FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Autotuning Parallel Application in Heterogeneous Systems

João Alberto Trigo de Bordalo Morais

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Jorge Manuel Gomes Barbosa

February 10, 2017

Autotuning Parallel Application in Heterogeneous Systems

João Alberto Trigo de Bordalo Morais

Mestrado Integrado em Engenharia Informática e Computação

February 10, 2017

Abstract

Nowadays computational platforms have been evolving to the high computational power direction, however it requires a lot of energy to achieve such high performance with single but powerful processing unit. To manage this energy cost and keep with high performance, computers are built under the assumption of heterogeneous systems, in other words, computers that have different kind of processing units with different functions, such as CPU, GPU, Xeon Phi and FPGA. So, developers should take advantage of parallel activity and scheduling tasks by using the various parts of the heterogeneous systems.

Now the problem is how to efficiently achieve the highest performance possible when running software applications by taking the most advantage of such heterogeneous systems without jeopardizing the application performance and its results. Overall, the problem consists in the co-existence work of multicore specs, its parallelism and its shared cache problems; CPU parallelism and scheduling tasks; performance.

For this problem's solution is expected to find patterns and variables that helps tuning application with the best performance main goal in mind. Detecting these patterns and variables that improve applications, by making them parallelized, helps in the progression of how to make automatic paralleled code, since now-a-days making parallel code requires a lot of effort and time.

This kind of solution requires some validation process and metrics to make sure that it is doing its work and with proper results. To do so, the idea of the process' validation is going to be about comparing the behaviour of three different codes: a version of a serialized code; a version of the same code but with an expert manually paralleling it; and a version of the serialized code but «automatically» parallelized by a tool called Kremlin. The metric that was used to compare these three code versions is execution time in different experience environment.

The application of this work will help developing better automatic tools to make parallel code, which means in a long term, developers will have less burdened about creating parallel code, consequently, saving them time.

applications will achieve its highest performance possible in an automatic way and developers will have less burdened about creating parallel code, consequently, saving them time.

Resumo

Atualmente as plataformas computacionais têm vindo a evoluir na direção do elevado poder computacional, no entanto, estas requerem uma quantidade enorme de energia para atingir elevado desempenho individualmente. De modo a gerir este custo energético e manter a elevada performance, os computadores são construídos sobre a assunção de sistemas heterogéneos, isto é, computadores compostos por diferentes tipos de unidades de processamentos com diferentes funcionalidades, como por exemplo, CPU,GPU, Xeon Phi e FPGA. É neste sentido que os programadores devem tirar proveito de atividade paralela e escalonamento de tarefas recorrendo às várias partes que compõem o sistema heterogéneo.

O problema incide sobre como atingir de forma eficiente o maior desempenho possível quando se corre uma aplicação de software, tirando o maior proveito dos sistemas heterogéneos sem prejudicar o resultado e o desempenho da aplicação.

Para solucionar este problema é esperado encontrar/padrões e variáveis que ajudem a afinar applicações com o principal objetivo de atingir a melhor performance em mente. Detetar estes padrões e variáveis que melhoram applicações, colocando-as paralelas, ajuda no avanço de como fazer código aparaelelo automáticamente, uma vez que nos dias de hoje fazer código paralelo requer muito tempo e esforço.

Este tipo de solução requer um processo de validação e métricas para assegurar que se está a fazer o trabalho corretamente e com resultados aceitáveis. Para tal, a ideia da validação do processo consiste em comparar o comportamento de três diferentes códigos: uma versão sequencial de um código; a versão deste mesmo código mas paralelizada por um perito; e a versão do código sequencial mas paralelizado «automaticamente» por uma ferramenenta denominada de Kremlin. A métrica que foi utilizada para comprar estas três versões de código é o tempo de execução em diferentes ambientes experimentais

A aplicação deste trabalho irá ajudar a desenvolver melhores ferramentas para fazer código paralelo automático, siginificando que, a longo prazo, programadores estarão menos sobrecarregados a criarem código paralelo o que, consequentemente, poupar-lhes-á tempo.

Acknowledgements

This dissertation is a milestone in my academic career. I have been fortunate learn theories and concepts which would have been impossible if I had not extensively carried out the needed research. I am grateful to a number of people who have guided and supported me throughout the research process and provided assistance for my venture, both technically and emotionally.

I would like to divide these acknowledgements in two parts: firstly, I would like to mention those who helped me, directly or indirectly, in terms of technical assistance and knowledge, expertise, advices and guidance during the development of this dissertation. And the other part is related to those who guided me and made never give up and maintaining focused until the end.

Beginning with the first part, and the most important person during the development of my dissertation, my supervisor, Pofessor Jorge Barbosa, who guided me in selecting the final theme for this research. My advisor was there throughout my preparation of the proposal and the conceptualization of its structure. I would not have been able to do the research and achieve learning in the same manner without his help and support. His recommendations and instructions have enabled me to assemble and finish the dissertation effectively. Additionally, He, also, gave me the opportunity to collaborate in *Antarex* project, a n European project that «proposes a holistic approach capable of controlling all the decision layers in order to implement a self-adaptive application optimized for energy efficiency.»

I would also like to thank all the people from the laboratory where I performed my experiences and collaborated in Antarex's project, such as Professor João cardoso, João Bispo, Hamid Arabnejad, Tiago Carvalho, Luís Reis, Nuno Paulino, Ricardo Nobre and Pedro Pinto. They made my integration in this project much easier and helped me the best way they could and any time i asked.

A special thanks to Kremlin team, in the name of Saturnino Garcia, always ready to answer my questions and requests related to the Kremlin software.

For the second part, I would like to thank again my supervisor, Professor Jorge Barbosa, who accompanied me until the end of this process and did not allow me to give up, even in the most frustrating and dire moments.

I would also like to thank all my instructors and teachers, from the beginning of my education until the end of this fase, who throughout my educational career have supported and encouraged me to believe in my abilities. They have directed me through various situations, allowing me to reach this accomplishment.

Along side with my instructors and teacher, I would like to mention my friends from my year's course: João Pereira, Henrique Ferrolho, João Soares, Maria Marques, José Paulo, Leonardo Faria, Pedro Castro, Rita Ferreira, Jorge Teixeira, Maria Miranda, José Cardoso, Sofia Reis, Pedro França, Gabriel Souto, David Azevedo, Pedro Faria, Vitor Teixeira and João Almeida, Sara Reis, who accompained me through this five years in university and without them it would a lot harder. The lessons learned, good and bad moments are a part of me, like this dissertation.

I would also like to specially thank more friends from my course, such as Simão Felgueiras, Francisca Paupério, Vitor Esteves, João Leal, Daniel Nunes, Cristiano Seabra, Gonçalo Moreno, Filipe Reis, Eduarda Cunha, Daniela João, Carolina Azevedo, Mariana Silva, João Carvalho, Catarina Correia, Sofia Silva, António Ramadas, Sérgio Domingues, Tiago Frutado, Diogo Vaz, Gustavo Silva, João Almeida, Beatriz Baldaia, Daniel Machado and João Monteiro, who, in some way, helped, guided, given me advices and lived with me during these five years in university.

I would like to mention mu friends form my birth place an elementary school who now-a-days I still make contact: Rik Rodrigues, Hugo Fernandes, Simão Teixiera, Rita Pinto, Rita Franco, Maria Silva, Catarina Cadilha, Sara Carvalho, Daniela Peixoto, Tiago Cunha, João Novo, Francisca Painhas, Ana Teixeira, Windy Noro and Guilherme Polónia, who still mean a lot to me and are always present when I need.

A special thanks to Rui Castro and Nino Rocha who were always so comprehensive, supportive and helpful every time i needed.

Finally, my mother, Maria Bordalo Morais, my father, Alberto Morais, my brothers, André and Pedro Bordalo Morais, and my grandfather, Humberto Bordalo Xavier, I would like to thank them for all the support, advices and help during all times, specially during the development of this dissertation.

In the end, to all that had supported and helped me along the course of this dissertation by giving encouragement and providing the moral and emotional support I needed to complete my thesis. To them, I am eternally grateful

João Bordalo

*“**Katsumoto:** You believe a man can change his destiny?*
***Nathan Algren:** I think a man does what he can until his destiny is revealed to him.”*

In movie: *THE LAST SAMURAI*

Contents

List of Figures

List of Tables

Abbreviations

CPU	Central Processing Unit
GPU	Graphic Processing Unit
FPGA	Field-Programmable Gate Array
OpenCL	Open Computing Language
CHC	Cooperative Heterogeneous Computing
OpenMP	Open Multi-Processing
OS	Operative System
WWW	*World Wide Web*

Chapter 1

Introduction

1.1 Context

Previously, computer systems were built to maximize their processing power in compactness and individually because programs were developed with a sequential approach. With the advance in microchips' technology, computers increased their processing capacity per volume, however some issues arose, such as high energy cost, high temperature and low equipment durability. To solve these issues some measures needed to take place in order to make computers systems more reliable, durable, efficient, and powerful.

Recently, the computing industry has moved away from exponential scaling of clock frequency toward chip multiprocessors in order to better manage trade-offs among performance, energy efficiency, and reliability [Dat08]

Combining different computer processing components, such as CPU, GPU Xeon Phi and FPGA, in a single computer system removed some heavy burden in the main processing core, making the computer system with better performance and reliable. However some concerns arose: how to properly use these components without jeopardizing the computer system and application performance. Some processing components can handle specif jobs better then others and combined the computer can achieve a whole new performance level; for instance, the use of a GPU together with a CPU to accelerate deep learning algorithm, analytics, and engineering applications [Nvi], however this kind of utility is not yet well optimized and its utility is only recently emerging.

1.2 Motivation and Goal

My motivation for this thesis is to advance a little further on the field of the automatic code parallelization and replace the manual parallelization labour because it requires a lot of time and effort to achieve significant performance.

1.3 Statement of the Problem

In the filed of achieving the highest possible performance in applications, it could be divided in for levels: hardware components level; transition between hardware and software level, the operative system level; the software level, related to the programming language; and the user level, the applications / program job, algorithms that it uses. The problem this dissertations approaches is related to the software level and the transition between hardware and software level.

Parallelizing code requires getting in touch with the programming language properties, software level, and using threads to make code paralleled, transition level between hardware and software, since the threads handling is done by the operative system. The real problem is parallelizing code requires a lot of effort, time and knowledge because, firstly, to apply parallelism, the target application must be analysed in order to find if it is possible to be parallelized. To do this, and if the applications uses complexed algorithms that requires expertise in other fields, such as biologic, mathematics, physics, it requires time to understand the algorithm and, then, if it has blocks that can be parallelized. Secondly, if the target programming language is suited to implement such application in order to take advantage of the parallelism. And, finally, how the parallelism is achievable with out jeopardizing applications result and performance.

In conclusion, it requires a lot of time, effort and knowledge to achieve notable impacts in applications' performance, since it is made manually.

1.4 Purpose of the Work

Applying parallel methods requires a lot because there is to many factors an variables to take in to account and this is done manually.

The purpose of this dissertation is to find patterns and what variables makes the code parallelizable and find what exists in the state of art, and their performance impact, that helps in achieving one step closer to the automatic parallelization.

1.5 Significance of the Work

Making parallel code increases applications performance and, if making code paralleled was a possible future, that would increase applications performance and. as well, increase developers performance in building applications and solutions efficiently because it would spare time and effort.

With this dissertation it is hopped that attaining automatic ways to parallelize code is achievable and prove that code being parallelized automatically is viable and trustworthy in term of performance, results and speed.

1.6 Research Hypothesis

This dissertations pretends to state that, in first place, parallelizing code is worth using and achieves grreat results in terms of performance. Secondly, automatically parallelizing code is possible and thirdly, has almost, if not, the same results as doing it manually. Additionally, this work pretends, as well, to state that Kremlin tool is an excellent starting point to make parallel code automatically.

1.7 Research Questions

The aim of this research is to find and answer for the following questions:

- Is it possible to achieve high performance level in applications in an automatic way?
- If so, how can it be achievable? Can it totally replace an expert?
- Since exists tools, like Kremlin, which help to, automatically, parallelize code, how acceptable are their results?
- How this specific tool can help in getting one step closer to automatic code parallelization?
- In which way can Kremlin be better than an expert?

1.8 Dissertation's Structure

This dissertation is divided in the following chapters:

1. **Introduction** 1

   This addresses the overall context of my dissertation, motivation and goal of the developed work, what is the problem, the purpose of the work, th impact that this work will have, the hypothesis that this research wants to prove, the questions behind the research and how this dissertation is structured.

2. **Achieving the Highest Processing Power** 2

   This chapter is the state of the art of my thesis' scope and establishes the information, knowledge and work developed so far. In this chapter there are three sections. The first section is related to the context of the state of the art in the filed. The other two sections are two different but complementary approaches which help and describe the state of the art.

3. **Matrix Multiplication** 3

   Still related to this dissertation state of the art, in this chapter it will presented and overview about the current state of the art for matrix multiplications, more specifically, two possible algorithms, their pseudo code, vantages and disadvantages. So, this chapter is divided in three sections: Matrix multiplication overview, Matrix multiplication generic algorithm and Matrix multiplication by line algorithm.

4. **Methodology** 4

   The Methodology chapter explains how the work will unfold, starting with the followed steps, executes experiences and what data was obtained and how this data will be analysed a validated. So, this chapter is divided in four sections: Research method, Data collection from conducted experiences, Data analysis method and Data validation.

5. **Results and Discussion** 5

   In this chapter is explained, in detail, the results obtained from all conducted experiences and the meaning and conclusions drawn from each one, the relation between them and the overall impact. So, this chapter is divided in two parts: firstly the analysis made from Kremlin's report and the comparison between Original code, Manual parallized code by an expert and Kremlin's indications to parallize code using the data from Kremlin's report.

