Multi-Kernel Auto-tuning on GPUs: Performance and Energy-Aware Optimization

João Filipe Dias Guerreiro

Thesis to obtain the Master of Science Degree in
Electrical and Computer Engineering

Supervisors: Doctor Pedro Filipe Zeferino Tomás
Doctor Nuno Filipe Valentim Roma

Examination Committee
Chairperson: Doctor Nuno Cavaco Gomes Horta
Supervisor: Doctor Pedro Filipe Zeferino Tomás
Member of the Committee: Doctor Hêrve Miguel Cordeiro Paulino

October 2014
Acknowledgments

I would like to thank my supervisors: Dr. Pedro Tomás, Dr. Nuno Roma and Dr. Aleksandar Ilic, for all their helpful insight, guidance and support through the development of the work presented in this thesis and especially when the deadlines started to tighten up.

Furthermore, I would like to thank INESC-ID for providing me the tools to develop the experimental works for this thesis.

I would also like to thank my family, for all their support given me during the course of this work.

Finally, I would also like to thank my colleagues that have helped me through my studying period in IST.

To all a big and heartfelt thank you.

The work presented herein was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under projects Threads (ref. PTDC/EEA-ELC/117329/2010) and P2HCS (ref. PTDC/EEI-ELC/3152/2012).
Abstract

Prompted by their very high computational capabilities and memory bandwidth, Graphics Processing Units (GPUs) are already widely used to accelerate the execution of many scientific applications. However, programmers are still required to have a very detailed knowledge of the GPU internal architecture when tuning the kernels, in order to improve either performance or energy-efficiency. Moreover, different GPU devices have different characteristics, causing the transfer of a kernel to a different GPU typically requiring the re-tuning of the kernel execution, in order to efficiently exploit the underlying hardware. The procedures proposed in this work are based on real-time kernel profiling and GPU monitoring and automatically tune parameters from several concurrent kernels to maximize the performance or minimize the energy consumption. Experimental results on NVIDIA GPU devices with up to 4 concurrent kernels show that the proposed solution can achieve, in a very small number of iterations, near optimal configurations. Furthermore, significant energy savings can be achieved by using the proposed energy-efficiency auto-tuning procedure.

Keywords

Automatic Tuning, Performance Optimization, Energy-Aware Optimization, Frequency Scaling, General Purpose Computing on GPUs, NVIDIA GPUs
Resumo

Devido à sua grande capacidade computacional e alta largura de banda da memória, as Unidades de Processamento Gráfico (GPUs) são já amplamente usadas para acelerar a execução de várias aplicações científicas. No entanto, os programadores ainda necessitam de ter um alto nível de conhecimento da arquitetura interna do dispositivo em questão, para poderem melhorar a performance ou eficiência energética de uma determinada aplicação. Além disso, diferentes GPUs têm características distintas, o que faz com que a transferência de um kernel para uma GPU diferente, necessite que o programador volte a ajustar a execução do kernel, para que se explore de forma eficiente o hardware do dispositivo. Os procedimentos propostos nesta obra são baseados no profiling dos kernels e monitoração da GPU em tempo real, levando a um automático ajustamento dos parâmetros essenciais de execução de vários kernels que executam de forma concorrente na GPU, de forma a maximizar a sua performance ou a minimizar o consumo energético. Os resultados experimentais realizados em GPUs da NVIDIA, com até 4 kernels concorrentes, mostram que a solução proposta consegue atingir, em um número reduzido de iterações, configurações quasi-óptimas. Além disso, ganhos energéticos significativos podem ser atingidos usando o procedimento de auto-tuning proposto que visa o aumento da eficiência energética.

Palavras Chave

Sintonização Automática de Kernels, Optimizações de Performance, Optimizações de Eficiência Energética, Escalonamento da Frequência, Unidade de Processamento Gráfico, NVIDIA GPUs, GPGPU
## Contents

1. **Introduction**
   - 1.1 Objectives .................................................. 1
   - 1.2 Main contributions .......................................... 2
   - 1.3 Dissertation outline ...................................... 3

2. **State-of-the-art**
   - 2.1 GPU architectures ........................................... 7
     - 2.1.1 NVIDIA's Tesla microarchitecture .................... 8
     - 2.1.2 NVIDIA's Fermi microarchitecture ................... 9
     - 2.1.3 NVIDIA's Kepler microarchitecture ................ 10
   - 2.2 GPU programming ............................................ 11
   - 2.3 NVIDIA framework ......................................... 12
     - 2.3.1 NVML .................................................... 13
     - 2.3.2 CUPTI ................................................ 14
   - 2.4 GPU characterization ....................................... 15
   - 2.5 Related work ............................................... 16
   - 2.6 Summary ................................................... 17

3. **Auto-tuning procedures**
   - 3.1 Search space reduction .................................... 18
   - 3.2 Single-kernel optimization ................................ 19
     - 3.2.1 Optimizing for performance .......................... 20
     - 3.2.2 Optimizing for energy ................................. 21
   - 3.3 Multi-kernel auto-tuning .................................. 22
   - 3.4 Summary ................................................... 23

4. **Auto-tuning tool**
   - 4.1 Auto-tuning tool description ............................... 24
     - 4.1.1 Requirements ............................................ 25
     - 4.1.2 Profiling ................................................ 26
     - 4.1.3 Adaptive search ........................................ 27
     - 4.1.4 Performance and energy monitoring of application kernels 28
   - 4.2 Summary ................................................... 29
List of Figures

2.1 NVIDIA’s Tesla logical organization ........................................... 9
2.2 NVIDIA’s Fermi logical organization ........................................... 10
2.3 NVIDIA’s Kepler chip block diagram, example with 14 SMs .......... 11
2.4 NVIDIA’s Kepler SM .............................................................. 12
2.5 Grid of thread blocks .............................................................. 13
2.6 Memory hierarchy ................................................................. 14
2.7 Power optimization space for the MatrixMul Kernel on Tesla K40c . 16
2.8 Kernel parametrization optimization examples in terms of performance and energy ................................................................. 17

3.1 Auto-tuning optimization procedure for GPU kernels .............. 24
3.2 Flow-chart diagram of single-kernel performance auto-tuning procedure ................................................................. 28
3.3 Example of the the proposed execution of Algorithm 3.1 .......... 30
3.4 Flow-chart diagram of the proposed single-kernel energy auto-tuning procedure ................................................................. 31
3.5 Example of the execution of Algorithm 3.2 ......................... 33
3.6 Flow-chart diagram of the proposed multi-kernel performance auto-tuning procedure ................................................................. 34
3.7 Flow-chart diagram of the proposed multi-kernel energy auto-tuning procedure ................................................................. 36

4.1 Overview of the developed Auto-tuning tool ......................... 40
4.2 Layer diagram of the developed auto-tuning tool .................... 41
4.3 Flow-chart diagram of the developed auto-tuning tool applying the proposed single-kernel energy auto-tuning procedure, with the APIs used at each stage ................................................................. 43
4.4 Multi-kernel energy-aware optimization procedure for 2 MatrixMul and 2 Lud kernels, executed on Tesla K40c ................................................................. 45
4.5 Multi-kernel energy-aware optimization procedure for 2 MatrixMul and 2 Lud kernels, executed on Tesla K40c ................................................................. 46
4.6 Overview of the developed Auto-tuning tool when integrated with the GPU application ................................................................. 46
4.7 Auto-tuning tool example output: device properties ............... 48
4.8 Auto-tuning tool example output: iteration 2 ......................... 49
4.9 Auto-tuning tool example output: best execution configuration found ................................................................. 49
4.10 Example of usage of the power measuring tool ..................... 50
4.11 Example of power consumption of a GPU device over time .... 51
4.12 Example 2 of usage of the power measuring tool .................. 51
Chapter 4.13: Example of power consumption during the execution of GPU kernels.

Chapter 5.1: Execution of MatrixMul kernels with 16 blocks per grid on Tesla K20c.

Chapter 5.2: Execution of MatrixMul kernels with 16384 blocks per grid on Tesla K20c.

Chapter 5.3: Occupancy levels for different thread block sizes for the kernel FDTD3D, on different GPU devices.

Chapter 5.4: Execution time for different thread block sizes for the kernel FDTD3D, on different GPU devices.

Chapter 5.5: Performance optimization procedure for the Streamcluster kernel. The full optimization space includes 10 possible kernel configurations on all tested GPU devices.

Chapter 5.6: Performance optimization procedure for the Lud kernel. The full optimization space includes 9 possible kernel configurations on the GTX285 and GTX580 devices, and 10 on the remaining tested GPU devices.

Chapter 5.7: Energy-aware optimization procedure for the Streamcluster kernel. The full search space for energy minimization includes 50 and 40 possible configurations for the Tesla K20c and Tesla K40c GPUs, respectively.

Chapter 5.8: Energy-aware optimization procedure for the Lud kernel. The full search space for energy minimization includes 25 and 20 possible configurations for the Tesla K20c and Tesla K40c GPUs, respectively.

Chapter 5.9: Energy-aware optimization procedure for the ParticleFilter kernel. The full search space for energy minimization includes 45 and 36 possible configurations for the Tesla K20c and Tesla K40c GPUs, respectively.

Chapter 5.10: Multi-kernel performance optimization procedure for the 2 concurrent MatrixMul kernels. The full optimization space includes 25 possible kernel configurations on all tested GPU devices.

Chapter 5.11: Multi-kernel performance optimization procedure for the MatrixMul and Lud kernels. The full optimization space includes 50 possible kernel configurations on all tested GPU devices.

Chapter 5.12: Multi-kernel performance optimization procedure for the ParticleFilter and Lud kernels. The full optimization space includes 45 possible kernel configurations on all tested GPU devices.

Chapter 5.13: Multi-kernel energy-aware optimization procedure for 2 concurrent MatrixMul kernels, executed on Tesla K40c. The full optimization space includes 100 possible configurations.

Chapter 5.14: Multi-kernel energy-aware optimization procedure for the MatrixMul and Lud kernels, executed on Tesla K20c. The full optimization space includes 125 possible configurations.

Chapter 5.15: Multi-kernel energy-aware optimization procedure for the ParticleFilter and Lud kernels, executed on Tesla K20c. The full optimization space includes 225 possible configurations.

Chapter 5.16: Performance optimization procedure for 4 concurrent kernels, namely Lud, Streamcluster, Hotspot and ParticleFilter, executed on a Tesla K40c GPU. The full optimization space includes 2250 possible configurations.
5.17 Energy-aware optimization procedure for 4 concurrent kernels, namely \textit{Lud}, \textit{Streamcluster}, \textit{Hotspot} and \textit{ParticleFilter}, executed on a Tesla K40c GPU. The full optimization space includes 9000 possible configurations.

5.18 Energy-savings and performance trade-offs, obtained comparing the results found when optimizing for performance and energy-efficiency. The results for the $2 \times \text{MatrixMul}; 2 \times \text{MatrixMul}$ & $\text{MatrixTrans}$ & $\text{VectorAdd}$; $2 \times \text{MatrixMul}$ & $2 \times \text{Lud}$; and \textit{Streamcluster} & \textit{ParticleFilter} & \textit{Lud} & \textit{Hotspot} come from experiments executed on Tesla K40c, while the remaining are from Tesla K20c.
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Architecture and kernel-specific parameters</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>NVIDIA Tesla K40c Characteristics</td>
<td>42</td>
</tr>
<tr>
<td>4.2</td>
<td>Additional NVIDIA Tesla K40c Characteristics</td>
<td>43</td>
</tr>
<tr>
<td>5.1</td>
<td>Considered benchmark applications</td>
<td>54</td>
</tr>
<tr>
<td>5.2</td>
<td>Characteristics of the used GPU devices</td>
<td>56</td>
</tr>
<tr>
<td>5.3</td>
<td>Number of registers per thread for the \textit{FDTD3D} kernel, on different GPU devices</td>
<td>58</td>
</tr>
<tr>
<td>5.4</td>
<td>Summary of the results for the single-kernel performance and energy-efficiency auto-tuning procedures</td>
<td>67</td>
</tr>
<tr>
<td>5.5</td>
<td>Summary of the results for the multi-kernel performance and energy-efficiency auto-tuning procedures</td>
<td>68</td>
</tr>
</tbody>
</table>
List of Algorithms

3.1 Single-Kernel Performance Auto-Tuning Procedure ........................................... 29
3.2 Single-Kernel Energy-Aware Auto-Tuning Procedure ........................................... 32
3.3 Multi-Kernel Performance Auto-Tuning Procedure ............................................ 35
3.4 Multi-Kernel Energy-Aware Auto-Tuning Procedure .......................................... 37
List of Acronyms

ALU Arithmetic and Logic Unit
API Application Program Interface
BFS Breadth-first search
CPU Central Processing Unit
CUDA Compute Unified Device Architecture
CUPTI CUDA Profiling Tools Interface
ECC Error Correcting Code
FPGA Field-Programmable Gate Array
FPU Floating Point Unit
GPU Graphics Processing Unit
GPGPU General-Purpose computing on Graphics Processing Units
IPC Instructions Per Cycle
LD/ST Load/Store Units
MAD Multiply/Add
NVML NVIDIA Management Library
PAPI Performance Application Programming Interface
PDE Partial Differential Equation
PCle Peripheral Component Interconnect Express
PTX Parallel Thread Execution
QoS Quality-of-Service
SFU Special Function Unit
SIMT Single Issue Multiple Thread
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>SM</td>
<td>Streaming Multiprocessor</td>
</tr>
<tr>
<td>SP</td>
<td>Streaming Processor</td>
</tr>
<tr>
<td>SPA</td>
<td>Streaming Processor Array</td>
</tr>
<tr>
<td>SRAD</td>
<td>Speckle Reducing Anisotropic Diffusion</td>
</tr>
<tr>
<td>SSQ</td>
<td>sum of squared differences</td>
</tr>
<tr>
<td>TPC</td>
<td>Texture/Processor Cluster</td>
</tr>
<tr>
<td>TSC</td>
<td>Time Step Counter</td>
</tr>
</tbody>
</table>
1 Introduction

Contents

1.1 Objectives .................................................. 3
1.2 Main contributions ........................................ 3
1.3 Dissertation outline ...................................... 4
Exploiting the capabilities of Graphics Processing Units (GPUs), for general-purpose computing (GPGPU) is emerging as a crucial step towards decreasing the execution time of scientific applications. In fact, when compared with the latest generations of multi-core Central Processing Units (CPUs), modern GPUs have proved to be capable of delivering significantly higher throughputs [6], thus offering the means to accelerate applications from many domains.

