

UNIVERSIDADE FEDERAL FLUMINENSE

JOÃO GABRIEL FELIPE MACHADO GAZOLLA

**A framework for task distribution in multi-CPU and
multi-GPU Systems**

NITERÓI

2017

UNIVERSIDADE FEDERAL FLUMINENSE

JOÃO GABRIEL FELIPE MACHADO GAZOLLA

A framework for task distribution in multi-CPU and multi-GPU Systems

Thesis presented to the Computing Graduate Program of the Universidade Federal Fluminense in partial fulfillments of the requirement for the degree of Doctor of Science. Topic Area: Visual Computing.

Advisor:

Esteban Walter Gonzalez Clua

Co-Advisor:

Orlando Gomes Loques Filho

NITERÓI

2017

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e do Instituto de Computação da UFF

G291 Gazolla, João Gabriel Felipe Machado
 A Framework for task distribution in multi-CPU and
 multi-GPU systems. / João Gabriel Felipe Machado Gazolla. -
 Niterói, RJ : [s.n.], 2017.
 108 f.

 Tese (Doutorado em Computação) - Universidade Federal
 Fluminense, 2017.

 Orientadores: Esteban Walter Gonzalez Clua, Orlando
 Gomes Loques Filho.

 1. Computação paralela. 2. Escalonamento de tarefa. 3.
 Unidade de processamento gráfico. I. Título.


CDD 005.1

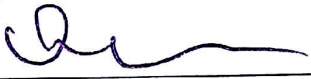
A framework for task distribution in multi-CPU and multi-GPU systems

João Gabriel Felipe Machado Gazolla


Thesis presented to the Computing Graduate
Program of the Universidade Federal Flumi-
nense in partial fulfillments of the require-
ment for the degree of Doctor of Science.
Topic Area: Visual Computing.

Approved in September 12th, 2017 by:

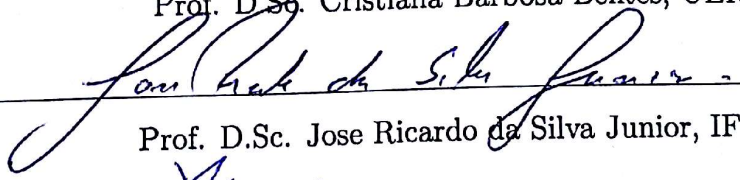

Prof. D.Sc. Esteban Walter Gonzalez Clua (Chair, Advisor), IC-UFF.

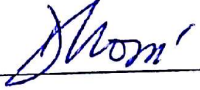

Prof. PhD Orlando Gomes Loques Filho (Co-Advisor), IC-UFF.


Prof. D.Sc. Lucia Maria de Assumpção Drummond , IC-UFF.


Prof. PhD Eugene Francis Vinod Rebello, IC-UFF.


Prof. D.Sc. Cristiana Barbosa Bentes, UERJ.


Prof. D.Sc. Jose Ricardo da Silva Junior, IFRJ.


Prof. PhD Daniel Mossé, University of Pittsburgh.

Stay hungry, Stay foolish. (Steve Jobs, 2005).

To my dear parents, Afrânio and Marisa...

Acknowledgements

Finally, after a few years, the moment to write this section of my thesis has come, and I have many people to acknowledge for all support. During this thesis project, several individuals have contributed one way or another, and to whom I owe my gratitude and will always be in my heart.

I wish to first thank my advisor Esteban Clua, for his encouragements, knowledge, patience, and trust. He not only advised me on my Ph.D. but on my mastering too. I learned many invaluable life lessons he taught me and by watching him on the day-by-day during a few years of my life.

I want to thank my co-advisor Orlando Loques for his continuous interest, help, patience, trust, support, guidance in all matters, feedback and for showing trust in me.

I want to thank my parents Afrânio and Marisa, without them none of this could have been possible, they gave up their dreams and wishes, to help me reach my own, making a dream come true. I would like to thank my brothers, Pedro and Patricia, my uncles Orlando and Cristina, Jorge and Marilene. Thank you all, each of you know that since many years ago, until now, you have somehow made this day a reality.

I want to thank my friends for helping to review this work: Cláudia Varassin, João Bentes, João Paulo, Luan Teylo and Pedro Gazolla.

Last but not least, I want to thank everyone at UFF, for all the support during all these years, all colleagues that participated in this journey during the classes, in conversations, counsels, and the everyday routine. Great professors who I had the chance to be their student, observe and admire their way of teaching and how they dedicate their life for their work: Aura, Anselmo, Bianca, Debora, Esteban, Murta, Orlando, Plastino, Simone and Vanessa.

Thanks to MediaLab for running the experiments on their CUDA machines.

I wish to thank UFF and Computing Institute staff for making everything working

fine and doing their jobs kindly and flawless. Finally, I want to thanks CAPES for the scholarship which provided vital financial support.

Finally, I want to thank again all my family for their love, support and encouragement they have given me throughout my Ph.D. and during all my life. I hope I have the wisdom to transmit all this support, patience and kindness you gave me to the people whom I have the chance to cross path on a daily basis.

Resumo

Desenvolver *software* para tirar proveito de recursos de *hardware* heterogêneos em cenários de computação alto desempenho é uma tarefa complexa e possui muitas variáveis associadas, criando porém uma vasta possibilidade e oportunidades para resolver problemas grandes que um único processador não seria capaz. Dependendo do mapeamento de tarefas para o *hardware* disponível, são possíveis diferentes tempos de execução. Este trabalho propõe e implementa um modelo inovador para otimizar o tempo geral de execução de tarefas independentes em ambientes heterogêneos, através da utilização do conceito abstrato de grupo virtual de processadores implementado pela biblioteca StarPU. Foi projetado e implementado um modelo que usa a combinação de grupos virtuais de processadores, políticas de agendamento e estimativas, para otimizar o tempo de execução geral, utilizando apenas uma pequena fração do tempo necessário para executar as tarefas. Essa abordagem nos permite alcançar automaticamente a otimização do tempo de execução geral, levando a minimização do tempo geral de execução das tarefas, permitindo aos desenvolvedores extrair mais desempenho a partir do mesmo *hardware*.

Palavras-chave: StarPU, CUDA e Computação Heterogênea.

Abstract

Developing software to take advantage of heterogeneous hardware resources in High-Performance Computing (HPC) scenarios is a complex task and has many associated variables, but also creates many possibilities and opportunities for solving larger problems that a single processor cannot handle. Depending on the mapping of tasks to the hardware, vastly different execution times are possible. This thesis proposes and implements a novel model for self-optimizing the overall execution time of independent tasks in heterogeneous environments by using the abstraction concept of a virtual group of processor cores provided by StarPU library. We design and implement an innovative model that uses the combination of virtual groups of processors, scheduling policies and estimations, optimizing the overall execution time and using just a small fraction of the time required to execute the tasks. This approach allows us to automatically achieve the self-optimization of the overall execution time, leading to the minimization of the overall execution time of tasks, allowing developers to automatically take advantage and extract more performance from the same available hardware.

Keywords: StarPU, CUDA and Heterogeneous Computing.

List of Figures

1.1	Set of independent tasks of various sizes are dispatched for execution on a multiple CPU and GPU architecture.	3
1.2	Task characterization.	3
2.1	Architecture overview comparison between CPU and GPU, having multi-processors (MPs) with many cores each.	8
2.2	Execution of a task within StarPU. Adapted from Augonnet et al. [6]. . . .	10
2.3	Data transfer and consistency state managed by the DSM. Adapted from Augonnet et al. [6].	11
2.4	StarPU data model scheme.	12
2.5	Five different examples of scheduling context configurations (A, B, C, D and N) on a hypothetical machine with six workers, four CPU cores and two GPUs.	14
4.1	Framework overview.	25
4.2	Measured times and estimations, split into two phases.	35
4.3	Execution of same type tasks over time on a scheduling context with different workers with different performances.	38
5.1	Phase where the historical of executions of the framework is generated. . .	43
5.2	Sum of vectors benchmark example.	44
5.3	Subtraction of vectors benchmark example.	45
5.4	Multiplication of two squared matrices benchmark example.	45
5.5	Multiplication of a vector by a scalar benchmark example.	45
5.6	Tasks execution timeline, based on observations using the NVIDIA Visual Profiler [54].	48

5.7	Transference of output data viewed on the timeline, based on observations using the NVIDIA Visual Profiler [54].	49
6.1	A comparison between measured times and its estimation for 100K tasks in every combination of the eager scheduler and scheduling context configurations from 2 to 15 (Table 5.1).	55
6.2	A comparison between measured times and its estimation for 100K tasks in every combination of the LWS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).	56
6.3	A comparison between measured times and its estimation for 100K tasks in every combination of the WS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).	57
6.4	Comparison between measured times and its estimation for transferring the execution results of 100K tasks in every combination of the eager scheduler and scheduling context configurations from 1 to 15 (Table 5.1).	59
6.5	Comparison between measured times and its estimation for transferring the execution results of 100K tasks in every combination of the LWS scheduler and scheduling context configurations from 1 to 15 (Table 5.1).	60
6.6	Comparison between measured times and its estimation for transferring the execution results of 100K tasks in every combination of the WS scheduler and scheduling context configurations from 1 to 15 (Table 5.1).	61
6.7	Comparison between measured times and its estimation for overall execution of 100K tasks in every combination of the eager scheduler and scheduling context configurations from 2 to 15 (Table 5.1).	63
6.8	Comparison between measured times and its estimation for overall execution of 100K tasks in every combination of the LWS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).	64
6.9	Comparison between measured times and its estimation for overall execution of 100K tasks in every combination of the WS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).	65
A.1	A comparison between measured times and its estimation for 25K tasks in every combination of the eager scheduler and scheduling context configurations from 2 to 15 (Table 5.1).	77

A.2	A comparison between measured times and its estimation for 50K tasks in every combination of the eager scheduler and scheduling context configurations from 2 to 15 (Table 5.1).	78
A.3	A comparison between measured times and its estimation for 25K tasks in every combination of the LWS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).	78
A.4	A comparison between measured times and its estimation for 50K tasks in every combination of the LWS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).	79
A.5	A comparison between measured times and its estimation for 25K tasks in every combination of the WS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).	79
A.6	A comparison between measured times and its estimation for 50K tasks in every combination of the WS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).	80
A.7	Comparison between measured times and its estimation for transferring the execution results of 25K tasks in every combination of the eager scheduler and scheduling context configurations from 1 to 15 (Table 5.1).	80
A.8	Comparison between measured times and its estimation for transferring the execution results of 50K tasks in every combination of the eager scheduler and scheduling context configurations from 1 to 15 (Table 5.1).	81
A.9	Comparison between measured times and its estimation for transferring the execution results of 25K tasks in every combination of the LWS scheduler and scheduling context configurations from 1 to 15 (Table 5.1).	81
A.10	Comparison between measured times and its estimation for transferring the execution results of 50K tasks in every combination of the LWS scheduler and scheduling context configurations from 1 to 15 (Table 5.1).	82
A.11	Comparison between measured times and its estimation for transferring the execution results of 25K tasks in every combination of the WS scheduler and scheduling context configurations from 1 to 15 (Table 5.1).	82

A.12 Comparison between measured times and its estimation for transferring the execution results of 50K tasks in every combination of the WS scheduler and scheduling context configurations from 1 to 15 (Table 5.1).	83
A.13 Comparison between measured times and its estimation for overall execution of 25K tasks in every combination of the eager scheduler and scheduling context configurations from 2 to 15 (Table 5.1).	83
A.14 Comparison between measured times and its estimation for overall execution of 50K tasks in every combination of the eager scheduler and scheduling context configurations from 2 to 15 (Table 5.1).	84
A.15 Comparison between measured times and its estimation for overall execution of 25K tasks in every combination of the LWS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).	84
A.16 Comparison between measured times and its estimation for overall execution of 50K tasks in every combination of the LWS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).	85
A.17 Comparison between measured times and its estimation for overall execution of 25K tasks in every combination of the WS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).	85
A.18 Comparison between measured times and its estimation for overall execution of 50K tasks in every combination of the WS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).	86

List of Tables

4.1	Segment of the historical of executions, extracted from the historical database.	36
4.2	Average of executions measured times and data transference of different tasks types per workers.	37
4.3	Average of executions measured times of different tasks types per scheduling context.	38
5.1	List of all considered configurations of scheduling contexts in the implementation.	46

List of Abbreviations

ALU	: Arithmetic Logic Unit.
API	: Application Programming Interface.
CPU	: Central Processing Unit.
CUDA	: Compute Unified Device Architecture.
FIFO	: First In First Out.
GPGPU	: General Purpose GPU.
GPU	: Graphics Processing Unit.
HC	: Heterogeneous Computing.
HCSP	: Heterogeneous Computing Scheduling Problem.
PU	: Processing Unit.
RAM	: Random Access Memory.
SIMD	: Single-Instruction Multiple-Data.
SIMT	: Single-instruction Multiple-Threads.
SM	: Streaming Multiprocessor.
SP	: Streaming Processor.
WS	: Work Stealing.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Problem statement	2
1.3	General Research Objective	4
1.4	Challenges	4
1.5	Research questions	4
1.6	Key contributions	5
1.7	Relevance and applicability	5
1.8	Thesis outline	6
2	Background	7
2.1	GPUs	7
2.2	Compute Unified Device Architecture (CUDA)	8
2.3	StarPU	9
2.3.1	Scheduling contexts	12
2.3.2	Scheduling policies for scheduling contexts	15
2.4	Heterogeneous Computing Scheduling Problem (HCSP)	16
3	Related Work	17
3.1	Task scheduling strategies for heterogeneous architectures	17
3.1.1	Scheduling algorithms for heterogeneous architectures	18

3.1.2	Impact of data transfers	19
3.2	Performance	19
3.3	Tasks lengths estimations	21
3.4	Load balancing of tasks	21
3.5	Spatial and temporal locality	22
3.6	Conclusion	23
4	Framework	24
4.1	Framework Overview	24
4.2	Framework details	26
4.2.1	Estimation of input/output transference and execution time	35
4.2.1.1	Input data transference and execution estimation for a task submitted to a worker	35
4.2.1.2	Input data transference and execution estimation for a task in a scheduling context	37
4.2.1.3	Output data transference estimation	39
4.3	On the scalability of the framework	40
4.4	Conclusion	41
5	Implementation	42
5.1	Supporting tools	42
5.1.1	Synthetic task generator	42
5.1.2	Benchmarks	43
5.1.3	Scheduling contexts generator	45
5.1.4	Execution validation	47
5.2	Measuring times	47
5.2.1	Input data transference and execution time	47
5.2.2	Output data transference time	48

5.3	Additional frameworks, libraries and technologies	49
5.3.1	NVIDIA visual profiler	49
5.3.2	NVIDIA management library (NVML)	50
5.3.3	Measuring times during execution	50
5.4	Conclusion	50
6	Experimental Results	52
6.1	Platform and environmental settings	52
6.2	Software settings	52
6.3	Historical of executions	53
6.4	Experiments	53
6.4.1	Input data transfer and execution comparison experiment	54
6.4.1.1	Eager scheduler vs Scheduling context configurations vs Number of tasks	54
6.4.1.2	LWS scheduler vs Scheduling context configurations vs Number of tasks	55
6.4.1.3	WS scheduler vs Scheduling context configurations vs Num- ber of tasks	56
6.4.1.4	Random scheduler vs Scheduling context configurations vs Number of tasks	57
6.4.1.5	Conclusion of the input data transfer and execution esti- mation experiment	57
6.4.2	Output data transfer comparison experiment	58
6.4.2.1	Eager scheduler vs Scheduling context configurations vs Number of tasks	58
6.4.2.2	LWS scheduler vs Scheduling context configurations vs Number of tasks	59
6.4.2.3	WS scheduler vs Scheduling context configurations vs Num- ber of tasks	60

6.4.2.4	Random scheduler vs Scheduling context configurations vs Number of tasks	61
6.4.2.5	Conclusion of the output data transfer experiment	61
6.4.3	Overall execution comparison experiment	62
6.4.3.1	Eager scheduler vs Scheduling context configurations vs Number of tasks	62
6.4.3.2	LWS scheduler vs Scheduling context configurations vs Number of tasks	63
6.4.3.3	WS scheduler vs Scheduling context configurations vs Num- ber of tasks	64
6.4.3.4	Random scheduler vs Scheduling context configurations vs Number of tasks	65
6.4.3.5	Conclusion of the overall execution comparison experiment	65
6.5	Limitations	65
6.6	Conclusion	66
7	Concluding Remarks and Future Works	67
7.1	Concluding Remarks	67
7.2	Future Works	68
7.2.1	Short term goals	68
7.2.2	Long term goals	69
	References	71
	Appendix A - Extra Results	77
	Appendix B - Glossary	87

Chapter 1

Introduction

This chapter presents the motivation behind this thesis, problem statement, research general objective, applicability, challenges, research questions, key contributions and finally an overview of the thesis outline.

1.1 Motivation

The number of heterogeneous computing (HC) architectures in use in the world are drastically increasing. Many computing devices, not just personal computers, already include a multicore Central Processing Unit (CPU) and a high-performance Graphics Processing Unit (GPU), including laptops, tablets, mobiles, game consoles and even autonomous cars.

A total of 64 systems on the top 500 supercomputing sites [66] are already using GPUs technology, an attractive asset for High-Performance Computing (HPC), meaning that computers in the world no longer necessarily rely solely on the CPU, but they are mostly heterogeneous machines. Among the top 10 supercomputing sites in the green500 [65], most of them are heterogeneous machines.

GPUs have been evolving at a rapid pace in the last decade [13], providing a cost-effective and massively available solution for HPC. Every year, powerful CPUs and GPUs models with distinct specifications are released in the market, with more computing power, better power efficiency [43], thousands of cores and sophisticated unified architecture. Despite this rapid evolution, there is still a huge gap between simply using and taking full advantage of this heterogeneous computing power potentially leading to a number of profitable search avenues [44]. In fact, it is not a trivial for an application to use all of a

device's resources efficiently. As this trend continues, it is relevant to start to design new parallel software for taking advantage of these heterogeneous architectures [63].

GPUs are highly parallel computing machines specializing in vector arithmetic and have a large number of Arithmetic Logic Units (ALUs), while CPUs are better at number crunching and have high-end branch predictors, making a hybrid solution the best of both worlds for many applications. Execution time depends on each device's settings, communication, and current workload.

The choice of a strategy to assign and map tasks to each processor can make a substantial difference in the overall execution time of the tasks [6], this framework helps the programmer in doing so.

This scenario opens possibilities for the development of frameworks to manage heterogeneous environments and transparently execute tasks, helping the developers to obtain more performance from the same available hardware, reducing the overall execution time of tasks.

1.2 Problem statement

Given an arbitrary set of different independent tasks with distinct sizes (Figure 1.1), all of them implemented and able to be executed either on the available CPUs or GPUs, the problem consists in solving the problem of minimizing the overall execution time of these tasks by dispatching them to be executed on a multiple CPU and GPU system, finding the best virtual configuration of processors that optimizes the overall execution time of the tasks.

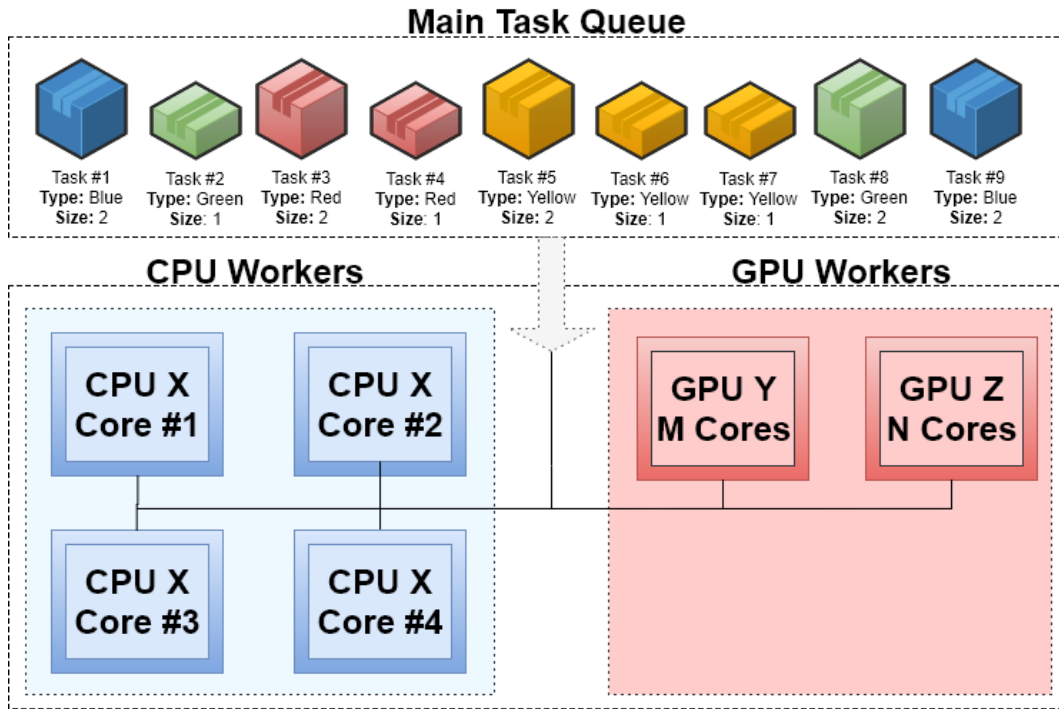


Figure 1.1: Set of independent tasks of various sizes are dispatched for execution on a multiple CPU and GPU architecture.

These independent tasks will run on a dedicated and exclusive heterogeneous machine, meaning that there will be no competition for resources, except for the operating system processes and the tasks itself.

It is a pre-condition that the tasks must be independent of each other, which means that they do not necessarily need to execute in a pre-defined order. They must follow the “atomic” encapsulated architecture on Figure 1.2, where the task, composed of its input data and corresponding function implementation is sent for execution on the matching processor. The task atomically executes from its start to the end, and on termination, output data (results) returns.

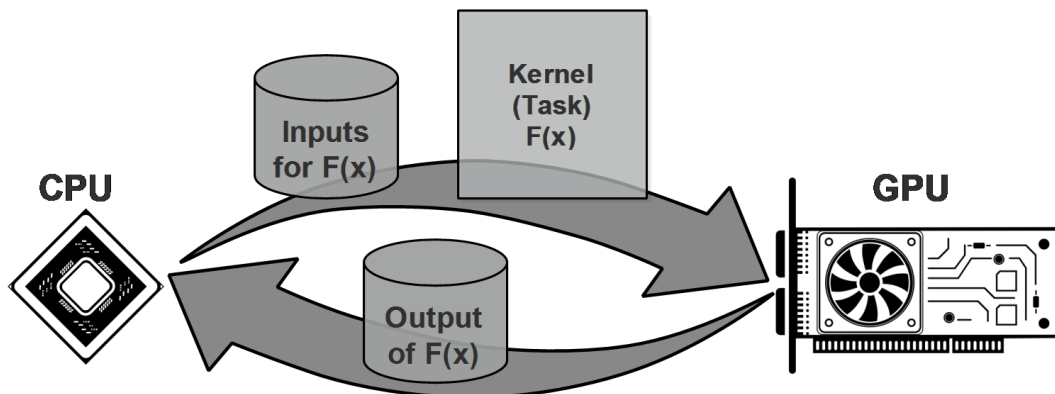


Figure 1.2: Task characterization.

1.3 General Research Objective

The general objective of this research is to design, develop, implement and test proof a framework that uses an efficient heuristic strategy to map distinct independent tasks to different processing units minimizing the overall execution time of the tasks. To reach this objective, tasks are executed in a seamless and intelligent way, so the programmer does not have to worry about the computer architecture, processing units or scheduling algorithms. The front-end itself tries to improve the performance based on the available hardware.

1.4 Challenges

During the design and development of this thesis, a set of main challenges was identified:

- **Using third-party library** - Relying on others code is not an easy task, due to the factors of support, bugs, code extension, features, and efficiency. In general, these libraries are in beta development and are produced by research groups.
- **Fully utilize devices** - Optimizing code for a task to fully take advantage of CPUs and GPUs, it is not simple since StarPU does not differentiate between CPU and GPU Workers.
- **Benchmarks** - Deciding which benchmarks to use is an important decision, since it is necessary to test the software with different types of code. However, for benchmarks to work in task based systems it is necessary to reimplement applications [62]. It is imperative not lose focus on this research since parallelizing codes for StarPU is not the objective or scope of this thesis, but parallelizing tasks execution.
- **Mapping kernels to a task model** - To rely on a framework and its abstractions it is necessary to transform and adapt code and data into the proper abstract data type of the library, which is not an easy task and sometimes requires a methodology [21]. For instance, to run a matrix multiplication function using the framework, it is necessary to adapt it to the abstract data type used by the framework.

1.5 Research questions

The research questions addressed by this thesis are presented as follows:

1. Is it possible to design a generic framework in order to transparently optimize each independent task execution on a multiple CPU and GPU machine?
2. Is it feasible to implement this framework that optimizes independent tasks overall execution time while adapts itself for multiple CPU and GPU machines without modifying the implementation?
3. Is it possible to estimate with enough precision the execution time of independent tasks under different scenarios without having to execute them?
4. Is it viable to determine the duration of independents tasks with sufficient precision in a fraction of the required time to execute them?

1.6 Key contributions

Following are the major contributions of this work:

- **Development of a framework for task distribution in multi-CPU and multi-GPU Systems** - Conceptual design and definition of the framework (Chapter 4).
- **Front-end that minimizes the overall execution time of tasks seamlessly** - Manual development in a heterogeneous system requires a lot of effort from the software programmer since it is necessary to take manual care of memory management, load balancing and synchronization. Most of the time, it is not enough to only use a parallelization framework to obtain performance. One needs to optimize and tune the application to a particular architecture. The proposed front-end helps the programmer in doing so (Chapter 5).
- **Validation of the framework** - Validation of the framework using common scientific operations as benchmarks and testing its efficiency (Chapter 6).

1.7 Relevance and applicability

The contributions of this thesis have wide applicability in Computational Science and areas that demand massive high-performance computing ranging from Physics [33, 60, 71], Genetics [11], Medicine [2, 36], Biology [70], Chemistry [10], Artificial Intelligence [46],

Weather Forecasting [22], Oil Industry [37], Financial Market [59], and others scientific and business fields.