6. **Conclusion** 6

   To sum up the work and reinforce the arguments the Conclusion chapter is divided in two sections: General conclusions and Future work. In General Conclusions section, is described the conclusion of each experiment and the overall conclusion for the experiences and the relation with the research hypothesis and the answer for the questions made. In the Future work section is presented what could be the next step and what can be done with the developed work made in this dissertation.

7. **Appendices ??**

   The last chapter of this dissertation, Appendices, presents, in two sections, the code developed for the conducted experiences and, for the second section, the information provided from Kremlin's report.

Chapter 2

Achieving the Highest Processing Power

The introduction describes a brief overview about each content of each chapter this report is made up with. This chapter will focus on the state of the art in how to achieve the highest processing power. Related work and already known technologies are the main point in this chapter.

2.1 Introduction

Following the context introduced in the previous chapter, the idea of having different processing components in a computer system doesn't improve the applications performance on its own. This is where the developers' work is crucial to take advantages of such different systems. The developers' work is to schedule the application's tasks to the different components so that these components can work simultaneously, avoiding overheads caused by their parallel activity, accessing memory at the wrong moment, memory conflicts, task dependency, wrong application's results compared with the sequential application. [LR]

As mentioned previously, trying to create parallelized code can arise many problems and must be handled so the applications don't lose their functionalities. In order to do so, it requires a lot of time and effort to make it correctly parallelized. So trying to make code parallelization automatic is the next step in the direction of taking the most advantage of heterogeneous systems which, consequently, improves applications performance.

This chapter is divided in two parts: one part will focus in the system's heterogeneity, how they can be used in favor of enhancing performance; and the main point of the other part is taking advantage of parallel activity by transforming sequential code into parallelized code.

2.2 Using Computers' Heterogeneous Components

Technologies and frameworks in this field have been developed in order to manipulate and control efficiently the different processing components. The main goal of this technologies is to optimize

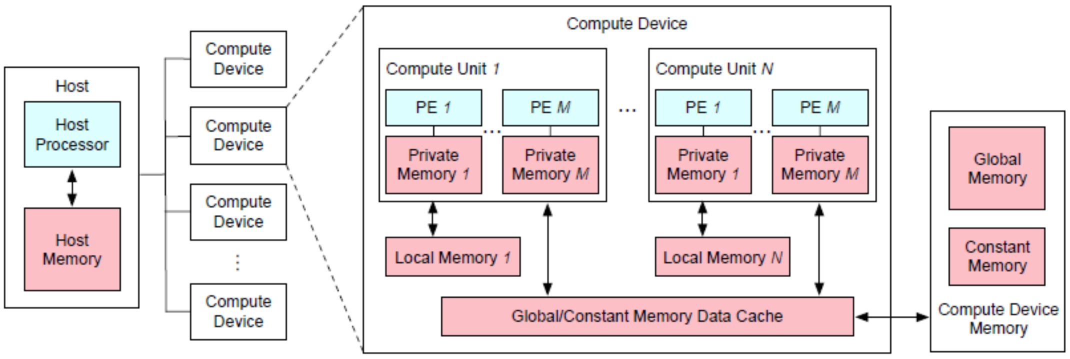

Figure 2.1: The OpenCL platform model and the OpenCL memory model

application parallelization; application memory management; application workload; application scheduling queue and application kernel dimension.

An interesting fact is that the following software/frameworks that will be present are built, as its bases, under OpenCL programing language due to the fact that this language's use is targeted to heterogeneous parallel programing with CPUs and GPUs.

2.2.1 OpenCL

OpenCL is a programming language for heterogeneous parallel programing targeted to CPUs, GPUs and other processors [She]. In a small brief, this language is designed to take advantage of different types of processors and facilitates heterogeneous computing integration in applications' code. The user programs in a virtual platform and the source code that has been developed there is compatible for any system that supports OpenCl. Additionally, OpenCL allows users to control the applications' tunning parallelism through its hardware abstraction. In figure 2.1 there is an idea of the OpenCL plataform model and memory model for a better understanding of this hardware abstractions that was previously mentioned.

The OpenCL's programs has two parts: the compute kernels that are executed depending the number of processing devices; and the program that will be run. The program creates a set of commands and puts them in a queue for each device, additionally, to manage the execution of each kernels, additional commands are queued in the different kernels. When the computation is finished, the result data, from the previous kernels activity, return back to the original program.

2.2.2 StarPU

StartPU is a software tool with the purpose for programmers to use the computing power available in CPUs and GPUs, wihtout needing to care about if their programs are adapted to a specific machine and its processing components. [Sta] In fact, StartPU is a run-timpe support library that provides scheduling applications-provided tasks on heterogeneous environments, such as CPUs and GPUs. Additionally, it comes with programming language support, for the programming C language extensions and for OpenCL.

Programs submit computational tasks, with CPU and/or GPU implementations, and StarPU schedules these tasks and associated data transfers on available CPUs and GPUs. The data that a task manipulates are automatically transferred among accelerators and the main memory, so that programmers are freed from the scheduling issues and technical details associated with these transfers.

StarPU takes particular care of scheduling tasks efficiently, using well-known algorithms from the literature (Task Scheduling Policy). In addition, it allows scheduling experts, such as compiler or computational library developers, to implement custom scheduling policies in a portable fashion (Defining A New Scheduling Policy).

2.2.3 Twin Peaks

"Software platform that enables applications originally targeted for GPUs to be executed efficiently on multicore CPUs", mentioned by Jayanth Gummaraju, Laurent Morichetti, Michael Houston, Ben Sander, Benedict R. Gaster, Bixia Zheng, in the paper *Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors* [Gum10]. This is a small definition of the Twin Peaks' job. The aim of this software is, firstly, to program applications using an API written in OpenCL; secondly, to compile the applications code to, for instance, add syntactic and semantic checks to make sure that the kernels meet the OpenCL requirements; and execute applications in the heterogeneous environment using CPUs and GPUs.

2.3 Using Code Parallelization

Great advances have been made in the code parallelization. However, currently this kind of practice (the code parallelization) mostly is done by programmers and it requires a lot of effort, time and knowledge. It requires knowledge in the best practices related to what should and can't be parallelized, good knowledge on the code: its functionalities and its correct outputs because without these knowledges the chances to parallelize code correctly would be low since it is important to know if, firstly, is possible to parallelize and if, secondly, the parallelization doesn't jeopardize the programs results, outcomes and performance; to sum up, it requires time and effort to get a deep understanding of the code and to try if the code is correctly parallelized. [Jeo]

Since this practice is very costly, although grants great results at performance levels, this field has been developing ways to have results less costly, mostly in effort and time-consuming. These developments created tools to help programmers develop parallelized code, using OpenMP directives, or software tools which recommend possible parallelized regions and its theoretical speed up gain, with Kremlin, or even a way to estimate how much can a program be parallelized, with Kismet software. [GJ]

The following software tools or instrument languages that will be presented have, as its base support, OpenMP directives to help in parallelizing code, or at least, to measure performance, excluding the last one, ATuner-IL.

2.3.1 OpenMP

OpenMP was designed to be a flexible standard, easily implemented across different platforms. the main objectives are: control structure, the data environment, synchronization, and the runtime library.

In terms of how it really does its job, OpenMP was designed to exploit certain characteristics of shared-memory architectures. The ability to directly access memory throughout the system , combined with fast shared memory locks, makes shared-memory architectures best suited for supporting OpenMP. in practice, OpenMP is a set of compiler directives and callable runtime library routines that extend Fortran (and separately, C and C++) to express shared-memory parallelism. [Nc98]. To be more precise, OpenMP provides standard environment variables to accompany the runtime library functions where it makes sense and to simplify the start-up scripts for portable applications. This helps application developers who, in addition to creating portable applications, need a portable runtime environment. OpenMP has been designed to be extensible and evolve with user requirements. The OpenMP Architecture Review Board was created to provide long-term support and enhancements of the OpenMP specifications.

2.3.2 Kremlin

The true purpose of Kremlin lies in asking the following question: "What parts of this program should I spent time parallelizing?" [Par]. So, in overall, Kremlin profiles a serial program and tells the programmer not only what regions should be parallelized, but also the order in which they should be parallelized to maximize the return on their effort. Giving a non parallelized code, Kremlin guides the programmer how to achieve better performance in its program though parallelization by presenting a list of code regions that could be parallelized. this list contains a plan that will minimize the number of regions that must be parallelized to maximize the programs performance, though parallelization.

At the core of the Kremlin system is a heavyweight analysis of a sequential program's execution that is used to create predictions about the structure of a hypothetical, optimized parallel implementation of the program. These predictions incorporate both optimism and pessimism to create results that are surprisingly accurate. [GJLT11]

Overall, Kremlin is an automatic tool that, given a serial version of a program, will make recommendations to the user as to what regions (e.g. loops or functions) of the program to attack first. [GJL$^+$12]

2.3.3 Kismet

Opposed to Kremlin, Kismet helps mitigate the risk of parallel software engineering by answering the question, "What is the best performance I can expect if I parallelize this program?" [Par]. Kismet profiles serial programs and reports the upper bound on parallel speedup based on the program's inherent parallelism and the system it will be running on.

Kismet performs dynamic program analysis on an unmodified serial version of a program to determine the amount of parallelism available in each region(e.g. loop and function) of the program. Kismet then incorporates system constrains to calculate an approximate upper bound on the program's attainable parallel speedup. [Tay]

In order to estimate the parallel performance of a serial program, Kismet uses a parallel execution time model. Kismet's parallel execution time model is based on the major components that affect parallel performance, including the amount of parallelism available, the serial execution time of the program, parallelization platform overheads, synchronization and memory system effects which contribute in some cases to super-linear speedups.

2.3.4 Atune-IL

Atune-IL is a general instrument language that helps finding the best values to tune parallel applications. It is based on a language-independent #pragma annotations that are inserted into the code of an existing parallel application [SPTK09].

Atune-IL is based on the assumption that programmers want to change the values of a variable between subsequent tuning runs in order to observe the relative performance impact. The Atune-IL purpose is to help automating this process by allowing programmers manually defining the tuning variables.

Firstly, in order to define those variables,and everything else, all Atune-IL statements must be preceded by the #pragma atune prefix. Defining variables is not the only thing that Atune-IL can do. This is what Atune-IL can define:

- **Defining Numeric Parameters**

  Considering that the tuning variable is the number of threads, numThreads, to let the autotuner vary this variable, the programmer adds a #pragma annotation after the variable, followed by SETVAR numThreads to mark it as tunable. Using TYPE int, the domain of trial values is constrained to integers. The value range is defined by VALUES 2-16 STEP 2, implying that numThreads will be set to the values 2,4,... ,16.

- **Defining Architectural Variants**

  A powerful feature of Atune-IL is that the TYPE of values in a SETVAR statement need not be numeric. Assuming that this program implements a sorting routine in a generic way, we can go to the point where the employed sorting algorithm is first instantiated and insert an annotation with TYPE generic; this allows us to include host language code for the creation of each algorithm instance. While the auto-tuner just sees two options that can be tried out in different tuning runs, it will actually try out two architectural variants of the program

- **Additional Support for the Optimization**

  The SETVAR keyword has additional options that were not mentioned yet. A value in the specified interval may be defined as the START value that is tried out first. This is useful when a variable that controls the number of threads should be tried out first with the number of available hardware threads.A WEIGHT number may quantify the importance of the annotated variable for the overall optimization, and the SCALE nominal or SCALE ordinal keyword may inform Atune that this variable has nominal or ordinal scale. With this information, the optimizer may treat such variables in a different way

2.4 Overview

As mentioned before, the previously presented software tools, for both cases (using computers' heterogeneous components and using code parallelization) have their base support even being a programming language, for OpenCL, or a set of compile directives, for OpenMP, excluding Atune-IL. Those software tools have improved applications performance somehow, which is already good. However, looking as a software that can do everything on its own, with the minimum programmer's input, in other words, that can do things almost automatically, none of them can make it. The only software tool that is close to that automation is Kremlin because it gives what a developer should do in their code in order to increase its efficiency and performance.

Both approaches, using computers' heterogeneous components and using code parallelization, have the role to answer the state of the art premise: "achieving the highest processing power".

Chapter 3

Matrix Multiplication

This chapter is about matrix multiplication algorithms. Two possible versions for the same problem are presented in the following sections. The last sections of this chapter is a recap of both algorithms.

3.1 Introduction

In the state of the art regarding matrix multiplication, there is an abundance in algorithms trying to optimize these computational operations [GG08] [YZ05], since it is used in, for instance, GPUs [FSH04].

Two different algorithm versions for the same problem are presented because they are going to be used as mean to prove that, firstly, parallel code increases performance in a huge scale, and, secondly, parallel code made in an automatic way can have similar results as the same code being manually parallelized by an expert.

The first algorithm is the general and classic approach of the problem. The second algorithm is an enhanced version of the same algorithm but with a new perspective, taking advantages of the hardware components, such as cache memory.

3.2 Classic Algorithm)

The classic version of this algorithm [FSH04], computes the multiplication of two matrix with the same dimension as it follows:

$$C_{ij} = (AB)_{ij} = \sum_{k=0}^{n-1} a_{ik}b_{kj} = a_{i1}b_{1j} + ... + ai_{(n-1)}b_{(n-1)j} \tag{3.1}$$

Where:

C$_{ij}$ Resulting matrix from matrix multiplication between A and B;

$(AB)_{ij}$ matrix multiplication between A and B;

n Matrix size NxN;

a_{ik} Element from line i and column k belonging to Matrix A;

b_{kj} Element from line k and column j belonging to matrix B.