Nonetheless, achieving the maximum computing throughput and energy efficiency has been shown to be a difficult task [25]. Despite the set of existing GPU programming Application Program Interfaces (APIs) (e.g., Compute Unified Device Architecture (CUDA) [6] and OpenCL [15]), the programmer is still required to have detailed knowledge about the GPU internal architecture, in order to conveniently tune the program execution and efficiently exploit the available computing capabilities.

To improve performance of a GPU kernel, current optimization techniques rely on adjusting the parameters of kernel launching, such as the number of threads and thread blocks as they are called in CUDA (work-items and work-groups in OpenCL), as a relative measure of kernel-architecture interrelationship. However, many modern GPUs already allow adjusting their operating frequency, thus providing the opportunity for novel energy-aware GPU kernel optimizations. In fact, tight power and dissipation constraints may also limit the possibility of efficiently employing all available GPU resources, a problem which is expected to worsen in the presence of the Dark Silicon phenomenon [22]. Consequently, future kernel optimization strategies must also consider the energy efficiency of the developed codes, despite the greater difficulties to analyse and design for simultaneous performance and energy efficiency optimization.

In particular, by varying the thread block size or the device operating frequency, one can achieve significant energy savings, which are tightly dependent on the application kernel under execution. Although a more flexible parametrization might offer convenient compromises between computing performance (throughput) and energy efficiency, such trade-offs are not easy to establish and are often device-dependent. This variability is also observed when considering code-portability, where significant performance and energy efficiency degradations can be observed not only when executing the same application on other devices characterized by a different amount of resources (or even a different architecture), but also when running the same program in a different GPU characterized by a rather similar architecture and resources.

Recent research works have suggested that the hardware utilization could be maximized if multiple applications are simultaneously scheduled to run on the GPU. Contrasting with commonly used time-multiplexing techniques, such an approach essentially consists on exploiting spatial-multitasking techniques, which allow multiple different kernels to share the GPU resources at the same time. As an example, Adriaens et al. [18] state that spatial-multitasking offers better performance results than cooperative- and preemptive-multitasking, because General-Purpose computing on Graphics Processing Units (GPGPU) applications often cannot fully utilize the hardware resources, which means those unused resources could be used by other applications.

Optimization of multi-kernel environments is however, a much more complex problem than the one of single-kernels. The reason for this, is that even though the possible configuration parameters for each separate kernel may be low, when combining all the possible combinations of the concurrent kernels,
the search space will increase exponentially. For this reason, it is important to find a new approach that allows the optimization of concurrent GPU kernels, without requiring the exploration of the whole optimization design space.

1.1 Objectives

To circumvent previously described challenges, this thesis focuses on the following objectives:

- Derive a novel optimization methodology for the optimization of applications relying on single and concurrent GPU kernels.

- Investigate methodologies for performance and energy optimizations across different NVIDIA GPUs, by relying on GPU internal counters to measure energy consumption.

- Development of an auto-tuning tool for the automatic application of the optimization methodologies on CUDA kernels.

1.2 Main contributions

In this dissertation it is proposed a novel methodology for the optimization of GPU kernels. It consists of a set of new auto-tuning procedures targeting performance or energy-aware optimization, across different architecture/generation and for single or several simultaneously executing kernels. These procedures are able to find the most convenient thread block and operating frequency configurations, in relation to the considered device and application. In order to expedite the procedure execution, a reduction of the design optimization space is performed. The proposed methodology start by looking at a measure of the utilization associated with each possible kernel configuration, and filter the ones with high utilization. Afterwards, one of the four presented algorithms is executed, depending on the type of optimization desired: single-kernel optimizing for performance, single-kernel optimizing for energy, multi-kernel optimizing for performance or multi-kernel optimizing for energy.

To complement the proposal of the new procedures, it is also presented in this work an auto-tuning tool for GPU kernels, that consists in an implementation of the proposed procedure for CUDA kernels. This tool is able to query the GPU device and application kernels, in order to obtain the profiling information used during the execution of the algorithms. Besides running the proposed algorithms, this tool also has mechanisms to perform frequency scaling of the GPU core, and it includes an additional developed application that measures the power and energy consumed by the GPU, applying the appropriate corrections to the power readings, as mentioned by Burtscher, et al. Even though the developed implementation of the proposed procedures only works for CUDA kernels, a different implementation could be developed with support for OpenCL kernels.

To evaluate the proposed procedures, extensive experimental evaluation with several applications from GPU benchmark suites is done. The obtained results are compared with the corresponding best possible setups, obtained by “brute-force” searching approaches within the whole search space. The benefits of the search space reduction can be specially seen in the tested situation with 4 concurrent kernels, where
the procedure reduces the optimization space in 98%. Experimental results show that the proposed procedures discover near-optimal configurations with a significant reduced number of optimization steps. Furthermore, it shows that up to 13% energy savings can be achieved by applying the proposed energy-aware optimization.

The scientific contributions of this work have been submitted for communication in the following conference:


1.3 Dissertation outline

This dissertation is organized in 4 chapters with the following outline:

- **Chapter 2 - State-of-the-art**: This chapter presents a summary of the current state-of-the-art related with the subject in study. It starts by presenting the GPU architectures that were used during the execution of this work, namely the three NVIDIA microarchitectures used in the experiments. Furthermore, it briefly presents the CUDA and OpenCL programming models, which are used to program general purpose applications on GPUs, followed by a description of the NVIDIA Management Library (NVML) and CUDA Profiling Tools Interface (CUPTI) libraries, which are used to monitor and control the state of the GPU and to profile the GPU application kernels. Afterwards, it describes some interesting characteristics of the current functioning of GPUs, in order to contextualize the problem that this work is trying to address. To finalize the chapter it is presented an overview of the state-of-the-art studies that have been done in this area of work, namely works that address performance portability on GPUs, concurrent programming on GPUs and energy-efficiency on GPUs.

- **Chapter 3 - Auto-tuning procedures**: In this chapter, the procedures to optimize the execution of GPU kernels proposed in this dissertation is presented. The chapter starts by presenting the methodology to perform the reduction of the design optimization space, used by all of the presented procedures. This search space reduction is done not only to fasten the execution of the procedures, but also as a way of guaranteeing high utilization of the GPU resources. Afterwards, the algorithms used for performance and for energy in single-kernel or multi-kernel environments are presented, as well as a description for each of them.

- **Chapter 4 - Auto-tuning tool**: In this chapter is presented the auto-tuning tool that implements the procedures described in the previous chapter. Using this tool, programmers can rapidly find the kernel configuration that best suits their application kernels for the given GPU device. Furthermore, it is also presented the tool developed for measuring the power and energy consumption of GPU devices, which can be used integrated in the auto-tuning tool or independently.
• **Chapter 5 - Experimental results** In this chapter are presented the experimental results obtained when testing the auto-tuning procedures and tool presented in the previous chapters. It starts by describing the GPU applications used, as well as the GPU devices where the experiments were executed, followed by the presentation of some interesting results important to take into consideration. Finally, the chapter presents the results obtained applying the auto-tuning procedures to a wide set of GPU kernels.

• **Chapter 6 - Conclusions** This final chapter presents the conclusions taken from this work, and possibilities for future evolutions for the work are mentioned.
2 State-of-the-art

Contents

2.1 GPU architectures ................................................................. 8
  2.1.1 NVIDIA’s Tesla microarchitecture .................................. 9
  2.1.2 NVIDIA’s Fermi microarchitecture .................................. 10
  2.1.3 NVIDIA’s Kepler microarchitecture ................................. 11

2.2 GPU programming ............................................................... 12

2.3 NVIDIA framework ............................................................... 14
  2.3.1 NVML ................................................................. 14
  2.3.2 CUPTI ............................................................... 15

2.4 GPU characterization ............................................................ 16

2.5 Related work ................................................................. 18

2.6 Summary ................................................................. 21
In order to reflect the evolution of the state-of-the-art in terms of GPU programming, this chapter provides an overview of the basic architectural concepts and functionalities of modern GPU architectures. It also presents a brief description of the main characteristics of the CUDA programming model, used to create GPGPU applications, as well as the framework provided by NVIDIA, that allows the monitoring and control of the application kernels and GPU devices (NVML and CUPTI).

Furthermore, an overview of the characteristics of the execution of kernels on modern GPU devices is provided, highlighting some more interesting features that emphasize the problem addressed with this work. Finally, the chapter finishes with a round-up of the current state-of-the-art studies related with this work, including those addressing the following subjects: i) performance portability on GPU devices; ii) concurrent programming of GPU kernels; and iii) energy-efficiency on GPU devices.

### 2.1 GPU architectures

Research on GPU architectures has been mainly focused on the acceleration of computer graphics, with special focus on the gaming industry, leading to specialized real-time graphics processors. However, due to the increasing design complexity and evermore requirements for additional programming support, significant architecture changes were recently introduced, leading into a fully programmable co-processor with very high performance, surpassing the modern CPUs in terms of arithmetic throughput and memory bandwidth. Accordingly, recent GPU architectures include many simple processing cores making use of single instruction multiple thread (SIMT) technology to improve performance, making GPUs the ideal processor to accelerate data parallel applications.

The first attempts to use GPUs with non-graphical applications started in 2003 and required the use of graphics APIs. However, these APIs required the programmers to have deep knowledge of the graphics processing pipeline and API, as well as in the GPU architecture, since programs had to be written using high-level shading languages such as DirectX, OpenGL and Cg, which greatly increased the program complexity. Another major fault at the time, was the non-existent support for double precision, which meant several scientific applications could not be ported into GPU devices.

These problems prompted NVIDIA to create a new GPU architecture, called the G80 architecture (launched in late 2006); as well as a new programming model called CUDA (released in 2007). CUDA offered a programming environment to create GPU application using C-like constructs, making GPU programming available to a whole new range of programmers. In 2009, an alternative to CUDA appeared with OpenCL, which benefited from the fact that its use was not constrained to NVIDIA devices. The G80 architecture was the first of many architectures compliant with the CUDA programming model, however since 2006 NVIDIA has released 3 new microarchitectures. The first one, which the G80 is a part of was the Tesla, followed by the Fermi microarchitecture released in 2009. In 2012 NVIDIA release a new microarchitecture called Kepler, which was kept until early 2014 when it was released the most recent Maxwell microarchitecture. Since the experimental results for this dissertation were obtained using GPUs from the first three GPGPU capable NVIDIA microarchitectures, the next sections will feature a brief description of their characteristics, followed by a brief description of the CUDA programming model.
2.1.1 NVIDIA’s Tesla microarchitecture

The Tesla microarchitecture is based on a multiprocessor chip with a Streaming Processor Array (SPA), responsible for the GPU programmable calculations. As seen in Figure 2.1(a), the SPA is composed of several different blocks, called the Texture/Processor Cluster (TPC). These TPCCs are constituted by multiple Streaming Multiprocessors (SMs), which are themselves composed by multiple Streaming Processors (SPs), also called CUDA cores. The connection with the Host CPU is made via the Peripheral Component Interconnect Express (PCIe). There are 3 different levels of GPU memory: a global memory accessible by all processing units; a shared memory associated with each SM, accessible by the processing units of that SM; and a set of read-only on-chip caches for textures and constants. Each TPC has a texture unit, a texture cache and a geometry controller, which are the structures specially designed to support graphics processing.

![Diagram of NVIDIA’s Tesla microarchitecture]

(a) Chip block diagram, example with 9 SMs.

(b) SM.

Figure 2.1: NVIDIA’s Tesla logical organization.