Every problem that is convertible to the form of an independent atomic task and requires to process a massive number of these tasks can take advantage of the proposed framework on this research. For instance, processing of a massive number of images for artificial intelligence, hashing functions as in cryptocurrencies or scientific functions.

Consequently, this work helps to fill the gap on developing better resource management software to extract more performance from powerful heterogeneous hardware that is already improving every year. This work also opens paths, raises questions and possibilities for new research and future work on the subject.

1.8 Thesis outline

This thesis consists of seven chapters, the remaining of this document is organized as follows: Chapter 2 introduces the background and important concepts. Chapter 3 discusses the related work. Chapter 4 presents the framework. Chapter 5 details the implementation and its details. Chapter 6 discusses the experimental results. Finally, Chapter 7, presents the concluding remarks and future work.

Chapter 2

Background

This chapter presents and discuss relevant and fundamental concepts that are important to understand the remainder of this work. First, this chapter gives a brief overview of the GPUs, CUDA architecture [49] and the StarPU Library [8].

2.1 GPUs

GPUs are highly parallel processors originally dedicated to graphics computation and has become part of mainstream computing systems [57]. Thousands of lightweight processing cores compose ordinary GPUs, large bandwidth access to on-chip memory and gigabytes of Random Access Memory (RAM) [55], being specially designed for processing tasks that make use of their vector arithmetic.

GPUs use the Single Instruction Multiple Data (SIMD) model, where a single instruction controls multiple processing elements [48], which is different from the traditional model used by CPUs. To take advantage of the GPUs highly parallel architecture, it is necessary to adapt data structures to this type of paradigm.

The main difference between a CPU and a GPU is about how they execute tasks. A CPU consists of a few number cores optimized for sequential execution and processing while the GPU has a parallel architecture consisting of thousands of smaller cores designed for handling multiple kernels simultaneously (Figure 2.1).

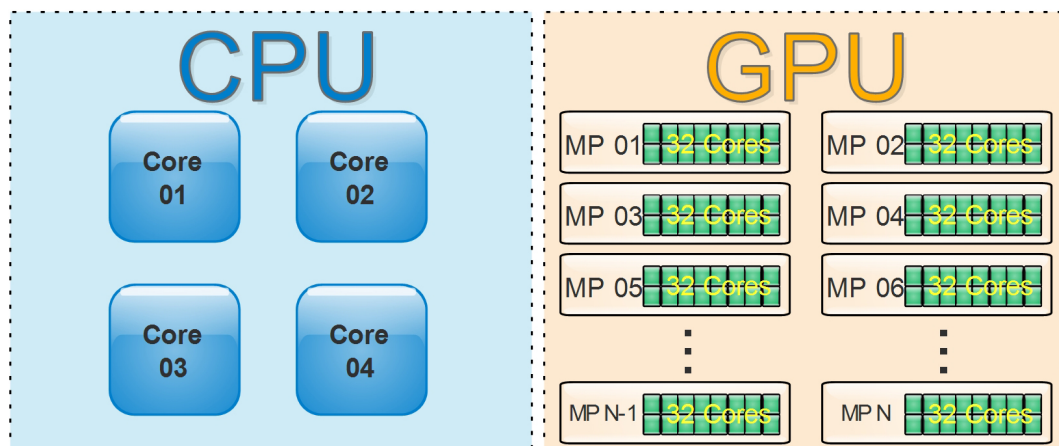


Figure 2.1: Architecture overview comparison between CPU and GPU, having multiprocessors (MPs) with many cores each.

By definition, GPU devices have limited control capabilities compared to CPUs, due to their high density of execution units. These limited instructions and simple logic allow GPUs to benefit by having more Arithmetical Logical Units (ALUs) on the chip, compared to CPUs.

2.2 Compute Unified Device Architecture (CUDA)

Compute Unified Device Architecture (CUDA) [49] is a unified parallel computing architecture developed by NVIDIA. Currently, programming in CUDA is possible through an API that extends the C programming language on CUDA-enabled devices.

The parallel portions of code that execute on the GPU are known as kernels and called from CPUs or GPUs sections of code. In CUDA, groups of threads will execute the same code, which is inherent to the SIMD model, these groups are called warps and executed on the same Streaming Multiprocessor (SM), through the abstraction of blocks in CUDA.

CUDA allows the software designer to launch a massive number of SIMD threads, where threads inside the same block are all mapped to the same SM on the device. Each GPU has its specific architecture with a different number of SMs, memory, and cores. Execution of the same kernel on different GPUs can lead to different execution times.

There are large sets of possible optimization configurations that can be applied. Each kernel may require a specific configuration to achieve the best performance, and running an application to a new hardware often requires a new optimization configuration for each kernel [26].

2.3 StarPU

Among the investigated frameworks that support tasks, and due to the desired objectives and demands of this research the StarPU was chosen due to its features, support for CPUs and GPUs, rich documentation and several recent publications.

StarPU [8] is a unified runtime system for heterogeneous multicore architectures, hybrid CPU, and GPU execution. It was developed to give support for heterogeneous multicore architectures and present a unified view of the computational resources, CPUs and GPUs, simultaneously.

StarPU tries to efficiently map and dispatch tasks to different processing units, while transparently handling low-level issues such as data transfers. The core of StarPU is its run-time support library, which is responsible for scheduling tasks on heterogeneous machines. StarPU runtime and programming language extensions support a task-based programming model, which allows developers to implement custom scheduling policies in a portable fashion.

StarPU provides GPU support and documentation, which is essential in making StarPU useful as a tool for development. The StarPU team also provides constant and fast response support. Google Scholar also reveals that interest in this framework is growing and the increasing number of publications involving StarPU over the last years.

StarPU allows a single application to run on different architectures, but it also allows performance portability since the application can benefit from StarPU optimizations independently of the underlying hardware [7]. There are a few important terms that must be understood before understanding the starPU architecture itself [39]:

- **Task:** represents the unit of work to be run on the processor defined by the programmer. It has input and output data that must be transferred to the processors.
- **Codelet:** records pointers to various implementations of the same abstract function.
- **Data handle:** keeps track of copies of the same data over various memory nodes. This data is registered by the application. The Distributed Shared Memory (DSM) abstraction and implementation is responsible for keeping them coherent.
- **Scheduling context:** a mapping between a set of tasks and processors. Scheduling contexts can be created and destroyed on-the-fly. If a context is created, the programmer can statically or dynamically map tasks to a set of processors.

- **Worker:** execute the tasks, there is typically one per CPU computation core and one per GPU.
- **Scheduler:** schedules tasks to workers when they are ready to be executed, which means that the tasks do not have data or task dependencies. The workers pull tasks one by one from the scheduler.

Each task assigned to a GPU worker runs as an independent piece of code without any control on how the computation is distributed on the device. StarPU does not make distinctions between GPU workers.

There are two basic principles in StarPU: every task has its implementation and data transfers are handled transparently. This is why StarPU can dynamically schedule tasks to any processor. For each implementation, a task will have a different execution performance on each different processor. The runtime system also keeps track of data copies linked to each processor, avoiding copies whenever is possible [6].

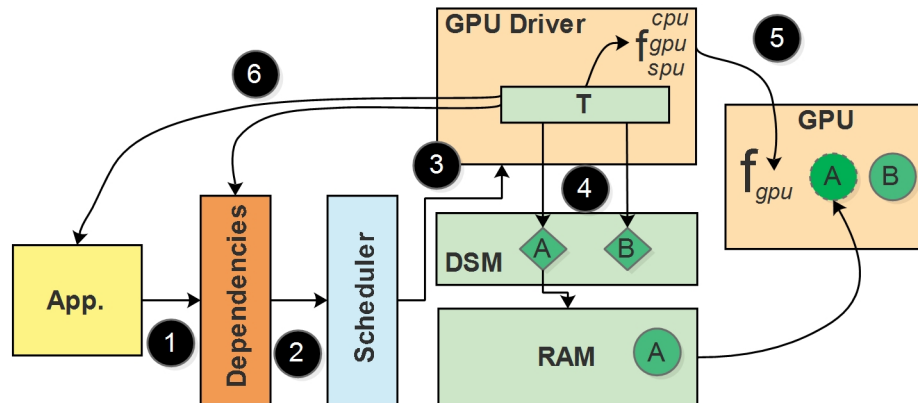


Figure 2.2: Execution of a task within StarPU. Adapted from Augonnet et al. [6].

An overview of StarPU framework flow can be seen in Figure 2.2, giving the reader a basic idea of how StarPU modules work together. In (1), tasks from the application are submitted to StarPU. When a task is ready (2), it is dispatched to one of the device drivers by the scheduler (3). As every function f can have multiple implementations for multiple workers, the required correspondent implementation is also offloaded (5). The Distributed Shared Memory (DSM) module (4) is responsible for making all necessary data available to where the task will be executed. By the time that the given task finishes, an optional callback for the task is executed (6).

The DSM is responsible for data replication and memory consistency. When the application registers data to the DSM, StarPU allocates an array to record the memory

state of that memory in different nodes, which means that data can be replicated on different nodes keeping read-write consistency. StarPU uses Modified Shared and Invalid (MSI) states to keep data consistency.

In Figure 2.3, GPU X #2 requests Read and Write (RW) access to the data on CPU RAM or GPU RAM X #1, represented by the pink circle. This data has shared status, since it is replicated in at least two places, but still has consistency, avoiding new transfers. After this data is transferred from the CPU or GPU RAM to the GPU X #2 it is marked as invalid on the CPU RAM and also on GPU RAM X #1, since GPU X #2 requested RW access on its memory. This data modification is represented by the empty circle, invalid, transitioning to a yellow circle that represents the modified data.

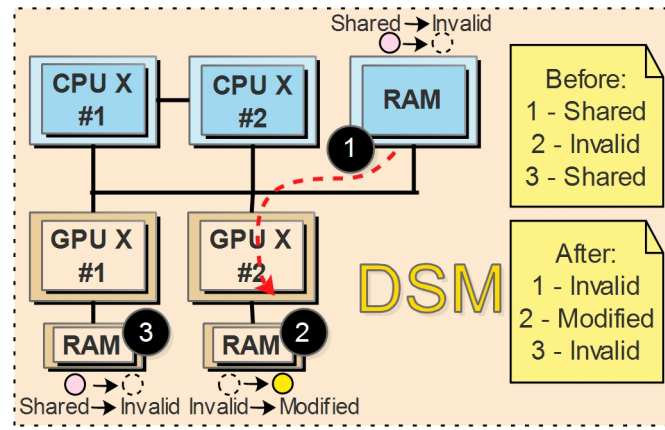


Figure 2.3: Data transfer and consistency state managed by the DSM. Adapted from Augonnet et al. [6].

Basic knowledge of the StarPU Data Model is necessary to understand how to take advantage of its architecture. After the codelet is created by the software, which record pointers to various implementations of the same abstract function, now it is possible to point to two different implementations of the same function, one to a CPU and another to a GPU as shown in Figure 2.4. The codelet defines characteristics common to a set of tasks and relates an abstract computation kernel to its implementations so that tasks can be instantiated, in this case, T1 and T2. The codelet also records pointers to various implementations of the same abstract function.

A task is an instantiation of a codelet and atomically executes a kernel from its beginning to the end, receiving some input and producing some output. A task needs to be associated with a data handle, which designates a piece of data managed by StarPU and DSM.

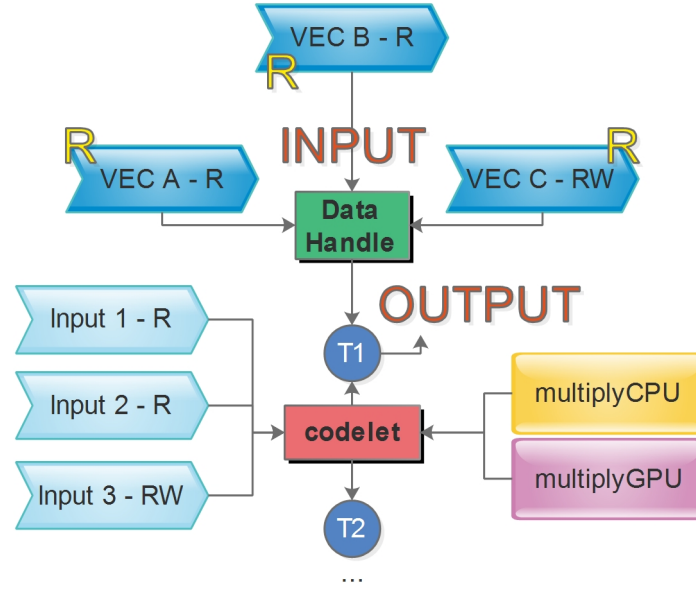


Figure 2.4: StarPU data model scheme.

The basics of the scheduling policies in StarPU are that the scheduler gets to schedule tasks when they are ready to be executed, which means that they are not waiting for data or task dependencies. CPU or GPU workers pull tasks one by one from the scheduler. Scheduling policies usually contain at least one queue of tasks that store them between the time they become available and the time a worker retrieves them.

It is relevant to mention that the use of StarPU helps to keep the scope of this proposal focused since advantage is taken of already implemented abstractions and features. Implement each starPU framework function would be time consuming. The decision to use StarPU also add implementation constraints and some computation overheads on the software. StarPU was chosen as a tool to evaluate the contributions and any similar framework could be used instead. However, the proposed solution on this research are concepts that can apply to any other tool or framework.

2.3.1 Scheduling contexts

Scheduling contexts [28] is an abstraction for sets of workers that allow the programmers to control the distribution of tasks to resources, virtual groups of CPUs and GPUs [39]. The main motivation to use scheduling contexts according to StarPU authors is to minimize interference among multiple parallel kernels, through the creation of a partition of available workers. In the starPU framework, tasks are submitted to a default scheduling context, which disposes of all the computation resources, workers, available to StarPU.

Hugo et al. [29] show the benefits of the isolation capabilities of scheduling contexts, with no modification to the original code of the kernel. Scheduling contexts in this situation can be understood as a virtual subset of workers. This abstraction can lead to significant performance gains compared to situations where a parallel code is mixed over a pool of processors, instead of using a sub-group. This feature developed by Hugo et al. have been incorporated to StarPU. A similar approach was designed by Sun et al. [67] which is called as a work pool, which is a structure that contains a collection of processors.

Although the construction of scheduling contexts is already an abstraction provided by StarPU that allows the programmer to precisely control the distribution of computational resources in concurrent parallel kernels, the objective was to exploit this abstraction to design, develop and program strategies, leading to speedups. In the proposed framework, statistics related to the previous execution of tasks are collected and a mapping is defined, which minimizes the overall execution time of the tasks.

According to the StarPU handbook [39], if the developer needs to execute many parallel kernels simultaneously, by default these kernels will execute within the default scheduling context, which contains all workers and using a single scheduler policy. Each scheduling context has only a limited view of hardware resources [29]. If the application programmer has more information about the kernels he wants to run, it is possible to split the workers in various scheduling contexts, isolating the execution of each kernel, and allowing the use of specific scheduling policies for each scheduling context.

Considering a hypothetical machine with six workers, one processor with four CPU cores and two GPUs, many possible combinations of workers and scheduling contexts can be assembled, as illustrated on Figure 2.5: (A) one scheduling context with six workers; (B) two scheduling contexts, one with four CPU cores and the other with two GPUs; (C) two mixed scheduling contexts, both with two CPU cores and one GPU each; (D) three scheduling contexts, two of them with two CPU cores each and the other with two GPUs; (N) each scheduling context has only one worker.

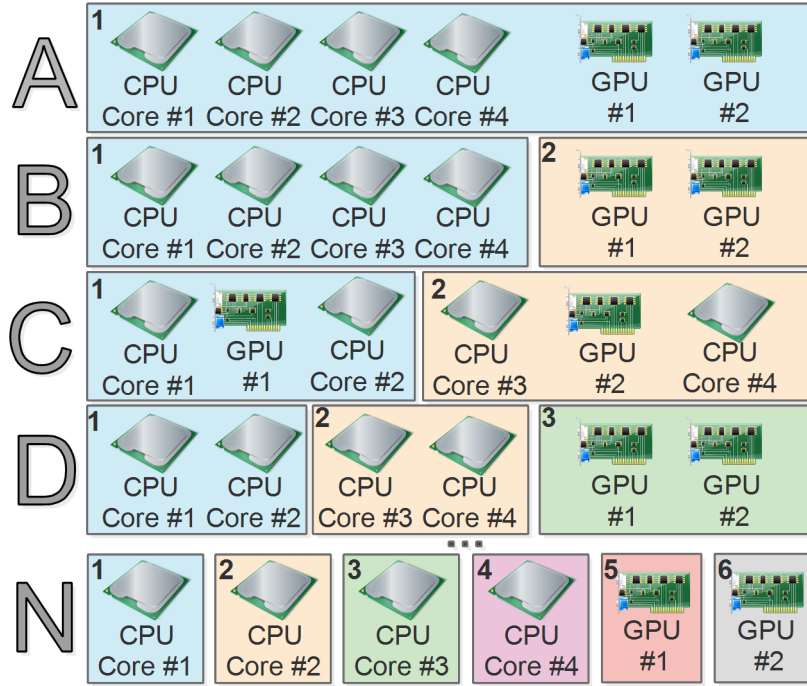


Figure 2.5: Five different examples of scheduling context configurations (A, B, C, D and N) on a hypothetical machine with six workers, four CPU cores and two GPUs.

One of the most significant assumptions of this thesis is that scheduling contexts with different configurations of workers produce different execution times for the same group of tasks. This experiment occurred and was confirmed during the development of this thesis.

All virtual groups of workers or scheduling contexts are disjoint sets, in simple words, they have no worker in common, which is equivalent to say that a worker can not be part of two scheduling contexts at the same time. This decision was a project design used for the sake of simplification of the research, making it possible to gain focus on the topics of more significance for this study. The use of joint scheduling contexts features as a suggestion for future works on Section 7.2.

In the scenario of creating scheduling contexts composed of mixed workers, as in the case of letter C in Figure 2.5, although workers are in the same virtual group, they are still physically apart, meaning that they still need to transfer data between them to communicate. The exchanging of data between CPU and GPU is much slower than from cores on the same CPU. They do not need to communicate through the bus because all the CPU cores are on the same chip. If a scheduling context has one or more GPUs, they still need to communicate explicitly through the bus.

The supported and allowed operations for contexts are:

1. **Creation of a scheduling context** - Creation of scheduling contexts by the programmer indicating the set of processors corresponding to each context. It is also possible to set its scheduling policy, instead of the default one which is the eager policy. The details of the scheduling policies are on Subsection 2.3.2.
2. **Modification of a scheduling context** - Allows to add additional workers to a context or remove if no longer needed.
3. **Submission of tasks to a scheduling context** - The kernel which runs inside a task is sent to be executed by a context or group of processing units.
4. **Deletion of a scheduling context** - In this operation, the application that is running and using StarPU can indicate which scheduling context should keep the resources of a deleted one. All the tasks of the scheduling context should have terminated their execution before doing this.

2.3.2 Scheduling policies for scheduling contexts

Among the schedulers available on StarPU framework, four of them are used. The following definitions were obtained from the StarPU handbook [39]:

1. **Eager** - It has a central task queue, from which all workers draw tasks to work on concurrently. It does not permit to prefetch data since the scheduling decision is taken late.
2. **Work Stealing (WS)** - It has a queue per worker and schedules a task on the worker who released a finished task. When a worker becomes idle, it steals a task from the most loaded worker. Chatterjee et al. [16] demonstrated the effectiveness of WS in distributing tasks across devices using different benchmarks.
3. **Locality Work Stealing (LWS)** - Uses a queue per worker, and schedules a task on the worker who released it by default. When a worker becomes idle, it steals a task from other workers.
4. **Random** - The random scheduler uses a queue per worker, and distributes tasks randomly.

2.4 Heterogeneous Computing Scheduling Problem (HCSP)

A Heterogeneous Computing (HC) architecture can be made up of different kinds of processing units, with CPUs, GPUs and other accelerators acting as co-processors for arithmetic intensive data-parallel workloads [64]. The trend towards heterogeneous computing and highly parallel architectures creates a high demand for software development infrastructure in the form of parallel programming languages and libraries supporting heterogeneous computing. The Heterogeneous Computing Scheduling Problem (HCSP), became especially important due to the popularization of heterogeneous distributed computing systems and libraries [47].

Given an HC system composed of a set of machines and a collection of atomic tasks to be executed on this system, $T^{n,m}$ represents the time required to execute the task N in the processing unit M . The goal of the HCSP is to find an assignment of tasks to processors, which minimizes the sum of all $T^{n,m}$.

The HCSP is an NP-Hard problem since the homogeneous scheduling problem is also NP-Hard [23] and heterogeneity just adds complexity to the problem even harder, which means that exact methods are generally not viable for solving large instances.

In this thesis, a set of tasks is executed on an heterogeneous system, with an independent task being an atomic unit of workload. The execution time of a task varies for every different machine, which means that tasks will compete for the processors or machines that can execute them in the shortest interval.

Scheduling Problems (SP) are mostly concerned with minimizing the time spent to execute all of the tasks, the common metric to minimize in this model is the makespan [47].

Chapter 3

Related Work

This chapter discusses the most relevant prior research work related to the areas covered in this thesis, categorized in topics of interest.

3.1 Task scheduling strategies for heterogeneous architectures

According to Hugo et al. [29] and Andrade et al. [4], there is no perfect scheduling strategy suitable for running every parallel kernel library, which makes the scheduling problem suitable for heuristic-based solutions.

Greg et al. [27] affirm that a greedy local scheduling decision of tasks, leads to imbalance, causing underutilization of devices and contention of others. His suggestion is to make global scheduling decisions and assign some applications to a slower device. This decision can increase system throughput and decrease individual application runtimes.

In the proposed framework, a greedy local scheduling decision is utilized to achieve an initial unbalanced solution. Although this is not an efficient solution at first, this strategy is an initial step towards a balanced and efficient final solution, which is improved through a series of iterations. The significant advantage of using a greedy strategy is that it is possible to rapidly create an initial solution and refine it, and consequently solve the problem fastly.

Wen et al. [73] presented an effective OpenCL task scheduling scheme which schedules multiple kernels from multiple programs on heterogeneous platforms, which determines at runtime which kernels are likely to utilize best a device. He shows that the speedup is a good scheduling priority function and developed a novel model that predicts a kernel's

speedup based on its static code structure, obtaining significant performance improvement over other approaches.

Zhou et al. [74] consider that for task mapping in heterogeneous embedded systems for fast completion time, three factors need to be taken into account to make mapping decisions: (i) Treating kernels as the mapping entity yields better performance than mapping each entire application to a processor; (ii) If a kernel have much shorter execution times on a GPU compared to CPU, this kernel should be prioritized somehow, having higher possibilities to be assigned to its favorite processor and (iii) check if data partition worth and consider the transferring overhead.

The proposed framework follows the recommendation of Wen et al. [73] by determining through previous executions which tasks are likely to utilize best a device through the use of a historical database. The framework also follows Zhou et al. [74] conclusions, which says that tasks should be prioritized and assigned to its favorite processor.

3.1.1 Scheduling algorithms for heterogeneous architectures

Among the relevant works that englobe scheduling algorithms, heterogeneous architectures and tasks, Gautier et al. [24] introduces a general scheduling algorithm for multi-CPU and multi-GPU systems that enforces a locality-aware work stealing (LWS), which is one of our four scheduling algorithms or policies defined on Subsection 2.3.2 and considered on this work, which are set inside each scheduling context configurations defined on Subsection 5.1.3.

Greg et al. [27] described and demonstrated a dynamic scheduling algorithm for heterogeneous architectures that utilize a historical database of executions, which is in part a similar idea of this work. Greg et al. showed that using a historical information database of executions; the scheduler can determine the best way to assign applications to heterogeneous processors, using all devices available and increasing the throughput. This work utilizes the historical information database for creating the best Scheduling Context that optimizes time, while Greg et al. use for optimizing the scheduling.

Kaleem et al. [34] presented a set of scheduling algorithms that use load balancing to minimize application execution time on heterogeneous architectures. Our work uses a similar approach defined in our framework, while Kaleem et al. make a low-cost online profiling, and this work an offline approach.

It is simply not the focus of this work to design new scheduling algorithms but to use

the already defined ones as a tool to develop a task distribution system that transparently optimizes the overall execution time of independent tasks, while at the same time, forming virtual groups of processors or scheduling contexts.

3.1.2 Impact of data transfers

Data transfers can take much time due to the PCI Express 3.0 bus between the CPU and the GPU being a bottleneck regarding global memory bandwidth [61], which acts as a barrier for efficient execution of GPU kernels and tasks. In several applications, the cost of data transfer operations is comparable to the computation time [68].

Nowadays, acceleration cards support asynchronous data transfer, this way transfers can be overlapped with computations, leading to better performance and speedups. This makes prefetching of data a significant advantage, as soon as a task is scheduled to be run by some processor, the data transfer order can be queued, so when the task is going to start, probably its data is already available, thanks to data being transferred in parallel, optimizing the whole execution [6].

Data transfers have a critical impact on efficient execution [68], which is why a scheduler favoring locality can increase the benefits obtained by reusing data with caching techniques [6]. It is important to enforce locality between related tasks, leading to gains of up to 35% [29]. Hugo et al. believe that an increase in processing units sometimes does not compensate for the costs of data transfers to isolated cores. This behavior is seen especially for small matrices for which the computations do not counterbalance the data transfers [29], which is a case that the proposed framework stays aware of to reach better performances.

In the proposed framework, advantages are taken from the abstraction of scheduling contexts developed by Hugo et al. [29] which allows the creation of virtual groups of processors which can favor locality and consequently leads to gains of time. The transfer time estimations are used to check if it would be worth executing a certain scheduling context configuration on a set of processors after transferring the required data.

3.2 Performance

The architectural and algorithmic differences between devices and applications can cause profound effects on the achieved performance, which is why scheduling and selection can

help by deciding which implementation and device to use by considering previous historical execution information along with information about current load balance, transfer costs and data locality [20].

Navarro et al. [45] finds that three factors are critical to achieve ideal performance and maximize utilization on multiple CPU and GPU in heterogeneous architecture: computational speed of each computing resource should be accurately measured, the assignment of chunks to the computational resources, which means that load imbalance must be minimized and finally, the chunk size for GPUs might be fixed to minimize data transfers.

