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 and keeping the energy cost at the minimum level without jeopardizing the application performance and its results. Overall, the problem consists in the coexistence work of multicore specs, its parallelism and its shared cache problems; CPU and GPU parallelism and scheduling tasks; performance; and energy costs.

For this problem's solution is expected to find/create an autotuner, or at least a concept proof, that can achieve the best performance in a software application by enhancing the application's code automatically in a level that takes the best benefit of the available hardware without elevated energy costs. To do so, after creating its code, the developer runs the autotuner and it will enhance, automatically, the code to get the best performance.

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. The metrics that will be used to compare these three code versions are the following: processing power; execution time; number of memory accesses; and energy consuming.

With this solution, 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 e mantendo o nível de custo energético o mais baixo possível sem prejudicar o resultado e o desempenho da aplicação.

Para solucionar este problema é esperado encontrar/criar um autotuner, ou pelo menos uma prova de conceito, que consegue atingir o melhor desempenho numa aplicação de software, aprimorando automaticamente o código da aplicação a um nível que take o melhor proveito do hardware disponível sem custos elevados de energia. Para tal, após o código criado, o programador correrá o autotuner e este irá aprimorar, automaticamente, o código para atingir o melhor desempenho.

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. As métricas que serão utilizadas para comprar estas três versões de código são as seguintes: poder de processamento; tempo de execução; número de acessos a memória; e custo energético.

Com esta solução, as aplicações conseguiram atingir o seu melhor desempenho possível de forma automática e sobrecarregando menos os programadores a criarem código paralelo o que, consequentemente, poupar-lhes-á tempo.

Contents

CONTENTS

List of Figures

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 [**?**]

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 [**?**], 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 labor because it requires a lot of time and effort to achieve significant performance.

1.3 Structure of the Report

This report is divided in three more chapters. The next one is called *Achieving the Highest Processing Power*, and it is related to the state of the art of my thesis' scope. In this chapter there are three sections. The first section is related to 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.

The third chapter addresses the problem involved in my thesis and how I propose to solve it, including the approach, the methodology and solution's validation. The last chapter includes final consideration related to the work developed so far in *Preparação da Dissertação* course, expected results with my proposed solution and work plan to develop the solution. In the end of this report there are the references used to develop this report.

Chapter 2

Achieving the Highest Processing Power

The introduction describes a brief overview about each content of each chapter that 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. [**?**]

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

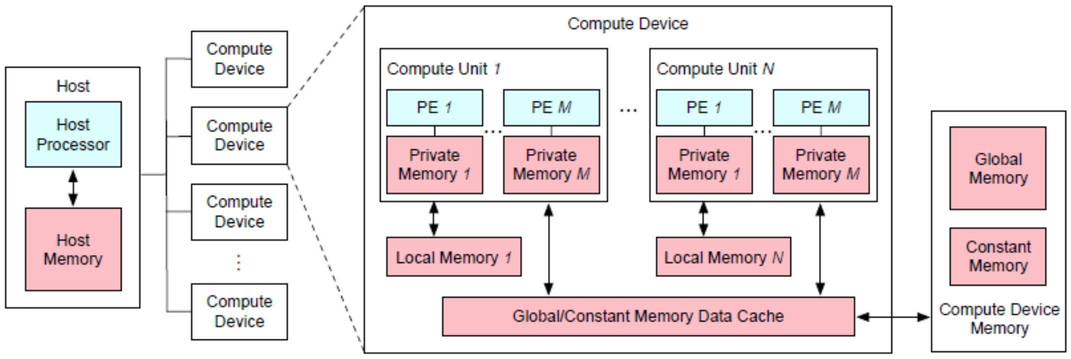

Figure 2.1: The OpenCL platform model and the OpenCL memory model

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 [**?**]. 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 3.1 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. [**?**] 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* [**?**]. 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. [**?**]

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. [**?**]

The following software tools that will be presented have, as its base support, OpenMP directives to help in parallelizing code, or at least, to measure performance.

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. [?]. 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?" [?]. 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. [?]

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. [?]

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?" [?]. 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. [**?**]

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.

2.4 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. 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

The introduction describes a brief overview about each content of each chapter that 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.

3.1 Introduction

3.2 Using Computers' Heterogeneous Components

3.2.1 OpenCL

3.3 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. 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".

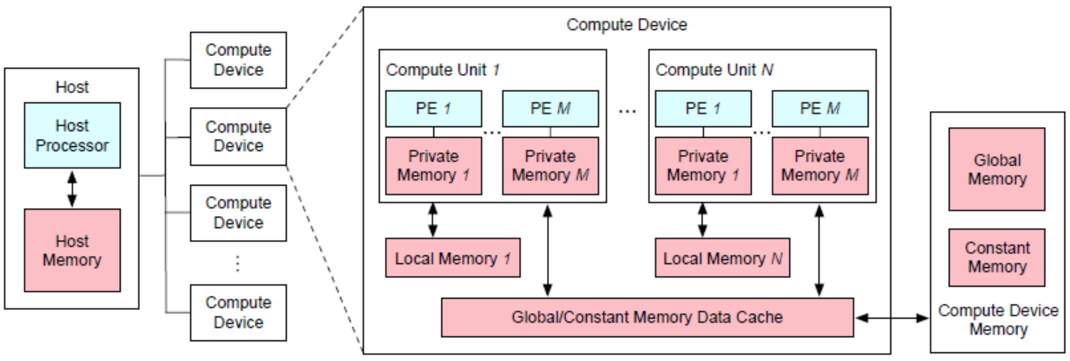

Figure 3.1: The OpenCL platform model and the OpenCL memory model

Chapter 4

Methodology

4.1 Introduction

According to the state of the art presented in chapter three, 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 applications programs. For this propose, Kremlin had a crucial impact in other to understand the viability of automatically parallelize code.