This algorithm multiplies the first line of matrix A for each column of matrix B, as it is presented in 3.1

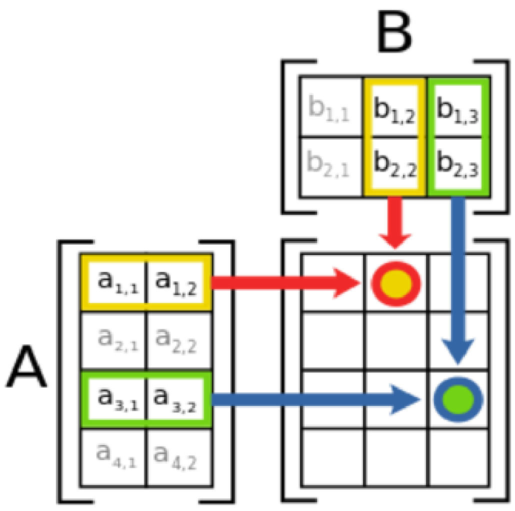

Figure 3.1: Matrix multiplication between matrix A(4x2) and B(2x3), representative matrices.

So, the pseudo code used to implement 7.1 is the following:

```
for(i=0; i<n; i++) {
    for( j=0; j<n; j++) {
        for( k=0; k<n; k++) {
            C[i][j]+= A[i][k] * B[k][j];
        }
    }
}
```

3.3 In Line Algorithm

This algorithm version computes matrix multiplication differently comparing to the classic algorithm:

$$C_{ij} = (AB)_{ij} = \sum_{j=0}^{n-1} a_{ik}b_{kj} = a_{i1}b_{1j} + ... + ai_{(n-1)})b_{(n-1)j} \tag{3.2}$$

Where:

C$_{ij}$ Resulting matrix from matrix multiplication between A and B;

(AB)$_{ij}$ matrix multiplication between A and B;

n - Matrix size NxN;

a$_{ik}$ Element from line i and column k belonging to Matrix A;

b$_{kj}$ Element from line k and column j belonging to matrix B.

This algorithm multiplies an element form matrix A for the correpondent line of matrix B, as it is suggested in 3.2:

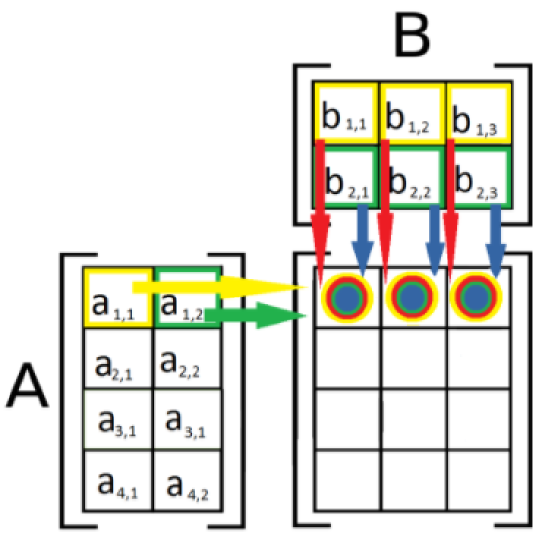

Figure 3.2: Matrix multiplication in line between matrix A(4x2) and B(2x3), representative matrices.

So, the pseudo code used to implement 7.2 is the following:

```
for(i=0; i<n; i++) {
    for( k=0; k<n; k++) {
        for( j=0; j<n; j++) {
            C[i][j]+= A[i][k] * B[k][j];
        }
    }
}
```

3.4 Overview

In terms of code implementation, the biggest difference in these two algorithms is in the permutation between the second and third inner *for* loop, which, consequently, has an huge impact in performance [LWR].

This slight alteration in the classic algorithm makes matrix multiplication in line much faster when implemented in an application. The way and order of the operations are made for this algorithm is what makes this algorithm better: the order the calculation is made takes advantage of what is preloaded in cache memory. By doing though, it reduces the number of times a cache miss happens, which, consequently, reduces the number of times information is replaced in cache memory, reducing the number of times memory cache is written, therefore, reducing the waiting time until cache memory has the required information, reducing the overall application time, which improves application performance.

However, the matrix multiplication in line algorithm is best suited for long matrix sizes, for instances, if the matrices fit in cache memory, this algorithm, that reduces the number of cache memory swaps, makes almost no differences in overall performance.

Chapter 4

Methodology

4.1 Introduction

According to the state of the art presented in chapter two, there are many means to, in some kind of automatic way, improve an applications performance. During my research, my focus was to find ways to automatically enhance the execution time in applications and programs. For this propose, Kremlin had a crucial impact in other to understand the viability of automatically parallelize code.

To study the utility and impact of automatic tools, the matrix multiplication algorithm will be used as a reference to make the performance comparison between original algorithm, an expert manually parallelizing the original algorithm and using the Kremlin's indications to parallelize the original algorithm.

To increase the credibility of this experiment, two similar algorithms for the matrix multiplication were used. As mentioned and explained in the chapter two, there is the traditional way of multiplying square matrices, naming as a quick reference *Mult* algorithm, see in the appendix's list 7.1 this algorithm implementation, written in C++ programming language; and the optimized algorithm that multiplies each element from the first matrix with the correspondent line of this matrix element but for the second matrix, naming this algorithm as *MultLine*, see in the appendix's list 7.2 this algorithm implementation, written in C++ programming language. These algorithms differs from one another in the variables preparation and the order of the loops, which differs how the memory accessed. The *MultLine* algorithm is an optimized version for matrix multiplication because it takes advantages of what is preloaded in cache and starts pre-calculating the intermediate values that will lead to the final and correct result of the multiplication, which means that won't be needed to load unnecessary values to cache memory and/or will need afterwards.

Several experiments were conducted to understand the influence of Kremlin's indications versus code being manually parallelized by an expert. The data's length, in this case, the matrix size; the number of threads used and if the code was parallelized were the used metrics to evaluate the results, based on a comparison of the execution time.

In this chapter it is explained the methodology and the steps followed to report in the Results and Discussion chapter the results and conclusions obtained from the performed experiences. This chapter also includes detailed information of the acquired data from the conducted experiences, as in, how it is obtained and its meaning; also includes the methods that were used to analyse the obtained data and the reason behind those methods; and, in the end, how the data was validated in order to verify its correctness, accuracy and reliability.

4.2 Research Method

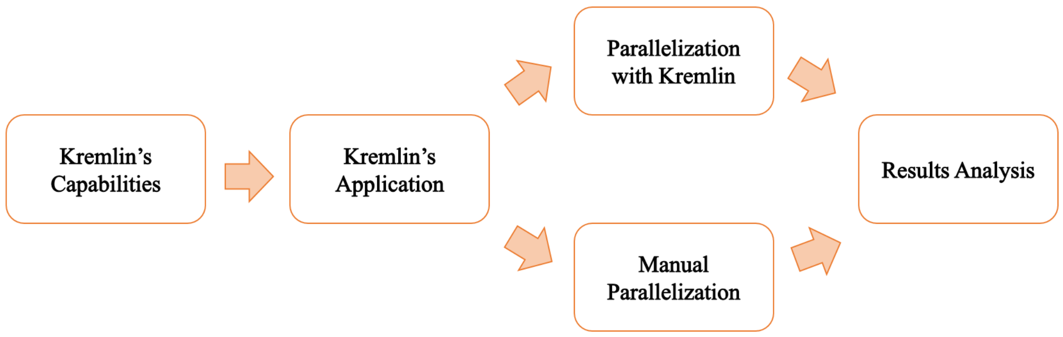

Figure 4.1: Followed up methodology

In Figure 4.1 is outlined the steps that were followed to study the impact of the code being automaticaly parallelized. This methodology has five states. Firstly and using simple applications, an evaluation was made for Kremlin's tool in order to understand how to use this software tool and evaluate the results that it can achieve, for instance, if it has similar results comparing with an expert parallelizing manually the same code. After this, Kremlin will be applied to a set of codes with specific characteristics.

Before applying Kremlin, I manually parallelized the same sample of code in order to evaluate the results and, afterwards, compare with the Kremlin's output. Since these states (the experiences with Kremlin and the manual code parallelization) required several attempts there were transitions between these states.

Finally, in the last state, after several attempts and tuning exercises applied to both code cases (Manual and Kremlin), data was collected from this experience to evaluate and validate its correctness in order to conclude how helpful can automatic parallelization can be.

To sum up, this methodology as three main stages: learn and evaluate Kremlin's uses and results; finding the tuning parameter through several attempts using Kremlin's outputs and manually parallelize the application's code; and, in the end, compare and analyse the results in every attempt to take conclusions;

4.2.1 Deep learn on Kremlin's usage

Firstly, and according to all tools/frameworks mentioned presented in the second chapter, *Achieving the Highest Processing Power*, in the *Using Code Parallelization* section 2.3, Kremlin was chosen because it presented the best results, easy usage and accessibility comparing to the others 2.3.2.

Kremlin is a tool that indicates, for a sequential program, which block can be parallelised and some metrics theoretical calculated, such as, overall speedup; self parallelism for each block; the ideal time reduced for each block, in percentage; the actual time reduced for each block, in percentage; and the block coverage, in percentage. The way this tool was used is as it follows: first, an object file, **.o* extension, is required from the compilation of a sequential code. Afterwards, it is time to use the Kremlin's compiler with the generated object file so that it can profile the application. In order to do so, Kremlin's compiler runs the program as it is supposed to work. Now that the profiling is done, Kremlin generates the indications that should be followed to parallelize de provided sequential code. It also includes the blocks that can be parallelized and the impact of this theoretical parallelization with the calculations done during the profiling. Since this parallelization report is done, the developer has to interpret it, confront with the code an apply.

The Kremlin's usage seems easy, linear and fast forward, however it has some limitations that I experienced during the learning of Kremlin's capabilities: Kremlin's requires a specific environment mentioned in the Kremlin's repository [Kre]. It requires several software, libraries, compilers installations and a modern Unix operative system as its bases, such as MAC OS, RHEL 7 or other Linux distribution compatible with the software specification required. Additionally, when installing the Kremlin's tool, some minor fixes are required in order to successfully install.

From the experiences that I have been through, Kremlin has another limitation: it can not compile and profile all kind of programs: it can only profile programs that use C/C++ as its programming language; programs that take advantage of data structures from the *Standard Library*, such as, stack, list, priority queue, queue, list, hash table, map, multimap, heap, and rest, since it doesn't recognize these structures; another Kremlin's limitations is its capability of compiling programs that have a deep function call level greater that seven. By deep function call level I mean the depth a function has starting from the *main()* function until it is called, like a functions call tree. For instance: in a program there is the *main()* function, a first level, that calls a *foo1()* function, and this function calls a *foo2()*, that this calls a *foo3()* function, and so on. In this case, the depth of *foo3()* function is four. Another small issue that Kremlin's tool has is the definition of the iterator variable used in the *for*'s loops must be defined outside of the loop, as it is in C programming language.

4.2.2 Kremlin's application in specific code samples

After all the experiences made in the previous state and as mentioned in the introduction of this chapter, the matrix multiplication algorithm was used to see the potentialities of Kremlin's compiler to profile and identify the regions that can be parallelized. So, Kremlin was used in two similar, relatively similar in code structure, matrix multiplication codes. The reason behind the choice was because these two versions of the algorithm are really close to one another, which means that the testing environment is similar to one another, consequently, the results should be similar.

4.2.3 Code parallelization with Kremlin's data

Kremlin's tool just points the regions/blocks where the program can be parallelized. In both code samples there are various numbers of inner *for* loops, for the *Mult* code there are three inner *for* loops, which one of them has a degree of three and the rest a degree of two; and for the *MultLine* code there are four inner *for* loops, which one of them has a degree of three and the rest a degree of two as well. At this time, after reading the report provided by Kremlin's tool, the developer must locate the loops, apply which loop should be parallelized, if it should be, and in case of inner *for* loops, what loop should be parallelized using the OpenMP *pragma* directives.

In my case, I followed all the instructions provided by Kremlin, located all the *for* loops blocks indicated by kremlin's tool and applied the OpenMP *pragma* directives.

Following the two reports, 7.8 7.9, and looking at the code's structure for both codes, it can be divided in 2 bigger parts: the *for* loops used for matrices initialization and the *for* loop for the matrix multiplication. With this information, understating the code and using an expert knowledge, the code parallelization was done.

4.2.4 Manually Code parallelization

In order to not be manipulated by the Kremlin's indications, the both codes were previously manually paralleliszd, this way it was guaranteed that the expert parallelization wasn't bias nor influenced.

For this parallelization, as mentioned before, it requires knowledge in, firstly, matrix multiplication algorithm; code understanding; best practice in what can and can not be parallelized, taking into account the overhead that could occur; and understand the thread behaviour in order to make it do the proper job without jeopardizing the programs outputs and/or performance.

Analysing the code, only the *for* loop for the matrix multiplication was parallelized and applied the OpenMP *pragma* directives to the innermost *for* loop. In this case, each code as a slight difference because for *Mult* code each value of the result matrix must be calculated individually, so each thread is responsible for it and must treat that value as a private variable that isn't shared by the other threads. In the opposite, and since the *MultLine* code calculates the values by adding the multiplication to the respective matrix's cell, each thread does not need to have their own private variable.

The bigger part of the code responsible for the matrices initialization wasn't parallelised, unlike in kremlin's case, because the gain would be noticeable on a large matrix size or it even could cause thread trampling, which could lead an overhead increase.

4.2.5 Results analysis

To obtain the final execution time of each implementation (Original Matrix Multiplication 7.1, Original Matrix Multiplication by line 7.2, Manual Matrix Multiplication by an expert 7.3, Manual Matrix Multiplication by line by an expert 7.4, Kremlin Matrix Multiplication 7.5 and Kremlin Matrix Multiplication by line 7.6), these six implementations suffered many modifications and tweaks since this process is a try-error until it is found the believed best parallelization. It is hardly possible to parallelize a whole program at the first try.

