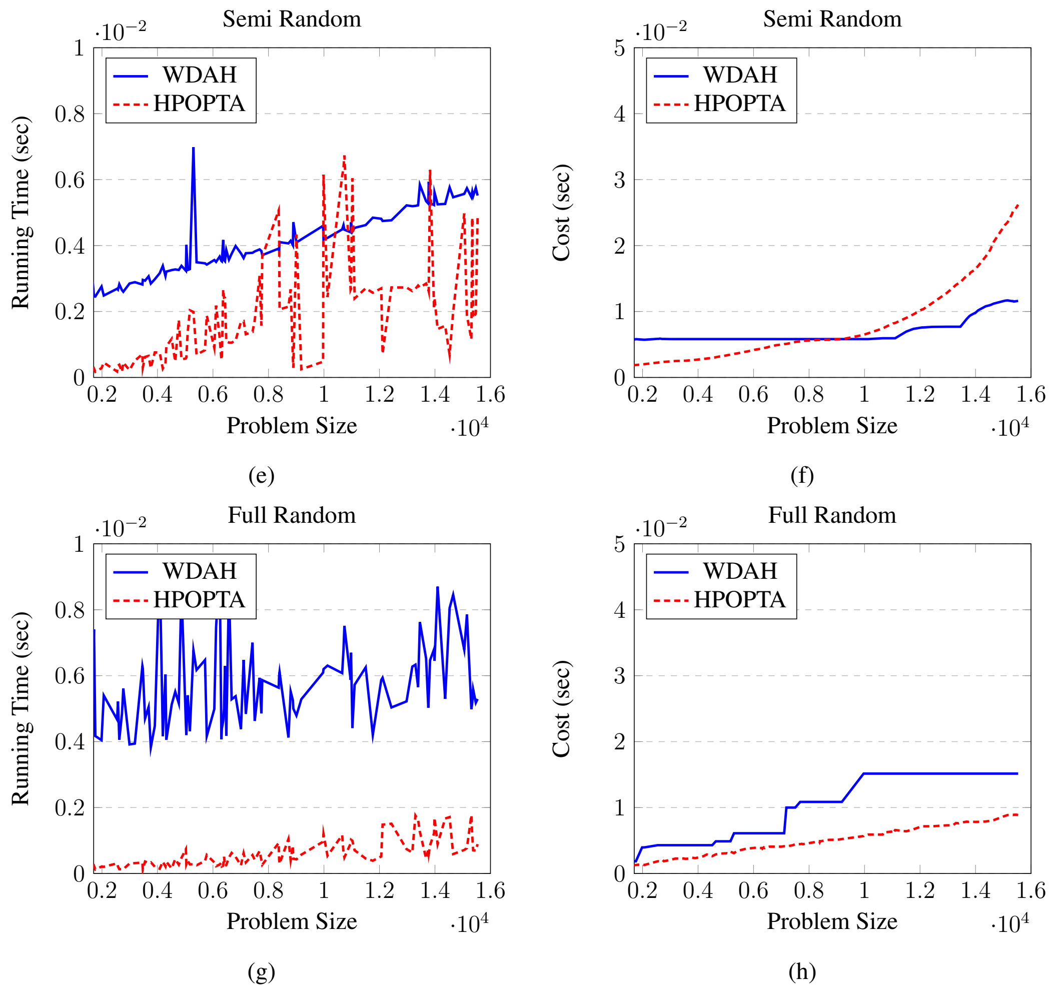


Figure 5.10: Algorithm Running Time and Solutions Cost

Figure 5.10 shows that the running time of WDAH-Branch-and-Cut algorithm is increasing linearly when problem size increases while number of computing elements is fixed, except for Full-Random case where running time is unforeseeable. Furthermore, from a comparison point of view, it is observed that the HPOPTA has running time lower than our proposal. Figure 5.10 illustrates running time and solutions cost for each case. The generated solutions by our proposed method has lower cost than HPOPTA solution. Since the difference in running times of both methods is in the order of microseconds, the difference is not humanly observable and almost negligible, except possibly for critical real time problems.