SMs are the structural units that create, manage, schedule and issue the threads (in groups of 16 or 32), in a Single Issue Multiple Thread (SIMT) fashion. SMs can execute graphics related computations, as well as general purpose programs. The logic organization of each SM is presented in Figure 2.1(b). As it can be concluded from the figure, SMs are composed of 8 SPs, an instruction cache, a multithreaded instruction fetch and issue engine, a read-only cache for constant data, a 16 KB read/write shared memory and a register file. Each SM also includes two Special Function Units (SFUs), used for transcendental functions (such as sin, cosine, etc), where each contain four floating-point multipliers. Each SP contains a Multiply/Add (MAD) unit.

Aside from the initial Tesla microarchitecture, a few other generations were released by NVIDIA (from 1.0 to 1.3), each including specific architecture improvements aiming at a better performance in GPGPU programming. The main modifications were: increase of the number of SMs from 16 to 30; increase in the register file size from 8K to 16K 4-byte registers per SM; higher number of threads running in parallel per...
from 768 to 1024; introduction of native support for double precision arithmetics, and modifications in the memory architecture, such as the support for parallel execution of coalesced memory accesses.

2.1.2 NVIDIA’s Fermi microarchitecture

The NVIDIA’s Fermi microarchitecture [12], whose logical hierarchy is presented in Figure 2.2, introduced a few improvements over the previous generation. As it can be seen on Figure 2.2(a) the chip was divided into 16 SMs each including 32 SPs, resulting in 512 CUDA cores.

![Chip block diagram](image1)

![SM](image2)

![SP](image3)

(a) Chip block diagram
(b) SM.
(c) SP.

Figure 2.2: NVIDIA’s Fermi logical organization.

The Fermi microarchitecture implements a single unified memory request path for loads and stores, with an L1 cache per SM multiprocessor and a unified L2 cache that services all operations (load, store and texture). The 64KB L1 cache in each SM is configurable at compile time, as either 48KB of shared memory and 16KB of L1 cache (for caching of local and global memory operations), or 16KB of shared memory and 48KB of L1 cache. This way the programmer can chose the configuration that best suits the application requirements. Fermi also features a unified L2 cache of 768KB that services all load, store and texture requests. This architecture also introduced a mechanism of Error Correcting Code (ECC), used to protect the integrity of data along the memory hierarchy, which is specially useful in high performance computing environments.

The logic of each SM is presented in Figure 2.2(b). Each SM is composed of 32 SPs, 16 Load/Store Units (LD/ST) allowing source and destination addresses to be calculated for 16 threads at each clock cycle; 4 SFUs, 1 register file with 32K 4-byte registers; 1 instruction cache; 2 warp schedulers; 2 dispatch units; and 1 configurable shared memory/L1 cache. The organization of each SP is presented in Figure 2.2(c) and is constituted by a fully pipelined integer Arithmetic and Logic Unit (ALU) and a Floating Point Unit (FPU).

A major improvement introduced in the Fermi microarchitecture is its two level, distributed thread scheduler called the *GigaThread*. The highest level distributes groups of threads to each SM. Then, at the SM level, there are two warp schedulers and two instruction dispatch units. The dual warp scheduler is able to select two warps, and issue one instruction from each warp. Since warps execute independently, there is no need to check for dependencies. However, double precision instruction do not support dual
dispatch with any other operation. Another addition to the capabilities of the scheduler, is the ability to support concurrent kernel execution, meaning the GPU can execute at the same time different kernels from the same application context, allowing for a higher utilization of the GPU resources.

2.1.3 NVIDIA’s Kepler microarchitecture

The evolution from the Fermi to the Kepler microarchitecture [13] followed a similar trend than the previous evolution from the Tesla to the Fermi microarchitecture. Figure 2.3 presents the architecture of the Kepler, where it can be seen that despite the number of SMs having decreased to 14 (some implementations have 13 or 15 SMs), the amount of computational resources has substantially increased. Accordingly, each SM now includes 192 SPs (contrasting with Fermi which had 32), 32 LD/ST units (Fermi had 16), 32 SFU (Fermi had 4) and 62 double precision units (Fermi had 16). Each SM now has 4 warp schedulers and 8 instruction dispatch units, allowing four warps to be issued and executed concurrently, as shown in Figure 2.4. The quad warp scheduler works by selecting 4 warps, and 2 independent instructions per warp to be dispatched in each cycle. Unlike Fermi, in Kepler double precision instructions can be paired with other instructions.

While substantial changes were made to the SM architecture, the memory subsystem for the Kepler microarchitecture remains similar to the one used in Fermi architecture. Notwithstanding, the configurable shared memory and L1 cache of each SM now has support for a third configuration, allowing for a 32KB/32KB split between shared memory and L1 cache. In addition, the L2 cache was increased to 1546KB, doubling the amount present in the Fermi microarchitecture.

A new feature introduced in Kepler’s microarchitecture is Dynamic Parallelism, which enables kernels to launch other GPU kernels, managing the workflow and any needed dependencies without the need for host CPU interaction. Kepler also increases the total number of connections (work queues) between the host and the GPU device, by allowing 32 simultaneous, hardware-managed connections (in Fermi only 1 connection was allowed). This feature is called the Hyper-Q, and allows separate

Figure 2.3: NVIDIA’s Kepler chip block diagram, example with 14 SMs.
CUDA streams to be handled in separate work queues, enabling them to execute concurrently and eliminating situations of false serialization that could happen in previous microarchitectures. Finally, Kepler introduces the NVIDIA GPUDirect, which allows for direct transfers between GPU and other third party devices.

2.2 GPU programming

At the same time the complexity of GPU architectures increased, in order to provide significant performance improvements to applications, the management of the hardware also became more complex. To ease the programming of GPU devices, in 2007 NVIDIA released the Compute Unified Device Architecture (CUDA), a parallel computing and programming model. With a combination of hardware/software platforms it enables NVIDIA’s GPUs to execute programs written in C, C++, Fortran and other languages. Two years later Khronos Group released the OpenCL, a framework inspired by CUDA used for writing programs that execute across heterogeneous platforms CPUs, GPUs, Field-Programmable Gate Arrays (FPGAs), etc.). Like CUDA, OpenCL defines a C-like language for writing programs that execute on compute devices.

Using either one of these platforms, programmers can easily use GPUs for general purpose processing (GPGPU). When executing a program on a GPU using either of the aforementioned platforms, a Kernel will be invoked, that will execute across several parallel threads (or work-items in OpenCL). The programmer or the compiler further organizes the threads in thread blocks (work-groups in OpenCL), where the whole range of thread blocks is called a grid (NDRange in OpenCL).

CUDA and OpenCL have defined these terminologies, allowing an abstraction to the actual execution environment. As a consequence, there is an easier automatic scalability of GPU programs between GPU
devices, since each thread block can be assigned to any SM within a GPU in any order, concurrently or sequentially. Furthermore, compiled GPU programs are capable of executing on different GPUs with different number of SMs as only the runtime system needs to know the physical processor count.

As it can be seen on Figure 2.5, blocks can be organized in a one-dimensional, two-dimensional or even three-dimensional grid of thread blocks. The same applies to the organization of threads within a thread block. The number of threads within each thread block is defined by the programmer, and that value together with the problem size usually settles the total number of thread blocks in a grid.

![Figure 2.5: Grid of thread blocks.](image)

Each thread will execute an instance of the kernel, having its unique information and data such as thread IDs, thread block IDs, program counter, registers, per-thread private memory, inputs, and output results. Threads within a thread block can cooperate by using data in shared memory, and through barrier synchronization.

CUDA threads may access data from multiple memory locations as illustrated in Figure 2.6. Each thread has a per-thread private memory space used for register spills, function calls and C automatic array variables. Each thread block has a per-block shared memory space used for inter-thread communication, data sharing and result sharing. Grids of thread blocks can share results in the global memory space after kernel-wide global synchronization.

CUDA’s hierarchy of threads maps to the GPU hardware hierarchy the following way: each GPU executes one or more kernel grids; each SM executes one or more thread blocks; CUDA cores (SPs) and other execution units within the SM (SFUs, LD/ST) execute thread instructions. The SM schedules and issues threads in groups of 32 threads, called warps, which should be taken into account by programmers, in order to improve performance. As an example, this can be done by making sure threads in a warp execute the same code path, and memory accesses are done to nearby addresses.
2.3 NVIDIA framework

NVIDIA provides many tools that allow programmers to get more information about the GPU or applications kernels running on the device. Two of those tools, which were used during the course of this work, are NVML [9] and CUPTI [8], and are described in the following subsections.

2.3.1 NVML

The NVIDIA Management Library (NVML) [9] is a C-based API that can be used for monitoring or managing the state of NVIDIA GPU devices. It is the underlying library that supports the NVIDIA-SMI [10] command-line application. It provides the tools that allow programmers to create third-party applications and have direct access to the queries and commands related with the state of the GPU device.

Some of the queries that can be made, regarding the GPU state, and that are particularly relevant in terms of the subject of this dissertation, are the following:

- **GPU utilization**: Retrieves the current utilization rates for the device’s major subsystems. The returned utilization rates have two values, one related with the utilization of the GPU, the other related with the utilization rate of the memory subsystem. The first one is calculated as the percent of time, over the past sample period during which one or more kernels was executing on the GPU. The memory utilization rate is calculated as the percent of time, over the past sample period during which the global device memory was being read or written. The sample period is a characteristic of the device being queried, and its value may be between 1 second and 1/6 seconds.
• **Current clock rates:** Retrieves the current values for the memory and graphics clock rates of the GPU device.

• **Supported clock rates:** Retrieves the list of possible clocks for the device’s major subsystems (memory or graphics clocks). First the programmer retrieves the memory clocks, i.e., the frequency values that the memory system is allowed to take. For each of those values, the programmer can also get the list of possible graphics clocks, i.e., the values that the SM frequency is allowed to take. If desired, these values can later be used to set the memory and SM frequency levels.

• **Power usage:** For the compatible devices, e.g. NVIDIA’s Tesla K20c and Tesla K40c, it retrieves the current power usage for the GPU device, in milliwatts.

NVML also provides tools that allow the programmer to change the state of the GPU device, such as by using the following commands:

• **Set clock rates:** Set the clocks that applications will lock to. The command will set the frequency levels for the memory and graphics systems, that will be used when compute or graphics applications run on the GPU device.

• **Reset clock rates:** Resets the clock rates to the default values.

Apart from the mentioned functions, which were the ones used during this thesis, NVML provides many other ways to query and control the state of GPU devices. Other queries that can be made to the GPU device, include retrieving the current fan speed or temperature, the GPU operation mode or information regarding whether ECC is enabled or not, amongst many others. One particular inquiry that can be made to the GPU device, is regarding its power limit constraints, i.e., retrieving the values for the minimum and maximum power limits allowed for the GPU device. The result of this query could be later used in a call to the function that sets a new value for the power limit restriction for the device. Other commands that NVML provides, which allow changing the state of the device, are changing the ECC mode, the compute mode or the persistence mode.

### 2.3.2 CUPTI

The CUDA Profiling Tools Interface (CUPTI) [8] is a library that enables the profiling and tracing of CUDA applications, and supports NVIDIA profiler tools, such as `nvprof` or `Visual Profiler` [5], and just like in NVML, the CUPTI APIs can be used by third party applications. CUPTI can be divided into four APIs:

• **Activity API**, allowing programs to asynchronously monitor the activity of a CPU and GPU CUDA application.

• **Callback API**, allowing the programmer to register a callback into the code, which is invoked when the application being profiled calls a CUDA runtime or driver function, or when a specific CUDA event occurs.
- **Event API**, providing the tools to query, configure, start, stop and read the event counters on a CUDA-enabled device.

- **Metric API**, providing the tools to collect the application metrics that are calculated from the event values.

By combining these four APIs, CUPTI allows the development of profiling tools that can give important insights about the behaviour of CUDA applications at the level of both the CPU and the GPU.

### 2.4 GPU characterization

As stated before, within each SM the computations from the currently assigned thread block are performed in a group of data-parallel threads, which are referred to as thread warps. Hence, the number of assigned thread blocks directly influences the utilization of the available SMs while the characteristics and the number of threads (warps) in the block affects the utilization of hardware resources within each SM. As a result, the overall GPU utilization can highly vary depending on how the total amount of kernel work is organized in terms of the number of thread blocks and the number of assigned threads per block. Additionally, on highly data-parallel GPU architectures, good performance is usually achieved with high utilization of the GPU resources, since it will mean the total amount of kernel work will be distributed along all SMs.