After compiling all these implementations and registering all the execution time for different matrix sizes and number of threads (not applied to the Original codes), this data was organized so it could be used to compare results and conclude about the performed experiences.

4.3 Data collection from conducted experiences

From all the developed work, the obtained data can be divided in two moments: Kremlin's indications reports and the execution times for the six code variations.

4.3.1 Kremlin's indications reports

For the *Mult* 7.8 and *MultLine* 7.9 codes were generated a report done by Kremlin. This report displays for each parallelizable block:

Time reduced percentage of time reduced if parallelization is implemented;

Ideal Time reduced percentage of ideal time reduced if parallelization is implemented;

Coverage percentage of sequential execution time in a block;

Self Parallelism amount of parallelism in a block;

Parallelism type classification of the parallelizable block;

Loop location block lines range of the parallelizable block;

Function location function name of the parallelizable block, mentioning the line where the function is defined;

File location file name of the parallelizable block, mentioning the line where the function is called;

The guidelines given by the report must be followed by the order it is suggested because the first detected block has the biggest impact in programs performance and should be parallelized first. The Coverage and Self Parallelism are metrics that indicates the speedup of the block, which, consequently, interferes with the time reduced. This block speedup must be equal or less then the overall program speedup, based on Amdahl's law, and can be calculated as it follows [GJ]:

$$speedup \leq \frac{1}{(1 - Coverage) - \frac{Coverage}{SelfParalelism}} \tag{4.1}$$

4.3.2 Execution times for the code variations

After running the six implementations, the results can be divided in three major groups and each group has got the respective implementations of the *Mult* and *MultLine* algorithms. Each group has its own experimental environment, with their own variables and their own meaning according to the given context.

4.3.2.1 Original code

The Original code only variable is the matrix size. This group is a reference group to compare the results of the others groups and to quantify the impact of the others groups. The results of this group are the execution times running both matrix multiplications algorithm versions.

4.3.2.2 Manual code parallelization

The Manual code parallelization by an expert has as variables the matrix size and number of threads used for each run. This group's results are the execution times for both matrix multiplication algorithm versions. These results were used to confront with the following group in order to evaluate the improvement that a guided parallelizations, using Kremlin, can for an application.

4.3.2.3 Kremlin's code parallelization

Like the previous group, Kremlin's code parallelization indications use the same variables and provide the same type of the results. However, this group is responsible to define if it is advantageous to use software tools to help with code parallelization, in an automatic way.

4.4 Data analysis method

Analysing data is a very important stage because it is necessary to have correct conclusions. From the experiments made a lot of data as been generated and without a proper organization it is hard to understand the meaning, therefore, hard to take good conclusions from its analysis. There are three groups of code and in each group two different algorithms implementation. In order to make a correct analysis from the execution times for each situation and comparing with the others cases, the collected results were stored in tables along with the experimental related variables. Additionally, calculations were required, such as, difference between executed times between groups and algorithm implementation; ratio between these executed times; the percentage of the increase/decrease for these executed times; and the impact in the execution time that the others two groups have comparing with the Original code group. After these data manipulation, the best way to analyse all this generated and calculated data is by a dispersion plot. Using a dispersion plot, it transforms data into information visually understandable and easier to conclude because these plots display the variation of results according to the experimental environment variables.

4.5 Data validation

A considerable amount of data was generated and, more importantly, it is important that this data is scientific correct, or at least there is an explanation. In order to keep its fidelity, it is crucial to validate each and every piece of data. The first measure is to have a critical position every time by questioning if the obtained values makes sense when compared with theoretical, or expected or referenced value. In this current case, for instance, the Original group implementation is the reference, which means if the other groups have a lower execution time, or the data has some defect caused by some hardware component, or the implementation isn't good enough, or any other reason that can justify the data invalidation.

Criticism can not be the only measure because it could be luck and the gathered data happened to be correct. It is important consistency. To do so, the tests must be performed several times under the same circumstances and with a plausible and considerable amount of values to find patterns. For this particular case it is used a matrix size large enough, [1000,2000,3000,4000], and a wide range for the number of threads, [1,2,3,4,5,6,7,8]. For instance, if the matrix size was small, such as one hundred, the execution time would be so low and with so much error accumulated since the CPU executed fast enough that it couldn't count the time with precision.

Finally, to make reasonable comparisons and analogies between results, the experience environments must have some connection in its variables or environment. Without a connection, the data as no meaning, therefore, it turns impossible to take conclusions. In the performed experiences, it was used the same algorithm, Matrix Multiplication, with small modifications in the implementations but with the same structure. Additionally, the environment variables were the same: number of threads and matrix size.

Chapter 5

Results and Discussion

5.1 Introduction

Generally, the first objective a developer has when building software is making it work. After some experience and good practices, the development becomes faster, elegant and with concerns about its performance. When dealing if performance issues, there is a lot of measures to pay attention from the lowest hardware level to the highest software level. Now-a-days performance is as much important as the creating software, because it makes the work faster, with less cost, and, in the end, more revenue. However it is really hard for a single person masters performance as a whole because there are to many variables, conditions, aspects, and realities making humanly impossible mastering everything.

The approach to achieve high performance level is to have handful of expertises in each concrete area. Even so, mastering specific fields in the performance level, it is hard, takes time, lots of effort and most of the times impossible to achieve the perfect performance. Since achieving high level of performance is so important and requires a lot of effort to try to achieve it, then, first of all, is it possible to achieve high performance level in applications in an automatic way? If so, how can it be achievable? Can it totally replace an expert?

Focusing these question to the field of code parallelization, more will rise, not necessarily related to this specific filed: since exists tools, like Kremlin, which help to, automatically, parallelize code, how acceptable are their results? How this specific tool can help in getting one step closer to automatic code parallelization? In which way can Kremlin be better than an expert?

In order to answer all previous questions, this chapter is divided in two main sections: the first section is related to the Kremlin's activity and how is it helpful. The second section is the confrontation of all the gathered data to verify what is better and how can it contribute to the future of automatic code parallelization.

5.2 Kremlin's reports

When Kremlin compiles and profiles a sequential code, it provides a report with locations of the blocks that can be parallelized. Additional it gives some values that indicates the theoretical gain if the parallelization is implemented.

For the *Mult* and *MultLine* algorithm, Kremlin gave these reports 7.8 7.9, respectively. Taking into account that the manual code parallelization was done in the first place, the risk of being bias is null and, additionally, helps to understand if Kremlin is reporting things correctly.

In this case, Kremlin detected the block code with the most impact on application performance for both implementations (*Mult* and *MultLine*). Additionally, Kremlin's report pointed the locations of more blocks to be parallelized, however, the impact of these blocks being parallelised might have a low impact on applications' performance, also for both implementations. The impact of the parallization made in the others block is analysed in the next section because a verdict can be made after comparing the execution times of the Manual's group against Kremlin's group.

The justification behind these analysis is based on the *time reduced*, *ideal time reduced*, *coverage* and *self parallelism* values and the block location, provided by the report, comparing with the expected result and manual code parallelization by an expert, in both implementation.

Getting a close look in these reports, at the left side is *Mult* report values, 7.8, and on the right side is *MultLine* report values, 7.9:

Table 5.1: Kremlin's report values for the matrix multiplication block

Time reduced	66.38%
Ideal time reduced	70.96%
Coverage	88.51%
Self parallelism	5.05

Time reduced	63.01%
Ideal time reduced	63.20%
Coverage	84.02%
Self parallelism	4.03

In both reports, the high percentage of the reduced time and time reduced means that parallelizing these blocks the execution time of this block is, theoretically, reduced in between those two values.

Taking a close look in the others blocks, their locations refers to the matrices initialization and the values of timed reduced and ideal timed reduced are really low, around 3% in both implementations, which means that the improved performance is insignificant and might cause delay in during de applications executions. However, this situations is confirmed in the next section.

5.3 Comparison between Original, Manual and Kremlin

The previous section has an important role because the reports credibility and correctness influences the the results in this chapter, consequently, could lead to misguided and wrong conclusions. Since the report gave correct feedback and it is well justified, the following values are valid.

To get a satisfying answer for the initial questions, it is necessary to, in first place, understand context of each experience and respective results; following the evolution and the comparison is made between data. So, in a first instance, each group (Original, Manual, Kremlin) is going to be analysed individually to stablish the context and basis knowledge. Then, the second subsection will focus in measuring the impact of Manual and Kremlin group have to the Original group to prove that these measures, in practical terms, have a huge impact improving applications performance. After this knowledge also as been established, the results will prove if the Kremlin's guidelines make the code with better performance comparing to the expert's results parallelizing the code manually, and respond to the question if automaticly parallelizing code is a reliable and good practice.

5.3.1 The Three groups individually analysed

As mentioned in the previous chapter, each group has its purpose based on the variables used and results obtained. To understand the overall impact of these implementations, it is important to firstly understand the experiences that were made in each group separately and analyse their results, step by step.

5.3.1.1 Original

This group has the sequential code version of the *Mult* and *MultLine* algorithms. As mentioned earlier, this group stablish the base reference for the execution time. From now on, all experiences should have better performance, unless there is an explanation for the Original Group has better results, in some particular cases.

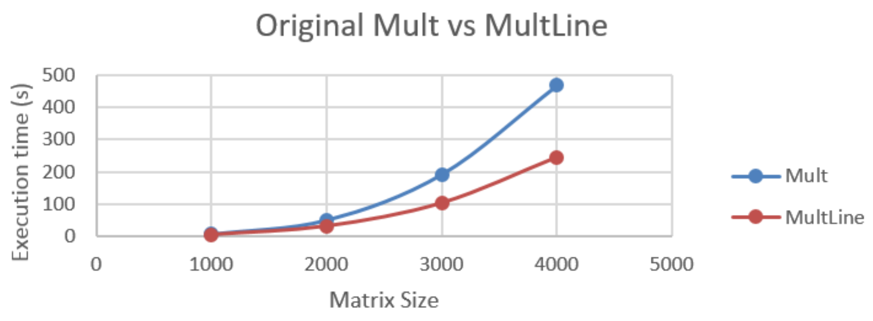

Figure 5.1: Dispersion plot that represents the evolution of execution time with the matrix size increase in both *Mult* and *MultLine* sequential implementations

According to Figure 5.1, the *MultLine* algorithm has better results the higher the matrix size is, as expected, since this algorithm takes advantage of the values preloaded in memory cache.

Another aspect noteworthy is the increase of the function inclination variation, in both implementations, as the matrix size increases. Specially for the *Mult* implementation. This is related to the memory cache size. The smaller the size the higher will be de variation of the function inclination.

5.3.1.2 Manual

Manually parallelizing a code requires a lot of effort, time and know-how since the way it is done requires de expert to understand the code, have practice in detecting potential parallelized blocks of code, identify the best way to parallelize those blocks and test the work until it gives a reasonable result. So, this group corresponds to this situation and, in theoretical perspective, has the bests results.

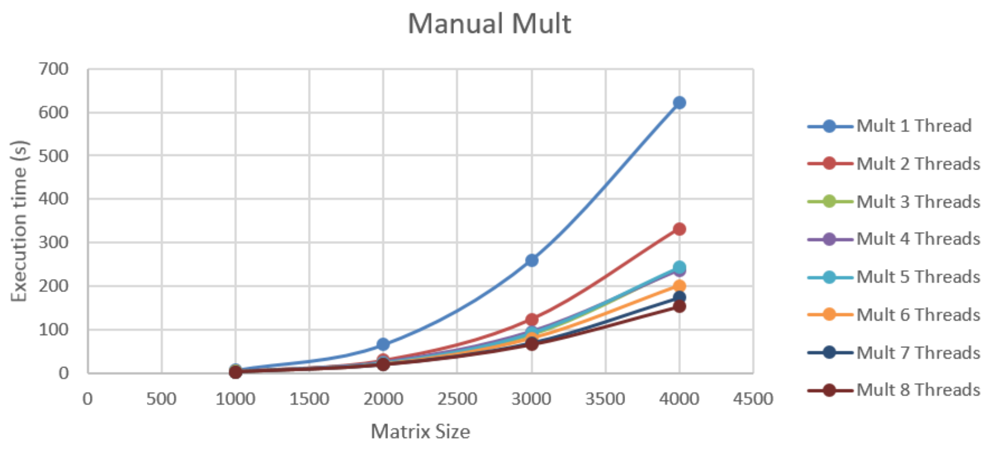

Figure 5.2: Dispersion plot that represents the evolution of execution time with the matrix size increase in function of the number of threads. Version of the manual code parallelization for *Mult* algorithm

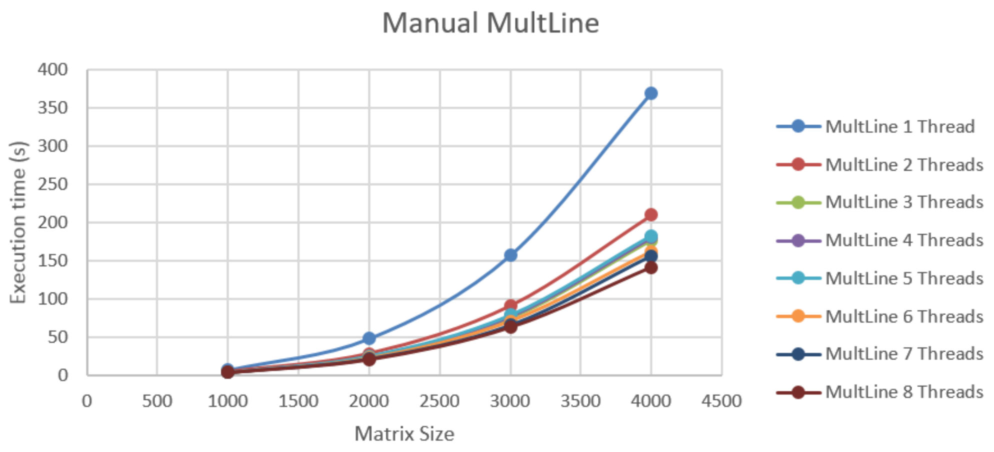

Figure 5.3: Dispersion plot that represents the evolution of execution time with the matrix size increase in function of the number of threads. Version of the manual code parallelization for *MultLine* algorithm

Figure 5.2 and Figure 5.3 have similar behaviours including the fact that using eight threads makes the application with the best performance because the executed time is inferior as long as the matrix size increases.