Agullo et al. [3] proposed a three-step methodology to run algorithms as fast as possible on complex architectures composed of multicore CPUs and GPUs. The first step consists in writing the algorithm in a sequence of multiple tasks of fine granularity. The second phase consists of providing high-performance kernels with CPU and GPU implementations of the tasks. The final step consists in integrating these tasks on a runtime system which is responsible for scheduling the different tasks onto the processing units taking into account dependencies, data availability and coherency.

The framework in this thesis considers all these relevant factors described by the authors. The history of execution similar to the one proposed by Dastgeer et al. [20] is used, which helps to better understand the behavior of each task on a certain processor. The load imbalance which Navarro et al. [45] points as a critical issue for performance is minimized, this issue is solved by using estimations to guess the finish time of each group of tasks, giving just the correct load of tasks for each group, making them finish their jobs simultaneously. This work also follows the recommendations of Agullo et al. [3] by using a framework that integrates these tasks on a runtime system which is the case of StarPU, transparently handles data availability and coherency of data.

Along with a few related works of this research, Cojean et al. [19] fill the performance gap between accelerators and individual cores by using multiple tasks granularities. His work extends starPU and uses the concept of scheduling contexts, which requires in-depth modifications to the data layout used by existing implementations. Cojean et al. reduce the performance gap between processing units by forming scheduling contexts of CPUs, exploiting tasks' inner parallelism. The performance of these clusters of CPUs can better compete with one of the powerful accelerators such as GPUs.

Martinez et al. [42] developed an experiment on modeling seismic wave propagations using StarPU and demonstrated significant performance efficiency and acceleration of 32% over the heterogeneous version. The significant impact on performance is considered to

be caused by the granularity and scheduling strategies. This experiment points that the proposed framework is towards the right direction, it is used not only different scheduling strategies, but also scheduling contexts, which sets virtual groups of processors.

Breder et al. [12] developed a strategy to optimize and reorder the submission of kernels to the GPU, the results showed that changing the submission order lead to significative gains on the average time of turnaround and throughput of the system, in comparison to the default submission of kernels implemented on the modern GPUs.

3.3 Tasks lengths estimations

Greg et al. [27] propose an accurate estimation when an application is up to finish its execution on a given device. He proposes the use of a runtime information database, which keeps an average runtime for each application and information about the input data size. This information allows better scheduling decisions that are both globally and locally efficient, with higher computational throughput and device utilization for all processing units.

According to Augonnet et al. [5] it is not necessary to have perfectly accurate models to make correct decisions when mapping tasks to processors. The most important part is to capture the relative speedups and also the affinities between tasks and processors.

Luk et al. [41] use an analog approach of offline profiling to get information about the execution of tasks on different computing devices, but with different calculations. Teodoro et al. [69] proposed a solution that when a new application is implemented, it is benchmarked for a representative workload and the execution times are stored.

In the proposed framework, the recommendation and conclusion of Augonnet et al. [5] are essentially followed. Although the historical database of executions does not give precise information about the length of each task since the the average as main metric is utilized, just having a vague idea of the average time for that particular task it is already a good hint about the behavior and duration of that task on a specific processor.

3.4 Load balancing of tasks

To minimize a software overall execution time on heterogeneous hardware, most of the time it is fundamental to consider the load balancing of the distributed task workload [15].

The ideal size of computational task and data that is sent to an accelerator during each task assignment should also be considered. If the block size is too small, data transfer to the device will take most of the time, and the device initialization overhead will be excessive. On the other hand, large blocks will lead to more performance in GPUs; however huge blocks can result in work imbalance. The right block size can minimize underutilization and load imbalance, resulting in shorter execution time [9].

Load balancing and GPU resource utilization cannot be satisfactorily addressed by the current GPU programming paradigm, CUDA scheduler cannot handle the unbalanced workload efficiently [18]. It is fundamental to distribute the workloads equitably, based on the processing capacity of each processor, to reach balance and improve performance.

The use of multiple GPUs can considerably increase the computing power available for an application, which can cause workload imbalance by decreasing occupancy and lead to the underutilization of device resources. This means that devices may be idle between kernel executions, leading to reduced performance.

According to Odajima et al. [56] the task size should be considered for heterogeneous computing. A certain size or larger task size is required to achieve enough performance on the processing units. Otherwise, the efficiency to utilize GPU becomes low, since the GPU needs to be feed with more tasks and bigger ones. If there is a relatively small number of tasks, it is not possible to keep the load balance between CPUs and GPUs, because it was not assigned enough tasks.

In the proposed framework, it is considered the load balancing of tasks by estimating the duration time of each set of tasks associated with each group of processing units. Sets of workers with GPUs will tend to receive bigger tasks since they run more efficiently on that particular group, considering the capacity of each processor according to Chen et al. [18] and Odajima et al. [56] recommendations.

3.5 Spatial and temporal locality

It is of particular importance to consider spatial and temporal locality when considering the scheduling of tasks, which can save much time with memory access, since tasks that access one memory location tend to reaccess the same memory location, as well as nearby memory locations [44].

Hugo et al. [28] showed that the scheduling contexts defined by starPU and explained

on this work on subsection 2.3.1, helps better exploit data reuse and improves locality. On his experiments was observed that the use of scheduling contexts increased the hit rate by almost 10% in comparison to the regular case, and reduced the amount of data transferred by more than 50%, increasing the execution performance.

Our framework exploit the effect on spatial and temporal locality, by designing different scheduling contexts configurations on subsection 5.1.3, and automatically finding the one that optimizes the overall execution time of the independent tasks.

3.6 Conclusion

This Chapter discussed the most relevant prior research work related to the areas covered in this thesis. In order to build a framework for distribution of independent tasks in multi-CPU and multi-GPU systems, it is required to consider some critical factors: the impact of data transfers when executing a task in a processor; the scheduling strategy; the performance of each task on a given core processor; efficient and fast estimations; the load balancing of the tasks and finally, the spatial and temporal locality.

Chapter 4

Framework

This chapter presents the framework and give some generic details without diving down into specific of the implementation. The application and software implementation details are explained on Chapter 5. The framework was designed to be capable of adjusting itself during operational time. The objective is to find the best configuration that minimizes the overall execution time for a particular set of distinct independent tasks, helping to obtain speedups transparently.

4.1 Framework Overview

We summarize in phases the Framework overview on Figure 4.1. Each box on the workflow represents a a high level abstraction of the framework, presenting using this form, even having to make a few abstractions, eases readability and comprehension before presenting the details of the framework in depth. The numbers in parenthesis along this section text represent the position on Figure 4.1 that is being detailed.

The general simplified idea of the framework is to use previous historical information of past executions to infer the best mapping of tasks and the virtual group of processors that minimize the overall execution time of the tasks. The implementation starts with the initialization of the application (1) where is set the environment parameters of the framework. On the next phase, (2) historical data about how each task perform on each core of the machine is obtained, in the case of this information is not available, it can be generated at this point.

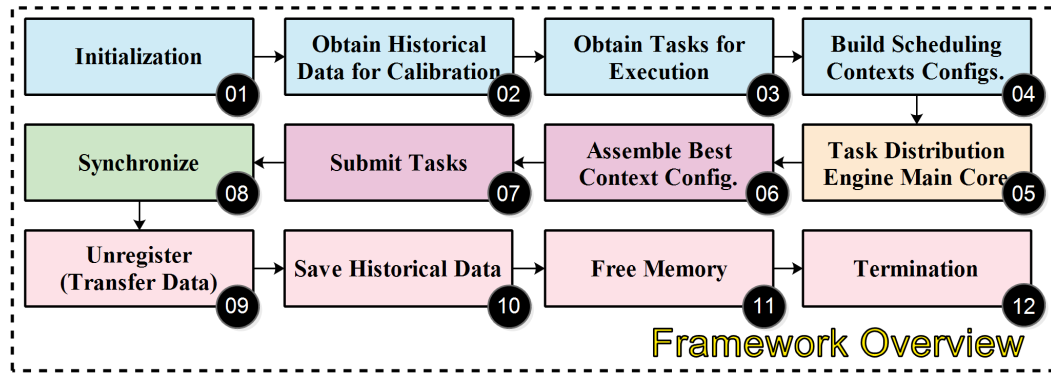


Figure 4.1: Framework overview.

At this phase, the framework statically receives all tasks that will execute (3). These tasks have various implementations of the same abstract function as seen on Figure 2.4. It makes it possible for a task to know how to execute seamlessly, independently of the worker architecture the task is assigned, whether it is a CPU or a GPU. After this, the tasks are finally instantiated but still pending to execute. It is now time to build the scheduling contexts configurations candidates (4) that are going for evaluation through estimation of the overall execution time of the tasks in our framework.

After the definition of the possible contexts configurations on 4, the task distribution engine main core (5) will go through the possibilities and analyze the time estimations for every configuration associated with that particular group of task.

Inside the task distribution engine main core (5), firstly, a greedy solution to map each task to the fastest virtual set of processors is created. Secondly, the solution is iteratively improved by trying to exhaustively move tasks from one virtual group of processors to another, until no improvements to the overall execution estimated time estimation can be made. This condition is finite and converges since the number of possible moves is limited. The framework builds various configurations of virtual sets of processors based on the machine hardware setup and scans through all. The virtual set of processor configurations is rapidly evaluated through the use of estimation of the execution time, finding the one that possibly minimizes the overall execution time of the tasks.

After the context with the best estimation is defined, it is now time to assemble this best context configuration (6) and submit the tasks (7) for execution, it scans the vector of tasks, and for each task, reads its designated context and dispatches the task for execution.

Since different contexts can have different execution times, the slowest context is our

limiting factor. It is necessary to synchronize (8) the execution, waiting for all to finish. After this, it is time to unregister the data (9) and transfer it. This operation allows the result to be collected and keep memory coherence. After the accomplishment of the tasks execution and data transfers, we must save the historical data of executions (10) and write these logs on disk, which is crucial to the post-analysis because it allows comprehending the events that happened during the execution and obtain new calibration data for new executions. After the necessary information is on disk, it is now time to free the memory (11) on CPU and GPU. On the termination function (12) we shut down the framework.

4.2 Framework details

On Procedure 1 is presented in depth the details of the framework. It is assumed that each task has a CPU and GPU implementation and can run in a transparent fashion once submitted to the corresponding context. The framework is divided into two parts: (i) the main function which wraps all the environment data required to run the framework; and (ii) the task distribution engine main core, which effectively does the optimization job, finding the best distribution of tasks among groups of core processors.

Procedure 1 Framework - Overview - Part I

```

1: Procedure TASK DISTRIBUTION FRAMEWORK
2: Input:      (1) tasks[N], (2) calibrationTasks[M] (3) machineConfigurations, (4)
   schedulerPolicies[S], (5) calibratingMode
3: Output: (1) Empty.
4:
5: tasks[N]  $\leftarrow$  set of tasks to execute
6: calibrationTasks[M]  $\leftarrow$  calibration set of tasks
7: submit tasks[N] to the main queue of tasks
8: analyzeTasks(Tasks[N])
9: machineConfigurations  $\leftarrow$  info from the current available hardware
10: contextsConfigurations[P]  $\leftarrow$  possible context configurations
11: schedulerPolicies[S]  $\leftarrow$  possible schedulers
12: historicalOfExecutions  $\leftarrow$  historical of executions
13:
14: if (historicalOfExecutions is empty and calibratingMode is on) then
15:   calibrationSummary  $\leftarrow$  empty
16:   tasks[N]  $\leftarrow$  calibrationTasks[M]
17: else
18:   calibrationSummary  $\leftarrow$  analysed historical data from the historicalOfExecutions
19: end if
20:
21: bestTaskSchedulingMap  $\leftarrow$  taskDistributionEngineMainCore()
22: submit tasks for execution based on the bestTaskSchedulingMap
23: wait for all contexts to finish tasks executions
24: transfer results
25: reevaluate strategy
26: update the historical of executions
27: save the historical of executions
28: terminate

```

Procedure 1 Framework - Overview - Part II

```

29: Procedure taskDistributionEngineMainCore
30: Input: (1) tasks[N], (2) contextsConfigurations[P], (3) schedulerPolicies[S], (4)
    calibrationSummary, (5) calibratingMode.
31: Output: (1) bestTaskSchedulingMap.
32:
33: if (calibrationSummary is empty and calibratingMode is on) then
34:   bestTaskSchedulingMap  $\leftarrow$  defaultContextConfiguration
35:   return bestTaskSchedulingMap
36: end if
37:
38: bestTime  $\leftarrow$  positive infinity value
39: ntt  $\leftarrow$  number of different task types
40: avgLengthOfWorkers[workers][ntt]  $\leftarrow$  info from the calibrationSummary
41: avgLengthOfContexts[contexts][ntt]  $\leftarrow$  info from the calibrationSummary
42: favWorkersByTaskType[ntt][workers]  $\leftarrow$  info from the calibrationSummary
43:
44: for each context configuration on contextsConfigurations[P] do
45:
46:   taskDistribution  $\leftarrow$  create greed initial solution
47:   estimationInputAndExecution  $\leftarrow$  estimateTransferInputAndExecutionLength()  $\triangleright$  function
    details on Procedure 2
48:
49:   while (estimationImproved is true) do
50:     simulate move tasks from context with longest duration to other contexts
51:     calculates if estimationImproved
52:     move the task and update time if the movement estimationImproved is true
53:   end while
54:
55:   estimationOutput  $\leftarrow$  estimateContextConfigTransferOutputLength()  $\triangleright$  function details on
    Procedure 3
56:
57:   currentTotalEstimation  $\leftarrow$  estimationInputAndExecution + estimationOutput
58:
59:   if (currentTotalEstimation < bestTime) then
60:     update the bestTime, bestContextConfigurationIndex and bestTaskDistribution
61:   end if
62:
63: end for
64:
65: bestTaskSchedulingMap  $\leftarrow$  best context configuration
66:
67: return bestTaskSchedulingMap

```

The input of the framework on **line 2** is detailed as follows:

1. ***tasks[N]*** - Vector of N tasks that will be submitted to the scheduler to execute on the processing units. Each task type has a CPU and GPU implementation of its function type to run in a transparent fashion on the processor it is assigned. Each task has also information about its parameters and data handles as described in Section 2.3.
2. ***calibrationTasks[M]*** - Vector of M tasks that will be used to generate historical data of executions for the framework. It contains distinct tasks for each existent type of tasks on vector *tasks[N]*. It is used for calibration since it discovers the general performance of each worker for each different type of tasks.
3. ***machineConfigurations*** - Stores information about the hardware where the tasks will execute: number of CPU cores, number of GPUs and memory available on each processor.
4. ***schedulerPolicies[S]*** - Vector of possible scheduling policies, according to the ones described in Subsection 2.3.2.
5. ***calibratingMode*** - A boolean variable which defines if the framework is going to have a historical of executions or not. Having a historical of executions helps the framework to know the performance of each worker which calibrates itself.

The output of the framework on **line 3** is empty since nothing is returned. The desired result of its execution, which is the optimization of the overall execution time of the tasks, occurs inside the main function procedure defined on line 1, consequently, there is no need to return.

At **line 5**, all produced tasks that will be executed are stored on *tasks[N]*. On **line 6**, calibrating tasks are stored on *calibrationTasks[M]* vector.

On **line 7**, enqueues all produced tasks stored in *tasks[N]* to the main queue of tasks. This vector of tasks is also submitted for analysis by the *analyzeTasks()* method on **line 8**, which updates the abstract data type who stores information and creates a summary about how many tasks of each type and its configurations.

The *machineConfigurations* stores information about the hardware which is provided by the current available hardware on **line 9**. It collects information about the

available hardware on the machine: number of CPU cores and GPUs available. The programmer does not need to know how many nodes or devices are on the machine. The acquisition of this data is transparent and a whole concern of the framework. Its capabilities as number of cores, number of SMs and available RAM is available through the use of NVIDIA Management Library (NVML) [53].

On **line 10**, manually build possible scheduling contexts configurations based on the *machineConfigurations* variable are stored on the vector of *contextsConfigurations*[*P*]. This module is key to the success of the framework. It is built based on the idea of scheduling contexts defined on Subsection 2.3.1 and available on StarPU [39]. Scheduling contexts are abstract sets of workers that allow the programmer to control the assignment of concurrent parallel tasks to computational resources. The main idea behind this concept is to minimize interference between the executions of multiple parallel kernels/tasks by partitioning the underlying pool of workers using scheduling contexts.

On **line 11**, all possible scheduling policies, according to the ones described in Subsection 2.3.2 are stored on the *schedulerPolicies*[*S*] vector. These policies are applied individually to each scheduling context. In the framework, the same scheduling policy is applied for all scheduling contexts inside a context configuration, which by definition contains one or more scheduling contexts.

The *historicalOfExecutions* variable on **line 12**, stores information returned from the function `loadHistoricalOfExecutions()` that tries to load information from the file pointer. If there is no historical available to be loaded, it will point to null.

On **line 14**, is checked if the historical of executions points to a null value and if the calibrating mode is activated. If this happens, the *calibratingSummary* points to empty on **line 15**, which is responsible for providing summarized calibrating performance information about how each task type behaves for each processor or worker. This information is obtained through analysis of the *historicalOfExecutions*.

Since no historical data is available and calibration mode is activated, *calibrationTasks*[*M*] receives the role of normal *tasks*[*N*] on **line 16**, this way it is possible to make the first run and generate historical data for knowing the performance of each worker. Notice that both sets of tasks are different, but this attribution is just made to the *calibrationTasks*[*M*] to follow the standard flow of *tasks*[*N*]. When no historical information is available, the priority becomes to generate calibration data. Optimizing the execution and minimizing the overall execution time of the independent tasks is just the main objective and priority when there is an historical of executions available.

In the case there is available historical of executions as on **line 18**, this data is analyzed and attributed to the results of the historical analysis to the *calibrationSummary* variable. This analysis is based on offline calibration since data obtained from relevant previous historical execution is loaded, information gathered in the current machine using real runs. The outcome of this strategy essentially relies on how accurately the calibration reflects about what happens during actual executions, for every new platform the calibration must be repeated [34].

The task distribution engine main core on **line 21**, will decide who is going to run each task and which scheduling context each of them will be assigned. It also estimates an execution time for each pre-defined possible settings of different scheduling contexts configurations, so the framework can decide which one will minimize the overall execution time of the tasks. It is the most relevant and complex part of the framework, since it puts all information together, makes the analysis and the task mapping final decision.

The procedure Task Distribution Engine Main Core on **line 29** receives as input the same parameters of the framework main function plus the *calibrationSummary* variable which provides processed historical information about the historical data. The output of the method is the *bestTaskSchedulingMap* variable which contains information about the best scheduling context configuration that optimizes the overall execution time of the independent tasks, task distribution among scheduling contexts and scheduling policy.

In the case where there is no processed information about historical data on the calibration summary variable and the application is in calibration mode on **line 33**, the best task scheduling map is updated with a default scheduling context configuration on **line 34**, detailed on the following paragraphs.

Since no historical data is available and the framework is not trained, the primary objective becomes to execute the *calibrationTasks*[*M*] vector to generate calibration data. At this time, there is no point to search for the best scheduling context configuration, so scheduling contexts with only one worker each are created and equally distribute all the tasks on the *calibrationTasks*[*M*] vector to these scheduling contexts. The creation of one worker scheduling contexts helps reduce interference and collect data more precisely. The scheduler set for popping tasks from each scheduling context is set to equally distribute tasks, which increases the chances for each worker to collect performance information about how different kinds of tasks behave when executed on different workers, and obtain helpful performance data about the execution itself. After this, a default configuration is set and the framework flow returns to line 21.

On **line 38** the *bestTime* variable stores the estimation of the overall execution best time, it receives the maximum possible positive value so any generated estimation will be better than this value. The *ntt* variable on **line 39** stores the number of different task types on the current execution.

The *avgLengthOfWorkers[workers][ntt]* matrix on **line 40** stores information about the average time each worker needs to receive input data and execute a certain task type. This information is obtained using the *retrieveAVGLengthOfWorkers()* which parses the *calibrationSummary* variable.

The *avgLengthOfContexts[contexts][ntt]* matrix on **line 41** stores information about an estimation of the average time each scheduling context, which is a group of workers, needs to receive input data and execute a certain task type on average. This information is obtained using the *retrieveAVGLengthOfContexts()* function which parses the *calibrationSummary* variable. For understanding the details of how the transference output data time is estimated, refer to Subsection 4.2.1.3.

The *favWorkersByTaskType[ntt][workers]* matrix on **line 42** stores information about the favorite worker or processor for each task type.

On the for statement of **line 44** it is initiated the search for the best scheduling context configuration that will optimize the overall execution time of the independent tasks, it scans every scheduling context configuration on the *contextsConfigurations[P]* vector from 1 to P, where P is the number of possible scheduling context configurations, and each configuration has virtual groups of workers or contexts.

Before starting each search iteration, it is necessary to create an initial solution for the problem. Each task stored in the *tasks[N]* vector must be distributed among scheduling contexts. An initial greedy solution is generated on **line 46**. The method simply chooses a scheduling context that execute each task in the shortest time. The previously defined *avgLengthOfContexts* matrix is used to find the necessary information and generate the initial solution, making it possible to work towards the improvement of this solution.

Now, it is necessary to estimate how much time all tasks in the initial solution will need to transfer their input and execute. On **line 47**, this result is achieved by executing the *estimateTransferInputAndExecutionLength()* procedure which uses the *TaskDistribution* variable and the *avgLengthOfContexts* matrix to discover this estimation which is stored on the *estimationInputAndExecution* variable. For understanding the details on how to estimate the input data transference and execution time, refer to Subsection 4.2.1.2 on

Procedure 2.

Naturally, the creation of a greedy solution has a high chance of generating an unbalanced solution. If a scheduling context with only the fastest workers for every type of tasks was created, the other scheduling contexts would not receive any tasks. An idle scheduling contexts is not a desired behavior, since all workers on other scheduling contexts would stay in waiting status while they should be solving tasks to help minimize the overall execution time and reduce the overload of the best scheduling context.

This unbalancement is solved on the statements on **lines 49-53**, by systematically trying to move tasks from the scheduling context with the longest estimation to the scheduling context with the shortest estimation, until no improvements are possible to be made. These moves will cause the scheduling contexts to reach ideal hypothetical balance, having approximately the same theoretical estimation on each scheduling context after a few iterations. These moves are evaluated in constant time since it is possible know the effect on time by adding or removing tasks to any of the scheduling contexts. It is simply not necessary to call the estimation method which would need to scan the whole *TaskDistribution* vector.

At this point, it is not possible to reduce the theoretical duration of the scheduling contexts by simply moving tasks from one scheduling context to another, since they already have the minimum duration. Now, an estimation time for transferring the input data and execution of these tasks is available. Following, it is necessary to estimate the output transferring time using the function `estimateContextConfigTransferOutputLength()` on **line 55**. It is necessary to estimate this transfer time since transferring data through the bus takes some considerable time. The transfer time is estimated through linear approximation using information from the offline calibration phase. For understanding the details of how it is estimated the transference output data time, refer to Subsection 4.2.1.3 on Procedure 3.

On **line 57**, the total estimation of the overall execution time of the tasks is calculated by adding the input and execution estimation to the output estimation, and attributing this value to the total estimation. Finishing the for loop, it is evaluated from **lines 59-61**, if the current estimation is better than the global best estimation time and save the best settings.

The for loop iterates for every possible scheduling context configuration, and the process repeats until the scheduling context configuration with smallest estimation of the overall execution time is found. The results are updated on the *bestTaskSchedulingMap*

variable which stores the best task distribution among scheduling contexts, best scheduling context configuration and the used scheduling policy. After the value on the best task scheduling map is updated, the method `taskDistributionEngineMainCore()` returns this object on **line 67**, and the execution flow returns to the main function on line 21.

After the execution of the `taskDistributionEngineMainCore()` the *bestTaskSchedulingMap* variable on **line 21** stores the configuration that optimizes the overall execution time of the tasks on the *tasks[N]*. Finally, these tasks are submitted for execution on **line 22** together with its parameters, the function knows exactly to what previously established scheduling context to submit each task on the *tasks[N]* vector. The function scans the vector and dispatch each task for the mapping stored on the *bestTaskSchedulingMap* variable.

Following, it is time for synchronizing all scheduling contexts and wait for them to finish on **line 23**.

After all tasks have finished executing, now it is time to transfer the output results data back to the CPU and keep memory coherence. In the case of GPUs, an explicit transfer must be done. All automatically allocated buffers needs to be freed, and a valid copy of the data is copied back to keep coherence of the memory on the CPU, which was possible invalid. These results are transferred on **line 24**. This approach is similar to Pandit et al. [58], where these transfers are done only when data is modified on the GPU. For tasks that have executed completely on the CPU, this data transfer is generally not required.

In case the duration of scheduling contexts are unbalanced and not behaving accordingly or deteriorating, and scheduling contexts are ending with a big enough difference in time, it is also possible to call the reevaluate the strategy on **line 25**, which can optionally reset the calibration data available, by deleting its historical data, forcing to generate new historical data for the proposed framework if necessary with an adequate set of tasks. This situation could happen if the calibration data differ substantially in comparison to the actual tasks data set used in subsequent runs.

On **lines 26-27**, it is possible to update or not the data type who stores the historical of executions and optionally write this data on disk.

Finally, the execution is terminated on **line 28**, where the memory of CPU and GPU will be freed and correctly shutdown the libraries to end the framework execution.

In this Chapter, the framework is explained in a more generic way and explained

apart from implementation specifics. Some clarifications on the framework are presented on Subsection 4.2.1. On Chapter 5, application and software implementation details are presented in depth.

4.2.1 Estimation of input/output transference and execution time

In this subsection, it is complemented the framework explained on Chapter 4 by showing how executions length and input/output transference time are estimated. Due to the nature of the tasks model, the process is split into two parts according to Figure 4.2. In the first part, the input data transference and execution time is estimated, and in the second, the duration of the output data transference time is estimated in Subsection 4.2.1.3.