4.2 Research Method

-como usar o kremlin -aplicar no meu código -paralelizar o código -retirar tempos -conclusões

4.2.1 Data collection

In order to achieve such performance in the conditions mentioned in the problem's section, I purpose an autotuner or a concept proof that will enhance the application. For that, this autotuner will receive the program's original source code and though parallel optimization a new code will emerge. This code's modification will increase applications performance without jeopardizing the application's outcome.

4.2.2 Data analysis method

To create this autotuner, the first step is to identify what parameters exist to tune and what impact they have in the application. For this matter, with the help of Kremlin and manual expert parallelization applications will increase its performance and parameters will be found. As Kremlin detect possible parallelized code and, additionally, measures the applications speedup with such modifications, and with manual expert parallelization, there will be a confront with these results

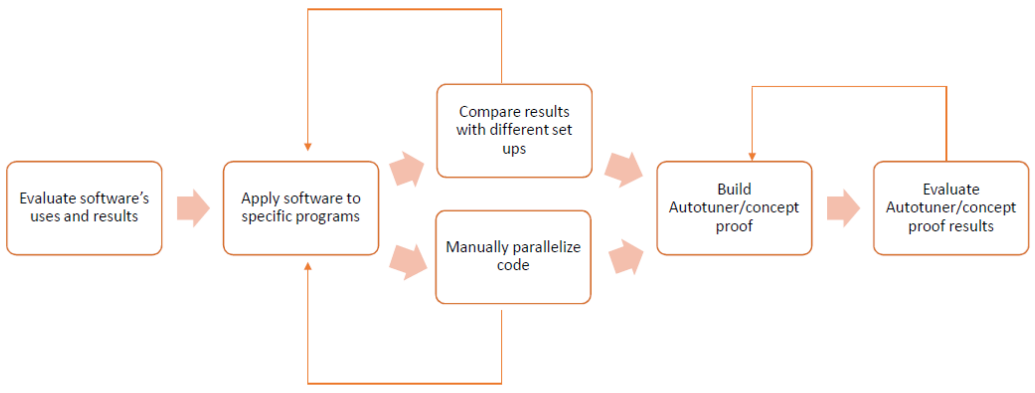
Evaluate software's uses and results
Apply software to specific programs
Compare results with different set ups
Manually parallelize code
Build Autotuner/concept proof
Evaluate Autotuner/concept proof results

Figure 4.1: Methodology that will be followed

and find what is better between these two scenarios. With this confrontation, and with different use cases, the tunning parameters will be found and the autotuner will be made.

4.2.3 Solution's Methodology

In the figure 4.1 is outlined the steps that will be followed to create the autotuner. This methodology has six states. Firstly and using simple applications, an evaluation will be made for Kremlin to understand how to use this software tool and evaluate the results that Kremlin can achieve, for instance, if it has similar results comparing with a expert parallelizing manually the same code. Then Kremlin will be applied to specific applications. After this state, I will compare results with different Kremlin set ups and it is possible to return to the previous state to apply Kremlin but in a different configuration set up.

Simultaneously and for the same application, I will manually parallelized code in order to evaluate the results and compare with the Kremlin results. As these states (the experiences with Kremlin and the manual code parallelization) requires several attempts there will be transitions between manual and Kremlin states.

The creations of the autotuner or concept proof happens when the tunning parameters are found after several attempts in the previous states. To make sure that the autotuner is being built correctly, evaluations will be made in order to verify the autotuner performance, so that the autotuner is correctly being made.

To sum up, this methodology as three main stages: learn and evaluate Kremlin's uses and results; finding the tunning parameter thorough several attempts using Kremlin's outputs and manually parallelize the application's code, and compare and analyze the results in every attempt; and, in the end, the constructions of the autotuner.

4.2.4 Data validation

To validate the whole methodology process, not only the autotuner itself but to compare the results between Kremlin outcomes and the code manually being parallelized, for evaluation metrics will be used: system energy consumptions when running the application; applications execution time; number of memory accesses and cache misses on the application; and the processing power measured by the number of instructions per secs. These evaluation metrics will be compared in three different cases: the original sequential code; manually paralleled code; and "automatic" paralleled code. In this last case it can be with Kremlin or with autotuner, depending in which of the methodology's state I am currently in.

To measure the above mentioned evaluation metrics, some libraries will be used: to measure energy consumptions RAPL will be used [**?**]; to measure application execution time OpemMP will be used [**?**]; to measure number of memory accesses, cache misses and processing power PAPI [**?**] will be used.

Finally, two use cases will be used to validate this solution: a biopharmaceutical HPC application for accelerating drug discovery; and a self-adaptive navigation system to be used in smart cities. These use cases will be the applications that, the autotuner will try to achieve the highest processing power without jeopardizing the applications' outcome and having a low energy consumption.

Chapter 5

2 Resultls and Discussion

Resultls and Discussion

Chapter 6

Conclusion

Conclusion

include11-appendix1