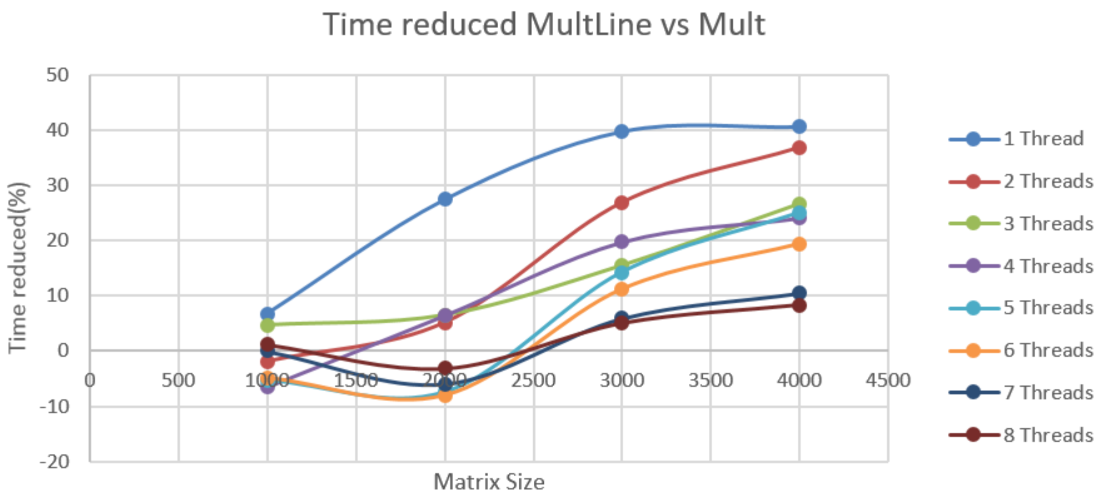

Figure 5.4: Dispersion plot that represents the evolution of time reduced with the matrix size in function of the number of threads. Versions of *Mult* and *MultLine* manual implementations.

However, there is a difference between these two plots is the value of the execution time. This difference can be analysed in Figure 5.4. Time reduced is a percentage of how much the *MultLine* implementation reduces comparing to the *Mult* implementation.

It is a fact, observed in Figure 5.4, that increasing the matrix size, the time reduced tends to a certain value which is independent of the number of threads. This value is the ceiling where *MultLine* algorithm can not be better than *Mult* algorithm for the same hardware components; meaning the existence of a hardware limitation (processor power, memory ram and cache size) since the increase of the matrix size will, proportionally, increase the number of loads, writes and cache missed for both algorithms.

Another fact is that the higher the thread number, lesser will be de timed reduced, and lesser will be the executed time, also related to the maximum capacity of threads a multicore processor can have and have working at the same time.

Another noteworthy fact is , for low values of matrix size, 1000 and 2000, and the higher the number of threads being used, the time reduced is negative, meaning executed time for *Mult* implementation is lower than *MultLine* implementations, concluding that *Mult* implementation is better suited for low size data in case a high number of threads are being used. This is due to many threads are being used simultaneously and they are trampling each other in order to complete their task, which increases the overall overhead, and so the reason behind ne time reduced negative value.

5.3.1.3 Kremlin

This group is constituted by the result of the implementations indicated by Kremlin's report. The experiences conducted in this group and the respective obtained results will demonstrate if Kremlin can actually bring acceptable results.

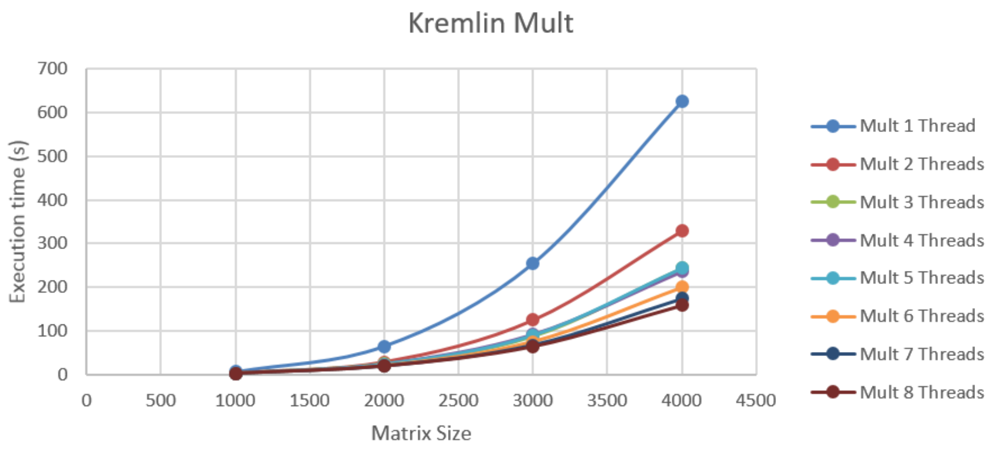

Figure 5.5: Dispersion plot that represents the evolution of execution time with the matrix size increase in function of the number of threads. Version of the Kremlin code parallelization for *Mult* algorithm

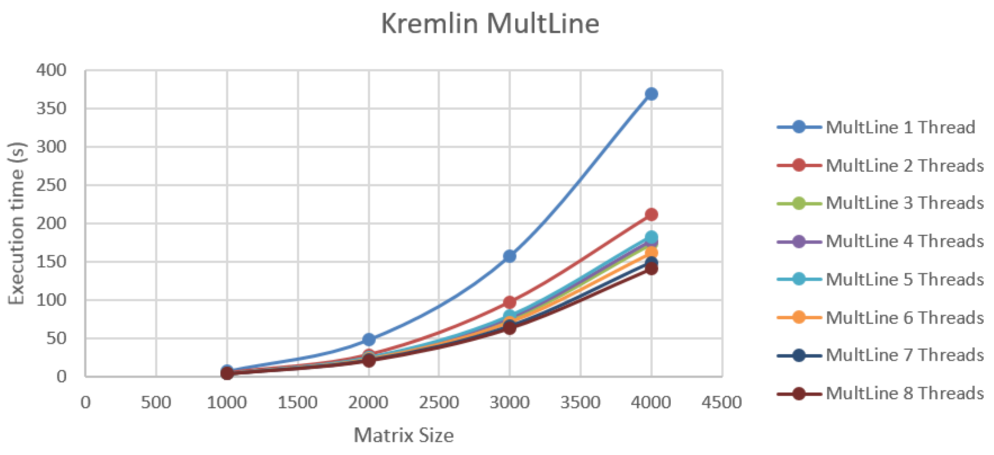

Figure 5.6: Dispersion plot that represents the evolution of execution time with the matrix size increase in function of the number of threads. Version of the Kremlin parallelization code for *Multline* algorithm

Figure 5.5 and Figure 5.6, as expected, have similar behaviour compared to manual group, since the code samples from both groups have the same structure and the experimental environment is the same: same experimental variables(matrix size, number of threads, parallelized code) and same result type (execution time).

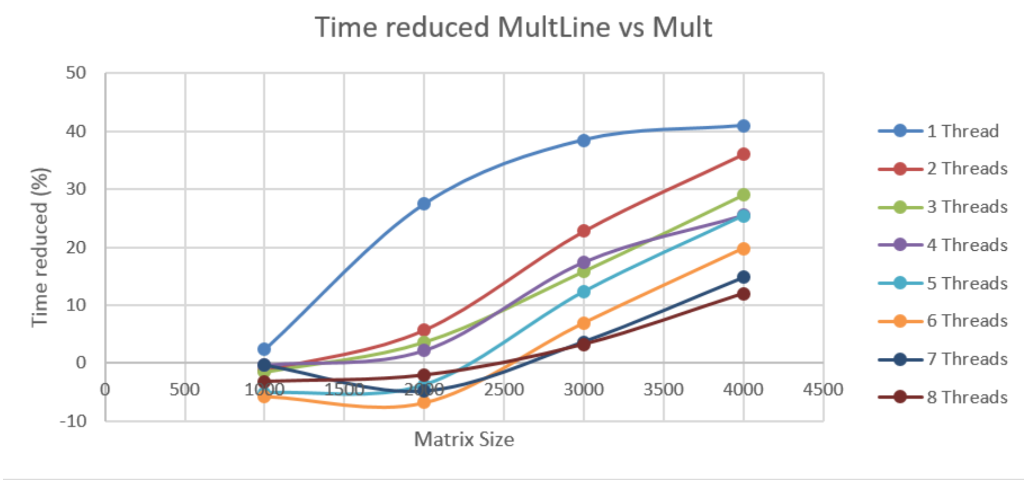

Figure 5.7: Dispersion plot that represents the evolution of time reduced with the matrix size in function of the number of threads. Versions of *Mult* and *MultLine* Kremlin's implementations.

Additionally, and for the same reason, Figure 5.7 also has the same behaviour has the manual group and, therefore, the analysis is the same.

5.3.1.4 Overall review

Through the analysis of these three tests groups, separately, and since these groups were tested under the same circumstances, it is possible to conclude that using these versions of the matrix multiplication algorithm do not jeopardize the results, since they have similar behaviours, moreover, they increase assurance and credibility for the following up analysis and conclusions. It is because of the similarity of behaviours that this comparison is valid and correct, even so the performance is different, which was expected, as explained before.

5.3.2 Measuring the performance's impact using code parallelization

The conducted analysis presented in previous section explains the characteristics of each group and the plausible connection with each other. To quantify how beneficial can code parallelization be, the next sub sub sections will prove its impact.

The first sub sub section compares how much better was the improvement for Manual group, using the Original group as base. The second sub sub sections compares the same ways as the previous sub sub sections but, instead using the Manual group, it will be the Kremlin group. Finally, overall conclusions will be presented about both sub sub sections.

For this analysis, either Manual or Kremlin groups, the number of threads that will be used is eight since, and for both groups, using eight threads gave the best performance results. This does not mean that for the other number of threads the results were worst than the Original group, far from that. The goal is achieving the highest performance, so the best results were picked.

5.3.2.1 Comparison between Manual and Original groups

In Figure 5.8 is presented a dispersion plot demonstrating that Manual group has an overwhelming better performance and that performance increases with the matrix size to a certain point, explained in 5.3.1.2. This includes both implementations.

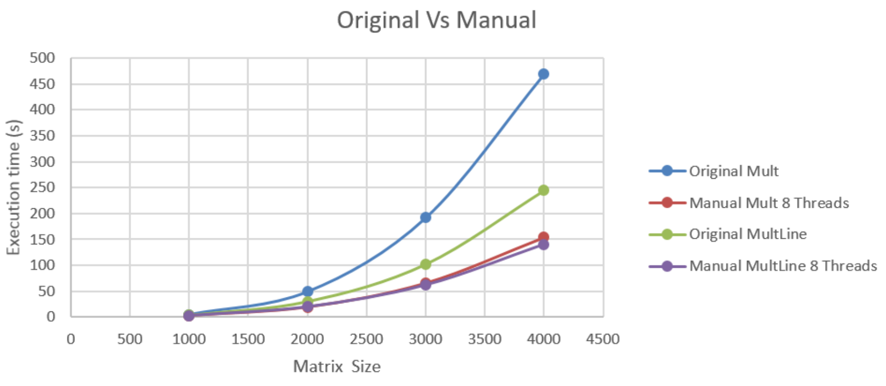

Figure 5.8: Dispersion plot that represents the evolution of execution time with the matrix size in function of Manual and Original.

With this plot. it is proved that using parallelization techniques in programs and applications can achieve higher performances. In this particular case, it can be more than twice better for the *Mult* case and almost twice for *MultLine*. These values are presented in 5.3.2.3.

5.3.2.2 Comparison between Kremlin and Original groups

Regarding the Kremlin experimental results, in Figure 5.9 is shown a dispersion plot revealing, as expected, and like in the previous sub sub section, that Kremlin's group surpassed the Original group in terms of execution time, in both implementations.

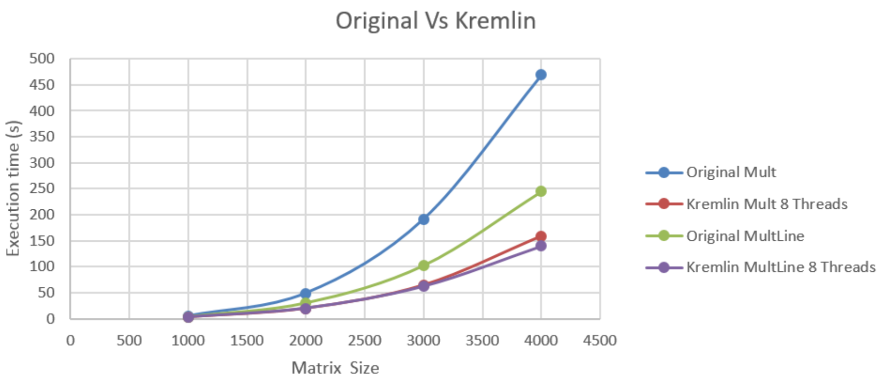

Figure 5.9: Dispersion plot that represents the evolution of execution time with the matrix size in function of Kremlin and Original groups implementations.