However, there are multiple possibilities for the thread block configuration of a single kernel, and when the frequency is also considered, the total number of combinations can quickly grow out of proportion. In Figure 2.7 is one example of the power consumption on the Tesla K40c for a matrix multiplication kernel, when changing the thread block size and frequency level of the GPU device. Even when only considering thread block where the dimensions are powers of 2, it can be seen that the optimization space will have $11 \times 4 = 44$ possible combinations. It is important to mention that, even though, modern GPUs allow setting separate frequency levels for two different subsystems (memory and graphics), during this thesis

![Figure 2.7: Power optimization space for the MatrixMul Kernel on Tesla K40c.](image)

---

1. 1. **SM**
2. 2. **CPU**
3. 3. **GPU**
only the graphics operating frequency level will be considered, since the memory only supports two very

distinct frequency levels, where one of them is used when the GPU is idle. For this reason, only the GPU

operating frequency in the graphics subsystem will be considered as an optimization variable.

In multi-kernel optimizations the search space will grow exponentially with the number of concurrent

kernels. For example, in one of the tested scenarios, where the procedure optimizing for energy-awareness

was applied in a situation with 4 concurrent kernels (Lud, Streamcluster, Hotspot and ParticleFilter) on

Tesla K40c, the full design optimization space is composed of 9000 combinations of the kernel parameters

and the GPU operating frequency. For this reason, it crucial to find a method to reduce the optimization

design space, without losing the quality of the final found solution.

In addition, the range of possible kernel configurations is limited by the characteristics of the kernel,

as well as by the capabilities of the GPU device on which that kernel is executed. As an example, at

least three constraints, intrinsic to each GPU device, can limit the size of a thread block: the number

of registers in each SM, the amount of shared memory per SM and the maximum number of thread blocks

per SM. Once the thread block size is determined, the problem size should fix the number of thread

blocks in the grid. Since all thread blocks run in independent compute units (the SMs), the execution

of the kernel is only possible if the existing limits in terms of the used shared memory and number of

registers of each SM are satisfied.

However, these limits are not always the same across different GPU devices. In fact, GPU characteristics

has been significantly changing for the latest GPU generations, which forces programmers to

re-tune the applications every time they are ported to a different device. Hence, the selection of the

most convenient distribution of the application kernel over the GPU has proven to be a non-trivial design

optimization. This is emphasized on the kernel configuration examples that are illustrated in Figure 2.8

where significant performance differences can be observed when varying the thread block size (see “time

range”).

A non-trivial relation that can be observed in Figure 2.8 concerns the configurations achieving maximal

performance and energy efficiency (see “time range” vs. “energy range”). In fact, even though performance

and energy optimization may result in the same kernel configurations for certain applications

(e.g., 128 in Figure 2.8(a)), for other applications the optimal energy-efficient configurations considers a

completely different configuration (e.g., 256 vs. 1024 in Figure 2.8(b)).

![Figure 2.8: Kernel parametrization optimization examples in terms of performance and energy.](image)

(a) *Streamcluster* kernel executing on Tesla K20c. (b) Lud kernel executing on Tesla K20c.
This unpredictable behaviour, in terms of kernel configuration, opens the door for the new set of auto-tuning procedures that are proposed in this thesis. By following the proposed methodology, the programmer is able to simply supply the kernel and GPU information and get, as output, the most convenient configuration (e.g., thread block size or GPU operating frequency) that allows him to further optimize the kernel for either the attained performance or the energy consumption. Furthermore, the procedures proposed in Chapter 3 are completely kernel and GPU agnostic: they can work with any set of kernels and on any GPU device, independently of their characteristics and architectures. Additionally, the proposed procedures are not limited to single-kernel environments, being able to optimize the fundamental parameters for a given set of concurrent GPU kernels.

2.5 Related work

Not many research works have addressed the performance portability of GPU kernels. Notwithstanding, there are interesting approached to such problems. One particularly interesting work is RaijinCL [23], which provides an auto-tuning library for matrix multiplication applications, capable of generating optimized code for a given GPU architecture. However, this approach only considered a specific application (matrix multiplication), which contrasts with this thesis proposed methodology, which is completely agnostic in terms of the considered application kernels. In addition to that, while the procedures proposed in this thesis find the best kernel parameters, related with how the kernel will distributed over the GPU, RaijinCL focuses on optimizing the kernel source code for a given architecture.

Kazuhiko et al. [29] studied the difference between the performance offered by OpenCL kernels versus CUDA kernels and provided some guidelines on performance portability. They studied the optimizations performed by both compilers, concluding that with the default compiling options, OpenCL kernels achieve a much worse performance when compared with the CUDA implementation of the same application. However, when manually optimizing the OpenCL kernels, and choosing the appropriate compiler optimization options, the sustained performance of the kernels became almost the same as the performance of the CUDA ones. They also studied the performance of OpenCL kernels across different GPU architectures (NVIDIA vs. AMD). They stated that the kernel configuration (thread block size), needs to be adjusted for each GPU device, in order to obtain the best performance, which lead the authors to conclude that an automatic performance tuning methodology based on profiling is the best way to achieve performance portability of GPU kernels.

Sean et al. [33] also studied the performance portability of GPU kernels, namely the portability of optimized kernels from one GPU device to a different one. The authors observe that different GPUs architectures have distinct sensitivities to different optimizations techniques, concluding that special measures must be considered to obtain performance portability. The authors also state that auto-tuning approaches might be a viable way to achieve such goal, but these kind of approaches would continue to be time-consuming, which in their optic is the price to pay in order to achieve performance portability.

A few studies concerning the scheduling of concurrent kernels on GPUs have also been presented in the last few years. Initial steps suggested the use of cooperative-multitasking when scheduling GPU kernels.
That was the case of the work by Ino, Fumihiko, et al. [27], where the authors presented a cooperative-multitasking method for concurrent execution of scientific and graphics applications on GPUs. What they proposed, was allowing the GPU to use idle cycles to accelerate scientific applications. These scientific applications should be divided in small pieces, in order to be sequentially executed in the appropriate intervals, without causing any significant slow-down of the graphics applications.

A different approach was suggested by Adriaens et al. [18], which stated that spatial-multitasking offers better performance results when compared with cooperative- or preemptive-multitasking. According to the authors, GPGPU applications often cannot fully utilize the GPU hardware, meaning those unused resources could be used by other applications. The authors also stated that this is a trend that could only keep increasing as GPU architectures evolve. The experiments made by the authors showed the existence of significant performance benefits when using the proposed resource partitioning, over the most commonly used cooperative-multitasking, for environments with two, three and four simultaneous applications. It is also mentioned that spatial-multitasking provides opportunities for flexible scheduling and Quality-of-Service (QoS) techniques that can improve the user experience.

Aguilera et al. continued the work on spatial-multitasking of GPGPU kernels, by using kernels with QoS requirements. In particular, the authors noticed that some QoS applications did not require the allocation of all GPU SMs in order to meet the QoS requirements. This meant that when running the QoS application, they could allocate the resources required to meet the requirements, and still manage to allocate the remainder of the SMs to other non-QoS applications. The authors showed that static allocation of the resources could not be a solution, since the performance of the QoS application would depend on the other applications running at the same time. They ultimately proposed a run-time resource allocation algorithm to dynamically partition the GPU resources between concurrently running applications.

Another study on concurrent scheduling of GPU kernels was performed by Kayıran, Onur, et al. [28]. The authors claimed that filling the computation units (SMs) with thread blocks is not always the best solution to boost performance. The reason behind this is the long round-trip fetch latencies primarily attributed to high number of memory requests generated by executing more threads concurrently. To address this matter, the authors proposed a dynamic scheduling algorithm of thread blocks for GPGPUs, which attempts to allocate the optimal number of thread blocks per-core based on application demands, trying to reduce contention in the memory subsystem. The algorithm is able to find the nature of the application (compute-bound or memory-bound), and with that knowledge it allocates the appropriate resources.

However, the previously mentioned multi-kernel studies and many others have been based on simulation environments like the GPGPU-SIM [19] simulator, as opposite to using real GPU systems. This is mainly because the NVIDIA Tesla, Fermi and Kepler microarchitectures do not provide any tool to manually turn on/off the computation units, or even the possibility to individually assign application kernels to specific computation units. Other limitations of current GPUs, were acknowledged by Pai et al. [32], when they studied concurrent execution of GPU kernels using multiprogram workloads. What they found out is that, the current implementation of concurrency on GPUs suffers from a wide variety
of serialization issues, that prevent concurrent execution of the workloads. The main issues that caused serialization were the lack of resources, and serialization due to exclusive execution of long-running kernels and memory transfers. The authors proposed the use of elastic kernels which allow for a fine-grained control over their resource usage. Adding to that, they proposed several elastic-kernel-aware concurrency policies that improve the concurrency of workloads. They also presented a technique to timeslice kernel execution using elastic kernels, addressing the serialization due to long-running kernels.

Hong et al. [25] studied the energy savings of introducing an integrated power and performance model, allowing the prediction of the optimal number of active processors for a given application. The developed model also takes into account increases in power consumption that resulted from the increases in temperature. Assuming the availability of power gating to limit the number of active cores, the authors analyse the Parallel Thread Execution (PTX) output code, and based on the number of instructions and memory accesses, use that information to model the power and performance of GPUs. This means, their approach needs to be made offline, not considering any analysis of the input data.

Abe et al. [17] presented a power and performance analysis of GPU-accelerated systems based on the NVIDIA Fermi architecture. They concluded that, unless power-gating is supported by the GPU, the CPU is not a significant factor to obtain energy savings in GPU-accelerated systems. On the contrary, they stated that voltage and frequency scaling of the GPU is very significant when trying to reduce energy consumption.

Kai et al. [31] presented GreenGPU, a holistic energy management framework for heterogeneous architectures, that is able to achieve an average 21.04% energy savings. The proposed framework, tries to balance the workload of the CPU and GPU, minimizing energy wasted while one side is waiting for the other. However, the framework is limited to scaling the operating frequencies of the GPU device, accordingly to the type of the application kernel being executed. It also scales the frequency and voltage levels at the CPU in a similar way.

Burtscher et al. [20] presented their work on the measuring of GPU power using the built-in sensor of the Tesla K20c GPU. They found out several unexpected behaviours when the measurements were done and present a methodology to accurately compute the true power and energy consumption using the sensor data.

Huang et al. [26] try to characterize and debunk the notion of a “power-hungry GPU”, through an empirical study of the performance, power, and energy characteristics of GPUs for scientific computing. This was done by taking a program that traditionally runs on CPU environments, and testing it on an hybrid CPU-GPU environment, using different implementations and evaluating each one in terms of performance, energy consumption and energy-efficiency. In the end, they were able to achieve an energy-delay product multiple orders of magnitude better than the one achieved in traditional CPU environments.
2.6 Summary

This chapter presented an overview of the main characteristics of the NVIDIA microarchitectures, namely the Tesla, Fermi and Kepler microarchitectures, followed by the programming models used to program general purpose applications on GPU devices (CUDA and OpenCL), which together, constitute the hardware devices and software tools that were used during the fulfilment of this thesis. Then, it was presented the tool provided by NVIDIA for monitoring and controlling the state of the GPU device (NVML), as well as the library used for the profiling of CUDA applications (CUPTI). Subsequently, it was presented a characterization of modern GPUs, followed by an overview of the studies that have been done on this field of work. Furthermore, the analysis of the current state-of-the-art makes clear that performance and energy-aware portability on GPU devices is still an open and unresolved problem in the GPGPU computing research area. State-of-the-art analysis, suggests that there has not yet been any previous work targeting energy-aware auto-tuning, and offering performance portability across GPU devices without any further programmer intervention.
3 Auto-tuning procedures

Contents

3.1 Search space reduction ........................................... 25
3.2 Single-kernel optimization ....................................... 27
  3.2.1 Optimizing for performance ............................... 28
  3.2.2 Optimizing for energy ...................................... 30
3.3 Multi-kernel auto-tuning ......................................... 33
3.4 Summary ............................................................ 38
As previously mentioned, the objective of this thesis is to provide a novel optimization methodology for GPU kernels, that envisages the selection of a convenient configuration for the fundamental kernel parameters and an appropriate value for the device operating frequency. Using performance metrics and the power counters of the GPU devices, the proposed procedures target the automatic determination of the most adequate configuration for the concurrent execution of a set of GPU kernels, such that different optimization goals can be achieved. In order to tackle the most fundamental aspects of modern GPU architectures and applications, two different auto-tuning procedures are derived that allow optimizing the GPU kernel execution for performance and energy-efficiency.

While ideally, it would be important to derive an analytic cost function, which would allow easily finding the optimal distribution in a single step, such an approach is especially difficult, since it is a non-linear discrete optimization problem. To tackle this problem, exhaustive characterization of the application kernels and GPU devices could be performed. However, this work is out of the scope of this thesis because of time constraints. Accordingly, the work herein proposed is based on an iterative algorithm that selectively tests a set of kernel configurations, in order to find the optimal case for each specific goal.

An overview of the structure of the procedures is presented in Figure 3.1. The procedures start by getting information about the GPU device and the application kernel, and subsequently use this information to reduce the search space of the possible distributions. By reducing the overall design optimization space to a subset of possible GPU kernel configurations (in terms of the number of threads and thread blocks), the proposed procedures do not only allow for a fast discovery of the most adequate configurations, but they also guarantee a high utilization of the GPU resources.