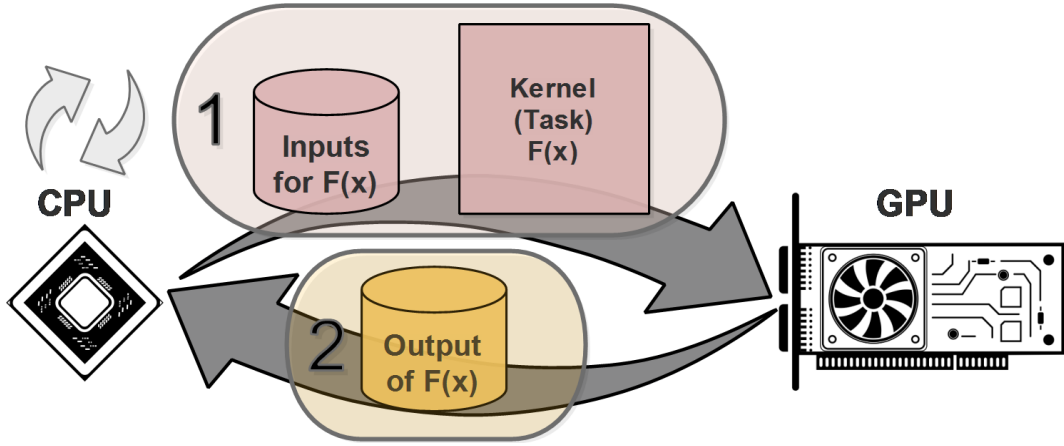


Figure 4.2: Measured times and estimations, split into two phases.

Before formalizing in a sequence of steps how to estimate the input data transference and execution time, it will be explained the intuitive idea about how it works, using an example. These estimations will be presented for a worker on Subsection 4.2.1.1 and for a scheduling context on Subsection 4.2.1.2. It is necessary two separate methods, since when a task is submitted to a scheduling context, there is no guarantee about who is going to execute that particular task.

4.2.1.1 Input data transference and execution estimation for a task submitted to a worker

After the calibration phase performs, the information in Table 4.1 is available for the framework in the historical database.

Table 4.1: Segment of the historical of executions, extracted from the historical database.

#	Worker	Task Type	Length (μs)
1	3	1	242
2	2	3	387
3	1	2	95
4	1	1	19
5	2	2	1795
6	3	2	1791
7	3	3	385
8	2	1	242
9	1	3	35

The information obtained from the calibration phase is valuable for the success of the framework. From lines five and six in Table 4.1 is possible to notice that workers two and three perform slowly for tasks of type two, with lengths of 1795 and 1791 microseconds (μs) respectively. This performance is considerably slow if compared to the performance of worker one, with a duration of 95 μs .

No private information of what the tasks are executing or using as parameters are utilized to write the log, and the framework only knows the type of the task that runs on each worker. The framework is “blind” on purpose, performing only black box measurements which reduce dependency on tasks information. Naturally, this approach decreases the amount, accuracy, and quality of information that is possible to extract from the tasks but makes the framework generic, feasible and appropriate for implementation since interdependency associated to the tasks is reduced.

When performing estimation analysis, there is a critical trade-off between estimate with enough precision and time spent during analysis. Most of the times, the evaluation time needs to be significantly faster than the time it will probably save optimizing the overall execution time of the tasks.

The history of executions saved in the historical database is read when the framework is initialized on line 12 of the Framework 1, and the averages of the history of execution are summarized and loaded on memory on line 18. Each cell in Table 4.2 represents an example of hypothetical average times in microseconds for the input data transference and execution of different task types.

Table 4.2: Average of executions measured times and data transference of different tasks types per workers.

	Average Tasks Execution Time in μs		
Workers	Type 1	Type 2	Type 3
#1	10	200	30
#2	15	100	15
#3	30	50	10
#4	30	150	60
#5	20	50	60
#6	10	25	60

This performance information is crucial for the framework since it provides a good hint about how every worker performs and behaves on average for each type of task on the current machine.

4.2.1.2 Input data transference and execution estimation for a task in a scheduling context

The average of executions time obtained in Table 4.2 is necessary for the calculation of the input data transference and execution estimation for a task in a scheduling context. It is an artificial metric that represents the average time each task type needs to transfer input data and execute in a scheduling context.

When a task is dispatched to a scheduling context to execute, there is no guarantee of which worker will receive and execute this task, then it is required to calculate a metric that somehow emulates the approximate time. First will be presented the intuitive idea through an example and after the definition the framework.

From the data obtained from Table 4.2 a scheduling context which contains workers from 1 to 3, will be created. Figure 4.3 represents how many tasks of type two worker two and three would solve if compared simultaneously to the slowest one in the group, in this case, worker one.

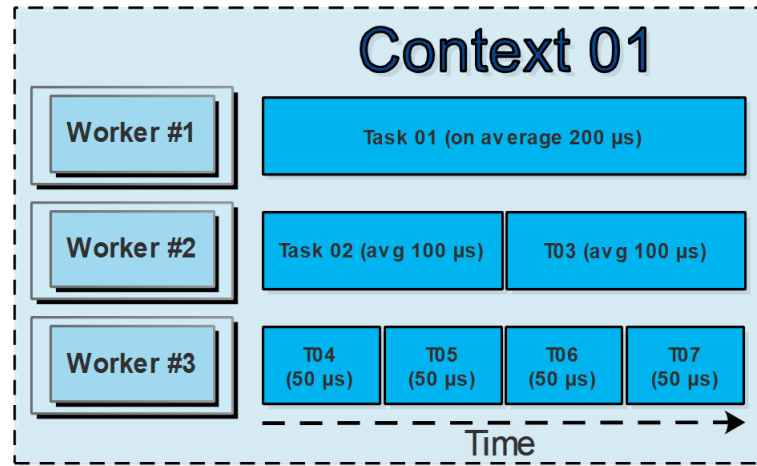


Figure 4.3: Execution of same type tasks over time on a scheduling context with different workers with different performances.

In 200 microseconds, worker one would have solved one task, while worker two would have solved two tasks and worker three would have solved four tasks, with a total of seven tasks. On this scenario, the average length per task is $200 \mu s / 7$ or approximately $29 \mu s$, which is a reasonable value since, with the help of other workers, the average should be at least smaller than the fastest one which is Worker 3 with $50 \mu s$ on average. Repeating this process for all combinations of scheduling contexts and types of tasks, Table 4.3 is achieved.

Table 4.3: Average of executions measured times of different tasks types per scheduling context.

	AVG Scheduling Context Execution Time in μs		
Context	Type A	Type B	Type C
#1	5	29	5
#2	5	15	20

Simply calculate the average time of all workers that compose the scheduling context to obtain the average length of a scheduling context for a particular task type is not an option since each worker solves different numbers of tasks on the same amount of time, which can lead to workers to remain idle for a certain length of time [1].

On Procedure 2, it is exhibited the procedure to calculate the average lengths of scheduling contexts, which corresponds to line 41 on Procedure 1, the framework main procedure.

Procedure 2 AVG Lengths of Scheduling Contexts Calculation Steps

```

1: Procedure avgLengthOfContextsPerTaskType
2: Input: (1) avgLength[numberOfWorkers][numberOfTaskTypes]
3: Output: (1) avgLengthPerContext[numberOfContexts][numberOfTaskTypes].
4:
5: for  $c \rightarrow 0$  to numberOfContexts do ▷ for each context
6:   for  $t \rightarrow 0$  to numberOfTaskTypes do ▷ for each task type
7:     sumOfUnits  $\leftarrow 0$  ▷ accumulate how many tasks are solved in that time
8:     smallest  $\leftarrow$  find the smallest average for task type t on context c
9:     longest  $\leftarrow$  find the longest average for task type t on context c
10:    for  $w \rightarrow 0$  to workers do ▷ for each possible worker
11:      if (workerBelongsToContext(w, c) = true) then ▷ checks if worker w belongs to context c
12:        sumOfUnits  $\leftarrow$  sumOfUnits + (avgLength[w][t] / smallest)
13:      end if
14:    end for
15:    avgLengthPerContextPerTaskType[c][t] = longest / sumOfUnits
16:  end for
17: end for
18:
19: return avgLengthOfContextsPerTaskType[ ][ ]

```

4.2.1.3 Output data transference estimation

The first necessary information to estimate the output data length is the number of tasks that will execute on each GPU, since explicit transference calls are just made from the device to host in this case. CPUs do not need to communicate through the bus because all the CPU cores are on the chip.

Since it is possible theoretically estimate the length of a scheduling context, and is also known how many tasks and which types of tasks will execute on that group of processing units. Using the average time each worker on this group needs to run a task, it is viable to approximately estimate how many tasks each GPU will run on that group of processing units.

In the case the scheduling context has only GPUs, there is no need to calculate how many tasks will execute on the device, since all mapped tasks will for sure execute on GPUs. In the case the scheduling context has no GPUs, there is also no need to calculate since no data is transferred through the bus. In the mixed scenarios, the calculations are required.

During the calibration phase, the initial overhead of this function is measured, which

is when no tasks are sent to the GPUs. Since is known the initial time the function needs to execute, how many tasks and which types of tasks will execute on each GPU, and the average transfer time for each type of task, it is viable to estimate the output data transfer time with good enough precision. Procedure 3 corresponds to the function `estimateContextConfigTransferOutputLength()` on line 55 on Framework 1.

Procedure 3 Output Data Transference Estimation

```

1: Procedure outputDataTransferenceEstimation
2: Input: (1) estNumberOfTasksPerWorker[w][t], (2) avgWorkerOutputDataTimePerTaskType[w][t]
3: Output: (1) longestOutputDataEstimation
4:
5: for w → 0 to workers do
6:   if (isWorkerGPU(w) = true) then
7:     outputDataEst[w] = initialOverhead                                ▷ measured on calibration phase
8:   else
9:     outputDataEst[w] = 0                                             ▷ just need the vector for GPU Workers
10:  end if
11: end for
12:
13: for c → 0 to numberOfContexts do
14:   for t → 0 to numberOfTaskTypes do
15:     for w → 0 to workers do
16:       if ((isWorkerGPU(w) = true) and (workerBelongsToContext(w, c) = true)) then
17:         outputDataEst[w] ← outputDataEst[w] + ( estNumberOfTasksPerWorker[w][t] *
18:           avgWorkerOutputDataTimePerTaskType[w][t])
19:       end if
20:     end for
21:   end for
22:
23: longestOutputDataEstimation ← max(outputDataEst[w])
24:
25: return longestOutputDataEstimation

```

4.3 On the scalability of the framework

This framework was initially designed to work on a multi-CPU and multi-GPU machine and can be easily escalated for a more significant number of heterogeneous machines. The basic idea of the framework remains the same, but a few modifications should be required. The main difference is that the vector of independent tasks must be split among these

machines according to its computing power, and after all machines finish its execution, the results must be transferred and centralized. For the case where machines have many multi-CPU and multi-GPU, the same framework applies, but the number of Scheduling Contexts or virtual group of processors configurations, must be limited and manually defined as on the case on Table 5.1.

4.4 Conclusion

This Chapter presented the conceptual design, definition and the details in depth of a novel framework for task distribution of independent tasks in multi-CPU and multi-GPU systems. It uses the combination of virtual groups of processors, scheduling policies and estimations to automatically achieve the optimization of the overall execution time of independent tasks, using just a small fraction of the time required to execute the tasks. It allows developers to automatically take advantage and extract more performance from the same available hardware.

Chapter 5

Implementation

This chapter describes and details the implementation based on the framework presented on Chapter 4. First, it is presented the supporting tools necessary to implement the framework, after it is explained in-depth how it was measured the time intervals. Finally, it is presented the additional frameworks, libraries and technologies used in this phase.

5.1 Supporting tools

On this Section are presented all the auxiliary supporting tools used for the implementation of the framework and generating the experimental results.

5.1.1 Synthetic task generator

This is an auxiliary module that is not present on the framework, it synthetically creates tasks of different sizes and with a diversity of inputs for each type of task loaded on the framework. This module allows changing the tasks configurations easily: number, types and sizes. It is also possible to effortlessly instantiate a massive number of distinct tasks on the assemble tasks module. This module is detailed on Subsection 5.1.2.

Figure 5.1 presents the conceptual idea of synthetically generating tasks of different types and sizes to generate a historical of execution for the framework model. Tasks are popped from the main task queue and equally distributed among scheduling contexts to generate historical and performance data that will represent the relative performance of each core.

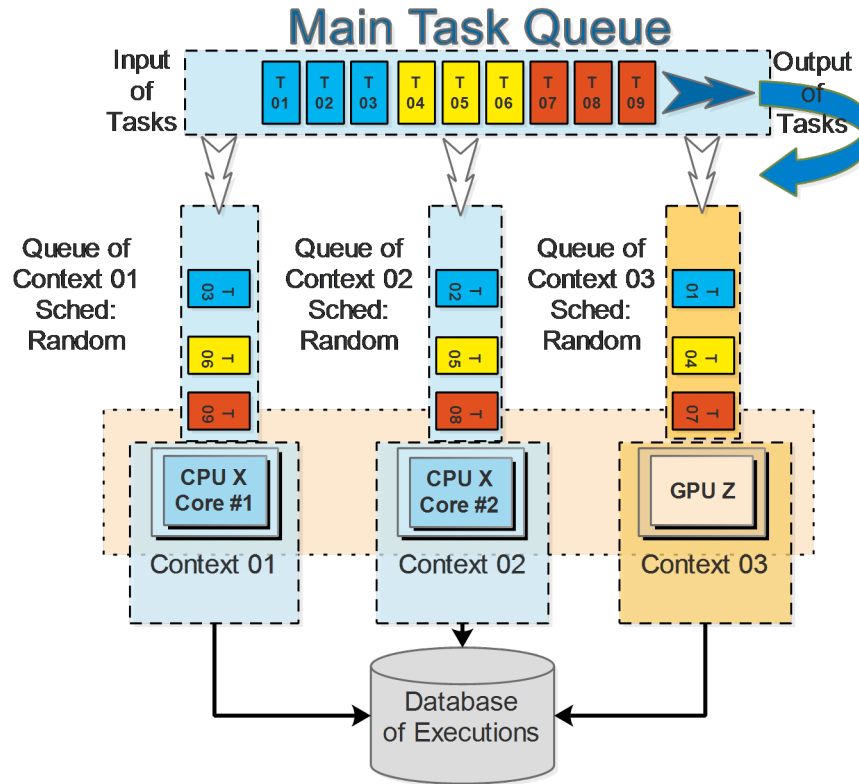


Figure 5.1: Phase where the historical of executions of the framework is generated.

5.1.2 Benchmarks

To evaluate the effectiveness of the framework, it was taken the strategic decision to benchmark the application building a StarPU-based synthetic task generator. It has four different kernels or task types, with hand-written C++ and CUDA versions for each task type to use as a benchmark baseline. The chosen benchmarks, detailed on this Section, contain frequently used operations in scientific applications, becoming an important choice to validate the framework.

The use of Rodinia [17], a well-known CPU and GPU benchmark suite for heterogeneous computing, was considered, but adapting their benchmarks to the StarPU model of tasks, understanding the nature of its kernels, checking the integrity and correctness of the real life algorithms implemented on Rodinia were not feasible. Other well known benchmarks also present the same limitation for the proposed validation.

Checking the integrity of the results of synthetic operations kernels is more practical and straightforward approach than choosing and adapting other benchmarks. Simple benchmarks have the advantage of allowing the outputs to be easily verified and permitting the research to focus on its subject, with a low level tuning and adjustments of desired

parameters.

Although it is an important active research topic, generating and porting efficient kernels for the CPU and GPU is beyond the scope of this thesis, since in most case it is required a specific domain knowledge. Contrary to many research works that focus on the accelerator performance, this work focus on efficient strategies and coordination of the execution in heterogeneous computing environments. The main interest of this work is in the correctness of the executed benchmark code and accuracy of the estimation than to the performance of the tasks inside a worker. The framework treat every task as a black box, so virtually any benchmarks that follow the StarPU task design based on atomic operations should work fine as a benchmark.

The scope of this thesis is on producing strategies to minimize overall execution time of the tasks, going through the challenges of porting benchmarks to StarPU format would be time-consuming and cause the research to lose focus. The development of a synthetic task generator allowed a higher level of control over many scenarios, creating the possibility to easily manipulate test scenarios, check results integrity and possible errors.

In the experiment, the StarPU benchmarks are implemented, with functions in C++ that executes on the CPU and in CUDA which executes on the GPU. Each implementation produces the same result, but each one executes in a different architecture. These implementations are associated to the starPU codelet described on Section 2.3. The size of the tasks is dynamic and has multiple sizes and different inputs for each task. The four implemented benchmarks are described below:

1. **Sum of vectors** - Sequential and parallel addition of two float arrays of same sizes to a third one who stores the result, as illustrated in Figure 5.2. According to Lee et al. [32] these benchmarks are extremely memory-bound data-parallel kernels, with chance to execute on CPUs or GPUs depending on the size.

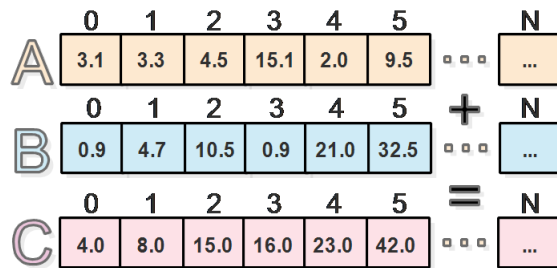


Figure 5.2: Sum of vectors benchmark example.

2. **Subtraction of vectors** - Sequential and parallel implementation of a subtraction

of two float arrays of same sizes to a third one who stores the result as in Figure 5.3.

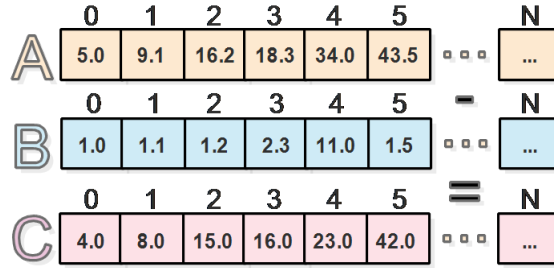


Figure 5.3: Subtraction of vectors benchmark example.

3. **Matrix multiplication** - Sequential and parallel implementation of a multiplication of two square matrices of float to a third one who stores the result, as illustrated in Figure 5.4.

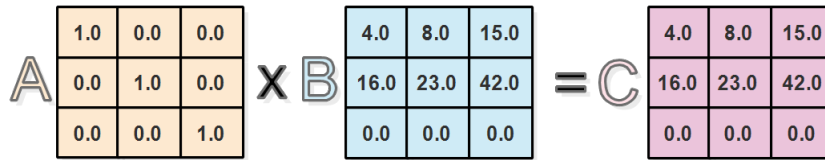


Figure 5.4: Multiplication of two squared matrices benchmark example.

4. **Multiplication of an array by a scalar** - Sequential and parallel multiplication of each position of a float array by a constant scalar, writing the results in a resultant array as illustrated in Figure 5.5.

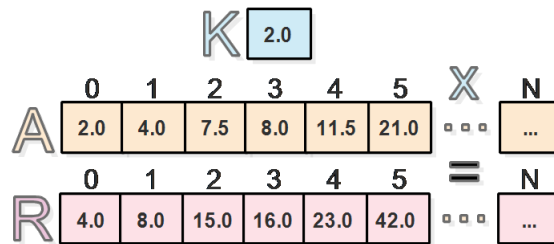


Figure 5.5: Multiplication of a vector by a scalar benchmark example.

5.1.3 Scheduling contexts generator

This module generates scheduling contexts configurations candidates that are going for evaluation through estimation of overall execution time in the framework. Table 5.1 lists all considered configurations of scheduling contexts for the platform specification detailed on Section 6.1.

Table 5.1: List of all considered configurations of scheduling contexts in the implementation.

	1		2		3		4		5		6	
CFG	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
1	4	0	-	-	-	-	-	-	-	-	-	-
2	0	2	-	-	-	-	-	-	-	-	-	-
3	4	2	-	-	-	-	-	-	-	-	-	-
4	0	2	4	0	-	-	-	-	-	-	-	-
5	2	1	2	1	-	-	-	-	-	-	-	-
6	2	2	2	0	-	-	-	-	-	-	-	-
7	0	1	0	1	4	0	-	-	-	-	-	-
8	1	1	1	1	2	0	-	-	-	-	-	-
9	0	2	2	0	2	0	-	-	-	-	-	-
10	2	2	1	0	1	0	-	-	-	-	-	-
11	1	1	1	1	1	0	1	0	-	-	-	-
12	1	2	1	0	1	0	1	0	-	-	-	-
13	0	2	1	0	1	0	1	0	1	0	-	-
14	1	1	0	1	1	0	1	0	1	0	-	-
15	0	1	0	1	1	0	1	0	1	0	1	0

Every line on Table 5.1 represents a possible configuration for the workers on the platform. The main idea behind the configurations is to create different possible context configurations, where each context inside a configuration has at least one processor or worker, and the maximum of all processors and workers available on the machine. Groups with mixed workers types, having CPUs and GPUs simultaneously are also created.

The testing machine comprehends of two GPUs and four CPU cores available. For instance, on configuration seven of Table 5.1, there are three virtual groups of processors, two groups with one GPU each and another group with four CPU cores. Every scheduling context is a disjoint group, and each available worker must be part of one and only one scheduling context.

15 different configurations possibilities are created to experiment and test its efficiency. These configurations are in ascending order, the smallest configuration number has the smallest count of scheduling contexts, and the highest one, the biggest count.

In configuration three, there is only one scheduling context composed of all available workers, while on configuration 15 there are six scheduling contexts, where each one consists of only one worker. The others configurations are variations in the number of scheduling contexts and mixed scenarios of different workers combinations per scheduling

context.

5.1.4 Execution validation

A developed auxiliary CPU function can also optionally be activated to validate the execution and check the correctness and integrity of the processed execution, checking for wrong results and data inconsistencies.

Another additional software was developed to parse the logs generated by each execution. This program uses log data for post-analysis and generating execution reports and sheets.

5.2 Measuring times

In this section, it is detailed how it was measured the execution and input/output transference time. The measured times are used as a baseline to compare the accuracy of the estimations.

Due to the nature of the tasks model and implementation concerns, the process is split into two parts according to Figure 4.2. First, times and estimations of input data transference and execution are measured, and secondly, times of the output data transference are measured.

5.2.1 Input data transference and execution time

In this Subsection, the process of measuring times and estimating the input data transference and execution time is detailed. It is also justified the implementation issues that prevent obtaining both data separately.

Before understanding how it was obtained the measured times of tasks, it is essential to comprehend how StarPU works; the NVIDIA Visual Profiler [54] timeline was used to look inside the pipeline.

StarPU only provides profiling data for each task when it starts and finish. Transference time for data parameters on each particular task it not logged individually. This implementation issue was bypassed by measuring the end time of execution for two neighbor tasks that execute on the same worker.

In the execution timeline in Figure 5.6, StarPU would only give information about

the duration of each task. On task one, the duration would be represented by interval $[T1, T2]$, and for task two should be interval $[T3, T4]$. The issue is that there is still no clue about the elapsed time from interval $[T0, T1]$ or $[T2, T3]$, which is the time that represents the necessary time to transfer data parameters to the tasks before the execution begins.

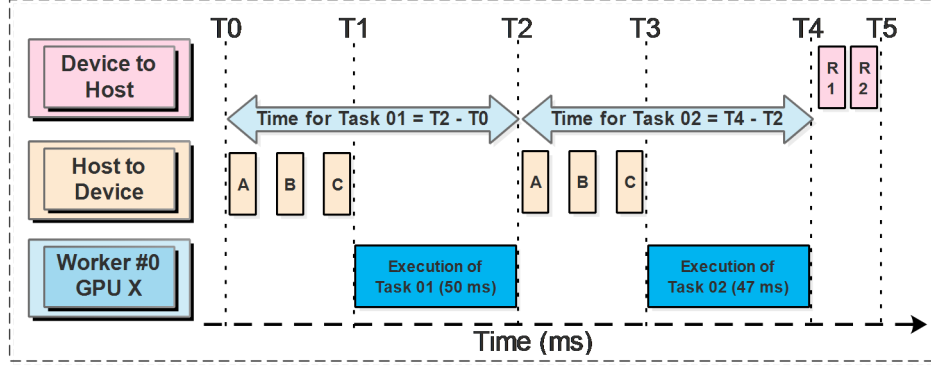


Figure 5.6: Tasks execution timeline, based on observations using the NVIDIA Visual Profiler [54].

A task needs three steps to execute: transfer data for receiving its execution parameters, execute and transfer back its results. It is not possible to measure the input/output transfer time due to framework design, the Application Programming Interface (API) simply does not provide this data. One solution is to measure the interval between the end of two tasks. Intervals $[T0, T2]$ and $[T2, T4]$ give us the sum of input transfer time plus execution time of the correspondent task.

This approach has some drawbacks: one of them is not having the separated times of input data and execution, but it is possibly the best practical solution possible to find since the application is being built on top of third party libraries, designed for other purposes and objectives. On the other side, it is also an interesting approach since it will leave us with almost no gaps in the timeline to measure, which is hard to estimate.

The only data still not present is the time required to transfer back the results, as an output data. The details of the output data transference time are explained on Subsection 5.2.2.

5.2.2 Output data transference time

The transference of output data occurs when all tasks finish executing. The synchronization unregister function on StarPU is called and the data is copied back, during the interval $[T4, T5]$, shown at Figure 5.7. The DSM on StarPU is then called, which updates

memory and keeps it in coherence.

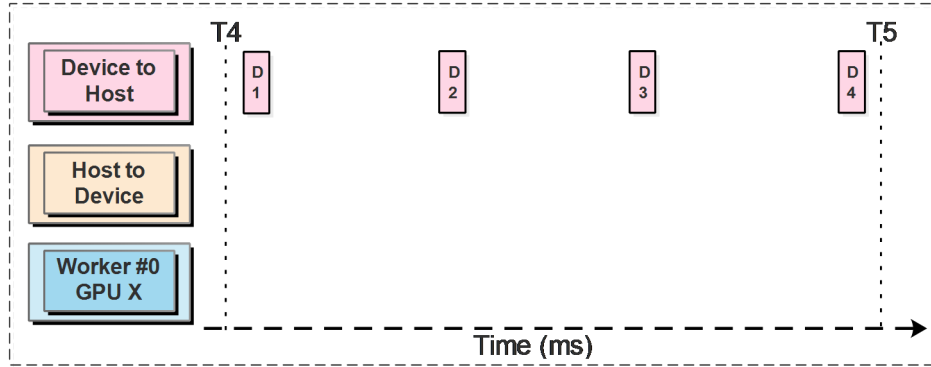


Figure 5.7: Transference of output data viewed on the timeline, based on observations using the NVIDIA Visual Profiler [54].