Like in previous case, this plot proves that using Kremlin as an automatic tool to help identifying parallelizable code blocks can enhance applications performance and is a good asset because it can save time by indicating where it is possible to make a block parallel, instead of the developer looking for them.

5.3.2.3 Overall comparison

If code parallelization would not bring any beneficial impact in programs and applications, obviously, it would make no sense using it. Additional, and for this concrete case, if Kremlin's performance was worst, this tool would be useless to help in making paralleled code automatically, and the test would and here.

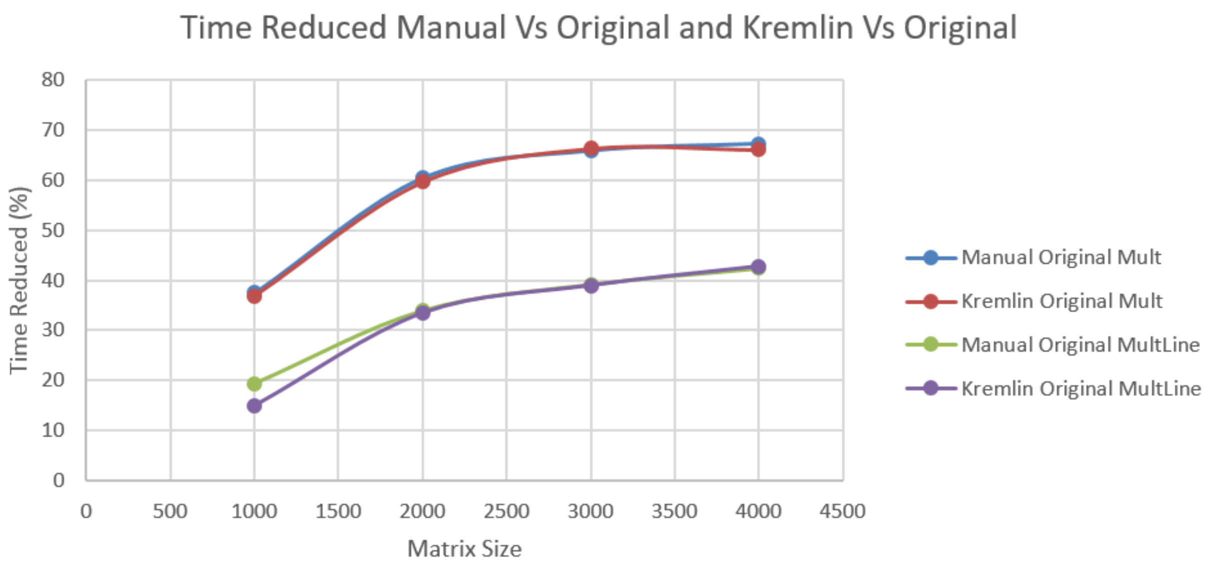

Figure 5.10: Dispersion plot that represents the evolution of time reduced with the matrix size in function of Manual and Kremlin groups implementations.

In Figure 5.10 is compared the time reduced for Manual and Kremlin groups comparatively to Original Group. From this plot it is drawn that *Mult* algorithm takes greater advantages of code parallelization, however *MultLine* has better execution times.

Quantifying such increase, for the *Mult* algorithm the time reduced can be around 67%, meaning that the *Mult* implementation, for both groups, can be 67% faster, more than twice, compared to the Original group, for the same algorithm. Regarding the *MultLine* algorithm, the time reduced can be around 42%. The 25% difference of both implementations represents the impact of the difference on matrix multiplication algorithm versions. Meaning that *MultLine* algorithm makes a huge difference on performance levels.

5.3.3 Comparison between Manual and Kremlin groups

Before starting the comparison between results obtained from running the parallelized code written by an expert and running the parallelized code with Kremlin's indication, the difference between them is that in the Kremlin group, more specifically, in the matrix initializations blocks, they are parallelized as well. These blocks were parallelized because Kremlin detected them, however the theoretical impact for these parallelized bocks could be small, positively or negatively, or even with no effect. Additional, in the expert perspective, they were not taken in consideration.

Now that the three groups were characterized and the comparison between Manual and Kremlin with Original group to stablish viability in both solutions, taking a close look at all this information and analysis and looking at Figure 5.10, the lines in the plot overlap or are really closed to one another in both algorithms. This observation confirms what is expected, which is manually parallelizing a code by an expert or following Kremlin's instructions gives the same results, approximately. However, they are not exactly the same because a small modifications was done to Kremlin's group code.

In order to evaluate if there is really a difference in results, the following plots present the the execution time ratio between Manual and Kremlin's group, for both implementations.

To correctly analyse Figures 5.11 and 5.12, it is important to visualize the range of the values:

Table 5.2: Interval of values for *Mult* (left table) and *MultLine* (right table)

Lower bound	0,966
Upper bound	1,049
Interval size	0,083

Lower bound	0,936
Upper bound	1,073
Interval size	0,137

Looking at the tabulated values, there are cases where Kremlin group had better performance then the Manual group, since there are values less then 1. However, looking at the interval size, for both cases, they are really small, meaning that the execution time for both groups is similar.

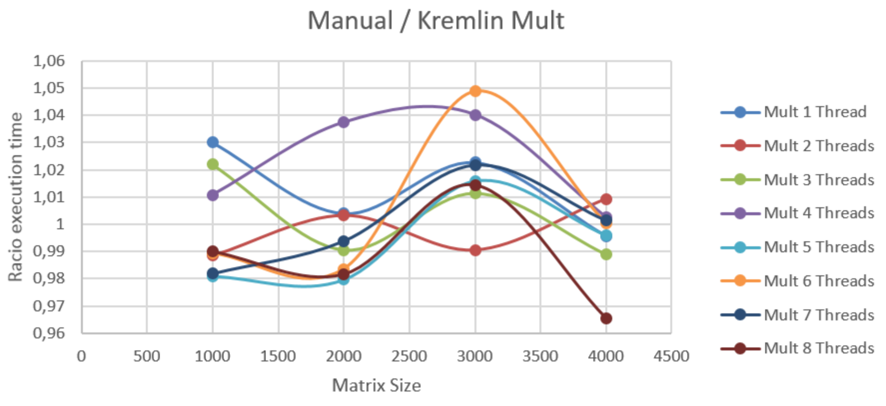

Figure 5.11: Dispersion plot that represents the evolution of execution time ratio with the matrix size in function of the number of threads, for the *Mult* algorithm .

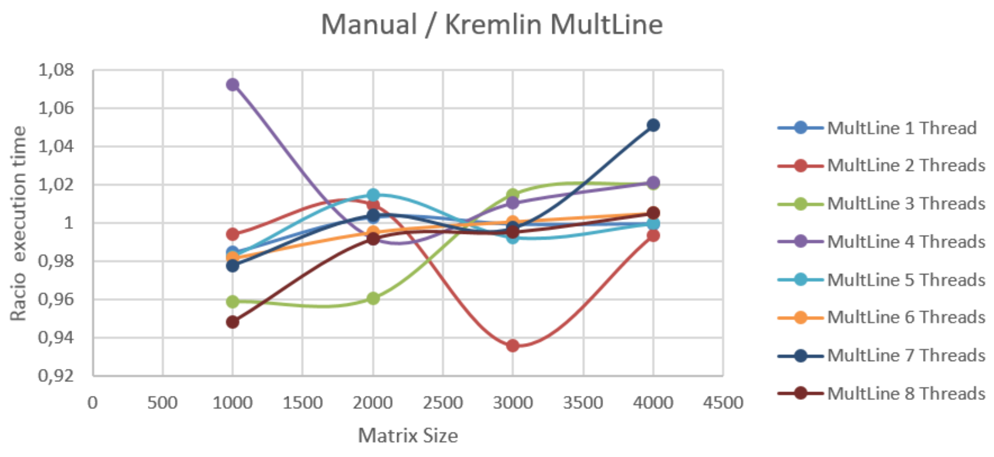

Figure 5.12: Dispersion plot that represents the evolution of execution time ratio with the matrix size in function of the number of threads, for the *MultLine* algorithm.

Confronting Figures 5.11 and 5.12, they seemed confusing, messed up, random, and that is partial true because the execution time values from both groups are that close form one another and a slight alteration makes, apparently, an huge impact. It is also partial false because there are patterns in these plots: for each matrix size there is a concentration of lines, meaning that it is not so random and the explanation is that both parallelizations have similar performances. However, and again, the interval value is small.

Trying to se the data in a different angle and perspective to understand if there is any correlation, pattern or relation, these Figures, 5.13 and 5.14, are an unsuccessful attempt of it. The point of these two figures is to understand the variation of execution time ratio with number of threads in function of matrix size, however there is no relation for both cases. So, the number of threads has no direct relation with the execution time ratio in function of the matrix size.

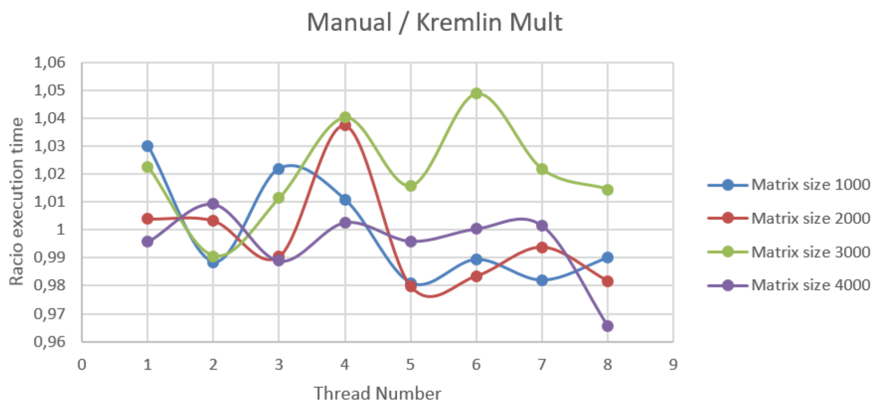

Figure 5.13: Dispersion plot that represents the evolution of execution time ratio with the number of threads in function of matrix size, for the *Mult* algorithm.

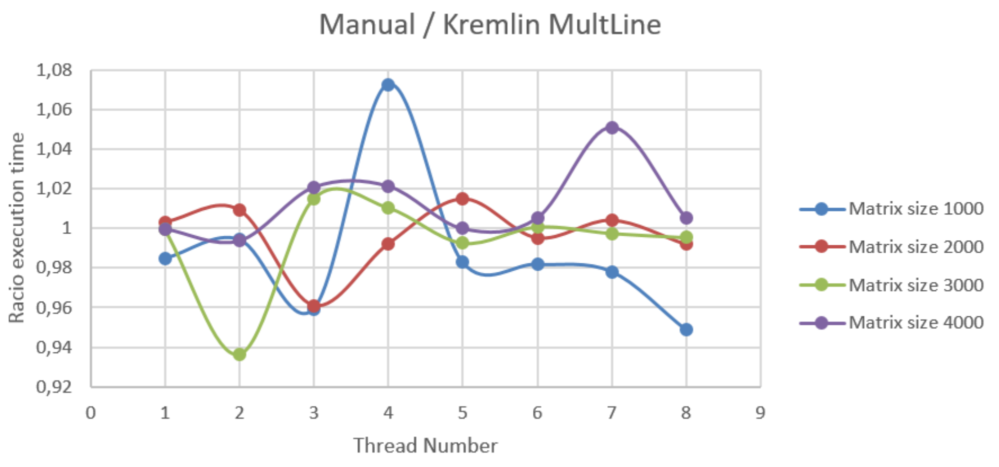

Figure 5.14: Dispersion plot that represents the evolution of execution time ratio with the number of threads in function of matrix size, for the *MultLine* algorithm.

So far, in this sub sub section, the analysis done is in a general plane. this Figure 5.15 is about
4 execution time ratio in function of matrix size using eight threads. This specific case was chosen because for both groups, using 8 threads had the best results, as previously mentioned and verified.
6 In the green line is established as a reference: the values under this line mean Kremlin performed better than Manual, and the values above this line mean the opposite. Looking at those values,
8 there is more values under the line than above, meaning that Kremlin group performed better, for both implementations.

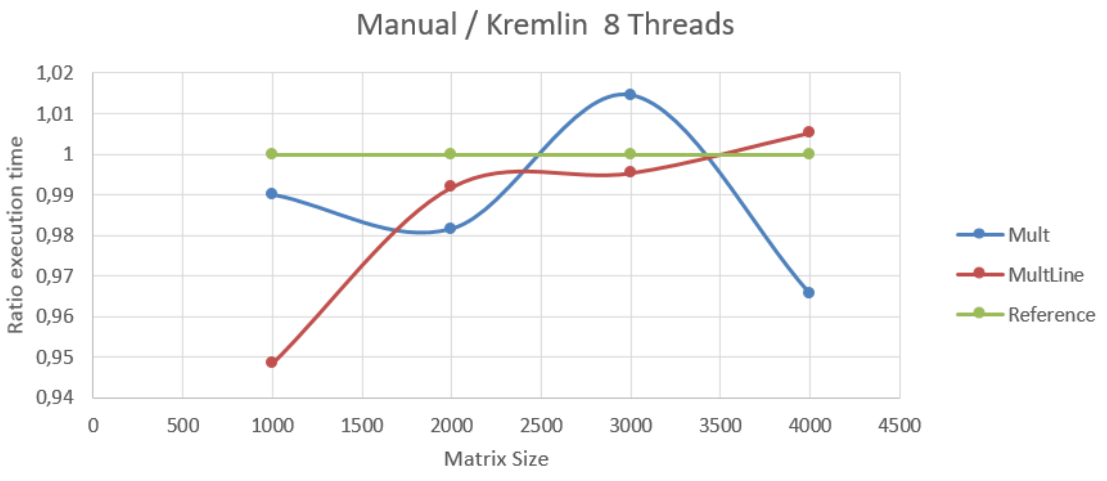

Figure 5.15: Dispersion plot that represents the evolution of execution time ratio with the matrix size in for 8 threads being used, using *Mult* and *MultLine* algorithms.