5.2.5 Utilization of Computing Elements

It is important also to study the performance of the algorithm in terms of its ability to utilize the computing elements efficiently. Avoiding overloaded and under-loaded computing elements is important from a balancing point of view. Throughout this test, the utilization of the computing elements is observed for five different problem sizes, conducted as a simulation test. There are three different computing elements CPU, GPU, and Xeon Phi. The utilization is evaluated using the following formula:

$$U_i = \frac{T_{pi}}{T_{comp}} \tag{5.3}$$

where U_i is the utilization of ith computing element and T_{pi} and T_{comp} are processing time of the ith computing element (non Idle time) and Total computational time of the application respectively.

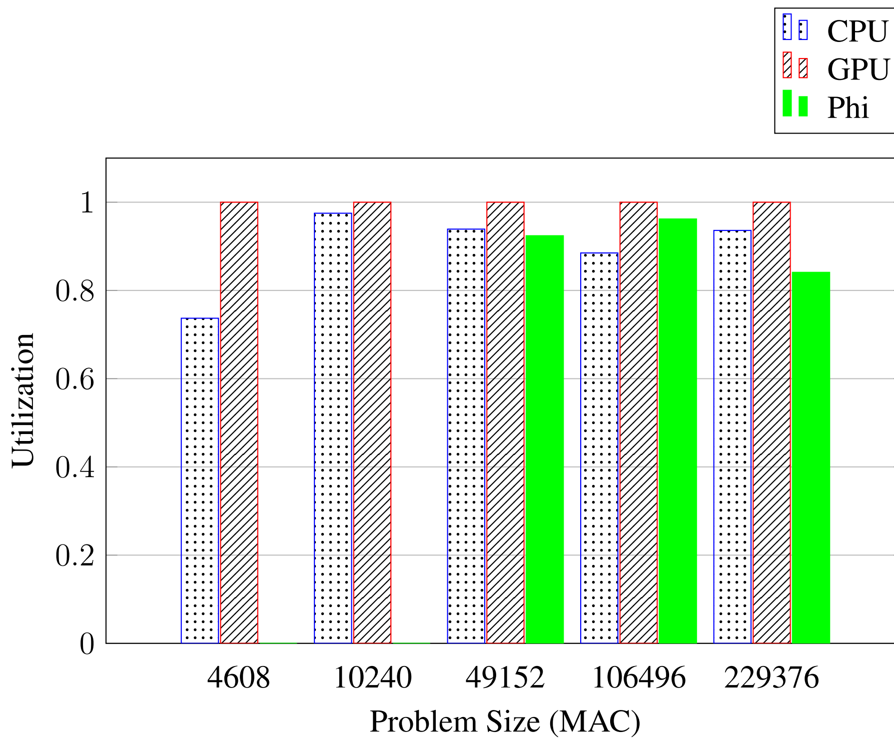


Figure 5.11: Utilization of computing elements in WDAH method

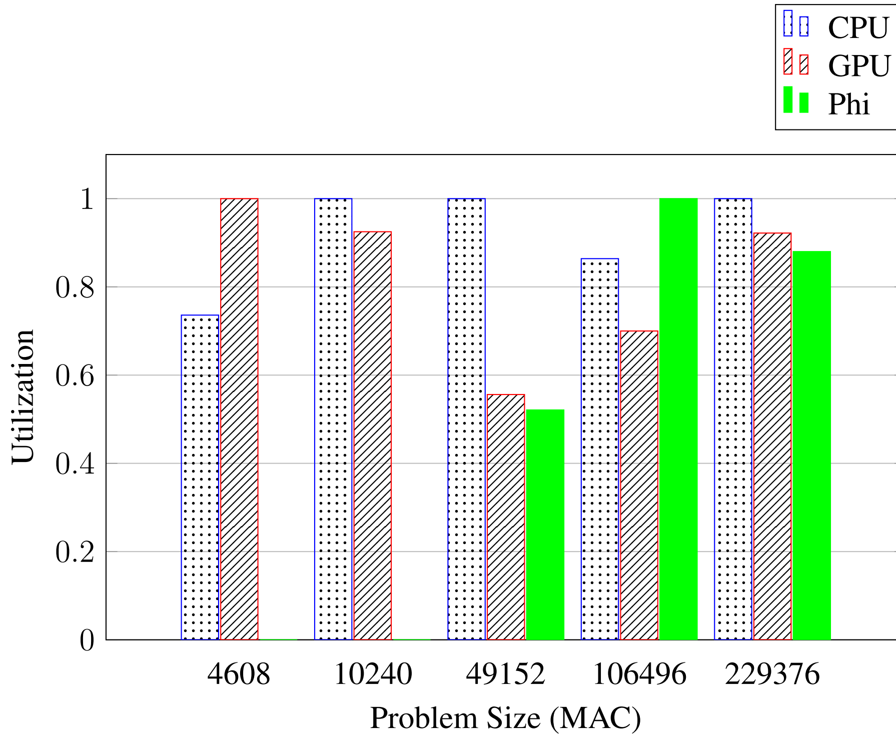


Figure 5.12: Utilization of computing elements in HPOPTA Method.

Figures 5.11 and 5.12 show the utilization levels of computing elements both for our proposal WDAH and HPOPTA methods. First observation is that WDAH efficiently balances the workload on computing elements. There are no overloaded or under-loaded cases. The cases where the problem sizes are 4608 and 10240 MAC, Phi processor has a utilization factor of $U = 0$, which indicates an under-loaded case, however, since cases are relatively small problems where using only two computing elements is enough. Comparing WDAH and HPOPTA methods, WDAH is observed to balance the computing elements more efficiently than HPOPTA.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

The present chapter discusses the merit of the proposed method and presents some potential improvement directions followed by final remarks.

6.1 Conclusion

There are various approaches and proposed methodologies employed to solve the workload distribution problem of data parallel applications. With all available techniques, it is crucial to highlight the significance of our proposal.

First, in comparison to other methods, our proposal uses a relatively small amount of prior information (FPM) in creating a feasible minimum solution. Minimum amount of prior information (FPM) brings the advantage of utilizing small memory to store FPMs and better running time.

Second, our proposal is capable of providing a solution for large size problems even without the FPM knowledge about large size partitions. For most methods that use the concept of FPM, in order to distribute the workload of a certain problem, FPM has to cover all possible partitions that spans the whole size of the targeted problem. For example, if the problem size is N, FPM should cover all possible partitions between size λ (minimal workload or size of a partition) and N, which is not the case for our proposal. This is due to the property that in batching multiple kernels to a computing element, it is sufficient for FPM to cover only partitions within the range defined in section 4.2 for our algorithm to find a solution.