In Figure 5.7 is possible to observe four data transferences over the interval $[T4, T5]$, which the length of each transference is exaggerated on the timeline for illustration purposes, but what happens are instant peaks of transference of only a few microseconds.

The PCI-Express 3.0 [31] bus can transfer gigabytes per second, and in these experiments, the predominance on the timeline is of empty spaces among the data transferred between system calls since the application is essentially transferring only results. This fact reduces the relevance of how much data is transferred and increases the relevance of the interval among transference calls.

The interval $[T4, T5]$ is measured by simply logging the time before and after calling this function. The total number of tasks the GPU executes corresponds to the number of transference calls in this range. Finally, this way it is possible to register the average transference time for each task type.

5.3 Additional frameworks, libraries and technologies

Some additional relevant technologies used during the implementation phase of the framework.

5.3.1 NVIDIA visual profiler

The NVIDIA Visual Profiler [54] is a cross-platform performance profiling tool that allows developers to have feedback and optimizing CUDA C/C++ applications and is available as part of the CUDA Toolkit [50].

Among the most useful functions of this tool, is the unified CPU and GPU timeline which allows viewing CUDA activity that is happening simultaneously on both CPU and GPU in a unified timeline, making it possible to identify opportunities for performance improvement and understand what is going on inside the GPU. It is also possible to verify CUDA API calls, memory transfers, kernel and CUDA launches.

The NVIDIA visual profiler introduced some delay in the framework code, but these functions are just used during development time to understand the mechanisms of what was occurring during the execution time. In production code, the instrumentation code is deactivated, removing the additional delays.

5.3.2 NVIDIA management library (NVML)

The NVIDIA management library is a C-based API for monitoring and managing various hardware of the NVIDIA GPU devices [53]. It provides direct access to the hardware via queries and commands.

5.3.3 Measuring times during execution

The high precision Chrono C++ Standard Library of C++11 is used to measure time. This library is a collection of classes to work with times and consists of three components: duration, clock and time point.

Every method is defined in the `std::chrono` namespace and requires the inclusion of the `<chrono>` header file. The standard library also provides different useful time units to measure time such as nanoseconds, milliseconds, minutes and even hours. Part of the `<chrono>` library includes support for various clocks such as system clock and a high-resolution clock, being appropriate for providing the timebase in a real-time C++ project [35].

5.4 Conclusion

In this Chapter, it was built a front-end that minimizes the overall execution time of independent tasks seamlessly, based on the framework proposed on Chapter 4. This front-end validates the framework using common scientific operations as benchmarks and testing its efficiency. Manual development of a heterogeneous system requires a lot of effort from the software programmer since it is necessary to take manual care of memory

management, load balancing and synchronization. Most of the time, it is not enough to only use a parallelization framework to obtain performance. One needs to optimize and tune the application to a particular architecture. This front-end, based on the proposed framework, helps the programmer in doing so.

Chapter 6

Experimental Results

This chapter details the description of the platform, software and training settings. It is also presented the experiments and results based on the framework.

6.1 Platform and environmental settings

The experiments were conducted on a dedicated and exclusive machine running Ubuntu 14.04.3 LTS [14] and powered by an Intel i7-4790 @ 3.60 GHz with four cores [30], 24 GB of RAM, with a Seagate Hard Disk of 1 TB Barracuda 3,5", 7200 RPM, 64 MB cache, SATA III and 2 NVIDIA GPUs TITAN X [51]. Each GPU has 3072 cores, 12 GB of RAM, maximum memory bandwidth of 336.5 (GB/sec), and peak single precision floating point performance of 6.6 TFlops and a maximum power consumption of 250W.

6.2 Software settings

The C++ codes were compiled with g++ version 4.8.4 [25] and CUDA source codes with NVCC [52], CUDA version 7.5 [50] and StarPU Runtime 1.1.2 [40]. The tests were performed automatically by a bash script performing ten executions for each configuration. For each execution, it is created a log file with the results and an automated parser in C++ compiles the results.

6.3 Historical of executions

The framework has a historical of executions which is created by previously executing the implemented application tasks of each benchmark type with different sizes until the RAM limit of the machine is reached, totalizing 100K tasks. All the workers and the random scheduler are utilized, with one worker per scheduling context. Once the historical of executions is available before starting the experiments, the subsequent executions are not added to the log data of the historical of executions database; otherwise, the historical should get significantly huge.

6.4 Experiments

The experiments are split and summarized into three parts:

1. **Input data transfer and execution comparison experiment** - The measured time the implemented framework takes to transfer input data and execute the tasks on the experimental platform in comparison with the estimations. The details and results of the experiment are exhibited on Subsection 6.4.1.
2. **Output data transfer comparison experiment** - The measured time the implemented framework takes to transfer output data compared to the estimations. The details and results of the experiment are exhibited on Subsection 6.4.2.
3. **Overall execution comparison experiment** - The measured time the implemented framework takes to complete the overall execution compared to the estimations. This experiment brings together the two experiments on Subsections 6.4.1 and 6.4.2. The details and results of the experiment are exhibited on Subsection 6.4.3.

For each experiment, it is presented all the results combinations of schedulers, scheduling context configurations and number of tasks. The scheduling context configurations range from 1 to 15 and are available on Table 5.1. The possible schedulers set for each scheduling context are eager, WS, LWS and random. The number of tasks experimented in each execution is 25K, 50K and 100K, equally split among the four task types benchmarks and in different sizes. Each test is performed ten times and the average result is computed.

6.4.1 Input data transfer and execution comparison experiment

In this experiment, it is taken into account the accuracy of the implementation to estimate the measured time the framework model takes to transfer input data and execute the tasks on the experimental platform.

In this Subsection of the experiment the **scheduler** is varied for all scheduling contexts inside every configuration of **scheduling contexts from 1 to 15**. It is also analyzed the input data transfer and execution for **25K, 50K and 100K tasks** in finding out which is the best configuration that optimizes the input data transfer and execution time.

6.4.1.1 Eager scheduler vs Scheduling context configurations vs Number of tasks

For **100K tasks**, the input data transfer and execution results and estimations are shown in Figure 6.1. The scheduling context configuration one from Table 5.1, had the worst performance. The measured time was 85009 ms. This configuration has been omitted from the plot in order to not compromise the scale. Scheduling context configuration six was the most efficient, in the measured time and estimation, where both show the smallest values of time for this set of configurations. The average GAP for all configurations is 12%.

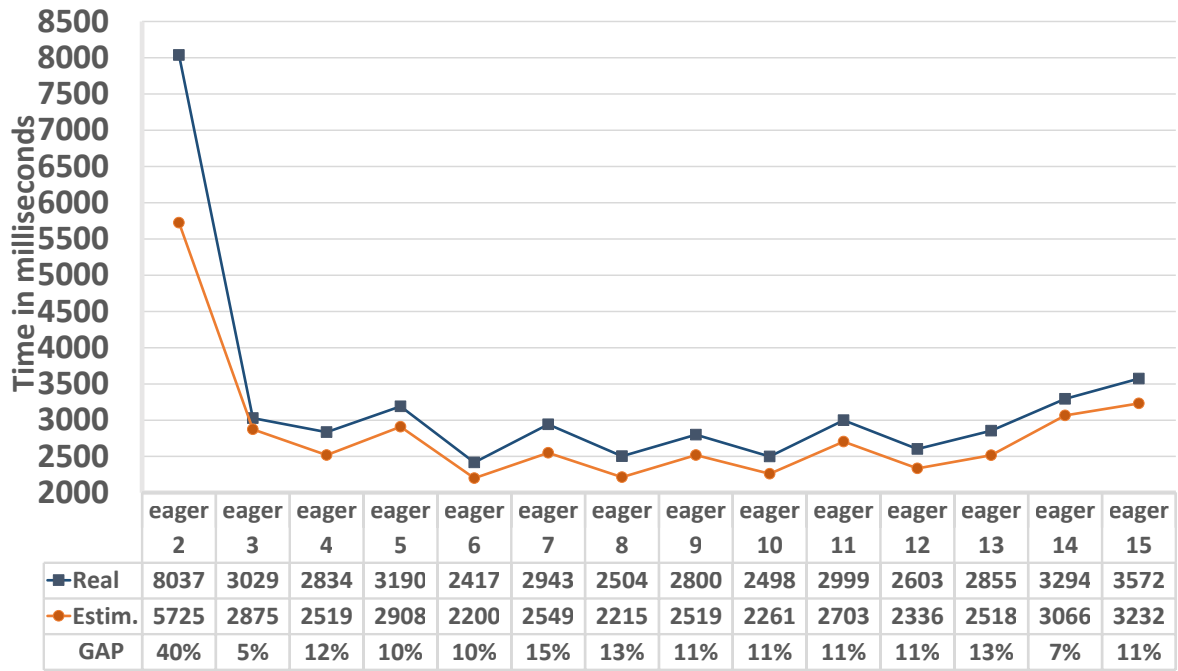


Figure 6.1: A comparison between measured times and its estimation for 100K tasks in every combination of the eager scheduler and scheduling context configurations from 2 to 15 (Table 5.1).

The results of this experiment for the most efficient scheduling context configuration are consistent. Independently of how many tasks are executed, it is possible to determine the most effective scheduling context configuration for receiving input data transfer and execute the tasks efficiently. Even though there is an average GAP from 10% to 12%, the estimation follows the trends of the execution. The results for **25K** and **50K** tasks for this configuration is available on the plots on Figures A.1 and A.2, on Appendix A.

6.4.1.2 LWS scheduler vs Scheduling context configurations vs Number of tasks

For **100K** tasks, the input data transfer and execution results and estimations are shown in Figure 6.2. The scheduling context configuration one from Table 5.1, had the worst performance. The measured time was 85009 ms. This configuration has been omitted from the plot in order to not compromise the scale. Scheduling context configuration six was the most efficient, in the measured time and estimation, where both show the smallest values of time for this set of configurations. The average GAP for all configurations is 20%.

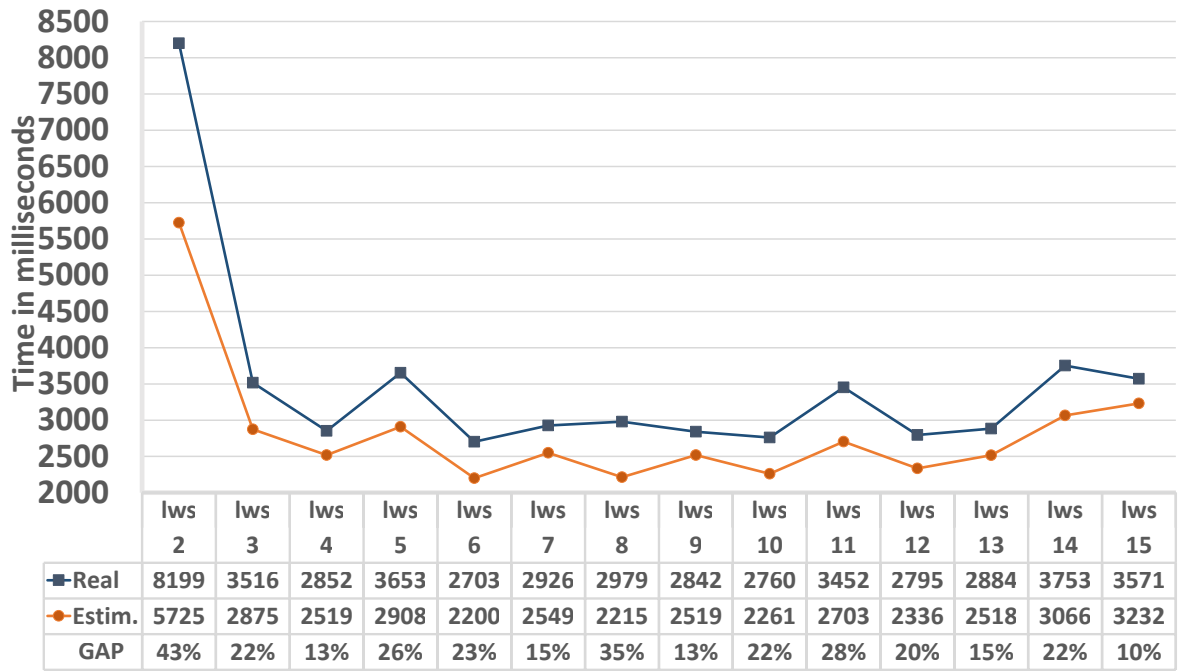


Figure 6.2: A comparison between measured times and its estimation for 100K tasks in every combination of the LWS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).

The results of this experiment for the most efficient scheduling context configuration are consistent. Independently of how many tasks are executed, it is possible to determine the most effective scheduling context configuration for receiving input data transfer and execute the tasks efficiently. Even though the average GAP range from 17% to 20%, the estimation follows the trends of the execution. The results for **25K** and **50K** tasks for this configuration is available on the plots on Figures A.3 and A.4, on Appendix A.

6.4.1.3 WS scheduler vs Scheduling context configurations vs Number of tasks

For **100K** tasks, the input data transfer and execution results and estimations are shown in Figure 6.3. The scheduling context configuration one from Table 5.1, had the worst performance. The measured time was 85038 ms. This configuration has been omitted from the plot in order to not compromise the scale. Scheduling context configuration six was the most efficient, in the measured time and estimation, where both show the smallest values of time for this set of configurations. The average GAP for all configurations is 19%.

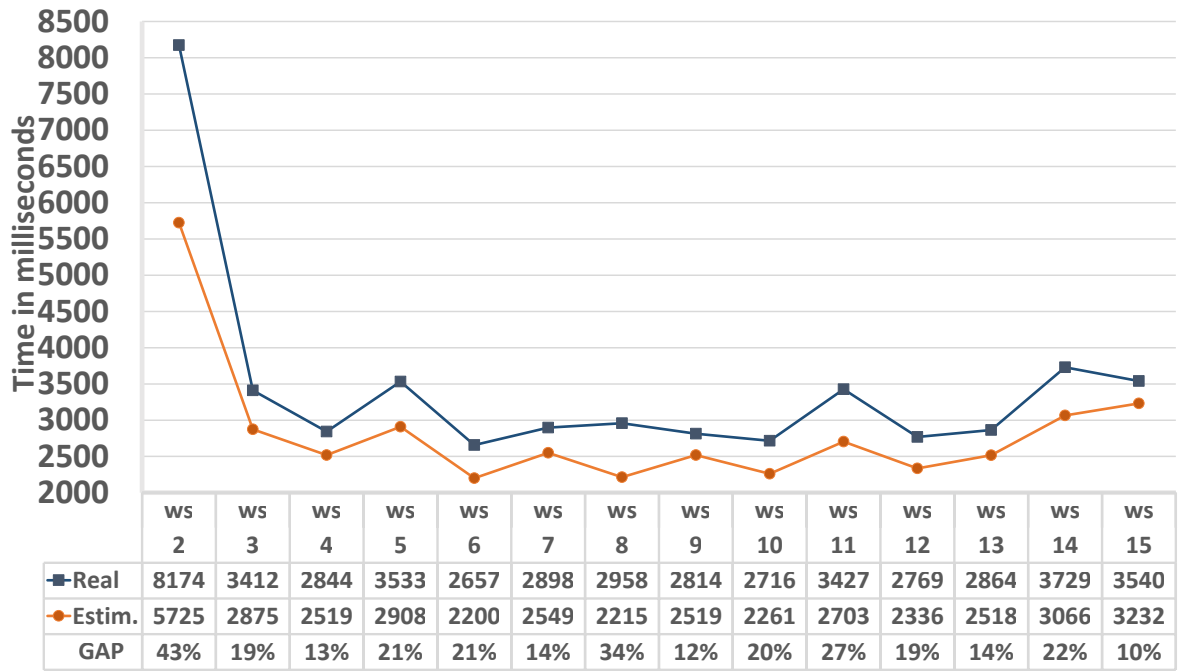


Figure 6.3: A comparison between measured times and its estimation for 100K tasks in every combination of the WS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).

The results of this experiment for the most efficient scheduling context configuration are consistent. Independently of how many tasks are executed, it is possible to determine the most effective scheduling context configuration for receiving input data transfer and execute the tasks efficiently. Even though there is an average GAP ranging from 17% to 19%, the estimation follows the trends of the execution. The results for **25K and 50K tasks** for this configuration is available on the plots on Figures A.5 and A.6, on Appendix A.

6.4.1.4 Random scheduler vs Scheduling context configurations vs Number of tasks

The results of this experiment for the most efficient scheduling context configuration were not consistent for the random scheduler, having GAPs of 303% for 25K tasks, 272% for 50K tasks and 263% for 100K tasks.

6.4.1.5 Conclusion of the input data transfer and execution estimation experiment

The eager scheduler had the best accuracy of estimating the input data transfer and execution estimation, ranging from GAPs of 10% to 12%. The results of this experiment

for the most efficient scheduling context configuration were not consistent for the random scheduler.

The data of this experiment is used together with the data collected on the experiment on Subsection 6.4.2 as partial results for the development of the overall execution experiment on Subsection 6.4.3.

6.4.2 Output data transfer comparison experiment

In this experiment, it is taken into account the accuracy of the implementation to estimate the measured time the framework takes to transfer output data from the executed tasks on the experimental platform.

In this Subsection of the experiment the **scheduler** is varied for all scheduling contexts inside every configuration of **scheduling contexts from 1 to 15**. It is also analyzed the input data transfer and execution for **25K, 50K and 100K tasks** in finding out which is the best configuration that optimizes the output data transfer time.

6.4.2.1 Eager scheduler vs Scheduling context configurations vs Number of tasks

For **100K tasks**, the transferring measured times and estimations are shown in Figure 6.4. The scheduling context configuration one, which comprehends of four CPU cores on the same scheduling context had the smallest time, the measured average time was 2912 ms and the estimated average time was 2835 ms, with an average GAP of 3%. The scheduling context configuration two, which comprehends of two GPUs had the longest time, in the measured time and estimation. This happens because in this configuration all tasks are executed on the GPUs, and requires explicit data transfer calls from the GPU to the CPU. The average transferring results GAP for all scheduling context configurations is 7%. The results for **25K and 50K tasks** for this configuration is available on the plots on Figures A.7 and A.8, on Appendix A.

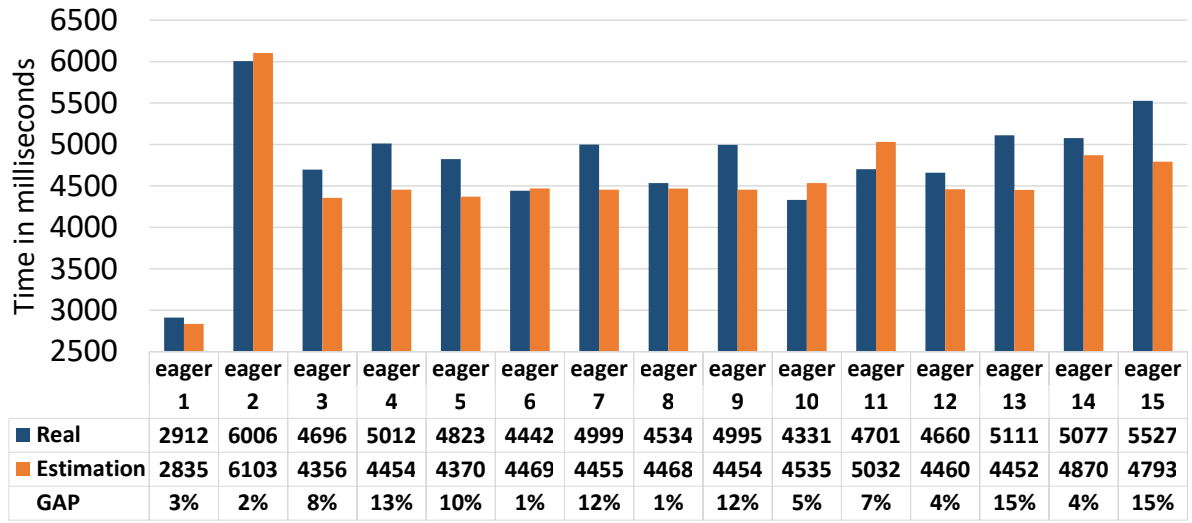


Figure 6.4: Comparison between measured times and its estimation for transferring the execution results of 100K tasks in every combination of the eager scheduler and scheduling context configurations from 1 to 15 (Table 5.1).

6.4.2.2 LWS scheduler vs Scheduling context configurations vs Number of tasks

For **100K tasks**, the transferring measured times and estimations are shown in Figure 6.5. The scheduling context configuration one, which comprehends of four CPU cores on the same scheduling context had the smallest time, the measured average time was 2791 ms and the estimated average time was 2835 ms, with an average GAP of 2%. The scheduling context configuration two, which comprehends of two GPUs had the longest time, in the measured time and estimation. This happens because in this configuration all tasks are executed on the GPUs, and requires explicit data transfer calls from the GPU to the CPU. The average transferring results GAP for all scheduling context configurations is 8%. The results for **25K and 50K tasks** for this configuration is available on the plots on Figures A.9 and A.10, on Appendix A.

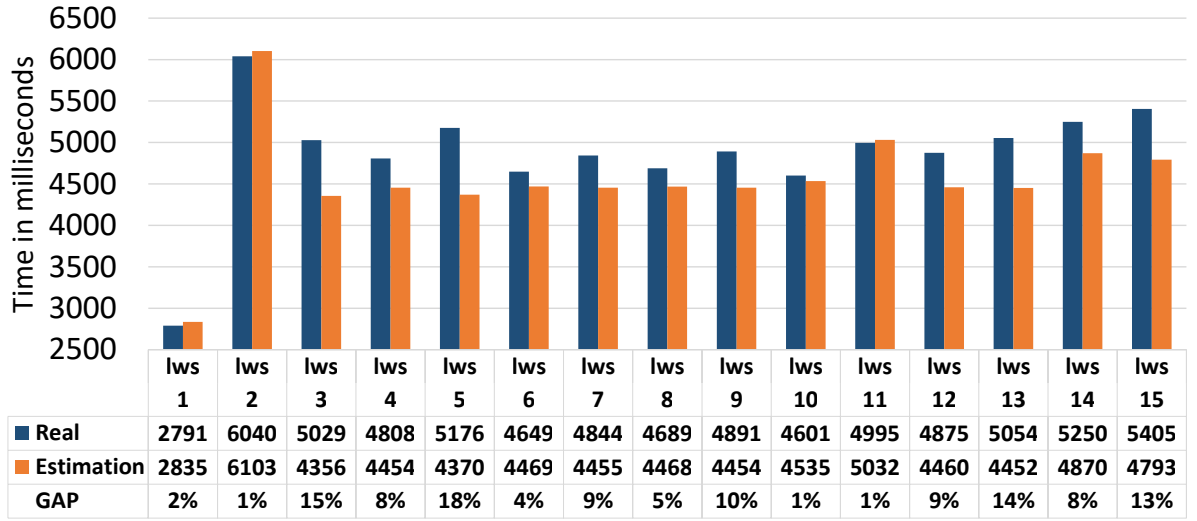


Figure 6.5: Comparison between measured times and its estimation for transferring the execution results of 100K tasks in every combination of the LWS scheduler and scheduling context configurations from 1 to 15 (Table 5.1).

6.4.2.3 WS scheduler vs Scheduling context configurations vs Number of tasks

For **100K tasks**, the transferring measured times and estimations are shown in Figure 6.6. The scheduling context configuration one, which comprehends of four CPU cores on the same scheduling context had the smallest time. The measured average time was 2753 ms and the estimated average time was 2835 ms, with an average GAP of 3%. The scheduling context configuration two, which comprehends of two GPUs had the longest time, in the measured time and estimation. This happens because in this configuration all tasks are executed on the GPUs, and requires explicit data transfer calls from the GPU to the CPU. The average transferring results GAP for all scheduling context configurations is 7%. The results for **25K and 50K tasks** for this configuration is available on the plots on Figures A.11 and A.12, on Appendix A.

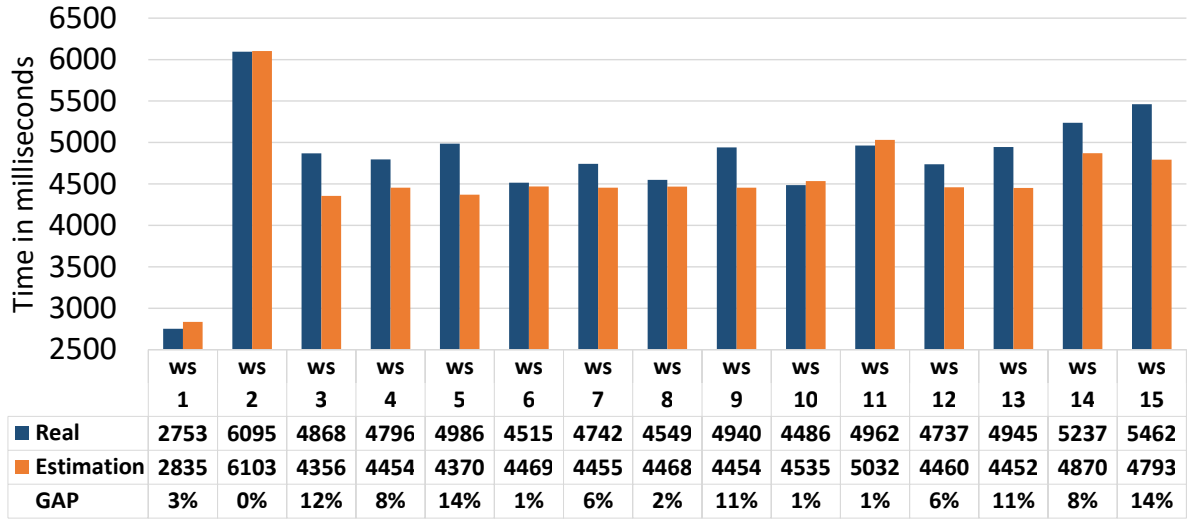


Figure 6.6: Comparison between measured times and its estimation for transferring the execution results of 100K tasks in every combination of the WS scheduler and scheduling context configurations from 1 to 15 (Table 5.1).