![Figure 3.1: Auto-tuning optimization procedure for GPU kernels.](image)

Accordingly, this chapter is organized as follows. Section 3.1 presents the kernel and GPU-aware search space techniques. Section 3.2 proposes performance and energy-aware optimization procedures for single-kernel configurations. By taking advantage of these procedures, multi-kernel procedures are also derived and presented in Section 3.3. Finally, Section 3.4 concludes the chapter by summarizing the main contributions.
3.1 Search space reduction

As it can be seen in Figure 3.1, the proposed procedures are based on an iterative algorithm, which will test possible kernel configurations and frequency ranges, in order to find the optimal combination of execution parameters. A possible approach to take, when searching for the most adequate configuration, is to explore the full range of optimization parameters. However, in most cases this approach would be a very time consuming and thus practically unfeasible, even at the level of a single GPU kernel, as it was previously stated on Section 2.4. Furthermore, as GPU architectures evolve, this problem is even expected to keep increasing, due to increased support for more threads per block (which has been increasing from one GPU microarchitecture to the next), and with the expected increase in the number of possible frequency levels for the memory and graphics subsystems (as it happens with modern CPU architectures).

In order to overcome this issue, it is important to create a mechanism to limit the number of combinations of parameters that require testing. For this reason, the proposed procedures for automatic execution tuning consider only a subset of all possible GPU kernel configurations, i.e., the configurations that maximize the utilization of GPU resources. This decision is motivated by the fact that, on highly data-parallel GPU architectures, good performance is achieved when the total amount of kernel work is distributed such that all SMs are kept busy [1].

In order to apply each procedure, information about the GPU device and application kernels is required. This is performed during an initial profiling stage that consists on querying both GPU device and application kernels. Accordingly, from the kernels, the search space reduction step requires knowing how many registers are used per thread (kerRegsT), and how much shared memory is required (kerSharedMemTB). Both this values can be obtained from the kernel at compile time. Additionally, the search space reduction step also needs to know what is the defined default value for the number of threads per thread block (kerT_TB), in the source file. Unfortunately, this value cannot be obtained at compile time, and must therefore be passed by the kernel programmer. This value must be known during the procedure, because it corresponds to the number of threads associated with the amount of shared memory allocated, given by the compiler and with both values, the procedure is able to extrapolate the amount of shared memory used in each of the kernel configurations that require being tested.

From the GPU device, the search space reduction step requires knowing the limits intrinsic to the device, such as: the maximum number of thread blocks per SM (devMaxTB_SM), the maximum number of threads per SM (devMaxThreads_SM), the maximum number of warps per SM (devMaxW_SM), the size of the warps (devWarpSize), the amount of shared memory per SM (devMaxSharedMemSM), the number of registers per SM (devRegsLimit_SM), and the allocation unit size for the shared memory (devSharedMemAllocUnit) and for registers (devRegAllocUnit).

Table 3.1 summarizes the set of architecture-specific and kernel-specific parameters mentioned, as well as the architecture- and kernel-specific information, that are obtained when combining the information relative to the application kernel and GPU device, and which are also used during search space reduction.

Regarding the identification of a full set of viable kernel configurations, performed during the search
Table 3.1: Architecture and kernel-specific parameters.

<table>
<thead>
<tr>
<th>Type</th>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture-specific</td>
<td>devMaxTB&lt;sub&gt;SM&lt;/sub&gt;</td>
<td>max. number of thread blocks per SM</td>
</tr>
<tr>
<td></td>
<td>devMaxW&lt;sub&gt;SM&lt;/sub&gt;</td>
<td>max. number of warps per SM</td>
</tr>
<tr>
<td></td>
<td>devMaxSharedMem&lt;sub&gt;SM&lt;/sub&gt;</td>
<td>shared memory per SM</td>
</tr>
<tr>
<td></td>
<td>devSharedMemAllocUnit</td>
<td>shared memory allocation unit size</td>
</tr>
<tr>
<td></td>
<td>devRegsLimit&lt;sub&gt;SM&lt;/sub&gt;</td>
<td>max. number of registers per SM</td>
</tr>
<tr>
<td></td>
<td>devRegAllocUnit</td>
<td>registers allocation unit size</td>
</tr>
<tr>
<td></td>
<td>devWarpSize</td>
<td>warp size</td>
</tr>
<tr>
<td></td>
<td>devMaxThreads&lt;sub&gt;SM&lt;/sub&gt;</td>
<td>max. number of threads per SM</td>
</tr>
<tr>
<td>Kernel-specific</td>
<td>kerSharedMem&lt;sub&gt;TB&lt;/sub&gt;</td>
<td>shared memory per block</td>
</tr>
<tr>
<td></td>
<td>kerReg&lt;sub&gt;T&lt;/sub&gt;</td>
<td>number of registers per thread</td>
</tr>
<tr>
<td></td>
<td>kerT&lt;sub&gt;TB&lt;/sub&gt;</td>
<td>number of threads per thread block</td>
</tr>
<tr>
<td>Architecture- and kernel-specific</td>
<td>W&lt;sub&gt;TB&lt;/sub&gt;</td>
<td>number of warps per thread block</td>
</tr>
<tr>
<td></td>
<td>allocSharedMem&lt;sub&gt;TB&lt;/sub&gt;</td>
<td>shared memory allocated per thread block</td>
</tr>
<tr>
<td></td>
<td>allocReg&lt;sub&gt;W&lt;/sub&gt;</td>
<td>kernels allocated per warp</td>
</tr>
<tr>
<td></td>
<td>TB&lt;sub&gt;lim1&lt;/sub&gt;, TB&lt;sub&gt;lim2&lt;/sub&gt;, TB&lt;sub&gt;lim3&lt;/sub&gt;</td>
<td>thread block limits</td>
</tr>
<tr>
<td></td>
<td>maxTB&lt;sub&gt;SM&lt;/sub&gt;</td>
<td>max. number of thread blocks</td>
</tr>
<tr>
<td></td>
<td>occupancy</td>
<td>hardware utilization measure</td>
</tr>
</tbody>
</table>

In space reduction stage (see Figure 3.1), it is firstly required to determine the full range of possible distributions of threads within a single thread block. This range is constituted by all the thread block configurations that do not violate any of the GPU devices intrinsic limits. Afterwards, in order to reduce the design optimization space, only the thread block distributions that suffice the feasibility criteria will be further considered.

In order to verify the referred feasibility of each kernel configuration, the maximum number of thread blocks that can be realistically assigned per SM (maxTB<sub>SM</sub>) is calculated by considering three main limiting factors, namely: 1) the maximum number of thread blocks per SM that is sustained by the device; 2) the register availability; and 3) the amount of shared memory. For each of these limiting factors, a distinct value for the maximum number of thread blocks per SM is calculated, i.e., TB<sub>lim1</sub>, TB<sub>lim2</sub> and TB<sub>lim3</sub>. Then, the minimum among these values is assigned as maxTB<sub>SM</sub>, such that:

\[
\text{maxTB}_{SM} = \min\{\text{TB}_{lim1}, \text{TB}_{lim2}, \text{TB}_{lim3}\}. \tag{3.1}
\]

For each considered kernel, TB<sub>lim1</sub> represents the maximum number of thread blocks per SM that does not violate the maximum number of thread blocks per SM (devMaxTB<sub>SM</sub>), and the maximum number of warps per SM (devMaxW<sub>SM</sub>):

\[
\text{TB}_{lim1} = \min\left\{ \text{devMaxTB}_{SM}, \left\lceil \frac{\text{devMaxW}_{SM}}{W_{TB}} \right\rceil \right\}, \tag{3.2}
\]

where \( W_{TB} \) represents the number of warps per thread block for the currently considered distribution, obtained using:

\[
W_{TB} = \frac{k\text{er}T_{TB}}{\text{devWarpSize}}. \tag{3.3}
\]
$TB_{lim2}$ is the maximum number of thread blocks that does not violate the shared memory constraints, computed as:

$$TB_{lim2} = \left\lfloor \frac{devMaxSharedMem_{SM}}{allocSharedMem_{TB}} \right\rfloor.$$  \hspace{1cm} (3.4)

The $allocSharedMem_{TB}$ is the amount of shared memory allocated for the kernel per thread block, which can be calculated by considering the amount of shared memory needed by the kernel, given at compile time ($kerSharedMem_{TB}$), and the shared memory allocation unit size ($devSharedMemAllocUnit$) according to the following equation:

$$allocSharedMem_{TB} = \left\lceil \frac{kerSharedMem_{TB}}{devSharedMemAllocUnit} \right\rceil.$$ \hspace{1cm} (3.5)

The $TB_{lim3}$ value represents the upper bound on the number of thread blocks, that is constrained by the number of registers in the SM. This value is calculated as:

$$TB_{lim3} = \left\lfloor \frac{devRegsLimit_{SM}}{allocRegs_{W}} \right\rfloor / W_{TB},$$ \hspace{1cm} (3.6)

where $devRegsLimit_{SM}$ is the device maximum number of registers per SM, $W_{TB}$ is the number of warps per thread given by Equation (3.3) and $allocRegs_{W}$ is the number of registers used by the kernel per warp, calculated the following way:

$$allocRegs_{W} = \left\lceil \frac{kerRegs_{T} \times devWarpSize}{devRegAllocUnit} \right\rceil.$$ \hspace{1cm} (3.7)

The $kerRegs_{T}$ value is the number of registers used per thread, obtained at compile time, $devWarpSize$ is the warp size of the current device and $devRegAllocUnit$ is the registers allocation unit size of the device.

Finally, to reduce the total optimization space of kernel configurations, the device occupancy is used as the relative measure of GPU utilization \cite{4}. In detail, once the $maxTB_{SM}$ value is determined, the theoretical GPU occupancy for the current kernel configuration/distribution is computed as:

$$occupancy = \frac{maxTB_{SM} \times W_{TB} \times devWarpSize}{devMaxThreads_{SM}}.$$ \hspace{1cm} (3.8)

This measure represents the percentage of the GPU hardware ability to process the active warps \cite{4}. As soon as the theoretical occupancy on a per distribution basis is calculated, only the distributions with higher occupancy are selected (e.g., occupancy $>$ 80%), as they provide the possibility of attaining the best performance, i.e., the minimum execution time. A similar approach to calculate the occupancy for each distribution that can also be adopted is the usage of the CUDA occupancy calculator \cite{4}.

It should however be noted that it is impossible to predict the exact kernel occupancy without executing it, since it depends on specific static and dynamic device characteristics, which are not even made public by GPU manufacturers. As an example, in many cases it is impossible to completely hide the latency of certain instructions (e.g., load instructions from global memory), leading to a decrease in actual GPU occupancy \cite{24}.

### 3.2 Single-kernel optimization

By taking into consideration the above described search space reduction approach, iterative optimization procedures will now be derived, considering single-kernel execution environments. Accordingly,
two procedures are derived, namely: i) performance optimization, where the kernel parameters are derived such as to minimize execution time; and ii) energy-aware optimization, which also considers GPU operating frequency as an energy minimization parameter.

Whenever kernels with significant amount of work to be distributed are considered, the proposed procedure will always converge with a significantly faster rate than the exhaustive search, due to the reduced (and finite) number of possible configurations. In addition, by iteratively exploiting the configurations from such a reduced optimization space, the proposed procedure allows the GPU developers to perform fast early stage evaluations of the developed codes (e.g., when evaluating how different kernel configurations affect the attainable execution time and how they fit to the underlying GPU hardware). In fact, the conducted performance evaluations can provide valuable insights when: i) detecting the possible execution bottlenecks; ii) obtaining further optimization guidelines; and iii) analysing the portability of the developed codes across different GPU architectures. This is especially relevant to enable the porting of kernels optimized for a given architecture to different generations of GPU architectures (e.g., NVIDIA Tesla, Fermi and Kepler, or future architectures) and/or across GPU devices with different capabilities from the same architecture (e.g., NVIDIA GTX680 vs. Tesla K40c).

3.2.1 Optimizing for performance

Current trends in GPU kernel optimization mainly consider the performance of the developed codes, i.e., the optimization process is guided such that the minimum execution time is attained. In order to ease

![Flow-chart diagram of single-kernel performance auto-tuning procedure.](image-url)
Algorithm 3.1 Single-Kernel Performance Auto-Tuning Procedure.

Input: set of architecture-specific and kernel-specific parameters.
Output: configuration (d) corresponding to the minimum execution time.

1: for all distributions of the number of threads in a block do
2: Calculate maxTB_{SM} and theoretical occupancy;
3: end for
4: Store distributions with high occupancy in vector D;
5: Sort D in non-decreasing order of occupancy (and thread block size);
6: if length(D) > 1 then
7: for i_d := 0 to length(D) do
8: Run kernel with distribution D[i_d] and measure run time T[i_d];
9: if (i_d > 0) ∧ (T[i_d] > T[i_d−1]) then
10: return distribution d=D[i_d−1];
11: end if
12: end for
13: end if
14: return distribution d=D[length(D)−1];

this process, the procedure proposed herein aims at automatic tuning of kernel configuration parameters, and its main functionality is outlined in Figure 3.2. When optimizing for performance, the frequency level is set to its highest possible level, since that is that level that will achieve the smallest execution time.