Chapter 6

Conclusion

Before verifying if parallel code made in an automatic way is a reality, first it is important to verify if making code parallel is viable. And to prove that, the experiments in 5.3.2 in shows that the performance is far greater for the bigger the size of the input data, under the hardware components capabilities. Since it was established the viability of making code parallel,it is now possible to focus on the goal of this dissertation. So, the main goal of this dissertations is to find the viability on making parallel code automatically. For that purpose, and mentioned in 2, it was necessary to know what were the basis, common mind set, knowledge's course, developments and future thoughts on this field. Proving that it is viable, the next course of action is finding how can it be made. One way to do it is using Kremlin's help. Kremlin not only proved assistance but also answered in one possible way to automatically make code parallel, for this case it is two circumstances: firstly, locating possibility parallizable blocks, which most likely are *for* loops, and, secondly, measuring the impact of these blocks imagining if they were parallelized.

However, and although Kremlin is a powerful tool to advice one of the most important jobs when parallelizing code, location of parallelizable blocks, it isn't perfect and has its own limitations:

- Can't use data structures from the *Standard Library*, such as, stack, list, priority queue, queue, list, hash table, map, multimap, heap, etc.;
- Compiling programs that have a deep function call level greater that seven;
- Can't detect *for* loops if the iterator iterator variable used isn't define outside of the loop;
- When dealing with inner *for* loops, it just detects the block, instead of pointing which loop should be parallelized.

During the conducted experience, when learning how to take the best profit from Kremlin and when collecting data from the performed experiences mentioned in the previous chapter, those where the problems found and the struggles that required to be overcomed.

For now, the location of a parallelizable block and the viability of this block being parallel in terms of performance are the patterns found for doing code parallel. Another aspect found when conducting the experiments is the number of threads used in function of the size of the data. For a big size of data, the higher the number of threads used, the performance was better. However, there is the limitations in hardware components, such as cache and RAM memory, the power of the processor, including the number of threads it can use.

These were the main aspects detected during these experimented and in the following sections it is justified the reasoning behind these choices.

6.1 Using Kremlin for code parallelization

In 5.2 it is analysed the information given by Kremlin's report. Adding to this information the analysis of the results made in 5.3, we can conclude Kremlin provided accurate and correct information. Even in the case of the parallelization of those matrix initialization blocks, Kremlin detected a low impact in performance and, according to the results, it had almost no impact at all.

For future reference, Kremlin is a powerful tool to help experts in identifying potential parallelizable blocks and measuring the quality of its performance in case being parallelized.

6.2 Automatic Parallelization is viable

From the conducted experiments in 5.3 and joining the information provided from Kremlin's report, it was proved and concluded that automatic parallelization is possible, viable, accurate and, the main purpose, spares time and efforts for the developer side. Although it still required expertise to understand and available the data provided, this is the first step in order to make automatic code parallelization even better. The time I used to follow Kremlin's instructions form the report was really small compared to the one used to manually parallelized the same code. Even doing the manual part first and using the knowledge from that time, it was still faster because I could skip the phase of detecting parallelizable code and evaliation of its relevance.

6.3 Future work

For the future, this work could be used as an orientation for the next steps. Those steps could be finding more patterns and variables; Using Kremlin information and number of threads according to the input data size, making a profile for this application to fin the bet performance; and, for the finale stage, rewrite applications code with these informations coming form the profiler, so that the application can have the best performance possible.

References

[Dat08] Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2008*, 2008. Cited on page 1.

[FSH04] K Fatahalian, J Sugerman, and P Hanrahan. Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication. pages 133–137, 2004. Cited on page 11.

[GG08] Kazushige Goto and Robert A V A N D E Geijn. Anatomy of High-Performance Matrix Multiplication. 34(3), 2008. Cited on page 11.

[GJ] Saturnino Garcia Jr. *A Practical Oracle for Sequential Code Parallelization*. Cited on pages 7 and 19.

[GJL$^+$12] Saturnino Garcia, Donghwan Jeon, Christopher Louie, Michael Bedford, and San Diego. the Kremlin Oracle for the Kremlin Open - Source Tool Helps Programmers By Automatically Identi-. pages 42–53, 2012. Cited on page 8.

[GJLT11] Saturnino Garcia, Donghwan Jeon, Chris Louie, and Michael Bedford Taylor. Kremlin : Rethinking and Rebooting gprof for the Multicore Age. 2011. Cited on page 8.

[Gum10] Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. *Proceedings of the 19th international conference on Parallel architectures and compilation techniques - PACT '10*, page 205, 2010. Cited on page 7.

[Jeo] Donghwan Jeon. *Parallel Speedup Estimates for Serial Programs*. Cited on page 7.

[Kre] kremlin: like gprof but for parallelization. https://bitbucket.org/elsaturnino/kremlin. Accessed: 2017-04-17. Cited on page 17.

[LR] Changmin Lee and Won W Ro. for Parallel Processing on CPU / GPU Hybrids. Cited on page 5.

[LWR] S Lam, E Wolf, and E Rothberg. The Cache Performance of Blocked. Cited on page 14.

[Nc98] S Ilicon G Raphics I Nc. OpenMP : An Industry-. pages 46–55, 1998. Cited on page 8.

[Nvi] What is gpu-accelerated computing? http://www.nvidia.com/object/what-is-gpu-computing.html. Accessed: 2017-02-09. Cited on page 1.

[Par] Saturnino (sat) garcia. http://home.sandiego.edu/ sat/. Accessed: 2017-02-12. Cited on pages 8 and 9.

[She] Jie Shen. *Efficient High Performance Computing on Heterogeneous Platforms*. Cited on page 6.

[SPTK09] Christoph A Schaefer, Victor Pankratius, Walter F Tichy, and D Karlsruhe. Atune-IL : An Instrumentation Language for Auto-Tuning Parallel Applications Atune-IL. 2009. Cited on page 9.

[Sta] *StarPU Handbook*. Cited on page 6.

[Tay] Michael Bedford Taylor. Kismet : Parallel Speedup Estimates for Serial Programs. pages 519–536. Cited on page 9.

[YZ05] Raphael Yuster and U R I Zwick. Fast Sparse Matrix Multiplication. 1(1):2–13, 2005. Cited on page 11.

Chapter 7

Appendices

7.1 Developed code

7.1.1 Original Matrix Multiplication (Mult)

Listing 7.1: Matrix Multiplication original algorithm, written in C++

```
double OnMult(int m_ar, int m_br)
{
    double Time1, Time2;
    double temp;
    int i, j, k;
    double *pha, *phb, *phc;

    //Matrixes Memory allocation
    pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

    //Starting counting time
    Time1 = omp_get_wtime();

    //Loading matrix values
    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            pha[i*m_ar + j] = (double)1.0;

    for(i=0; i<m_br; i++)
        for(j=0; j<m_br; j++)
            phb[i*m_br + j] = (double)(i+1);


    //Matrix Multiplication
    for(i=0; i<m_ar; i++)
    {   for( j=0; j<m_br; j++)
        {   temp = 0;
            for( k=0; k<m_ar; k++)
            {
                temp += pha[i*m_ar+k] * phb[k*m_br+j];
            }
            phc[i*m_ar+j]=temp;
        }
    }

    //Stoping time
    Time2 = omp_get_wtime();

    //Freeing memory used for matrixes
    free(pha);
    free(phb);
    free(phc);

    return Time2 - Time1;
}
```

7.1.2 Original Matrix Multiplication By line (MutLine)

Listing 7.2: Matrix Multiplication by line original algorithm, written in C++

```
double OnMultLine(int m_ar, int m_br)
{
    double Time1, Time2;
    double temp;
    int i, j, k;
    double *pha, *phb, *phc;

    //Matrixes Memory allocation
    pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

    //Starting counting time
    Time1 = omp_get_wtime();

    //Loading matrix values
    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            pha[i*m_ar + j] = (double)1.0;

    for(i=0; i<m_br; i++)
        for(j=0; j<m_br; j++)
            phb[i*m_br + j] = (double)(i+1);

    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            phc[i*m_ar + j] = (double)0.0;


    //Matrix Multiplication
    for(i=0; i<m_ar; i++)
    {   for( k=0; k<m_ar; k++)
        {
            for( j=0; j<m_br; j++)
```

```
                {
                    phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
                }

            }
    }

    //Stoping time
    Time2 = omp_get_wtime();

    //Freeing memory used for matrixes
    free(pha);
    free(phb);
    free(phc);

    return Time2 - Time1;;
}
```

7.1.3 Manual Matrix Multiplication (Mult)

Listing 7.3: Matrix Multiplication manually parallelised using OpenMP library, written in C++

```
double OnMultThreading(int m_ar, int m_br,int x)
{
    double Time1, Time2;
    double temp;
    int i, j, k;
    double *pha, *phb, *phc;

    //Matrixes Memory allocation
    pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

    //Starting counting time
    Time1 = omp_get_wtime();

    //Loading matrix values
    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            pha[i*m_ar + j] = (double)1,0;

    for(i=0; i<m_br; i++)
        for(j=0; j<m_br; j++)
            phb[i*m_br + j] = (double)(i+1);


    //Matrix Multiplication
    for(i=0; i<m_ar; i++)
    {   for( j=0; j<m_br; j++)
        {   temp = 0;
            #pragma omp parallel for reduction(+:temp) num_threads (x)
            for( k=0; k<m_ar; k++)
            {
                temp += pha[i*m_ar+k] * phb[k*m_br+j];
            }
            phc[i*m_ar+j]=temp;
        }
    }

    //Stoping time
    Time2 = omp_get_wtime();

    //Freeing memory used for matrixes
    free(pha);
    free(phb);
    free(phc);

    return Time2 - Time1;
}
```

7.1.4 Manual Matrix Multiplication By line (MultLine)

Listing 7.4: Matrix Multiplication by line manually parallelised using OpenMP library, written in C++

```
double OnMultLineThreading(int m_ar, int m_br,int x)
{
    double Time1, Time2;
    int i, j, k;
    double *pha, *phb, *phc;

    //Matrixes Memory allocation
    pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

    //Starting counting time
    Time1 = omp_get_wtime();

    //Loading matrix values
    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            pha[i*m_ar + j] = (double)1,0;

    for(i=0; i<m_br; i++)
        for(j=0; j<m_br; j++)
            phb[i*m_br + j] = (double)(i+1);

    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            phc[i*m_ar + j] = (double)0,0;


    //Matrix Multiplication
    for(i=0; i<m_ar; i++)
    {   for( k=0; k<m_ar; k++)
```

```
        {
            #pragma omp parallel for num_threads (x)
            for( j=0; j<m_br; j++)
            {
                phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
            }

        }
    }

    //Stoping time
    Time2 = omp_get_wtime();

    //Freeing memory used for matrixes
    free(pha);
    free(phb);
    free(phc);

    return Time2 - Time1;
}
```

7.1.5 Kremlin Matrix Multiplication (Mult)

Listing 7.5: Matrix Multiplication with Kremlin's indications for parallelization, written in C++

```
double OnMultKremlin(int m_ar, int m_br,int x)
{
    double Time1, Time2;
    double temp;
    int i, j, k;
    double *pha, *phb, *phc;

    //Matrixes Memory allocation
    pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

    //Starting counting time
    Time1 = omp_get_wtime();

    //Loading matrix values
    for(i=0; i<m_ar; i++)
        #pragma omp parallel for num_threads (x)
        for(j=0; j<m_ar; j++)
            pha[i*m_ar + j] = (double)1,0;

    for(i=0; i<m_br; i++)
        #pragma omp parallel for num_threads (x)
        for(j=0; j<m_br; j++)
            phb[i*m_br + j] = (double)(i+1);


    //Matrix Multiplication
    for(i=0; i<m_ar; i++)
    {   for( j=0; j<m_br; j++)
        {   temp = 0;
            #pragma omp parallel for reduction(+:temp) num_threads (x)
            for( k=0; k<m_ar; k++)
            {
                temp += pha[i*m_ar+k] * phb[k*m_br+j];
            }
            phc[i*m_ar+j]=temp;
        }
    }

    //Stoping time
    Time2 = omp_get_wtime();

    //Freeing memory used for matrixes
    free(pha);
    free(phb);
    free(phc);

    return Time2 - Time1;
}
```

7.1.6 Kremlin Matrix Multiplication By line (MultLine)

Listing 7.6: Matrix Multiplication by line with Kremlin's indications for parallelization, written in C++

```
double OnMultLineKremlin(int m_ar, int m_br,int x)
{
    double Time1, Time2;
    int i, j, k;
    double *pha, *phb, *phc;

    //Matrixes Memory allocation
    pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

    //Starting counting time
    Time1 = omp_get_wtime();

    //Loading matrix values
    for(i=0; i<m_ar; i++)
        #pragma omp parallel for num_threads (x)
        for(j=0; j<m_ar; j++)
            pha[i*m_ar + j] = (double)1,0;

    for(i=0; i<m_br; i++)
        #pragma omp parallel for num_threads (x)
        for(j=0; j<m_br; j++)
            phb[i*m_br + j] = (double)(i+1);

    for(i=0; i<m_ar; i++)
```

```
        #pragma omp parallel for num_threads (x)
        for(j=0; j<m_ar; j++)
            phc[i*m_ar + j] = (double)0.0;

    //Matrix Multiplication
    for(i=0; i<m_ar; i++)
    {   for( k=0; k<m_ar; k++)
        {
            #pragma omp parallel for num_threads (x)
            for( j=0; j<m_br; j++)
            {
                phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
            }

        }
    }

    //Stoping time
    Time2 = omp_get_wtime();

    //Freeing memory used for matrixes
    free(pha);
    free(phb);
    free(phc);

    return Time2 - Time1;
}
```