6.4.2.4 Random scheduler vs Scheduling context configurations vs Number of tasks

Since the results on Subsection 6.4.1.4 which estimates the input data transfer and execution time for the random scheduler were not consistent, it was not estimated the transfer time for the random scheduler on this experiment.

6.4.2.5 Conclusion of the output data transfer experiment

The results of this experiment were consistent, the scheduling context configuration one which is composed of only one group with all CPUs has systematically the smallest transfer time since no information is sent through the bus. The scheduling context configuration two has always the longest time since all tasks are executed on the GPU and need to explicitly call for transferring data from the GPU to the CPU. The implementation is capable of estimate the transfer time of the tasks with GAPs ranging from 4% to 8% on average.

It is necessary to associate the results generated in this experiment with the input data and execution time obtained on Subsection 6.4.1. To make the decision of choosing the best configuration that minimizes the overall execution time, it is necessary to know if the transfer time from the GPU to the CPU worth the execution advantage the GPU gives for executing particular types of tasks.

6.4.3 Overall execution comparison experiment

In this final experiment, it is taken into account the accuracy of the implementation to estimate the measured time the framework takes to overall execute the set of tasks on the experimental platform. After all times are estimated, the application chooses the configuration with the lowest time, looking for optimizing the overall execution time. This experiment merges two experiments, the first on Subsection 6.4.1 which estimates the input data transfer and execution time of the tasks and the second one on Subsection 6.4.2 that estimates the necessary time to transfer back the output data results. It is also measured the execution time the implementation needs to execute the algorithm.

In this Subsection of the final experiment the **scheduler** is varied for all scheduling contexts inside every configuration of **scheduling contexts from 1 to 15**. It is also analyzed the input data transfer and execution for **25K, 50K and 100K tasks** in finding out which is the best configuration that optimizes the overall execution time.

6.4.3.1 Eager scheduler vs Scheduling context configurations vs Number of tasks

For **100K tasks**, the overall execution time results and estimations are shown in Figure 6.7. The scheduling context configuration one from Table 5.1, had the worst performance. The measured time was 87921 ms. This configuration has been omitted from the plot in order to not compromise the scale. Scheduling context configuration ten was the most efficient, in the measured time and estimation, where both show the smallest values of time for this set of configurations. The average GAP for all configurations is 7% and the decision time to execute the algorithm and choose the best configuration takes 14.7 ms on average. Scheduling contexts configurations from 3 to 15 are the ones who use all the available CPUs and GPUs on the machine. The decision of using one of these configurations can impact the execution time in a variation from 6829 ms to 9099 ms. Taking the worst decision can increase the overall execution time up to 2270 ms if compared to the best configuration that the application needs just 14.7 ms on average to identify it. The results for **25K and 50K tasks** for this configuration is available on the plots on Figures A.13 and A.14, on Appendix A.

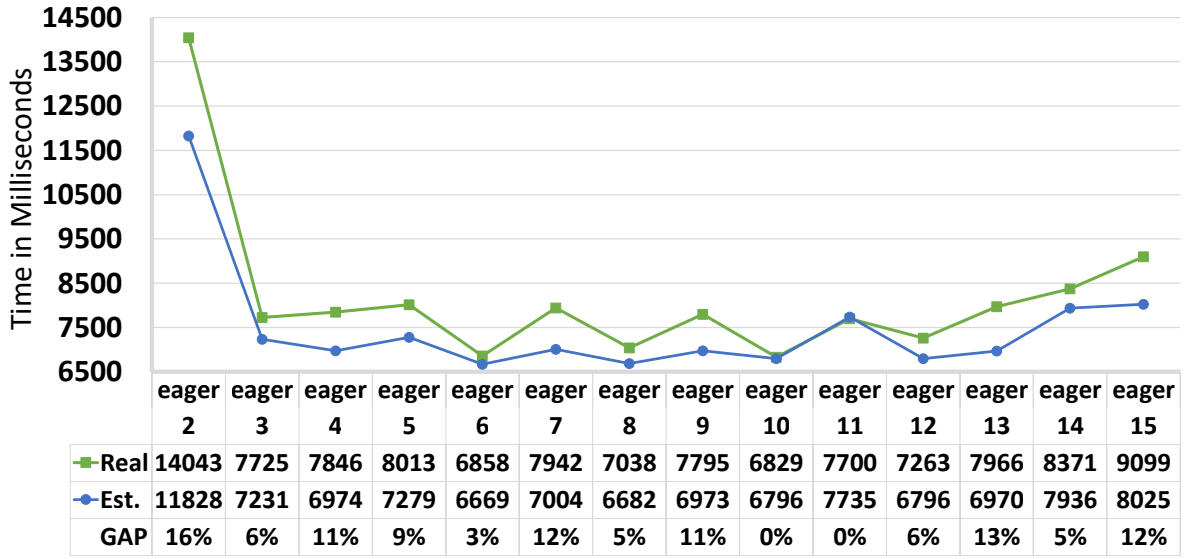


Figure 6.7: Comparison between measured times and its estimation for overall execution of 100K tasks in every combination of the eager scheduler and scheduling context configurations from 2 to 15 (Table 5.1).

6.4.3.2 LWS scheduler vs Scheduling context configurations vs Number of tasks

For **100K tasks**, the overall execution time results and estimations are shown in Figure 6.8. The scheduling context configuration one from Table 5.1, had the worst performance. The measured time was 87848 ms. This configuration has been omitted from the plot in order to not compromise the scale. Scheduling context configuration six was the most efficient, in the measured time and estimation, where both show the smallest values of time for this set of configurations. The average GAP for all configurations is 11% and the decision time to execute the algorithm and choose the best configuration takes 14.7 ms on average. Scheduling contexts configurations from 3 to 15 are the ones who use all the available CPUs and GPUs on the machine. The decision of using one of these configurations can impact the execution time in a variation from 7351 ms to 9002 ms. Taking the worst decision can increase the overall execution time up to 1651 ms if compared to the best configuration that the application needs just 14.7 ms on average to identify it. The results for **25K and 50K tasks** for this configuration is available on the plots on Figures A.15 and A.16, on Appendix A.

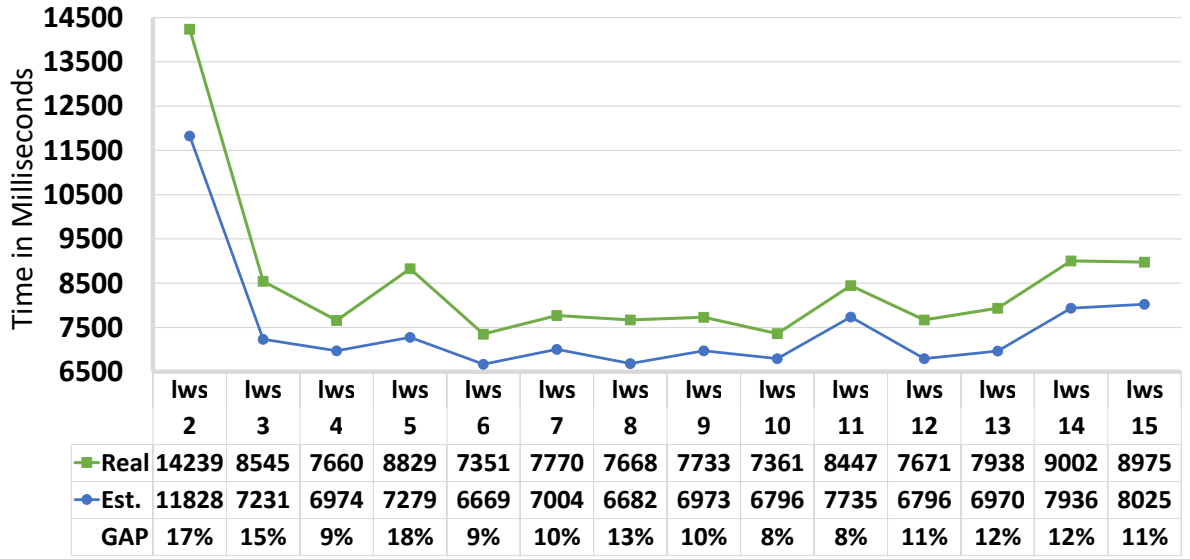


Figure 6.8: Comparison between measured times and its estimation for overall execution of 100K tasks in every combination of the LWS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).

6.4.3.3 WS scheduler vs Scheduling context configurations vs Number of tasks

For **100K tasks**, the overall execution time results and estimations are shown in Figure 6.9. The scheduling context configuration one from Table 5.1, had the worst performance. The measured time was 87791 ms. This configuration has been omitted from the plot in order to not compromise the scale. Scheduling context configuration six was the most efficient, in the measured time and estimation, where both show the smallest values of time for this set of configurations. The average GAP for all configurations is 10% and the decision time to execute the algorithm and choose the best configuration takes 14.5 ms on average. Scheduling contexts configurations from 3 to 15 are the ones who use all the available CPUs and GPUs on the machine. The decision of using one of these configurations can impact the execution time in a variation from 7172 ms to 9002 ms. Taking the worst decision can increase the overall execution time up to 1830 ms if compared to the best configuration that the application needs just 14.5 ms on average to identify it. The results for **25K and 50K tasks** for this configuration is available on the plots on Figures A.17 and A.18, on Appendix A.

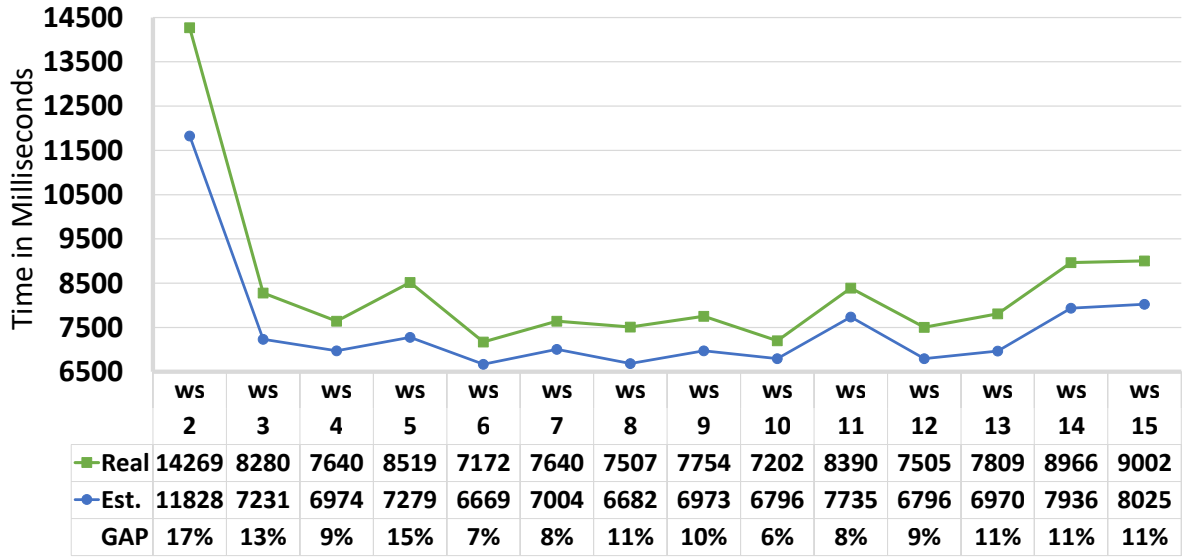


Figure 6.9: Comparison between measured times and its estimation for overall execution of 100K tasks in every combination of the WS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).

6.4.3.4 Random scheduler vs Scheduling context configurations vs Number of tasks

Since the results on Subsection 6.4.1.4 which estimates the input data transfer and execution time for the random scheduler were not consistent, it was not estimated the overall execution time for the random scheduler on this experiment. The scheduler randomly distributes tasks among workers inside each context, which creates an unpredictable pattern, making it harder to predict times above acceptable levels, leading to low accuracy.

6.4.3.5 Conclusion of the overall execution comparison experiment

The results are consistent, and independently of how many tasks are executed, it is possible to determine the most effective scheduling context configuration at least two order of magnitude faster in comparison with the order of time necessary to run the set of tasks. Even though the average GAP ranged from 3% to 7%, the estimation follows the trends and matches the plot pattern of the execution, as can be verified from Figures 6.7 to 6.9.

6.5 Limitations

Despite the good computational performance regarding optimizing overall execution time of tasks, this work aims to introduce new ideas and open horizons for heterogeneous

computing. It was not the goal of this research to overcome the algorithms considered as state-of-the-art, but rather to present the implementation of a novel framework as a proof of concept. Although this work designs and implements a framework to optimize the overall execution time of tasks showing that is efficient and brings improved execution times, much still can be done to enhance results and achieve even better times.

6.6 Conclusion

Different context configurations produce distinct overall execution times for a specific set of tasks. Using the tested benchmarks, the algorithm can predict the input/output data transfer and execution with an average GAP from 3% to 7%, the estimation has enough accuracy to follow the trends and matches the plot pattern of the execution. The implementation and execution of the algorithm can significantly reduce the overall execution time even in comparison with the most efficient configurations from 3 to 15 which use all workers and always surpass the configuration two which uses only GPUs. The time required to analyze the set of configurations is always at least two orders of magnitude faster than the necessary time to execute all the tasks, indicating that the time needed to run the algorithm is worth the benefits of potential saving times.

Chapter 7

Concluding Remarks and Future Works

This chapter concludes this thesis work. It briefly reiterates the achievements, and the results are wrapped up on section 7.1. It also checks how the outcome matched the general research objective and remarks achieved contribution. Finally, Section 7.2 present some directions for extension of this research.

7.1 Concluding Remarks

In this thesis, it was studied the problem of efficiently schedule independent tasks and minimize their overall execution time of the tasks on heterogeneous machines. This problem is solved by dispatching tasks to be executed on a multiple CPU and GPU system through the estimation and selection of the most efficient CPU and GPU virtual configuration of processors.

The solution of this problem is achieved through the design, development, implementation, and testing proof of the novel framework model, which points to being efficient for independent tasks. Regarding scalability, the framework expands seamlessly to as many CPUs or GPUs available on a single machine; the developer can abstract and does not need to worry about the current machine architecture.

The time required for the implementation to evaluate and estimate each execution configuration is at least two orders of magnitude less than the time required to execute a massive number of tasks, which points that the concept is feasible and remains efficient even with the presence of GAPS on the estimation.

The use of virtual groups of CPU cores and GPUs, together with fast estimations, can lead to the minimization of the overall execution time of tasks, allowing developers to

automatically take advantage and extract more performance. The contributions of this work have wide applicability in Computational Science and areas that demand massive high-performance computing and processing of generic independent tasks. This work is still an initial study and have interesting results, but still a lot of research is required. Finally, this work also helps to fill the gap on better resource management software to extract better computing performance while still relying on the same available powerful heterogeneous hardware.

This research also opens new horizons and possibilities for further experiments built on top of this thesis as described in Section 7.2.

7.2 Future Works

Due to the wide range of subjects and aspects of this thesis, the proposed and implemented framework opens numerous research opportunities for future works that can be improved and extended, which are classified as short and long term goals in Subsections 7.2.1 and 7.2.2.

7.2.1 Short term goals

In the short term, many directions appealing paths are possible for future works. A list of the most important ones and its description are the following:

1. **Test different schedulers for each scheduling context inside scheduling context configurations** - Assign multiple scheduling algorithms for different scheduling contexts inside a configuration. This work tests just one type of scheduler for all scheduling contexts inside a configuration.
2. **Test additional benchmarks** - Validate the framework with additional benchmarks.
3. **Tests with joint scheduling contexts** - In this work, there is no intersection among scheduling contexts. It would be relevant to investigate the effect of scheduling contexts that share particular workers at the same time and observe how they behave in comparison to disjoint scheduling contexts.
4. **Self-tuning of the GPU tasks for the specific GPUs on the machine** - Tune the performance of a kernel implemented inside a task for a particular GPU

architecture that this task is going to be assigned. This is feasible and can be done getting accurate information about the current status of each GPU architecture specifications, leading to time optimization.

5. **Increase the overlap of data transfers and computations** - All the communication happens after the end of all tasks execution to keep memory coherence. Instead of waiting for every task to finish, the idea is to parallelize communication and transfer the results while the CPU or GPU is still executing the tasks. This idea was not implemented due to framework computational limitations. According to Lima et al. [38], overlapping capability is at least as important as computing a good scheduling decision to reduce the completion time of a parallel program. The work of Verner et al. [72] uses multiple data streams and can inspire this future work.

7.2.2 Long term goals

In the long term, many directions appealing paths are possible for future works. A list of the most important ones and its description are the following:

1. **Expand for multiple machines** - Extend the implemented framework for a cluster of heterogeneous machines.
2. **Power optimization** - Develop a front-end that will optimize power spent by tasks. Optimize energy consumption at the price of simultaneously spending some more time for executing the set of tasks.
3. **Increase limit of tasks execution through the use of “waves”** - The implementation operates to the limit of the CPU and GPU RAM since all tasks are dispatched at once, executes and returns the results. The idea is to instead of having a huge number of tasks that is limited by the RAM; the objective is to increase the number of tasks splitting the total amount into waves, allowing the execution of a massive number of tasks.
4. **Dependent tasks** - Task dependency was a complicating factor for the novel study, since reduces parallelism and increases the interdependency of tasks. This suggestion refers to expand the same idea but for dependent tasks.
5. **Use internal information of tasks** - Many efforts were made for avoiding using internal private information of each task type for reducing interdependency of the

framework. Allowing the framework to access internal data of the tasks should make the optimization problem much more complicated, although it should collect valuable information that can be used for special optimizations.

References

- [1] ACOSTA, A.; BLANCO, V.; ALMEIDA, F. Dynamic load balancing on heterogeneous multi-GPU systems. *Computers & Electrical Engineering* 39, 8 (nov 2013), 2591–2602.
- [2] AGULLEIRO, J.; VÁZQUEZ, F.; GARZÓN, E.; FERNÁNDEZ, J. Hybrid computing: CPU+GPU co-processing and its application to tomographic reconstruction. *Ultramicroscopy* 115 (apr 2012), 109–114.
- [3] AGULLO, E.; AGULLO, E.; AUGONNET, C.; DONGARRA, J.; LTAIEF, H.; NAMYST, R.; THIBAUT, S.; TOMOV, S. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs, 2010.
- [4] ANDRADE, G.; RAMOS, G.; MADEIRA, D.; SACHETTO, R.; CLUA, E.; FERREIRA, R.; ROCHA, L. Efficient dynamic scheduling of heterogeneous applications in hybrid architectures. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing - SAC '14* (New York, New York, USA, 2014), ACM Press, pp. 866–871.
- [5] AUGONNET, C.; THIBAUT, S.; NAMYST, R. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. In *3rd Workshop on Highly Parallel Processing on a Chip (HPPC 2009)* (Delft, The Netherlands, 2010), pp. 56–65.
- [6] AUGONNET, C.; THIBAUT, S.; NAMYST, R. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Research Report RR-7240, INRIA, Mar. 2010.
- [7] AUGONNET, C.; THIBAUT, S.; NAMYST, R.; NIJHUIS, M. Exploiting the Cell/BE Architecture with the StarPU Unified Runtime System. In *9th International Workshop, SAMOS 2009, Samos, Greece, July 20-23, 2009. Proceedings* (2009), Springer Berlin Heidelberg, pp. 329–339.
- [8] AUGONNET, C.; THIBAUT, S.; NAMYST, R.; WACRENIER, P.-A. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198.
- [9] BELVIRANLI, M. E.; BHUYAN, L. N.; GUPTA, R. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Transactions on Architecture and Code Optimization* 9, 4 (jan 2013), 1–20.
- [10] BHASKARAN-NAIR, K.; MA, W.; KRISHNAMOORTHY, S.; VILLA, O.; VAN DAM, H. J. J.; APRÀ, E.; KOWALSKI, K. Noniterative Multireference Coupled Cluster Methods on Heterogeneous CPU–GPU Systems. *Journal of Chemical Theory and Computation* 9, 4 (apr 2013), 1949–1957.

- [11] BOUKERCHE, A.; CORREA, J. M.; MELO, A. C. M.; JACOBI, R. P. A Hardware Accelerator for the Fast Retrieval of DIALIGN Biological Sequence Alignments in Linear Space. *IEEE Transactions on Computers* 59, 6 (jun 2010), 808–821.
- [12] BREDER, B.; CHARLES, E.; CRUZ, R.; CLUA, E.; BENTES, C.; DRUMMOND, L. Maximizando o Uso dos Recursos de GPU Através da Reordenação da Submissão de Kernels Concorrentes. *WSCAD - Simpósio em Sistemas Computacionais de Alto Desempenho* 1, 1 (oct 2016), 88–99.
- [13] BRODTKORB, A. R.; HAGEN, T. R.; SÆTRA, M. L. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing* 73, 1 (jan 2013), 4–13.
- [14] CANONICAL. Ubuntu 14.04.3 LTS (Trusty Tahr). <http://releases.ubuntu.com/14.04.3/>, 2017.
- [15] CEDERMAN, D.; TSIGAS, P. On Dynamic Load Balancing on Graphics Processors. *Proceedings of the SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (2008), 57–64.
- [16] CHATTERJEE, S.; GROSSMAN, M.; SBÎRLEA, A.; SARKAR, V. Dynamic Task Parallelism with a GPU Work-Stealing Runtime System. In *24th International Workshop, LCPC 2011, Fort Collins, CO, USA, September 8-10, 2011* (2013), pp. 203–217.
- [17] CHE, S.; BOYER, M.; MENG, J.; TARJAN, D.; SHEAFFER, J. W.; LEE, S.-H.; SKADRON, K. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on* (2009), IEEE, pp. 44–54.
- [18] CHEN, L.; VILLA, O.; KRISHNAMOORTHY, S.; GAO, G. R. Dynamic load balancing on single- and multi-GPU systems. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)* (2010), IEEE, pp. 1–12.
- [19] COJEAN, T.; GUERMOUCHE, A.; HUGO, A.; NAMYST, R.; WACRENIER, P. A. Resource Aggregation for Task-Based Cholesky Factorization on Top of Heterogeneous Machines. In *Euro-Par 2016: Parallel Processing Workshops* (aug 2017), Springer, Cham, pp. 56–68.
- [20] DASTGEER, U.; KESSLER, C. Conditional component composition for GPU-based systems. In *Proc. Seventh Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG-2014)* (Vienna, Austria, 2014), HiPEAC NoE.
- [21] DELGADO, J.; GAZOLLA, J.; CLUA, E.; SADJADI, M. A case study on porting scientific applications to GPU/CUDA. *Journal of Computational Interdisciplinary Sciences* 2, 1 (2011).
- [22] DELGADO, J.; GAZOLLA, J.; CLUA, E.; SADJADI, S. M. A case study on porting scientific applications to gpu/cuda. *Journal of Computational Interdisciplinary Sciences* 2 (2011), 3–11.
- [23] GAREY, M. R.; JOHNSON, D. S. *A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

- [24] GAUTIER, T.; LIMA, J. V.; MAILLARD, N.; RAFFIN, B. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing* (may 2013), IEEE, pp. 1299–1308.
- [25] GNU. GCC and G++ Compiler Collection. <https://gcc.gnu.org/>, 2017.
- [26] GRAUER-GRAY, S.; XU, L.; SEARLES, R.; AYALASOMAYAJULA, S.; CAVAZOS, J. Auto-tuning a High-Level Language Targeted to GPU Codes. In *Innovative Parallel Computing* (2012), IEEE, pp. 1–10.
- [27] GREGG, C.; BOYER, M.; HAZELWOOD, K.; SKADRON, K. Dynamic Heterogeneous Scheduling Decisions Using Historical Runtime Data. In *Proceedings of the 2nd Workshop on Applications for Multi-and Many-Core Processors. San Jose, CA* (2011).
- [28] HUGO, A. E.; GUERMOUCHE, A.; WACRENIER, P. A.; NAMYST, R. Composing multiple StarPU applications over heterogeneous machines: A supervised approach. In *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum, IPDPSW 2013* (aug 2013), vol. 28, SAGE PublicationsSage UK: London, England, pp. 1050–1059.
- [29] HUGO, A.-E.; GUERMOUCHE, A.; WACRENIER, P.-A.; NAMYST, R. A runtime approach to dynamic resource allocation for sparse direct solvers. In *International Conference on Parallel Processing* (2014), IEEE, pp. 481–490.
- [30] INTEL. Intel Core i7-4790 Processor Specifications. <https://ark.intel.com/en/products/80806/>, 2017.
- [31] JACKSON, M.; BUDRUK, R.; WINKLES, J.; ANDERSON, D. *PCI Express Technology 3.0*, first ed. MindShare, 2012.
- [32] JANGHAENG LEE; SAMADI, M.; YONGJUN PARK; MAHLKE, S. Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques* (oct 2013), IEEE, pp. 319–330.
- [33] JETLEY, P.; WESOLOWSKI, L.; GIOACHIN, F.; KAL?, L. V.; QUINN, T. R. Scaling Hierarchical N-body Simulations on GPU Clusters. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (nov 2010), IEEE, pp. 1–11.
- [34] KALEEM, R.; BARIK, R.; SHPEISMAN, T.; LEWIS, B. T.; HU, C.; PINGALI, K. Adaptive heterogeneous scheduling for integrated GPUs. In *Proceedings of the 23rd international conference on Parallel architectures and compilation - PACT '14* (New York, New York, USA, 2014), ACM Press, pp. 151–162.
- [35] KORMANYOS, C. *Real-Time C++*. Springer, 2015.
- [36] LECRON, F.; MAHMOUDI, S. A.; BENJELLOUN, M.; MAHMOUDI, S.; MANNEBACK, P. Heterogeneous Computing for Vertebra Detection and Segmentation in X-Ray Images. *International Journal of Biomedical Imaging 2011* (2011), 1–12.