Based on the previously referred set of architecture- and kernel-specific parameters (see Table 3.1), the proposed approach determines the configuration (d) that minimizes the kernel execution time. For this, the proposed procedure starts with a reduction of the overall optimization space, by selecting a subset of configurations with high device occupancy (one possible way to implement it, is by considering the distributions that result in occupancies above 80%), according to the considerations (see also steps 1–5 in Algorithm 3.1) referred in Section 3.1. Accordingly, given the existence of n possible distributions \( \hat{d}_1, \ldots, \hat{d}_n \), with occupancy \( \hat{o}_1, \ldots, \hat{o}_n \), the set of distributions with occupancy greater or equal to max \( \{\hat{o}_1, \ldots, \hat{o}_n\} - \varepsilon \) is selected, where \( \varepsilon \) is left as an optimization parameter. The subset of considered distributions is then stored in a vector \( D \) which is sorted in a non-decreasing order, based on the occupancy of the configurations. If several distributions result in the same occupancy, they are subsequently sorted in decreasing order of the number of threads per thread block.

Whenever several different distributions need to be examined (steps 6–13 in Algorithm 3.1), the proposed procedure first selects the distribution with the best potential to provide good performance \( (D[0], \text{corresponding to the distribution with higher occupancy}) \). Afterwards, the application kernel is launched using that configuration and the respective execution time \( (T[0]) \) is recorded. Then, the next configuration from the vector of possible distributions is evaluated. After each examined configuration \( (D[i_d]) \), the currently recorded execution time \( (T[i_d]) \) is compared with the execution time obtained for the previously considered configuration \( (T[i_d−1]) \). As long as the currently obtained time results in a reduction of the execution time over the previously examined configuration, the proposed procedure continues with the auto-tuning by selecting the next configuration from vector D.

Once the currently examined configuration does not allow any further reduction of the execution
time, the auto-tuning procedure is finalized and the last examined configuration is marked as the one with the best discovered performance (see step 10 in Algorithm 3.1). As expected, in case the reduced optimization space contains only a single distribution, that configuration is marked as the best one (see step 14 in Algorithm 3.1).

Figure 3.3 presents an example of the evolution of the execution time over each iteration, when Algorithm 3.1 is executed. It can be seen that at each iteration, a different distribution for the application kernel is tested, resulting also in distinct values for the execution time. In this case, the output of the algorithm would be the last tested distribution (128), since that is the one with the lowest execution time. It is worth noting that the algorithm stops at iteration 4, because the next distribution to be tested (64 threads) would no longer achieve a high occupancy.

![Figure 3.3: Example of the proposed execution of Algorithm 3.1.](image)

### 3.2.2 Optimizing for energy

Although in the performance optimization, the frequency level could be considered a constant, when optimizing for energy-awareness that is not the case. In fact, energy-efficiency ($EE$) can be seen as the inverse of energy consumption ($E$), $EE = 1/E$ \[21\]. Since the latter is given by the product between average power consumption ($P$) and execution time ($t$) (or, equivalently, by the integration of the power consumption over the execution time), the energy-efficiency is simply written as:

$$EE = \frac{1}{Time \times AvgPower}.$$

(3.9)

Given that the execution time of a kernel is a property of the kernel, such as type number of instructions ($I$), and of the architecture, specifically on how the kernel maps onto the architecture, resulting in kernel-specific values for the number of Instructions Per Cycle ($IPC$), one can rewrite energy-efficiency, given in Joules per Instruction, as:

$$EE = \frac{1}{Time \times AvgPower} = \frac{f \times IPC}{Power/I},$$

(3.10)

where $f$ stands for the GPU operating frequency. The result is a non-trivial optimization function that not only has a non-linear dependency with $f$ (since the power consumption also depends on the operating frequency), but also largely depends on the kernel launch environment.
Accordingly, energy-aware optimizations requires dealing with a significantly larger design optimization space than when optimizing for performance, since besides the usual range of GPU kernel configurations, different frequency ranges at which the modern GPU devices can operate must also be taken into account. As a result, the optimization space for energy-efficiency represents a combination of the performance exploration space with the selection of the most suitable operating frequency. As it was previously stated (see Section 2.4), only the optimization of the graphics subsystem operating frequency is going to be considered, since the memory subsystem of the available GPU devices only supports one non-idle frequency level. Although, such a study could be considered by using state-of-art simulators (e.g., GPGPU-sim [19] and GPUWattch[30]), this is left for future work.

In order to ease the optimization process, a novel procedure for energy-efficiency auto-tuning is proposed herein, as outlined in Figure 3.4. This procedure allows the application programmer to obtain useful insights on the most suitable GPU kernel configuration in terms of energy-awareness, i.e., it determines the configuration \((d)\) and frequency level \((f)\) that allow high utilization of GPU resources with the minimum energy consumption.

As seen in Figure 3.4, the main body of the procedure can be divided into two parts, namely one for frequency optimization and another for kernel configuration optimization, which work in an alternate fashion in order to minimize the energy consumption. A detailed version of the algorithm is presented in Algorithm 3.2. Similarly to the performance auto-tuning, the proposed procedure starts by performing the previously referred reduction of the optimization space and by creating the ordered \(D\) vector with a set of configuration candidates (see steps 1–5 in Algorithm 3.2). In addition to having vector \(D\), now the algorithm also uses vector \(F\), with the allowed frequency levels for the current GPU device, ordered in non-decreasing order (see step 6).

Figure 3.4: Flow-chart diagram of the proposed single-kernel energy auto-tuning procedure.

Input: set of architecture-specific and kernel-specific parameters.
Input: $F$ vector with GPU core frequency levels.
Output: configuration $(d)$ corresponding to the minimum execution time.
Output: frequency $(f)$ corresponding to the minimum energy consumption.

1: for all distributions of the number of threads in a block do  
2: Calculate maxTB$_{SM}$ and theoretical occupancy;  
3: end for  
4: Store distributions with high occupancy in vector $D$;  
5: Sort $D$ in non-decreasing order of occupancy (and thread block size);  
6: Sort $F$ in non-decreasing order of frequency levels;  
7: Set $i_f=0, d=0, f′=−1, f_{\min}=F[length(F)];$  
8: Set $f=−1;$  
9: while $(i_f<length(F)) \& (f′=−1)$ do  
10: Set core frequency at $F[i_f]$ and run kernel with $D[d];$  
11: Measure execution time $T[d, i_f]$ and energy consumption $E[d, i_f];$  
12: if $(i_f>0) \& (E[d, i_f]>E[d, i_f−1])$ then $f=F[i_f−1];$ else $i_f=i_f+1;$  
13: end while  
14: if $(f′=−1)$ then $f=f_{\min};$  
15: if $(f′=f′)$ then goto step 29;  
16: Set core frequency at $f$ and $f′=f;$  
17: if $(length(D)>1)$ then  
18: for $i_d=0$ to length$(D)$ do  
19: Run kernel with $D[i_d];$ measure time $T[i_d, f]$ and energy $E[i_d, f];$  
20: if $(i_d>0) \& (E[i_d, f]>E[i_d−1, f])$ then  
21: if $(d\neq i_d−1) \& (f\neq f_{\min})$ then  
22: $d=i_d−1; \text{ goto step 8; }$  
23: else  
24: return distribution $d=D[i_d−1]$ and frequency $f;$  
25: end if  
26: end if  
27: end for  
28: end if  
29: return distribution $d=D[length(D)−1]$ and frequency $f;$

After sorting vectors $D$ and $F$, the first configuration candidate from $D$ is selected (see step 7), and the first part of the algorithm begins, using the candidate to assess the variation in energy consumption over different GPU frequency levels. In detail, the proposed procedure aims at reducing the GPU operating frequency $F[i_f]$ (from the ordered frequency range in vector $F$), as long as the reduction in the energy consumption is attained for the currently examined candidate $D[d]$ (see steps 9–13 in Algorithm 3.2). Once executing at a lower frequency does not result in further reduction of energy consumption, i.e., the increased execution time does not compensate the reduction in power consumption, the previously determined frequency level is stored $(f=F[i_f−1]).$

Afterwards, the GPU operating frequency is set to that level and the algorithm proceeds to the second part, with the evaluation of configuration candidates from $D$ (see steps 17–29 in Algorithm 3.2). The evaluation of different configurations $D[i_d]$ is performed as long as they allow further reduction in energy consumption (see step 20). If the determined configuration $(d)$ with the minimum energy consumption differs from the one previously used when testing the frequency ranges, the proposed procedure proceeds with the evaluation across the remaining frequency ranges (see steps 8–13). When the found distribu-
tion \((d)\) matches the one previously tested for frequency ranges, the auto-tuning finishes and the found distribution \((d)\) and frequency level \((f)\) are returned (see steps 24 and 29).

Each time a new possible optimal frequency level is found it is checked if that value is different from the previous found frequency level (see step 15 in Algorithm 3.2). If the values are the same, it means the thread block configuration optimization (second part of the procedure) does not need to be performed, since it was already executed for that frequency level and therefore the procedure can terminate.

Figure 3.5 presents an example of the execution of Algorithm 3.2, where it can be seen the evolution of the energy consumption over each iteration of the algorithm. In it, it is visible the two distinct stages of the algorithm: \(i\) in iterations 1–3 only the frequency level changes, trying to find the best frequency level for the initial kernel distribution, which corresponds to the one with the highest number of threads per block. Once that value is found, the frequency is fixed, and the procedure tries to find the best kernel configuration, i.e., the thread block size (see iterations 4–6). Finally, when a new optimal distribution is found, the next frequency level is tested to check if the optimal frequency is still the same for the new best found distribution (see iteration 7).

3.3 Multi-kernel auto-tuning

Based on the contributions of the above-referred auto-tuning approaches at the single GPU kernel level, a new set of procedures for simultaneous and automatic tuning of multiple concurrent GPU kernels is proposed, considering both optimization goals, namely performance and energy-efficiency. In brief, the objectives of the proposed multi-kernel procedures is to automatically determine the most adequate configuration for each kernel in respect to the other simultaneously running kernels, i.e., a set of per-kernel configurations (and frequency level) that maximize performance or minimize energy consumption at the level of whole execution. As a result, the proposed approaches provide the means for run-time multi-kernel execution optimization even in the presence of shared resource contention, when several
kernels are simultaneously co-scheduled.

The main functionality behind the proposed approach for the performance domain is presented in Figure 3.6. Like the single-kernel procedure, it starts by creating a configuration vector, this time one for each of the kernels, containing their high occupancy configurations. Afterwards, in each iteration the procedure tries to optimize one of the kernels, the one with the slowest execution time, this way trying to reduce the total execution time, until no more optimizations are possible.

Algorithm 3.3 presents the detailed guidelines of the procedure. As stated before, similarly to the single-kernel approach, this procedure first reduces the optimization space and creates a set of ordered $D_i$ vectors with configuration candidates for each considered kernel $i$ (see steps 1–7 in Algorithm 3.3). Afterwards, the proposed procedure selects the distributions with the best potential to provide good performance for each of the considered kernels ($D_i[0]$), marking such distribution as the preliminary candidate distribution ($\delta_i$). All kernels are then simultaneously launched with $\delta_i$ distributions and the execution time is recorded ($T_i[0]$). Based on this preliminary evaluation, vector $K$ is created with the kernel indexes sorted in decreasing order of the obtained execution time (see steps 8–9 in Algorithm 3.3).

In steps 10–22, the algorithm iteratively examines the individual configurations for each kernel from $K$ vector, such that the minimum overall execution time is attained. In detail, in order to determine the best configuration ($d_i$) for each currently examined kernel, the proposed procedure behaves in a similar way as the previous described Algorithm 5.1 for single-kernel performance auto-tuning. For one kernel at a time, the procedure will try to find the optimal kernel configuration ($d_i$), by running the kernel with different distributions until the one that results in the best execution time is found (see steps 11–20 in Algorithm 3.3).

Whenever the obtained distribution ($d_i$) differs from the $\delta_i$ distribution, the candidate $\delta_i$ distribution takes the value of the currently determined one (see step 21) and the iterative procedure is restarted (jump back to step 10 with $i = 0$). This is done, because a different distribution in one kernel may cause changes in the execution of previously analysed kernels. This way, when a different distribution is found for a certain kernel, the procedure makes sure to re-check all kernels, by running them concurrently with the kernel in question, using the new found distribution. The proposed procedure stops when all the candidate distributions match the currently determined ones.

As it can be observed, this procedure is based on the fact that the minimization of the overall execu-
Algorithm 3.3 Multi-Kernel Performance Auto-Tuning Procedure.

Input: architecture-specific and kernel-specific parameters for each kernel.
Output: configurations \(d_i\) for each kernel \(i\).