Third, the priority assignment to computing elements helps in finding good solutions

because of its positive impact on guiding the search to follow paths that have the potential to yield minimum solutions, hence, avoiding undesirable paths.

As a future work, the following improvements might be possible as follows: i) Real time deployment (switching from offline to online assignment) techniques ii) use of machine learning in acquiring prior information (constructing FPMs).

Offline to Online:
The current version of our algorithm is to be used offline or statically. In other words, partitioning and assignment step is applied at compile time. This is for sure not the only approach since in a variety of applications the workload changes over time, hence dynamical and online deployment may be required.

There are two parameters that need to be optimized for the algorithm to be real time or dynamically applicable: i) run time and ii) memory consumption of the algorithm. The requirements are to have a small overhead and memory usage. While for offline case this is not a critical concern, it is vital for the online deployment not to have a running time that is the order of minutes or even seconds.

In terms of memory usage, depicting the architecture in figure 5.1, the algorithm needs two different memory space, one to store prior information (FPMs) for the application and another one for storing the obtained solutions (assignment) or the lookup table in figure 5.1. For online case, there is no need for a look up table since the solutions will be assigned on the fly. Moreover, the profile information can be stored in cache memory, where content can be updated during run time using a predefined caching policy. Cache policies are not discussed in this work but the following scenario illustrates how it can be deployed.

We assume the architecture in figure 5.1 and consider that profiling cache is clear initially. When an application with a certain workload needs to be processed, there is no prior information about the application, hence the dispatcher uses any policy to distribute the workload. The dispatcher is given the freedom to choose any applicable policy to distribute the workload. During this process, a new back-end entity, called as $Recorder$ starts to build up information about the running application by recording its run-time behaviour on computing elements. After multiple iterations, $Recorder$

would have collected enough information about the application to create a functional performance model (FPM). How fast the FPM can be created is dependent on how frequent the application is invoked. Once there is a valid FPM, each time the application hits again, the system uses our proposed algorithm in order to distribute the workload.