- [37] LETA, F. R.; CLUA, E.; BARBOZA, D. C.; GAZOLLA, J. G. F. M.; BIONDI, M.; DO SOUZA, M. S. Real-Time Visualization and Geometry Reconstruction of Large Oil and Gas Boreholes Based on Caliper Database. In *Visual Computing: Scientific Visualization and Imaging Systems*. Springer, Berlin, Heidelberg, 2014, pp. 239–254.
- [38] LIMA, J. V.; GAUTIER, T.; MAILLARD, N.; DANJEAN, V. Exploiting Concurrent GPU Operations for Efficient Work Stealing on Multi-GPUs. In *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing* (oct 2012), IEEE, pp. 75–82.
- [39] LUCAS, A.; ROELAND, C.; AUGONNET, C.; COURTÈS, L.; FURMENTO, N.; AUMAGE, O.; THIBAUT, S. *StarPU Handbook*. INRIA France, 2017.
- [40] LUCAS, A.; ROELAND, C.; AUGONNET, C.; COURTÈS, L.; FURMENTO, N.; AUMAGE, O.; THIBAUT, S. StarPU Library SDK - Task Programming Library for Hybrid Architectures. <http://starpu.gforge.inria.fr/files/>, 2017.
- [41] LUK, C.-K.; HONG, S.; KIM, H. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture - Micro-42* (New York, New York, USA, 2009), ACM Press, p. 45.
- [42] MARTINEZ, V.; MICHEA, D.; DUPROS, F.; AUMAGE, O.; THIBAUT, S.; AOCHI, H.; NAVAUX, P. O. Towards Seismic Wave Modeling on Heterogeneous Many-Core Architectures Using Task-Based Runtime System. In *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)* (oct 2015), IEEE, pp. 1–8.
- [43] MITTAL, S.; VETTER, J. S. A Survey of Methods for Analyzing and Improving GPU Energy Efficiency. *ACM Computing Surveys* 47, 2 (aug 2014), 1–23.
- [44] MITTAL, S.; VETTER, J. S. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Computing Surveys* 47, 4 (jul 2015), 1–35.
- [45] NAVARRO, A.; VILCHES, A.; CORBERA, F.; ASENJO, R. Strategies for maximizing utilization on multi-CPU and multi-GPU heterogeneous architectures. *The Journal of Supercomputing* 70, 2 (nov 2014), 756–771.
- [46] NERE, A.; HASHMI, A.; LIPASTI, M. Profiling Heterogeneous Multi-GPU Systems to Accelerate Cortically Inspired Learning Algorithms. In *2011 IEEE International Parallel & Distributed Processing Symposium* (may 2011), IEEE, pp. 906–920.
- [47] NESMACHNOW, S.; CANABÉ, M. GPU implementations of scheduling heuristics for heterogeneous computing environments. In *XVII Congreso Argentino de Ciencias de la Computación* (2011).
- [48] NICKOLLS, J.; BUCK, I.; GARLAND, M.; SKADRON, K. Scalable parallel programming with CUDA. *Queue* 6, 2 (mar 2008), 40.
- [49] NVIDIA. Compute Unified Device Architecture (CUDA) Programming Guide. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>, 2017.

- [50] NVIDIA. Compute Unified Device Architecture (CUDA) Toolkit Documentation v8.0.61. <https://developer.nvidia.com/maxwell-compute-architecture>, 2017.
- [51] NVIDIA. GeForce GTX TITAN X Specifications. <https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications>, 2017.
- [52] NVIDIA. NVIDIA CUDA Compiler Driver NVCC. <http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/>, 2017.
- [53] NVIDIA. NVIDIA Management Library (NVML). <https://developer.nvidia.com/nvidia-management-library-nvml>, 2017.
- [54] NVIDIA. NVIDIA Visual Profiler. <https://developer.nvidia.com/nvidia-visual-profiler>, 2017.
- [55] NVIDIA. White Paper - GPU Maxwell Architecture. <https://developer.nvidia.com/maxwell-compute-architecture>, 2017.
- [56] ODAJIMA, T.; BOKU, T.; HANAWA, T.; LEE, J.; SATO, M. GPU/CPU Work Sharing with Parallel Language XcalableMP-dev for Parallelized Accelerated Computing. In *2012 41st International Conference on Parallel Processing Workshops* (sep 2012), IEEE, pp. 97–106.
- [57] OWENS, J. D.; HOUSTON, M.; LUEBKE, D.; GREEN, S.; STONE, J. E.; PHILLIPS, J. C. GPU computing. *Proceedings of the IEEE* 96, 5 (may 2008), 879–899.
- [58] PANDIT, P.; GOVINDARAJAN, R. Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (2014), ACM, p. 273.
- [59] PREIS, T.; VIRNAU, P.; PAUL, W.; SCHNEIDER, J. J. GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model. *Journal of Computational Physics* 228, 12 (jul 2009), 4468–4477.
- [60] RICARDO DA SILVA JUNIOR, J.; GONZALEZ CLUA, E. W.; MONTENEGRO, A.; LAGE, M.; DREUX, M. D. A.; JOSELLI, M.; PAGLIOSA, P. A.; KURYLA, C. L. A heterogeneous system based on gpu and multi-core cpu for real-time fluid and rigid body simulation. *International Journal of Computational Fluid Dynamics* 26, 3 (2012), 193–204.
- [61] RYOO, S.; RODRIGUES, C. I.; BAGHSORKHI, S. S.; STONE, S. S.; KIRK, D. B.; HWU, W.-M. W. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming - PPOPP '08* (New York, New York, USA, 2008), ACM Press, p. 73.
- [62] SGOGLAND, T. R.; ROUNTREE, B.; FENG, W.-C.; DE SUPINSKI, B. R. Heterogeneous Task Scheduling for Accelerated OpenMP. In *International Parallel and Distributed Processing Symposium* (May 2012), IEEE, pp. 144–155.

- [63] SONG, F.; DONGARRA, J. A scalable framework for heterogeneous GPU-based clusters. In *Proceedings of the 24th ACM symposium on Parallelism in algorithms and architectures - SPAA '12* (New York, New York, USA, 2012), ACM Press, p. 91.
- [64] STONE, J. E.; GOHARA, D.; SHI, G. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering* 12, 1-3 (2010), 66–73.
- [65] STROHMAIER, E.; HORST, S.; DONGARRA, J. The Green500's energy-efficient supercomputers, June 2017. <https://www.top500.org/green500/lists/2017/06/>, 2017.
- [66] STROHMAIER, E.; HORST, S.; DONGARRA, J. Top 500 Supercomputer Sites, June 2017. <https://www.top500.org/lists/2017/06/>, 2017.
- [67] SUN, E.; SCHAA, D.; BAGLEY, R.; RUBIN, N.; KAEI, D. Enabling task-level scheduling on heterogeneous platforms. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units - GPGPU-5* (New York, New York, USA, 2012), ACM Press, pp. 84–93.
- [68] SUNDARAM, N.; RAGHUNATHAN, A.; CHAKRADHAR, S. T. A framework for efficient and scalable execution of domain-specific templates on GPUs. In *2009 IEEE International Symposium on Parallel & Distributed Processing* (may 2009), IEEE, pp. 1–12.
- [69] TEODORO, G.; HARTLEY, T. D. R.; CATALYUREK, U.; FERREIRA, R. Run-time optimizations for replicated dataflows on heterogeneous environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing - HPDC '10* (New York, New York, USA, 2010), ACM Press, p. 13.
- [70] TEODORO, G.; KURC, T.; ANDRADE, G.; KONG, J.; FERREIRA, R.; SALTZ, J. Application performance analysis and efficient execution on systems with multi-core CPUs, GPUs and MICs: a case study with microscopy image analysis. *The International Journal of High Performance Computing Applications* 31, 1 (jan 2017), 32–51.
- [71] VASCONCELLOS, E. C.; CLUA, E. W.; ROSA, R. R.; GAZOLLA, J. G. F.; FERREIRA, N. C. D. R.; CARLQUIST, V.; COSTA, C. F. D. S. GPU Optimization for Data Analysis of Mario Schenberg Spherical Detector. *Procedia Computer Science* 80, 80 (2016), 2158–2168.
- [72] VERNER, U.; SCHUSTER, A.; SILBERSTEIN, M.; MENDELSON, A. Scheduling processing of real-time data streams on heterogeneous multi-GPU systems. In *Proceedings of the 5th Annual International Systems and Storage Conference on - SYSTOR '12* (New York, New York, USA, 2012), ACM Press, pp. 1–12.
- [73] WEN, Y.; WANG, Z.; O'BOYLE, M. F. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *2014 21st International Conference on High Performance Computing (HiPC)* (2014), IEEE, pp. 1–10.
- [74] ZHOU, H.; LIU, C. Task mapping in heterogeneous embedded systems for fast completion time. In *Proceedings of the 14th International Conference on Embedded Software - EMSOFT '14* (New York, New York, USA, 2014), ACM Press, pp. 1–10.

APPENDIX A - Extra Results

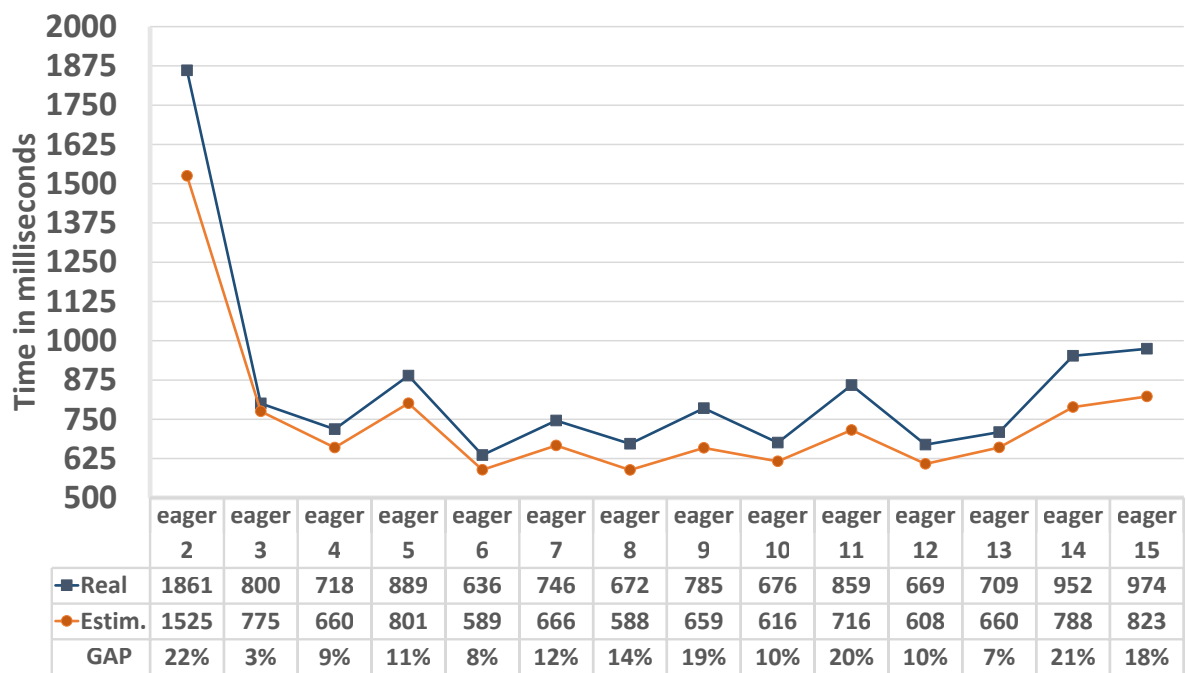


Figure A.1: A comparison between measured times and its estimation for 25K tasks in every combination of the eager scheduler and scheduling context configurations from 2 to 15 (Table 5.1).

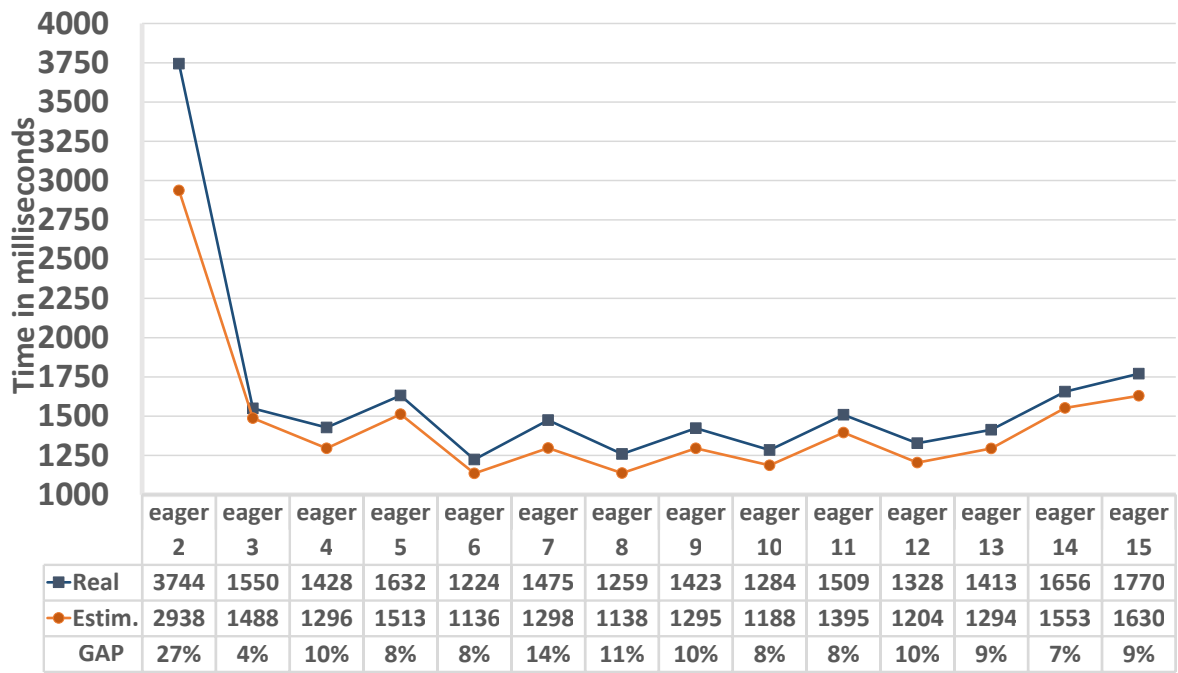


Figure A.2: A comparison between measured times and its estimation for 50K tasks in every combination of the eager scheduler and scheduling context configurations from 2 to 15 (Table 5.1).

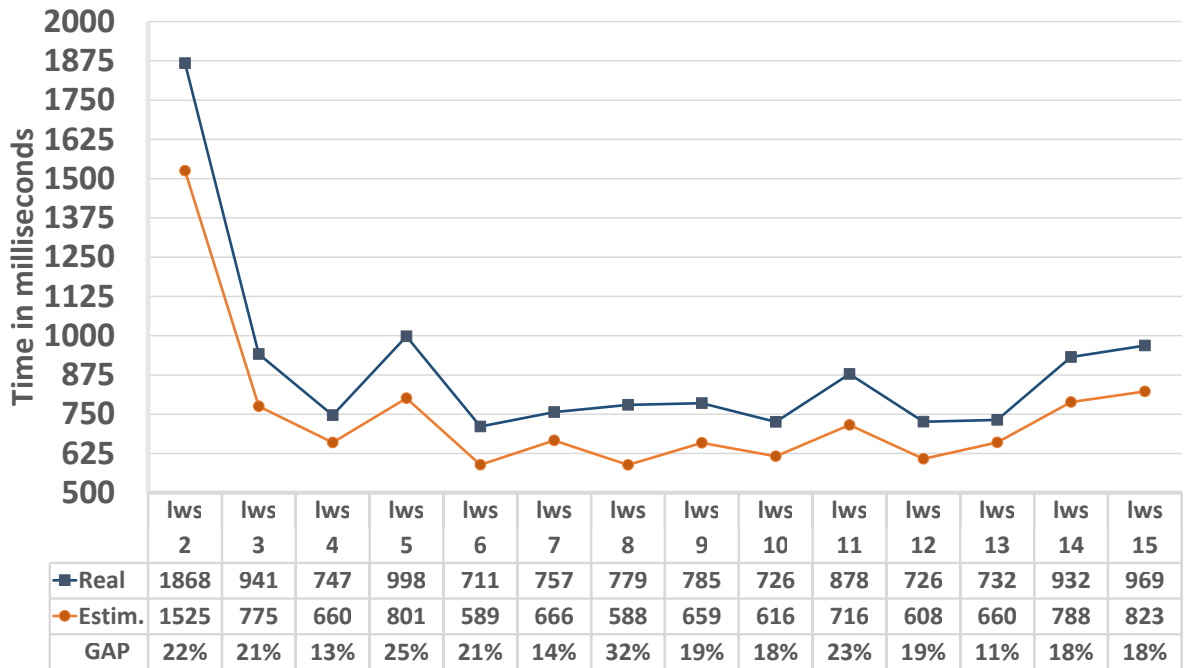


Figure A.3: A comparison between measured times and its estimation for 25K tasks in every combination of the LWS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).

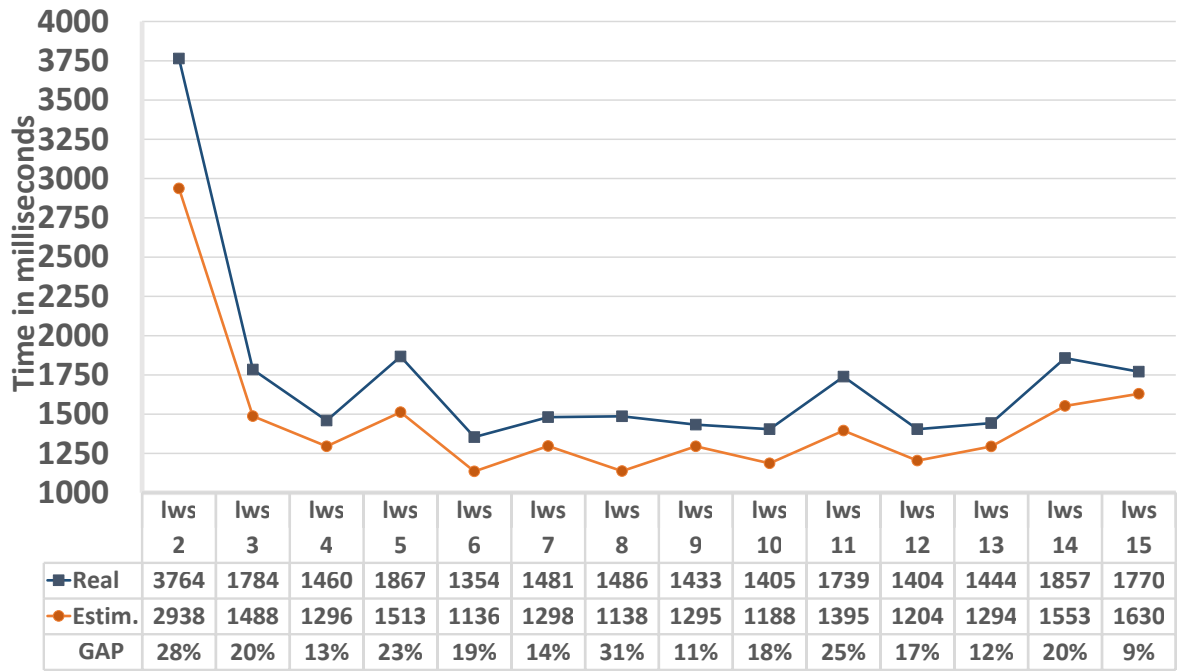


Figure A.4: A comparison between measured times and its estimation for 50K tasks in every combination of the LWS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).

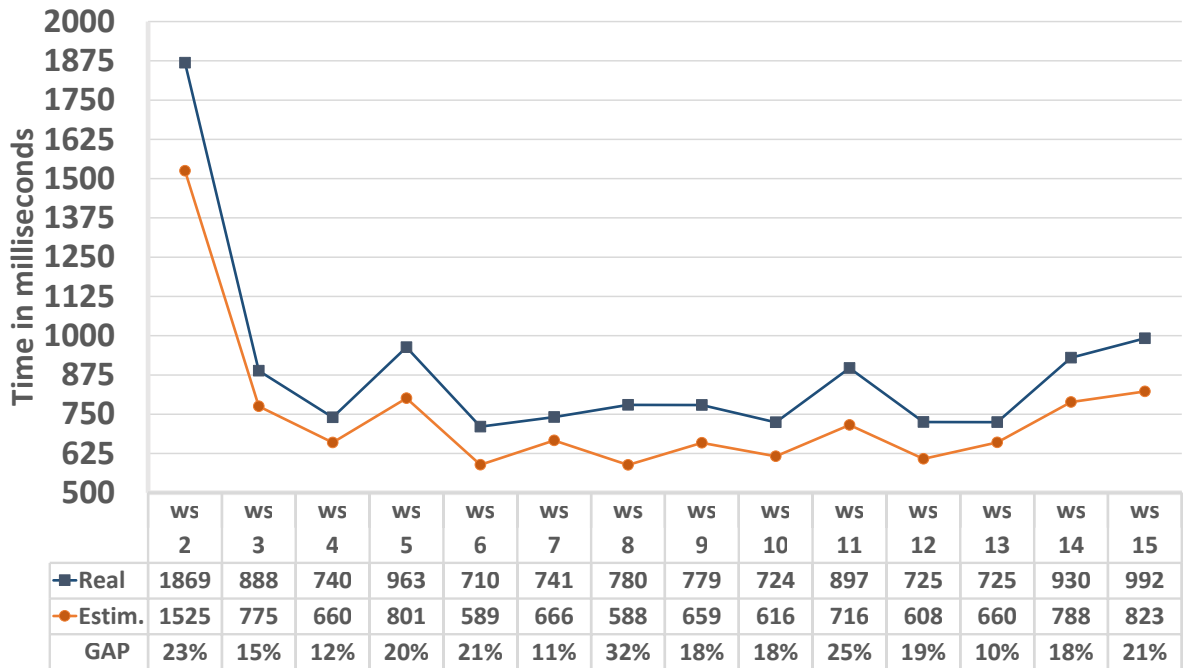


Figure A.5: A comparison between measured times and its estimation for 25K tasks in every combination of the WS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).

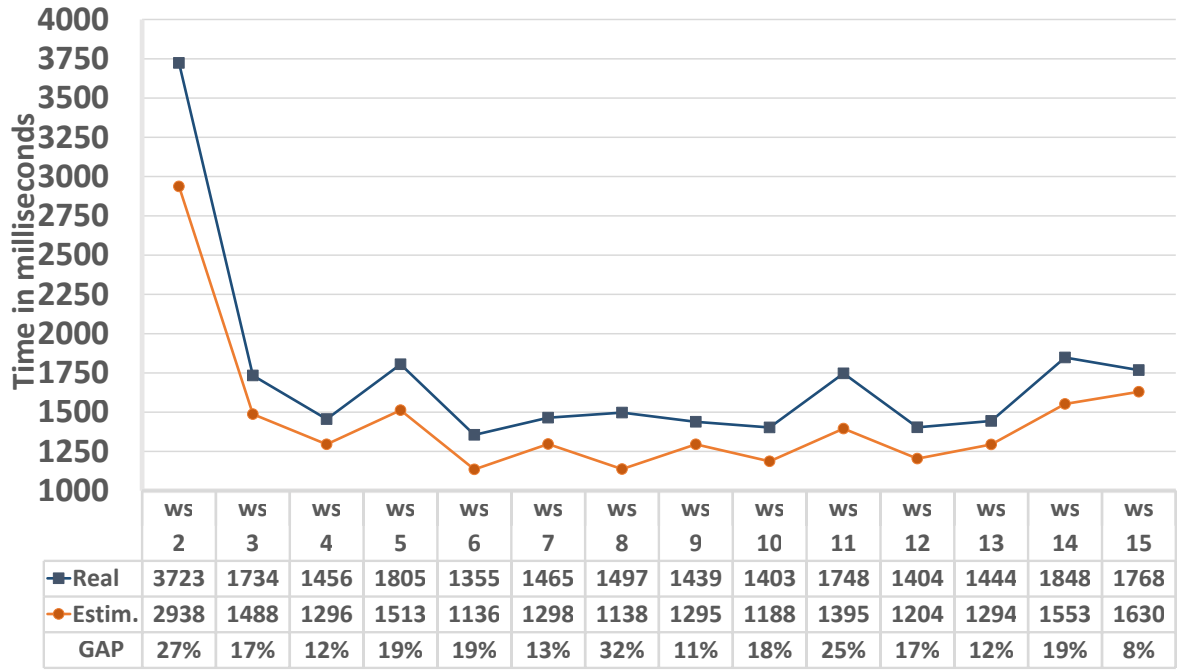


Figure A.6: A comparison between measured times and its estimation for 50K tasks in every combination of the WS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).

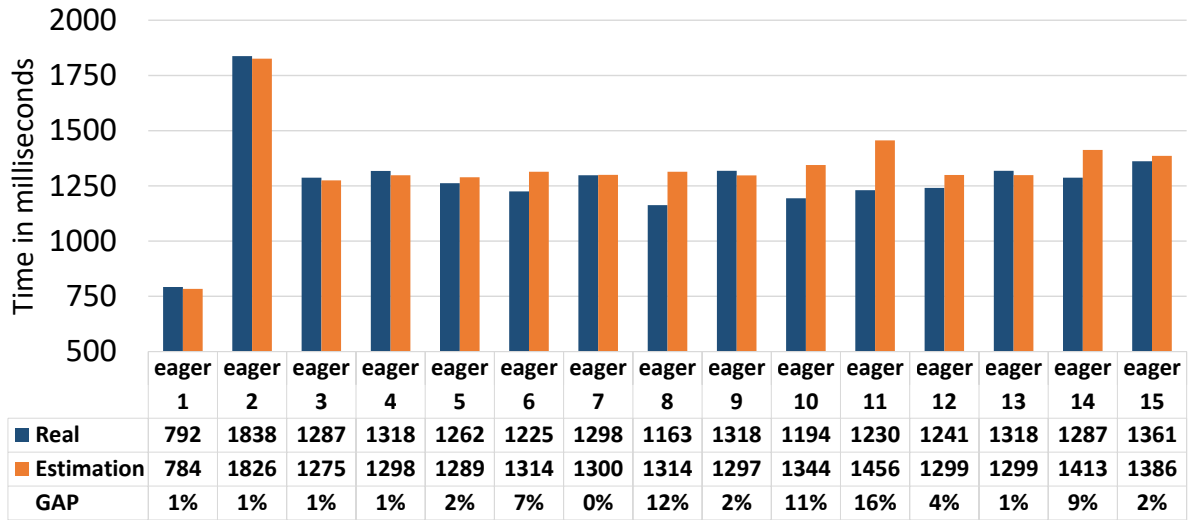


Figure A.7: Comparison between measured times and its estimation for transferring the execution results of 25K tasks in every combination of the eager scheduler and scheduling context configurations from 1 to 15 (Table 5.1).