1: for \(i=0\) to number of kernels do
2: for all distributions of the number of threads in a block do
3: Calculate maxTB\(_{SM}\) and theoretical occupancy;
4: end for
5: Store distributions with high occupancy in vector \(D_i\);
6: Sort \(D_i\) in non-decreasing order of occupancy (and thread block size);
7: end for
8: Run all kernels with distribution \(\delta_i=D_i[0]\) and measure run time \(T_i[0]\);
9: Create vector \(K\) with kernel indexes sorted in decreasing execution time;
10: for \(i=0\) to number of kernels do
11: if length\((D_i)\) \(>1\) then
12: for \(i_d=0\) to length\((D_i)\) do
13: Run kernels with distribution \(D_i[i_d]\) and measure total run time \(T[i_d]\);
14: if \((i_d>0)\land(T[i_d]>T[i_d-1])\) then
15: \(d_i=D_i[i_d-1]\);
16: goto step 21;
17: end if
18: end for
19: end if
20: \(d_i=D_i[\text{length}(D_i)-1]\);
21: if \((i>0)\land(d_i\neq\delta_i)\) then \(\delta_i=d_i\); \(i=0\);
22: end for
23: return distributions \(d_i\);

The execution time for several simultaneously running kernels can only be achieved if the minimum execution time is attained at the level of each individual kernel (when simultaneously co-scheduled with the other kernels). Furthermore, this procedure also adheres to the current parallel execution paradigm adopted in GPU architectures. For example, for several simultaneously launched applications, the GPU block scheduler will always try to satisfy the demands of the first application in terms of the number of SMs to be allocated (for the thread blocks). In fact, this allocation is usually performed in such a way that the thread blocks from a single application are spread as much as possible on the available SMs. As a result, the concurrent execution of the remaining applications highly depends on the availability of the GPU resources upon their reservation for previously launched kernels.

Still in the multi-kernel scenario, a second procedure was developed, targeting the energy-aware auto-tuning of multiple concurrent kernels, as presented in Figure 3.7. Similarly to the single-kernel algorithm, this procedure has two different optimizations stages, where the first tries to find the best frequency level for the whole set of kernels, in terms of the consumed energy. After finding the best frequency level, the procedure will try to optimize the kernel’s configurations, optimizing one in each iteration. When no more optimizations to the kernels are possible, the procedure will check if a different frequency level will result in an overall better consumed energy, by going back to the first stage. Only when there are no more possible frequency and distributions optimizations, does the procedure end.

It is worth mentioning that in an ideal situation the energy-aware procedure would give as output a
frequency level associated with each of the concurrent kernels. However, that would require the procedure
to be able to accurately predict exactly when each of the kernels would be executing on the GPU device,
which is currently still not possible. In addition to that, in current GPU architectures, the core frequency
can only be altered at the level of the whole GPU chip. For this reasons, the proposed procedure for
energy-aware optimization will only return a single frequency level that is the optimal frequency level
found for the whole set of concurrent kernels. As future work, it is envisaged the study of such situation
by taking advantage of state-of-art simulators such as GPGPU-sim [19] and GPUWattch [30].

Algorithm 3.4 presents the detailed functionality of the proposed procedure for simultaneous opti-
mization of several GPU kernels, such that the energy consumption is minimized at the level of the whole
execution. This procedure determines a set of per-kernel configurations \(d_i\) and a single frequency level
\(f\) for all currently running kernels. Once again, the proposed procedure relies on reducing the optimi-
mization space to create a set of ordered \(D_i\) vectors with configuration candidates for each considered
kernel \(i\) (see steps 1–7 in Algorithm 3.4). Then, the first configuration candidates from each \(D_i\) set are
selected and simultaneously run for different GPU frequency levels (see steps 8–17). Similarly to the
single-kernel procedure (see Algorithm 3.2), this process of reducing the operating frequency (from the
ordered \(F\) set) is performed as long as the overall energy consumption (for all kernels) is reducing. Once
lowering down the frequency does not provide any reduction of the energy consumption, the previously
determined frequency level is stored \(f=F[i_f-1]\).

Afterwards, the GPU core frequency is set to \(f\), the previously examined distributions are marked as
candidate \(\delta_i=d_i\) and vector \(K\) is created with the kernel indexes sorted in the decreasing order of energy
consumption (see steps 19–20 in Algorithm 3.4). Then, the individual configurations for each kernel from
the \(K\) vector are iteratively examined (see steps 21–30). The procedure starts by selecting the kernel
with the highest energy consumption, and tries to optimize it in terms of energy-awareness, by finding
the kernel distribution that results in the lowest overall energy-consumption (see steps 22–27). Then,
the same is done to the kernel with the second highest energy consumption, and so on, until all kernels

---

**Figure 3.7: Flow-chart diagram of the proposed multi-kernel energy auto-tuning procedure.**
The experimental results in Chapter 5. The proposed procedure stops when all preliminary distributions kernels, the optimal energy-consumption may be achieved at a different frequency level, as it is shown in step, i.e., step 10 in Algorithm 3.4. This is done because, with different distributions for the application optimization stage (see step 31). If that is the case the procedure is restarted from the frequency selection current kernel distributions is different from the preliminary distributions (∆
frequency level).

2, with

\( d_i \) with the value of the new determined one (\( d_i = d_i' \)) and run all kernels with \( D_i[d_i] \); 3. Measure total execution time \( T[i_f] \) and energy consumption \( E[i_f] \); 4. if \(( i_f > 0 ) \land ( E[i_f] > E[i_f-1]) \) then \( f = F[i_f-1] \); goto step 19; 5. else \( i_f = i_f + 1 \); end while

\begin{algorithm}
\begin{algorithmic}
\State \textbf{Input:} architecture-specific and kernel-specific parameters for each kernel.
\State \textbf{Input:} vector \( F \) with GPU operating frequency levels.
\State \textbf{Output:} configuration \(( d_i )\) for each kernel \( i \).
\State \textbf{Output:} frequency \(( f )\) corresponding to the minimum energy consumption.
\EndFor
\State \textbf{end for}
\EndFor
\State \textbf{end for}
\State \textbf{end for}
\State \textbf{end for}
\State \textbf{end for}
\State \textbf{end while}
\State \textbf{if} \( f = -1 \) then \( f = f_{\min} \);
\State \textbf{if} \( f = f' \) then goto step 32;
\State Set core frequency at \( f' = f \), \( \Delta_i = d_i \) and \( \delta_i = d_i \);
\State Create \( K \), with kernel indexes sorted in decreasing energy consumption;
\For \( i = 0 \) to number of kernels \( d_i \) to \( length(D_i) \) do
\State Run all kernels with \( D[i_d] \) and measure total energy \( E[i_d,f] \);
\State if \(( i_d > 0 ) \land ( E[i_d,f] > E[i_d-1,f] ) \) then \( d_i = i_d - 1 \); goto step 29;
\EndFor
\State \textbf{end if}
\State \textbf{if} \(( i > 0 ) \land ( d_i \neq \delta_i ) \) then \( \delta_i = d_i \); \( i = 0 \); goto step 21;
\EndFor
\State \textbf{return} distributions \( d_i = D_i[d_i] \) and frequency \( f \);
\end{algorithmic}
\end{algorithm}
are optimized. For each kernel currently being optimized, the best configuration is found by relying on an algorithm similar to the one used for the single-kernel energy-aware procedure (Algorithm 3.2). Whenever the determined distribution \(( d_i )\) differs from the candidate \( \delta_i \), the \( \delta_i \) distribution is assigned with the value of the new determined one \(( \delta_i = d_i \) \) (see step 29), and the procedure jumps back to step 21, with \( i = 0 \), in order to re-check all kernels with the new found distribution for kernel \( i \) (with the same frequency level).

After all kernels are optimized for the current frequency level, the procedure checks if any of the current kernel distributions is different from the preliminary distributions \(( \Delta_i \)\), used in the frequency optimization stage (see step 31). If that is the case the procedure is restarted from the frequency selection step, i.e., step 10 in Algorithm 3.4. This is done because, with different distributions for the application kernels, the optimal energy-consumption may be achieved at a different frequency level, as it is shown in the experimental results in Chapter 5. The proposed procedure stops when all preliminary distributions
match the currently determined ones (see step 32).

3.4 Summary

Throughout this chapter, it was presented the methodology proposed in this dissertation, that allows for an automatic tuning of GPU kernels, by finding the best execution configuration, i.e., the application kernel thread block size, and the GPU operating frequency, aiming at certain optimization goal. Accordingly, four different procedures were presented, for different optimization goals and different GPU environments: i) single-kernel performance auto-tuning; ii) single-kernel energy-aware auto-tuning; iii) multi-kernel performance auto-tuning; and iv) multi-kernel energy-aware auto-tuning. These devised procedures are integrated into an automatic optimization tool which will be presented in Chapter 4.
4

Auto-tuning tool

Contents

4.1 Auto-tuning tool description ............................................. 40
  4.1.1 Requirements .......................................................... 41
  4.1.2 Profiling ............................................................... 41
  4.1.3 Adaptive search ....................................................... 43
  4.1.4 Performance and energy monitoring of application kernels .... 44
  4.1.5 Frequency level ....................................................... 44
  4.1.6 Filtering bad distributions ......................................... 45
  4.1.7 Integrating the tool ................................................ 46

4.2 Users guide ................................................................. 47
  4.2.1 Usage ................................................................. 47
  4.2.2 Output ............................................................... 48

4.3 Power measuring tool ..................................................... 49

4.4 Summary ........................................................................ 52
While Chapter 3 presents the methodology that allows the automatic tuning of GPU kernels, the current chapter presents the tools developed during the course of this thesis, and that correspond to an implementation of the algorithms proposed in the previous chapter. Although the developed implementation was designed for CUDA programs, optimizing kernels defined in *.cu source files, a different implementation could easily be extended to also support the optimization of OpenCL kernels.

The developed auto-tuning tool is a command-line application, which can be used by programmers in order to find the best execution configuration for the application kernels (in terms of performance or energy-consumption), by providing the source code to their applications. An overview of the tool is presented in Figure 4.1 where it can be seen that the tool launches the GPU application which then launches the GPU kernels to be tuned.

Accordingly, this chapter is divided in three sections with the following outline: Section 4.1 presents the description of the developed tool, with its requirements and features; Section 4.2 presents how the users are able to use the developed auto-tuning tool; and since the development of the energy-aware optimization tool requires measuring energy consumption, Section 4.3 presents a energy-monitoring tool that can also be used by third-party applications to measure power and energy consumption of GPU devices.

4.1 Auto-tuning tool description

This section presents a description of the mechanisms behind the developed auto-tuning tool, which implements the optimization procedures presented in Chapter 3. Figure 4.2 presents a layer diagram of the developed auto-tuning tool, illustrating the main APIs used by the tool to communicate with the GPU device, in order to implement the proposed procedures.

The application called by the user, runs the developed auto-tuning tool in order to tune the GPU kernels specified by the user. The auto-tuning tool uses the PAPI, NVML and CUDA APIs to retrieve specific informations from the CPU and GPU devices. When applying the single- or multi-kernel energy-aware procedures the auto-tuning tool will also use the developed power-measuring tool, presented in Section 4.3. The following subsections present the functioning of the developed auto-tuning tool.

![Figure 4.1: Overview of the developed Auto-tuning tool.](image-url)
4.1.1 Requirements

The developed auto-tuning tool, is a command-line application that can be used to find the best execution configurations (thread block configuration or GPU operating frequency) for the kernels of a given GPU application, targeting performance or energy-awareness.

Since the proposed procedures rely on the execution of the application kernels with different thread block distributions, the application provided in the source code given to the tool must be able to execute with different configurations. In particular, the application to be tuned, when executed, must be able to receive as arguments the values of the thread block size, which should be used when defining the thread block size used to launch the application kernels.

In the developed implementation, in order to tune the kernels in the multi-kernel scenario, all kernels must be defined in the same source file, or at least be in different files that when compiled generate one single application. This means the tool will not optimize 2 kernels, defined in 2 independent *.cu files. If one application has more than one source file, a PATH to a Makefile that compiles the files should be given to the auto-tuning tool, using a specific argument. The specific arguments needed to launch the auto-tuning tool will be described later in Section 4.2.1.

4.1.2 Profiling

As stated in Chapter 3, all of the proposed procedures start with an initial stage where the procedure gathers information about the current GPU device and the application kernels to be optimized. The profiling information related with the application kernels, such as number of registers per thread and shared memory per block (see Table 3.1), can be obtained at compile time. The auto-tuning tool accomplishes this by launching a process that will run the NVIDIA compiler (nvcc) for the given source files, using flag \texttt{-Xptxas=-v}, in order to get an output like the following:

```
# ptxas info : Compiling entry function 'FiniteDifferencesKernel'
# ptxas info : Used 27 registers, 3884 bytes smem, 24 bytes cmem[1]
```
As it can be seen, the compiler will return the number of registers per thread and the amount of shared memory per thread block, for each of the kernels that are declared inside the given .cu file. In CUDA there are two ways to allocate shared memory for a GPU kernel. It may be done statically or dynamically, depending on whether the size of the memory to be allocated is known or unknown at compile-time. When allocated dynamically, the size of the shared memory to be used, is only specified within the function call used to launch the kernel. This is a problem, because when the flag `-Xptxas=-v` is used with the nvcc compiler, the compile output only considers the statically allocated shared memory, since it cannot know the size of the dynamically allocated one. To overcome such an issue, the programmer can also specify the size of the dynamically allocated shared memory, as an argument to the auto-tuning tool.