The above scenario contributes in three aspects: i) memory usage, ii) dynamic workload size and iii) real time behaviour. In such a scenario only a fixed region of memory is needed to store the FPM, which works as cache, thus its content can be changed over time. Moreover, lookup table needed to store the yielded solutions is not used anymore since the proposed method will be triggered during run time obtain a solution instantly. Finally, any additional processes such as $Recorder$ will work as daemon processes, whose job is only to collect information about the running application, which will not interrupt or effect the application negatively.

Acquiring Prior Information (Building FPMs):
Previous discussion assumes that FPM can be obtained offline via benchmarking of the application on each computing element. However, as an alternative approach, FPM construction can be done using machine learning and deep learning techniques. For instance, a neural network can be created and trained in order to simulate the behaviour of an application.

During the training phase, an application with a given workload will be fed as an input to the network, the network learns the features of the computing elements and the behaviour of the application throughout the process of feeding back the error between the actual output and expected one then updating the network coefficients accordingly. Each output neuron of the network resembles a computing element and the output value is the expected running time of the application on the resembled computing element.

After the learning process, the network can be used during run time to generate FPM for new applications without the need to consume any memory to store the FPM. Although this solution sounds tedious because it requires a large amount of applications during the training phase in order to improve the accuracy of the neural network, nowadays this is not a critical issue as there are millions of applications dis-

patched daily to servers and computing edges, hence the neural network can be trained quickly.

To conclude, the present thesis work proposes a new method for distributing a given total workload on multi-core heterogeneous devices. Our work has focused initially on applications that can be modeled as single program multiple data (SPMD) form. Although the proposed approach is shown to be promising, possible drawbacks also exist.

Since we are making all our selections as max entries from ordered sets, our solution is not guaranteed to be optimal but rather heuristic. The mathematical model we developed can be solved exactly either by i) an exhaustive search (for small problems) or ii) a suitably designed branch and bound algorithm combining appropriate upper bounding and lower bounding mechanisms and branching strategy (for moderate size problems). For the original problem, the complexity of finding an optimum solution is in the order of $O(n^p)$, where n is the number of workload types and p is the number of processors, hence using heuristics is reasonable for large problems.

The parallel architecture assumed in this work is a shared memory model. Since the communication between computing elements is not considered in the mathematical model of the algorithm, our proposed method may not be efficient in a distributed memory architecture, in which communication between computing elements shows a significant impact on running time. Therefore, for the algorithm to be more generic, it needs to be modified to be adapted also to distributed memory architectures.

Finally, although this work is focused on SPMD type applications, there are other possible application models also in which the running kernels or tasks are not of the same type. With the current version of our algorithm, it is not possible to offload such applications, therefore, further modifications need to be applied in order to make our proposal more generic and portable to other applications.

REFERENCES

Ahmed, U., Lin, J. C.-W., Srivastava, G., & Aleem, M. (2021). A load balance multi-scheduling model for opencl kernel tasks in an integrated cluster. *Soft Comput.*, *25*(1), 407–420. https://doi.org/10.1007/s00500-020-05152-8

Al-Hashimi, H. T., & Basuhail, A. A. (2021). A proposed data partitioning approach on heterogeneous hpc platforms: Data locality perspective. *IEEE Access*, *9*, 81432–81442. https://doi.org/10.1109/ACCESS.2021.3085774

Anthony, R. J. (2016). Chapter 5 - the architecture view. *Systems programming* (pp. 277–382). Morgan Kaufmann. https://doi.org/https://doi.org/10.1016/B978-0-12-800729-7.00005-4

Augonnet, C., Thibault, S., Namyst, R., & Wacrenier, P.-A. (2009). Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In H. Sips, D. Epema, & H.-X. Lin (Eds.), *Euro-par 2009 parallel processing* (pp. 863–874). Springer Berlin Heidelberg.