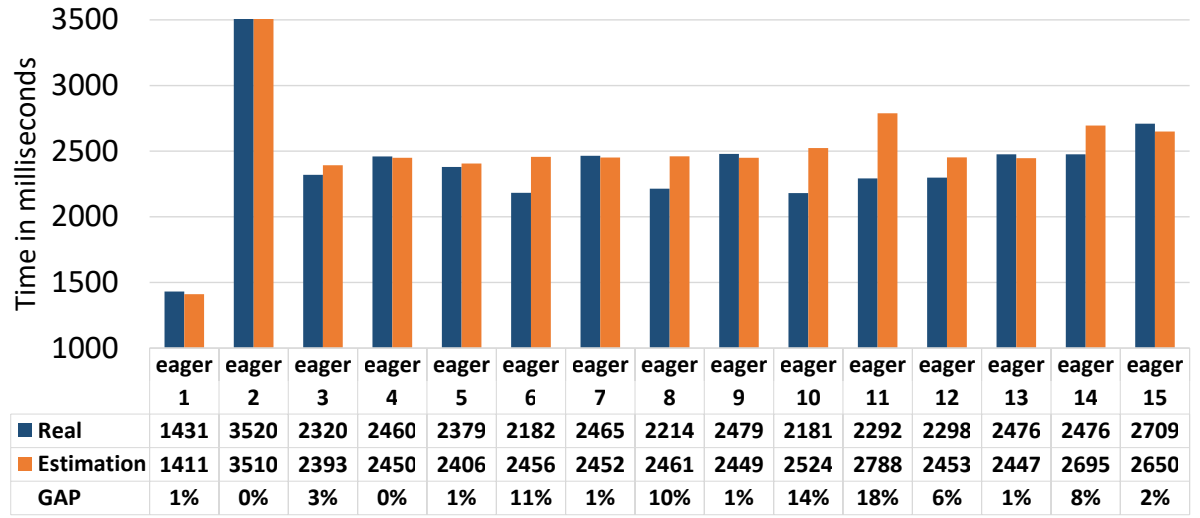


Figure A.8: Comparison between measured times and its estimation for transferring the execution results of 50K tasks in every combination of the eager scheduler and scheduling context configurations from 1 to 15 (Table 5.1).

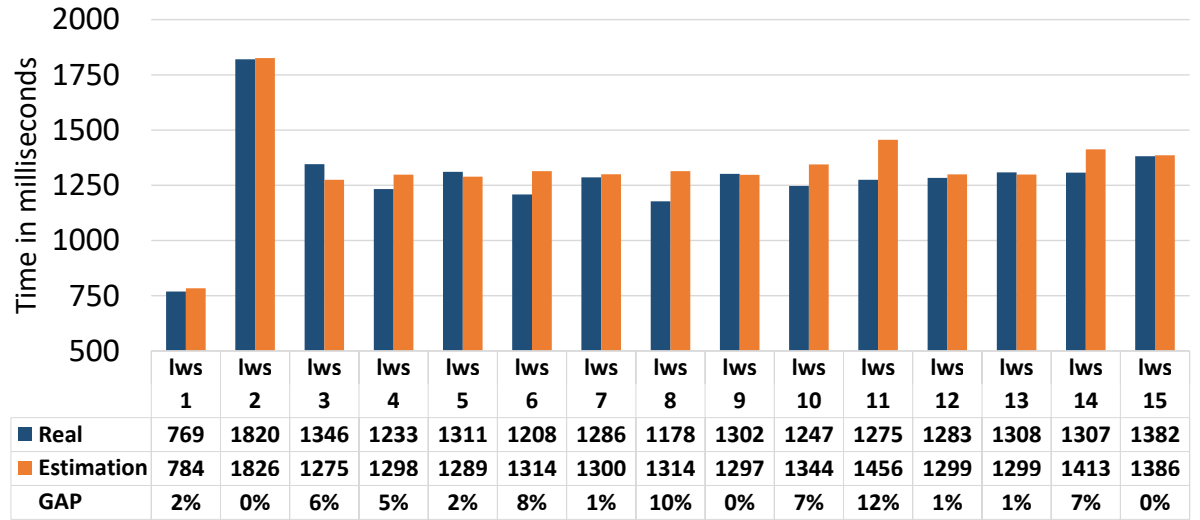


Figure A.9: Comparison between measured times and its estimation for transferring the execution results of 25K tasks in every combination of the LWS scheduler and scheduling context configurations from 1 to 15 (Table 5.1).

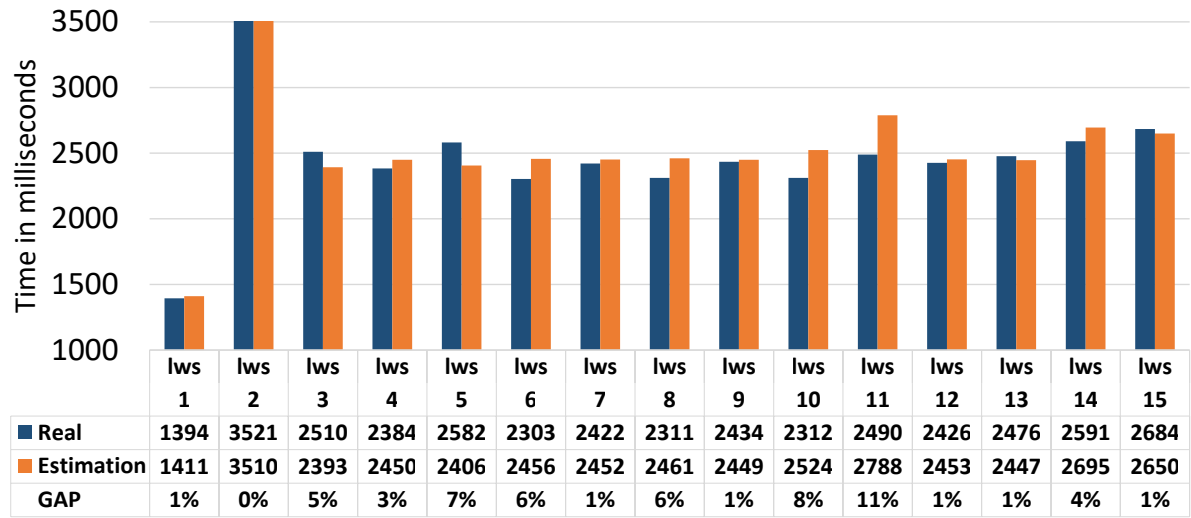


Figure A.10: Comparison between measured times and its estimation for transferring the execution results of 50K tasks in every combination of the LWS scheduler and scheduling context configurations from 1 to 15 (Table 5.1).

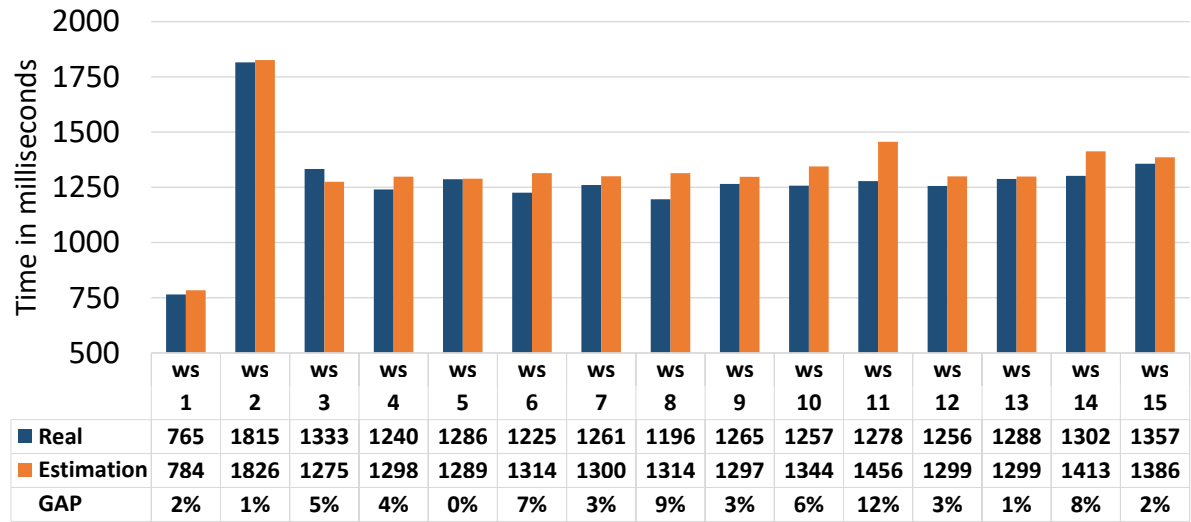


Figure A.11: Comparison between measured times and its estimation for transferring the execution results of 25K tasks in every combination of the WS scheduler and scheduling context configurations from 1 to 15 (Table 5.1).

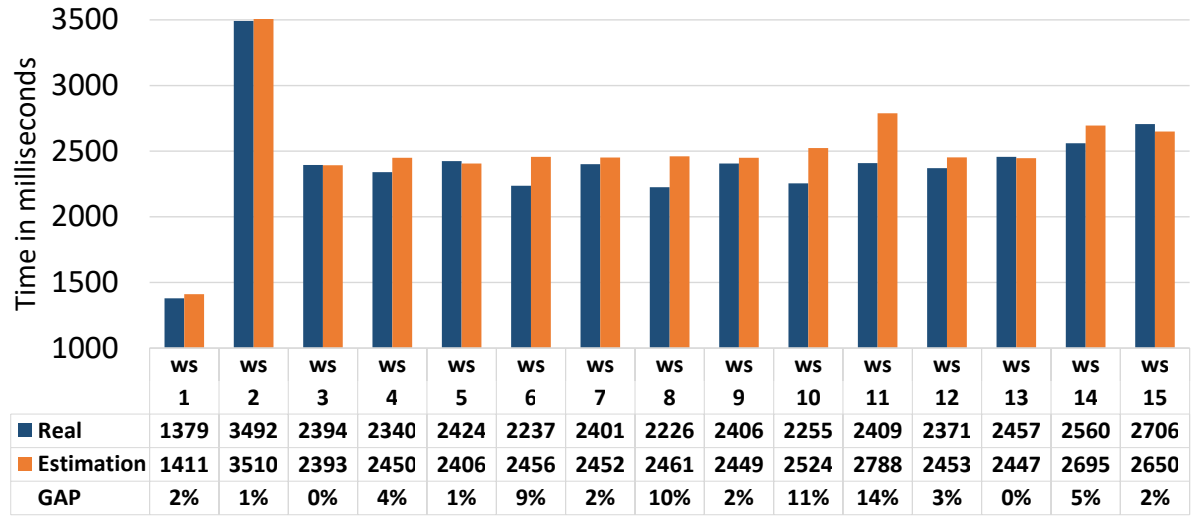


Figure A.12: Comparison between measured times and its estimation for transferring the execution results of 50K tasks in every combination of the WS scheduler and scheduling context configurations from 1 to 15 (Table 5.1).

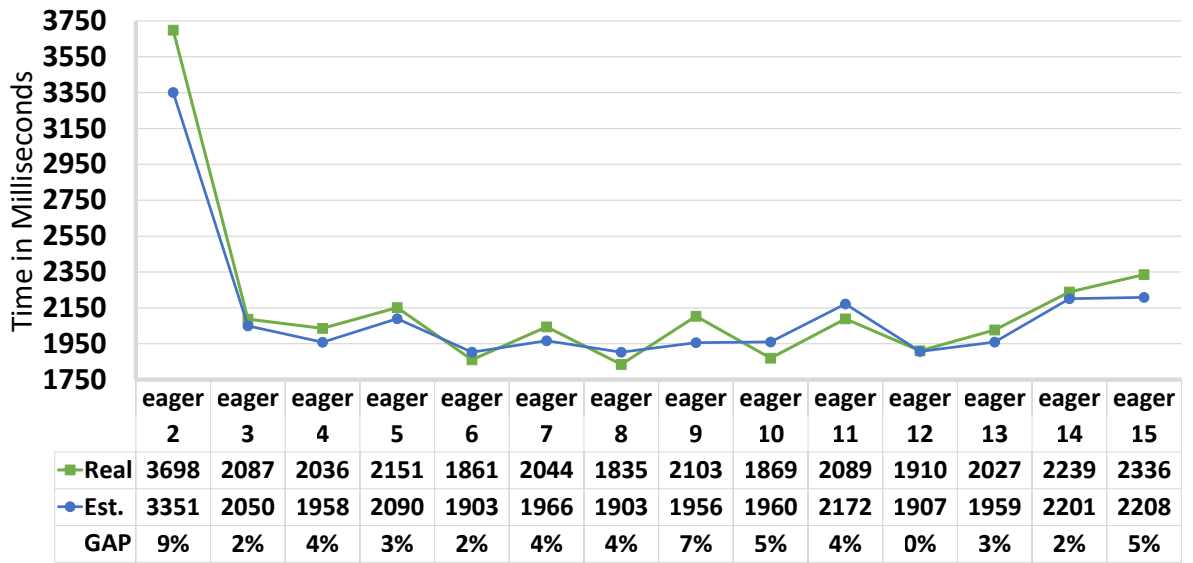


Figure A.13: Comparison between measured times and its estimation for overall execution of 25K tasks in every combination of the eager scheduler and scheduling context configurations from 2 to 15 (Table 5.1).

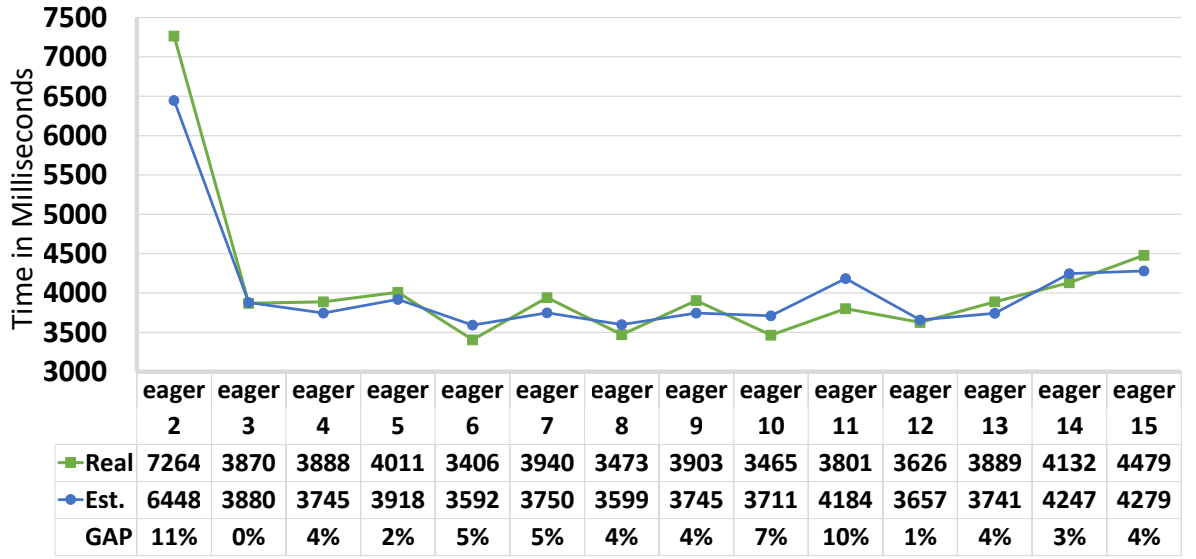


Figure A.14: Comparison between measured times and its estimation for overall execution of 50K tasks in every combination of the eager scheduler and scheduling context configurations from 2 to 15 (Table 5.1).

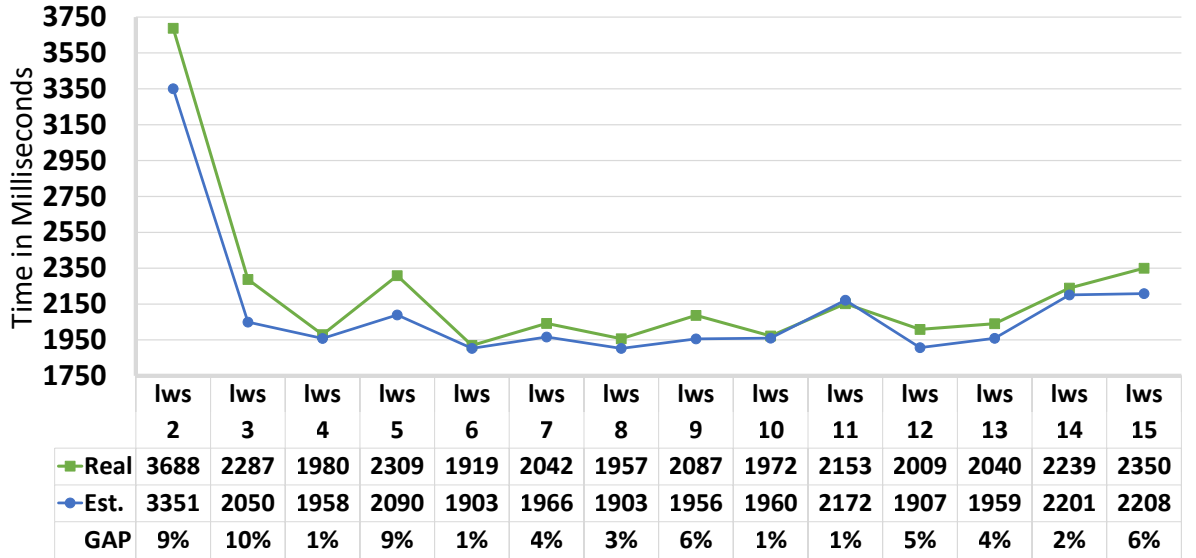


Figure A.15: Comparison between measured times and its estimation for overall execution of 25K tasks in every combination of the LWS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).

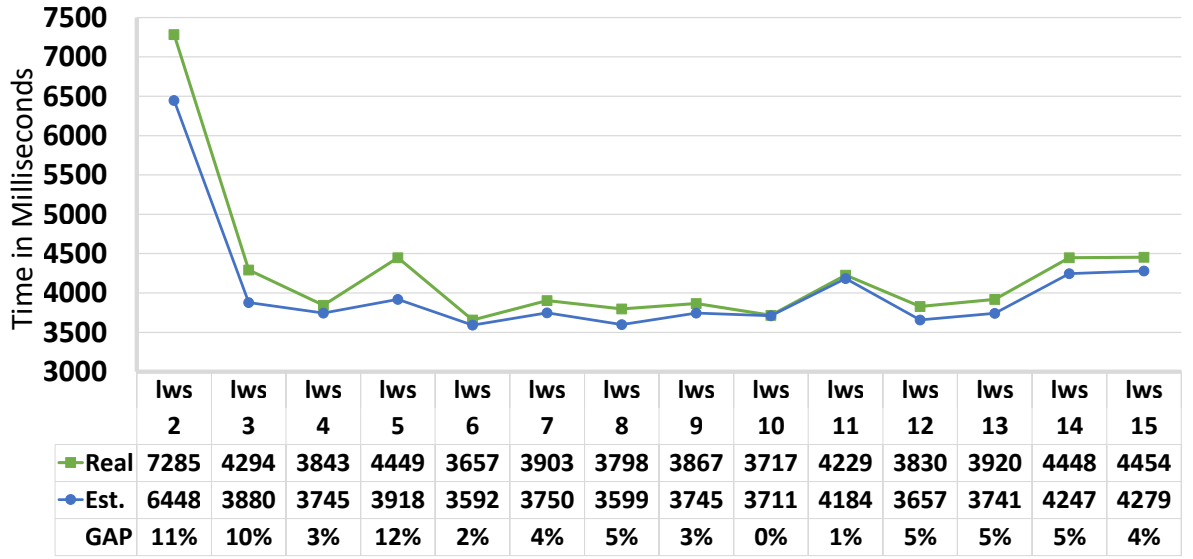


Figure A.16: Comparison between measured times and its estimation for overall execution of 50K tasks in every combination of the LWS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).

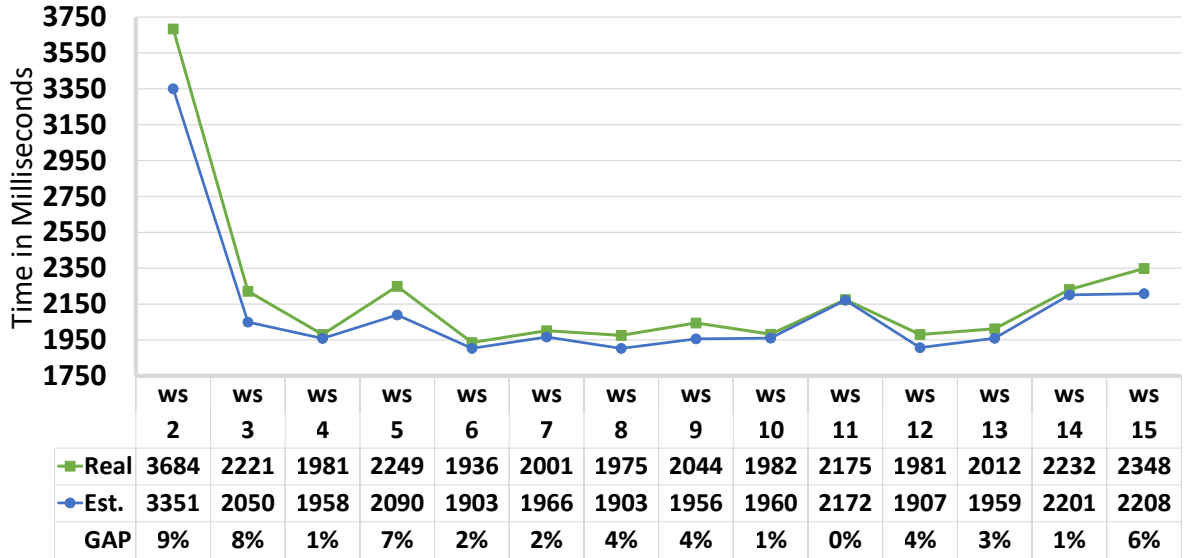


Figure A.17: Comparison between measured times and its estimation for overall execution of 25K tasks in every combination of the WS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).

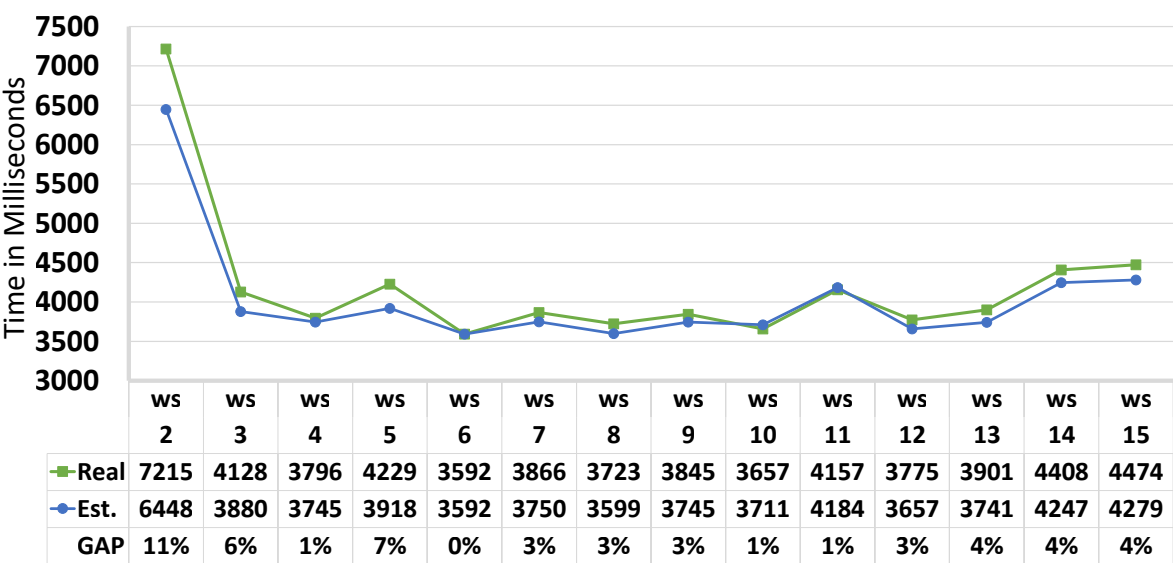


Figure A.18: Comparison between measured times and its estimation for overall execution of 50K tasks in every combination of the WS scheduler and scheduling context configurations from 2 to 15 (Table 5.1).

APPENDIX B - Glossary

- **Codelet:** records pointers to various implementations of the same abstract function.
- **Context:** see the definition of scheduling context on the glossary.
- **CPU:** Central Processing Unit. The microprocessor in a computer.
- **CUDA:** Compute Unified Device Architecture. The name of a parallel computing architecture for graphics cards by NVIDIA.
- **Data handle:** a StarPU component that keeps track of copies of the same data over various memory nodes. This data is registered by the application. The Distributed Shared Memory (DSM) is responsible for keeping them coherent.
- **Device:** a GPU connected to a host, a graphic card.
- **GPU:** Graphics Processing Unit. a microprocessor made specifically for accelerating graphics computations.
- **Host:** the main computer with possible devices connected to it. In this proposal it means a personal computer.
- **Kernel:** a piece of code that is executed on a device.
- **NVIDIA Corporation:** the company who developed CUDA architecture.
- **Scheduler:** schedules tasks to workers when they are ready to be executed, which means that the tasks do not have data or task dependencies. The workers pull tasks one by one from the scheduler.
- **Scheduler policy:** the rules that apply for pull tasks from the queue line. In our case, the scheduling policies are applied to contexts.

- **Scheduling context:** a StarPU extension that maps between a set of tasks and processors. Scheduling contexts can be created and destroyed on-the-fly. If a context is created, the programmer can dynamically map tasks to a set of processors.
- **Streaming Multiprocessor (SM):** a compute unit in the CUDA architecture, each CUDA device is made up of several streaming multiprocessors.
- **Task:** represents the unit of work to be run on the processing unit defined by the programmer. It has inputs and outputs data that must be transferred to the CPU or the CPU and GPU.
- **Warp:** a group of threads executed in parallel by one SM.
- **Worker:** execute the tasks, there is typically one per CPU computation core and one per GPU.