7.1.7 Sequential program compiled and profiled by Kremlin

Listing 7.7: Program compiled and profiled by Kremlin with both Mult and MultLine algorithms, written in C++

```
//#include <omp.h>
#include <stdio.h>
#include <iostream>
#include <iomanip>
#include <time.h>
#include <cstdlib>
//#include <papi.h>
#include <fstream>
#include <chrono>

using namespace std;

#define SYSTEMTIME clock_t
/*
double OnMultLineThreading(int m_ar, int m_br,int x)
{
    double Time1, Time2;

    char st[100];
    double temp;
    int i, j, k;

    double *pha, *phb, *phc;



    pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            pha[i*m_ar + j] = (double)1.0;



    for(i=0; i<m_br; i++)
        for(j=0; j<m_br; j++)
            phb[i*m_br + j] = (double)(i+1);

    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            phc[i*m_ar + j] = (double)0.0;



    Time1 = omp_get_wtime();

    for(i=0; i<m_ar; i++)
    {   for( k=0; k<m_ar; k++)
        {
            #pragma omp parallel for num_threads (x)
            for( j=0; j<m_br; j++)
            {
                phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
            }

        }
    }


    Time2 = omp_get_wtime();
    //sprintf(st, "Time: %3.3f seconds\n", (double)(Time2 - Time1));
    //cout << st;

    /*cout << "Result matrix: " << endl;
    for(i=0; i<min(10,m_ar); i++)
    {   for(j=0; j<min(10,m_br); j++)
            cout << phc[j] << " ";
    }
    cout << endl;*/
/*
    free(pha);
    free(phb);
    free(phc);
    return (double)(Time2 - Time1);

```

```
}*/
/*
double OnMultThreading(int m_ar, int m_br,int x)
{

    double Time1, Time2;

    char st[100];
    double temp;
    int i, j, k;

    double *pha, *phb, *phc;



    pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            pha[i*m_ar + j] = (double)1.0;



    for(i=0; i<m_br; i++)
        for(j=0; j<m_br; j++)
            phb[i*m_br + j] = (double)(i+1);



    Time1 = omp_get_wtime();

    for(i=0; i<m_ar; i++)
    {   for( j=0; j<m_br; j++)
        {   temp = 0;
            #pragma omp parallel for reduction(+:temp) num_threads (x)
            for( k=0; k<m_ar; k++)
            {
                temp += pha[i*m_ar+k] * phb[k*m_br+j];
            }
            phc[i*m_ar+j]=temp;
        }
    }


    Time2 = omp_get_wtime();
    //sprintf(st, "Time: %3.3f seconds\n", (double)(Time2 - Time1));
    //cout << st;

    /*cout << "Result matrix: " << endl;
    for(i=0; i<1; i++)
    {   for(j=0; j<min(10,m_br); j++)
            cout << phc[j] << " ";
    }
    cout << endl;*/
/*
```

```
    free(pha);
    free(phb);
    free(phc);
    return (double)(Time2 - Time1);

}*/

double OnMult(int m_ar, int m_br)
{

    //double Time1, Time2;

    char st[100];
    double temp;
    int i, j, k;

    double *pha, *phb, *phc;



    pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            pha[i*m_ar + j] = (double)1.0;



    for(i=0; i<m_br; i++)
        for(j=0; j<m_br; j++)
            phb[i*m_br + j] = (double)(i+1);



    //Time1 = omp_get_wtime();
    auto Time1 = std::chrono::high_resolution_clock::now();

    for(i=0; i<m_ar; i++)
    {   for( j=0; j<m_br; j++)
        {   temp = 0;
            for( k=0; k<m_ar; k++)
            {
                temp += pha[i*m_ar+k] * phb[k*m_br+j];
            }
            phc[i*m_ar+j]=temp;
        }
    }


    //Time2 = omp_get_wtime();
    auto Time2 = std::chrono::high_resolution_clock::now();
    //sprintf(st, "Time: %3.3f seconds\n", (double)(Time2 - Time1));
    //cout << st;

    /*cout << "Result matrix: " << endl;
    for(i=0; i<1; i++)
    {   for(j=0; j<min(10,m_br); j++)
            cout << phc[j] << " ";
    }
    cout << endl;*/

    free(pha);
    free(phb);
    free(phc);
    auto time = Time2-Time1;
    return (double) std::chrono::duration_cast<std::chrono::milliseconds>(time).count();

}


double OnMultLine(int m_ar, int m_br)
{
    //double Time1, Time2;

    char st[100];
    double temp;
    int i, j, k;

    double *pha, *phb, *phc;



    pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
    phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            pha[i*m_ar + j] = (double)1.0;



    for(i=0; i<m_br; i++)
        for(j=0; j<m_br; j++)
            phb[i*m_br + j] = (double)(i+1);

    for(i=0; i<m_ar; i++)
        for(j=0; j<m_ar; j++)
            phc[i*m_ar + j] = (double)0.0;



    //Time1 = omp_get_wtime();
    auto Time1 = std::chrono::high_resolution_clock::now();

    for(i=0; i<m_ar; i++)
    {   for( k=0; k<m_ar; k++)
        {
            for( j=0; j<m_br; j++)
            {
                phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
            }
            }
        }


        //Time2 = omp_get_wtime();
        auto Time2 = std::chrono::high_resolution_clock::now();
        //sprintf(st, "Time: %3.3f seconds\n", (double)(Time2 - Time1));
        //cout << st;

        /*cout << "Result matrix: " << endl;
        for(i=0; i<min(10,m_ar); i++)
        {    for(j=0; j<min(10,m_br); j++)
                cout << phc[j] << " ";
        }
        cout << endl;*/

        free(pha);
        free(phb);
        free(phc);
        auto time = Time2-Time1;
        return (double) std::chrono::duration_cast<std::chrono::milliseconds>(time).count();

}

void OutputToFile(int lin, int col, int inc, int limit/*, char* filename*/)
{
    int i;
    double temp;
    ofstream myfile;
    myfile.open (/*filename*/"matrixMultResult.csv");
    myfile << "i,Algoritmo a,Algoritmo b\n";
    for(i=lin;i <= limit; i=i+inc)
    {
        temp=OnMult(i,i);
        myfile << i << "," << temp << ",";
        temp=OnMultLine(i,i);
        myfile << temp << "\n";
    }
    myfile.close();
}




float produtoInterno(float *v1, float *v2, int col)
{
    int i;
    float soma=0.0;

    for(i=0; i<col; i++)
        soma += v1[i]*v2[i];

    return(soma);

}
/*)
```

```
void handle_error (int retval)
{
  printf("PAPI error %d: %s\n", retval, PAPI_strerror(retval));
  exit(1);
}

void init_papi() {
  int retval = PAPI_library_init(PAPI_VER_CURRENT);
  if (retval != PAPI_VER_CURRENT && retval < 0) {
    printf("PAPI library version mismatch!\n");
    exit(1);
  }
  if (retval < 0) handle_error(retval);

  std::cout << "PAPI Version Number: MAJOR: " << PAPI_VERSION_MAJOR(retval)
            << " MINOR: " << PAPI_VERSION_MINOR(retval)
            << " REVISION: " << PAPI_VERSION_REVISION(retval) << "\n";
}*/


int main (int argc, char *argv[])
{

    char c;
    int lin, col, nt=1, inc, limit,x;
    int op;
    char* filename;
    //int EventSet = PAPI_NULL;
    long long values[2];
    int ret;
    /*
    cout << "Numero de processadores: " << omp_get_num_procs() << endl;
    ret = PAPI_library_init( PAPI_VER_CURRENT );
    if ( ret != PAPI_VER_CURRENT )
        std::cout << "FAIL" << endl;


    ret = PAPI_create_eventset(&EventSet);
        if (ret != PAPI_OK) cout << "ERRO: create eventset" << endl;


    ret = PAPI_add_event(EventSet,PAPI_L1_DCM);
    if (ret != PAPI_OK) cout << "ERRO: PAPI_L1_DCM" << endl;


    ret = PAPI_add_event(EventSet,PAPI_L2_DCM);
    if (ret != PAPI_OK) cout << "ERRO: PAPI_L2_DCM" << endl;

    */


    op=1;
    do {
        cout << endl;
        cout << "1. Multiplication" << endl;
        cout << "2. Line Multiplication" << endl;
        cout << "3. outputToFile" << endl;
        cout << "4. multithreading on Multiplication" << endl;
        cout << "5. multithreading on LineMultiplication" << endl;
        cout << "Selection?: ";

        cin >>op;
        if (op == 0)
            break;

        printf("Dimensions: lins cols ? ");
        cin >> lin >> col;


        if(op == 3)
        {
            printf("Dimensional increment: inc ? ");
            cin >> inc;
            printf("Limit: limit ? ");
            cin >> limit;
            /*printf("outputFile: filename.csv (must be an existing file) ? ");
            cin >> filename;*/
        }
        /*if(op == 4 || op == 5)
        {
            printf("Number of threads: x");
            cin >> x;
        }*/


        // Start counting
        /*ret = PAPI_start(EventSet);
        if (ret != PAPI_OK) cout << "ERRO: Start PAPI" << endl;
*/
        switch (op){
            case 1:
                cout << OnMult(lin, col)<< endl;
                break;
            case 2:
                cout << OnMultLine(lin, col)<<endl;
                break;
            case 3:
                OutputToFile(lin, col, inc, limit/*, filename*/);
                break;
            /*case 4:
                cout << OnMultThreading(lin, col,x)<< endl;
                break;
            case 5:
                cout << OnMultLineThreading(lin, col,x)<< endl;
                break;*/
        }

        /*ret = PAPI_stop(EventSet, values);
        if (ret != PAPI_OK) cout << "ERRO: Stop PAPI" << endl;
        printf("L1 DCM: %lld \n",values[0]);
        printf("L2 DCM: %lld \n",values[1]);
        ret = PAPI_reset( EventSet );
        if ( ret != PAPI_OK )
            std::cout << "FAIL reset" << endl; */
```

```



    }while (op != 0);
/*
        ret = PAPI_remove_event( EventSet, PAPI_L1_DCM );
        if ( ret != PAPI_OK )
            std::cout << "FAIL remove event" << endl;

        ret = PAPI_remove_event( EventSet, PAPI_L2_DCM );


        ret = PAPI_destroy_eventset( &EventSet );
        if ( ret != PAPI_OK )
            std::cout << "FAIL destroy" << endl;
        */

}
```

7.2 Kremlin's Reports

7.2.1 Kremlin report for Matrix Multiplication, Mult version

Listing 7.8: Kremlin's indication of the blocks that should be parallelised and theorical variables that where calculated for Mult algorithm version

```
[ 0] TimeRed(4)=66.38%, TimeRed(Ideal)=70.96%, Cov=88.51%, SelfP=5.05, DOALL
  LOOP matrixmul.cpp [ 148 -  181]:      OnMult
  FUNC matrixmul.cpp [ 142 -  142]:      OnMult called at file matrixmul.cpp, line  397

[ 1] TimeRed(4)=3.10%, TimeRed(Ideal)=3.37%, Cov=4.13%, SelfP=5.44, DOALL
  LOOP matrixmul.cpp [ 149 -  167]:      OnMult
  FUNC matrixmul.cpp [ 142 -  142]:      OnMult called at file matrixmul.cpp, line  397

[ 2] TimeRed(4)=3.10%, TimeRed(Ideal)=3.37%, Cov=4.13%, SelfP=5.44, DOALL
  LOOP matrixmul.cpp [ 149 -  161]:      OnMult
  FUNC matrixmul.cpp [ 142 -  142]:      OnMult called at file matrixmul.cpp, line  397
```

7.2.2 Kremlin report for Matrix Multiplication, MultLine version

Listing 7.9: Kremlin's indication of the blocks that should be parallelised and theorical variables that where calculated for MultLine algorithm version

```
.................................
[ 0] TimeRed(4)=63.01%, TimeRed(Ideal)=63.20%, Cov=84.02%, SelfP=4.03, DOALL
  LOOP matrixmul.cpp [ 213 -  247]:  OnMultLine
  FUNC matrixmul.cpp [ 207 -  207]:  OnMultLine called at file matrixmul.cpp, line  400

[ 1] TimeRed(4)=2.86%, TimeRed(Ideal)=2.96%, Cov=3.81%, SelfP=4.45, DOALL
```

```
  LOOP matrixmul.cpp [ 213 -  235]:  OnMultLine
  FUNC matrixmul.cpp [ 207 -  207]:  OnMultLine called at file matrixmul.cpp, line  400

[ 2] TimeRed(4)=2.86%, TimeRed(Ideal)=2.96%, Cov=3.81%, SelfP=4.45, DOALL
  LOOP matrixmul.cpp [ 213 -  231]:  OnMultLine
  FUNC matrixmul.cpp [ 207 -  207]:  OnMultLine called at file matrixmul.cpp, line  400

[ 3] TimeRed(4)=2.86%, TimeRed(Ideal)=2.96%, Cov=3.81%, SelfP=4.45, DOALL
  LOOP matrixmul.cpp [ 213 -  225]:  OnMultLine
  FUNC matrixmul.cpp [ 207 -  207]:  OnMultLine called at file matrixmul.cpp, line  400
```