Brant, P. (2018). The importance of hardware raising all boats. https://education.emc.com/content/dam/dell-emc/documents/en-us/2018KS_Brant-The_Importance_of_Hardware_Raising_All_Boats.pdf

Calciu, I., Dice, D., Harris, T., Herlihy, M., Kogan, A., Marathe, V., & Moir, M. (2013). Message passing or shared memory: Evaluating the delegation abstraction for multicores. In R. Baldoni, N. Nisse, & M. van Steen (Eds.), *Principles of distributed systems* (pp. 83–97). Springer International Publishing.

Cardoso, J. M., Coutinho, J. G. F., & Diniz, P. C. (2017). Chapter 7 - targeting heterogeneous computing platforms. In J. M. Cardoso, J. G. F. Coutinho, & P. C. Diniz (Eds.), *Embedded computing for high performance* (pp. 227–254). Morgan Kaufmann. https://doi.org/https://doi.org/10.1016/B978-0-12-804189-5.00007-7

Cierniak, M., Zaki, M. J., & Li, W. (1997). Compile-time scheduling algorithms for a heterogeneous network of workstations. *The Computer Journal*, *40*(6), 356–372. https://doi.org/10.1093/comjnl/40.6.356

Deb, K., & Jain, H. (2014). An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, *18*(4), 577–601. https://doi.org/10.1109/TEVC.2013.2281535

Garey, M. R., & Johnson, D. S. (1990). Computers and intractability; a guide to the theory of np-completeness. W. H. Freeman & Co.

Glover, F., & Laguna, M. (1998). Tabu search. In D.-Z. Du & P. M. Pardalos (Eds.), *Handbook of combinatorial optimization: Volume1–3* (pp. 2093–2229). Springer US. https://doi.org/10.1007/978-1-4613-0303-9_33

Gupta, R., & De Micheli, G. (1993). Hardware-software cosynthesis for digital systems. *IEEE Design & Test of Computers*, *10*(3), 29–41. https://doi.org/10.1109/54.232470

Hossain, M. A., & Tokhi, M. O. (2002). Inter-processor and inter-process communication in realtime multi-process computing [15th IFAC World Congress]. *IFAC Proceedings Volumes*, *35*(1), 337–342. https://doi.org/https://doi.org/10.3182/20020721-6-ES-1901.00962

Kaeli, D., Mistry, P., Schaa, D., & Zhang, D. P. (2015). Chapter 1 - introduction. In D. Kaeli, P. Mistry, D. Schaa, & D. P. Zhang (Eds.), *Heterogeneous computing with opencl 2.0* (pp. 1–14). Morgan Kaufmann. https://doi.org/https://doi.org/10.1016/B978-0-12-801414-1.00001-6

Kalinov, A., & Lastovetsky, A. (2001). Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers. *Journal of Parallel and Distributed Computing*, *61*, 520–535. https://doi.org/10.1006/jpdc.2000.1686

Khaleghzadeh, H., Manumachu, R. R., & Lastovetsky, A. (2018). A novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous hpc platforms. *IEEE Trans. on Parallel and Distributed Systems*, *29*(10), 2176–2190. https://doi.org/10.1109/TPDS.2018.2827055

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*(4598), 671–680. https://doi.org/10.1126/science.220.4598.671

Lastovetsky, A., & Manumachu, R. (2007). Data partitioning with a functional performance model of heterogeneous processors. *IJHPCA*, *21*, 76–90. https://doi.org/10.1177/1094342006074864

Mittal, S., & Vetter, J. S. (2015). A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, *47*(4), 1–35. https://doi.org/10.1145/2788396

Nickolls, J., & Dally, W. J. (2010). The gpu computing era. *IEEE Micro*, *30*(2), 56–69. https://doi.org/10.1109/MM.2010.41

Pérez, B., Stafford, E., Bosque, J., & Beivide, R. (2021). Sigmoid: An auto-tuned load balancing algorithm for heterogeneous systems. *Journal of Parallel and Distributed Computing*, *157*, 30–42. https://doi.org/https://doi.org/10.1016/j.jpdc.2021.06.003

Puchinger, J., & Raidl, G. R. (2005). Combining metaheuristics and exact algorithms in combinatorial optimization: A survey and classification. In J. Mira & J. R. Álvarez (Eds.), *Artificial intelligence and knowledge engineering applications: A bioinspired approach* (pp. 41–53). Springer Berlin Heidelberg.

Rose, J., El Gamal, A., & Sangiovanni-Vincentelli, A. (1993). Architecture of field-programmable gate arrays. *Proceedings of the IEEE*, *81*(7), 1013–1029. https://doi.org/10.1109/5.231340

Texas Instruments. (2017). 66ak2hxx multicore dsp arm keystone ii system-on-chip (soc). https://www.ti.com/lit/ds/symlink/66ak2h12.pdf?ts=1638218914598&ref_url=https://google.com

Tse, A. H., Thomas, D. B., Tsoi, K., & Luk, W. (2010). Dynamic scheduling monte-carlo framework for multi-accelerator heterogeneous clusters. *2010 International Conference on Field-Programmable Technology*, 233–240. https://doi.org/10.1109/FPT.2010.5681495

Tsoi, K. H., & Luk, W. (2010). Axel: A heterogeneous cluster with fpgas and gpus. *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 115–124. https://doi.org/10.1145/1723112.1723134

Zhang, P. (2010). Chapter 17 - distributed operating systems. In P. Zhang (Ed.), *Advanced industrial control technology* (pp. 685–732). William Andrew Publishing. https://doi.org/https://doi.org/10.1016/B978-1-4377-7807-6.10017-8

Appendix A

OBJECT ORIENTED IMPLEMENTATION

The proposed algorithm is implemented in an object oriented programming model. This section discusses the proposed framework implementation, the framework portrays a computing architecture. There are mainly four classes defined: (i) $SolutionNode$, (ii) $SolutionEntity$, (iii) $Processor$ and (iv) $Dispatcher$. The following sections discusses the details of these classes.

A.1 Solution Node

$SolutionNode$ is the basic entity constructing the solution list, each node indicates a creation of a task with a given workload and assignment to a computing elements. Below snippet of code represents the implementation of the $SolutionNode$ class.

```
1  class SolutionNode:
2    def init(self, w, dp=0, prevNode=NULL, nxtNode=NULL):
3        self.TaskWorkload = w;
4        self.destProc = dp;
5        self.prevNode = prev;
6        self.nxtNode = nxt;
7        self.CostStamp = 0;
8
9    def GetPrevNode(self):
10       return self.prevNode;
11
12     def GetNxtNode(self):
13       return self.nxtNode;
14
15     def GetWorkload(self):
```

```
16        return self.TaskWorkload;
17
18      def GetDestProc(self):
19        return self.destProc;
20
21      def SetCostStamp(self,c):
22        self.CostStamp = c;
23
24      def GetCostStamp(self):
25        return self.CostStamp;
26
27      def SetNxtNode(self,nxt):
28        self.nxtNode = nxt;
29
30
31      def SetPrevNode(self,prev):
32        self.prevNode = prev;
```

Listing A.1: Solution Node

After each creation of an object of this type, the function $init$ is called. Each node has a pointers to next and previous nodes, w and dp parameters indicates the value of the assigned workload size and destination processing element. The rest of the member functions are used to read or write the contents of the node. Member functions $GetPrevNode$ and $GetNxtNode$ are used to get the previous and next nodes respectively. $GetWorkload$ returns the assigned workload and $GetDestProc$ returns the designated computing element. Functions $SetCostStamp$ and $GetCostStamp$ are used to set and get the currect cost of the solution at this specific node, this works as a stamp for the cost. Finally Functions $SetNxtNode$ and $SetPrevNode$ are used to update previous and next nodes, these functions are useful when the solution list gets modified.

A.2 Solution Entity

$SolutionEntity$ is the class resembling the solution list, it is made up of $SolutionNode$ objects. This class is used to store the current mapping or distribution scheme in the

form of a list. Each time a path within the search tree is extended its node is appended to the list. The implementation of this class is shown below:

```
class SolutionEntity:
  def init(self, peList):
        self.TopPtr = NULL;
      self.rootPtr = NULL;
      self.numNodes = 0;
      self.CurrentCost = 0;
      self.pePtr = peList;






  def InsertNode(self,w,dp):
      SNode = SolutionNode(w,dp,TopPtr,NULL);
      if numNodes == 0:
        self.rootPtr = SNode;
      else:
        self.TopPtr.SetNxtNode(SNode);

        self.TopPtr = SNode;
      self.numNodes++;

      self.pePtr[dp].UpdateCompTime(w);
      self.pePtr[dp].UpdatePriority(w);

      if self.CurrentCost < self.pePtr[dp].GetComptTime():
        self.CurrentCost = self.pePtr[dp].GetComptTime();

      self.TopPtr.SetCostStamp(CurrentCost);

    def RemoveTopNode(self):
      tmpPtr = self.TopPtr.GetPrevNode();
      self.pePtr[self.TopPtr.GetDestProc()].RetCompTime(self.TopPtr.
    GetWorkload());
      self.pePtr[self.TopPtr.GetDestProc()].RetPriority(self.TopPtr.
    GetWorkload());
      self.TopPtr = tmpPtr;
```

```
37        self.numNodes-- ;
38        if self.TopPtr == NULL:
39          self.CurrentCost = 0;
40        else:
41          self.CurrentCost = self.TopPtr.GetCostStamp();
42
43      def GetTopNode(self):
44        return self.TopPtr;
45
46      def GetRootNode(self):
47        return self.rootPtr;
48
49      def GetCost(self,c):
50        self.CurrentCost = c;
```

Listing A.2: Solution Entity

In $init$, variables $TopPtr$ and $rootPtr$ are pointers to last appended node to the list and first node of the list subsequently. $numNodes$ indicates number of current nodes in the list and $CurrentCost$ indicates the current cost. Finally $pePtr$ is a pointer to the list of computing elements.

When a node is inserted to the list, the function $InsertNode$ is called. It creates a $SolutionNode$ object with the assigned workload, destination computing element and previous node input arguments, then the function checks in case this is the first node appended to the list, if this is the case the $rootPtr$ is updated to point at this node, otherwise, the $TopPtr$ node updates its next node to point at the newly added node. After updating the $TopPtr$ and incrementing number of nodes, the function updates the current computational time of the destination computing element and its priority. Before returning, the function updates the current cost of the solution and sets the cost stamp of the $TopPtr$ node.

$RemoveTopNode$ is used in order to remove the top node of the list. After removing the top node, the computational time and priority of the computing element associated with the removed node are updated to retrieve their previous values. Moreover, to retrieve the previous cost, the function first checks whether there is a node in the list or not, in case the list is empty, the cost is assigned to 0, otherwise, the cost is set to

cost stamp of the current top node.

A.3 Processor

Objects of this class resembles computing elements. The implementation of this class is as follows:

```
1
2  class Processor:
3    def init(self, pe, fpm):
4        self.id = pe;
5        self.priority = 0;
6        self.compTime = 0;
7        self.wPool = fmp;
8        Sort(self.wPool);
9
10   def GetPriority(self):
11       return self.priority;
12
13     def UpdatePriority(self,w):
14       for i  in self.wPool:
15         if self.wPool[i].size == w
16           break;
17       self.priority −= self.wPool[i].size;
18
19     def GetComptTime(self):
20       return self.compTime;
21
22     def UpdateCompTime(self, w):
23     for i  in self.wPool:
24         if self.wPool[i].size == w
25           break;
26       self.compTime += self.wPool[i].time;
27
28     def SetPriority(self,pr):
29       self.priority = pr;
30
31     def AccessWTable(self):
32       return self.wPool;
```

```
    def RetPriority(self,w):
      for i  in self.wPool:
        if self.wPool[i].size == w
          break;
      self.priority += self.wPool[i].size;

  def RetCompTime(self, w):
    for i  in self.wPool:
        if self.wPool[i].size == w
          break;
      self.compTime -= self.wPool[i].time;

  def GetMaxSpeed(self):
      return self.wPool[0].speed;
```

Listing A.3: Processor Class

Upon object initialization, the object assigns its ID, initialize the priority and computational time to 0, gets the pool of workload packages $wPool$, each workload package is a data structure having three attributes: (i) $size$, (ii) $time$ and (iii) $speed$ where $size$ is the size of the workload, $time$ and $speed$ are the processing time and speed of this workload on the created computing element. $Sort$ function is called so that the workload packages will be sorted in descending order according to their speed.

$GetPriority$ and $SetPriority$ are functions used to get and update the priority of the computing element, while $AccessWTable$ is used to get the set of workload packages associated with the it. As discussed in the previous section, functions $UpdatePriority$, $UpdateCompTime$, $RetPriority$ and $RetCompTime$ are used by the $SolutionEntity$ object while appending or removing nodes from the list in order to preserve consistency.

A.4 Dispatcher

$Dispatcher$ object is the main orchestrator, it manages the all previously discussed objects and their interactions, the proposed tree search algorithm is embedded within the $Dispatcher$ class. The implementation of the dispatcher is depicted below:

```
1  class Dispatcher:
2    def init(self, ws, fpmList):
3        self.N = ws;
4          self.SetUpProc(dir);
5          self.sol = SolutionEntity(self.pePtr);
6          self.fsol = SolutionEntity(self.pePtr);
7
8    def SetUpProc(self,fpmList):
9        id = 0;
10       for fpm in self.fpmList:
11           self.pePtr = Processor(id,fpm);
12
13       for pe in self.pePtr:
14           pe.SetPriority(self.EvPriority(pe))
15
16     def EvPriority(self,pe):
17         Stot = 0;
18       for p in self.pePtr:
19           Stot += p.GetMaxSpeed()
20
21        return (pe.GetMaxSpeed()/Stot) self.N;
22
23     def Start(self):
24         self.BackTrack = 100;
25         self.GlobalCost = 100;
26         remSize = N;
27         self.ProposedScheduler(self.remSize);
28
29     def ProposedScheduler(remSize)
30         if(remSize   0):
31             tmppr = 0;
32         for pe in self.pePtr:
33           if tmpprty < pe.GetPriority():
34                     dp = pe;
35                     tmppr = pe.GetPriority();
36
37             wPool = dp.AccessWTable()
38
39             for w in wPool:
40                 if w.size <= remSize:
```

```
                self.sol.InsertNode(w.size,dp)
                if self.sol.GetCost() > self.GlobalCost:
                    self.sol.RemoveTopNode();
          continue;

      if self.BackTrack > 0:
          self.ProposedScheduler(remSize w.size);
          self.BackTrack-- ;

      self.sol.RemoveTopNode();

else:
    self.GlobalCost = self.sol.GetCost();
    self.fsol = self.sol;
```

Listing A.4: Dispatcher Class

When a dispatcher object is created, $init$ is called to initialize the parameters: N indicating the total workload size, list of computing elements $pePtr$ after calling $SetUpProc$ and empty solution lists sol and $fsol$, where sol is the current solution and $fsol$ is the final solution.

$SetUpProc$ function creates a list of computing elements using the $fpmList$, each computing element gets a single item from list $fpmList$ which represents the FPM associated with the created processing element. Afterward, each processing element is assigned a priority value using the function $EvPriority$.

In order to start the dispatcher, function $start$ is called. This function sets a backtrack and global cost to some large value and the remaining workload size to the total workload size. Then it calls a recursive function $ProposedScheduler$ which implements the proposed tree search algorithm. Indeed the structure of the $ProposedScheduler$ is similar to algorithm 2.

Creating only a dispatcher object is enough to starts the algorithm, the dispatcher itself handles the creation of other required objects and then the workload offloading process can be started. The dispatcher requires only as an input the size of the total workload and functional performance models (FPMs).

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Alasmar, Mahmoud
Nationality: Palestine
Date and Place of Birth: 25/02/1996, United Arab Emirates
Marital Status: Single
Phone: -
Fax: -

EDUCATION

Degree	Institution	Year of Graduation
B.S.	Middle East Technical University	2019

PROFESSIONAL EXPERIENCE

Year	Place	Enrollment
2018-2021	Vidyotek	Embedded Software and Hardware Engineer
2019-2020	Thales	Embedded Software Engineer
2021-	Kentkart	Embedded Software Engineer

PUBLICATIONS

International Conference Publications

M. Alasmar and C. F. Bazlamaçcı, "Workload Distribution on Heterogeneous Platforms," 2021 International Conference on Computer, Information and Telecommunication Systems (CITS), 2021, pp. 1-5, doi: 10.1109/CITS52676.2021.9618353.