In addition to these values, in order to run the procedures, the tool will also need information regarding the number of threads per thread block, used by default in the *.cu file, which must also be provided by the programmer. With this value, the tool knows how many threads correspond to the amount of shared memory given by the compiler and, as a consequence, the tool can then extrapolate the amount of shared memory used in each of the distributions which is useful during the search space reduction.

It is also important to notice that the developed auto-tuning tool aims at kernels where the used shared memory is always proportional to the thread block size, which is the case for most of the scientific applications designed to run on GPU devices. However, if for some reason there was a need to automatically tune a kernel where the amount of shared memory used was constant, regardless of the thread block size, an additional launch argument could be added to the tool, which would signal that the application and the kernel had those special characteristics.

The information relative to the GPU device is retrieved using the NVML library. The tool will call the NVML function `cudaGetDeviceProperties()` which queries the GPU device, getting as return the device’s characteristics. Table 4.1 presents a summary of the information possible to obtain, when using the `cudaGetDeviceProperties()` function on NVIDIA’s Tesla K40c.

<table>
<thead>
<tr>
<th>Compute capability</th>
<th>3.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of SMs</td>
<td>15</td>
</tr>
<tr>
<td>Max. threads per block</td>
<td>1024</td>
</tr>
<tr>
<td>Max. threads per SM</td>
<td>2048</td>
</tr>
<tr>
<td>Max. registers per block on each SM</td>
<td>65536</td>
</tr>
</tbody>
</table>

However, there are some additional informations about the GPU device that the tool requires, but which cannot be directly retrieved from the mentioned NVML function. Nonetheless, most of the characteristics of a GPU device can be known by taking into account the value of its compute capability. Accordingly, the tool will look at the GPU device compute capability, in order to discover the remaining architecture characteristics, which can be seen in Figure 4.2. This was implemented by creating a configuration file with all compute capabilities up to the most recent 5.0. Therefore, to extend the tool for future GPU devices, only the configuration file needs to be updated, by adding the characteristics associated with the new compute capabilities.
### Table 4.2: Additional NVIDIA Tesla K40c Characteristics

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. shared memory per SM</td>
<td>49152 bytes</td>
</tr>
<tr>
<td>Shared memory allocation unit size</td>
<td>256 bytes</td>
</tr>
<tr>
<td>Max. thread blocks per SM</td>
<td>16</td>
</tr>
<tr>
<td>Max. registers per thread</td>
<td>255</td>
</tr>
<tr>
<td>Allocation granularity</td>
<td>warp</td>
</tr>
</tbody>
</table>

#### 4.1.3 Adaptive search

After retrieving the profiling information related with the application kernels and GPU device, the tool will reduce the optimization search space, by finding the best possible execution configurations. As mentioned during Chapter 3, this is accomplished by checking, for each distribution, if the resulting number of registers, amount of shared memory and number of SMs are within the device’s limits. If that is the case, the tool computes the value of the occupancy resulting from that distribution. The developed auto-tuning tool considers that the optimal kernel configurations always lie on power of 2 solutions. Even though, for good performance to be achieved, the thread block size should in fact be a multiple of the warp-size, in order to make sure that there are no threads in a warp executing unnecessarily, the decision to only consider power of 2 solutions was made to further reduce the number of iterations required to find the best kernel configurations.

After the occupancies for all distributions are computed, the tool further creates an array of distributions for each kernel, with the ones that achieve high occupancies. The set of high occupancy distributions

![Flow-chart diagram of the developed auto-tuning tool applying the proposed single-kernel energy auto-tuning procedure, with the APIs used at each stage.](image-url)
is composed by all the distributions \( d_i \) with Occupancy > \( \text{Max}_{\text{occupancy}} - \varepsilon \), where the value of \( \varepsilon \) can be set by the programmer (the value used during this thesis was always 20%).

Figure 4.3 presents a flowchart of the single-kernel energy-aware auto-tuning procedure, already presented before, using the auto-tuning tool with the information relative to the APIs that the tool will use in each stage of the procedure.

### 4.1.4 Performance and energy monitoring of application kernels

At each iteration, the tool launches a specific kernel configuration (previously compiled during the profiling stage), as detailed in Chapter 3. It is important to notice that this procedure can take place under different circumstances, such as for allowing the tuning of an application for a specific GPU device (see Figure 4.1) and also for integration on iterative applications, where the same kernel is called multiple times during a single execution of the whole program. In the latter case, described in Section 4.1.7, it always aims for an automatic on-line optimization of the multiple applications kernels without any intervention of the programmer or the user.

In either circumstances, for performance auto-tuning, the tool should allow measuring the execution time of each of the application kernels. However, since the tool cannot get direct access to that information, the solution found is to have the application record the duration of each of its own kernels, as well as the total execution duration for the whole set of kernels, and write the values in an output file. When the application has finished its execution, the auto-tuning tool will read the written file, in order to know all the execution times. This way, the tool knows after each iteration, which of the kernels is the slowest, necessary in order to apply any of the procedures described in the previous chapter. The execution time of the kernels was measured using the Performance Application Programming Interface (PAPI) API to interface with the host CPU Time Step Counter (TSC). An alternative solution would be to measure the execution time of the kernels using the Events module of the CUDA runtime API, which allows the measure of the elapsed time between two events.

In the multi-kernel energy-aware auto-tuning procedure (see Algorithm 3.4), it was stated that at each iteration it was measured the energy consumption of each separated kernel. However, this is impossible to be done, since the auto-tuning tool has no means to know exactly when each of the kernels is executing and it is even possible that the kernels execution overlap each other, meaning the tool would not know to which kernel a specific power reading level would be related to. To bypass this problem, the compromise made consists on measuring the average power over all the application kernels, and multiplying that value with the duration of each kernel, in order to get the energy consumption for each kernel. Since the average power over the whole execution is constant over all kernels, the ordering of the kernels in terms of energy consumption will be the same as the ordering in terms of execution time. The power readings are obtained using the developed power-measuring tool presented in Section 4.3.

### 4.1.5 Frequency level

The proposed procedures require the execution of the application kernels on the GPU at certain specific levels of the GPU operating frequency. This means the auto-tuning tool needs to have a way of
setting the GPU operating frequency to the desired levels, which is done by using the NVML library.

To set the GPU operating frequency at a specific level, the auto-tuning tool starts by calling the functions `nvmlDeviceGetSupportedMemoryClocks()` and `nvmlDeviceGetSupportedGraphicsClocks()`, in order to get the frequency ranges allowed by the current GPU device. Afterwards, when it is required to set the frequency level at a certain value, the function `nvmlDeviceSetApplicationsClocks()` is called using as arguments the values of the desired frequency levels for the memory and graphics systems. As previously stated (see Section 2.4), for the tested modern GPU architectures (NVIDIA Kepler), there are only two very distinct allowed frequency levels for the memory subsystem. As an example, in the used Tesla K40c the values are 3004 MHz and 324 MHz, which is the value used when the GPU is active or idle, respectively. For this reason, the memory subsystem frequency level will always be set to its highest allowed value.

When optimizing for performance, in both single- and multi-kernel environments, the auto-tuning tool only needs to set the frequency level once, at the beginning of the execution, since as it was previously mentioned, the optimal performance is reached at the maximum frequency level. However, when using the energy-aware procedures, the GPU frequency level will not always be constant, and in fact, the frequency level is one of the optimization variables. This way, when required, the auto-tuning tool uses the `nvmlDeviceSetApplicationsClocks()` function, allowing the consequent executions of the applications kernels to be made at the desired frequency level.

4.1.6 Filtering bad distributions

During the iterative search for the optimal kernel configurations, certain kernel configurations chosen for one iteration may result in a significant decrease in the performance of the whole set of kernels. To decrease the occurrence of these situations, the developed auto-tuning tool also implements a mechanism to filter out the distributions that significantly decrease the performance (or increase the energy consumption, when optimizing for energy-efficiency). Figure 4.4 presents the evolution of the algorithm when optimizing four concurrent kernels, namely 2 MatrixMul and 2 Lud, where it can be seen that in iteration 2, the change in the distribution of one kernel will result in a significant increase of the execution time (14% higher than the previous value).

![Figure 4.4: Multi-kernel energy-aware optimization procedure for 2 MatrixMul and 2 Lud kernels, executed on Tesla K40c.](image-url)
The auto-tuning tool marks the distribution of the kernel that led to the decrease in performance, and if in a future iteration the kernel should be launched with that same distribution, the tool knows that it can skip that step and proceed immediately to the next iteration, which will further decrease the total number of iterations required to find the best kernel configurations. Figure 4.5 presents the results obtained with the same GPU environment, when auto-tuning tool implements this mechanism.

Figure 4.5: Multi-kernel energy-aware optimization procedure for 2 MatrixMul and 2 Lud kernels, executed on Tesla K40c.

4.1.7 Integrating the tool

In Figure 4.1 was presented a way to use the auto-tuning tool, that consisted in launching the tool to tune a certain GPU application, until the best execution configuration is found. However, the developed auto-tuning tool, also supports being integrated in third-party GPU applications, where the same GPU kernel is called multiple times during a single execution of the whole program.

This alternative way of using the developed auto-tuning tool, as presented in Figure 4.6, is particularly
interesting to use with GPU applications that implement iterative algorithms, because it allows the tuning of the GPU kernels during the first few iterations of the algorithm, making sure that the majority of iterations would already be executing with the best found execution parameters. With this methodology at each iteration of the iterative algorithm executed on the GPU, the auto-tuning tool measures the execution time of the GPU kernels, and is able to provide an automatic on-line optimization of the multiple applications kernels without any intervention of the programmer or of the user.

Integrating the tool can provide significant performance (or energy-savings) benefits, assuming that the algorithm implemented in the GPU application executes for more iterations than the number of steps required by the auto-tuning tool to find the best configuration parameters, which is not uncommon, since many GPU applications execute for thousands of iterations.

4.2 Users guide

This section presents the correct way for the users to use the developed auto-tuning tool, in order to find the best execution configurations for a specific GPU application. In Section 4.2.1 is presented the parameters that should be used as input to the tool, and in Section 4.2.2 are described the outputs provided by the tool during its execution.

4.2.1 Usage

In order to successfully use the developed auto-tuning tool, to optimize a specific set of application kernels, the user is required to provide some arguments. The first argument, is the number of kernels defined in the provided [CUDA] program. This value is needed, in order to facilitate the reading of some of the following arguments, which may require one value for each kernel.

Additionally, to execute the tool, it is also required to use as argument the name of the source file (along with the PATH to its location). Since this implementation was done considering [CUDA] kernels, the source code used as input of the tool, must be a file with type *.cu and, for this reason, the file type (*.cu) does not need to be provided in the file name.

The third argument to be used when calling the auto-tuning tool is the thread block size, used by default in the *.cu source files. The tool needs this value, because as previously stated, the procedures need to know what is the thread block size associated with the profiling results given by the compiler (number registers and amount of shared memory used). The value should be provided after the argument -tb. Given the fact that it is possible to use the tool to optimize programs with more than one GPU kernel, and since each kernel may have different default values for the thread block size, after the -tb argument it should be provided as many values for the thread block size, as the number of existent kernels.

Finally, the users of the auto-tuning tool, should also provide as final argument an index, indicating what is the type of tuning to be performed: i) 0 for single-kernel performance tuning; ii) 1 for single-kernel energy-aware tuning; iii) 2 for multi-kernel performance tuning; and iv) 3 for multi-kernel energy-aware tuning.

That being said, an example of a correct usage of the auto-tuning tool is the following:
With this command, the auto-tuning tool would be launched to optimize for performance, an application with source code in the file `kmeans.cu`, containing only one kernel, with default thread block size equal to 256.

If one application has more than one source file, a **PATH** to a **Makefile** that compiles the files may be given to the auto-tuning tool, using argument `-i makefile_path`. This argument is optional, and can be used as in the following examples:

```
# ./auto_tool 1 kmeans -i Makefile -tb 256 0 -- for single-kernel.
# ./auto_tool 3 three_multi -i Makefile -tb 256 512 64 0 -- for multi-kernel.
```

Finally, there is also another optional argument which may be used with the developed auto-tuning tool, which is `-mem`, used to specify the amount of dynamic shared memory used by each kernel. The value should be in bytes, and like the thread block size, it should be given a correspondent value for each of the kernels defined in the source program, as in the following example:

```
# ./auto_tool 2 kmeans_lud -i Makefile -tb 256 1024 -mem 0 3400 0
```

### 4.2.2 Output

Since the goal of the developed auto-tuning tool is to implement the procedures presented in Chapter 3, the tool will provide as output the best found execution configuration after the specified procedure is applied to the given GPU application.

During the execution of the optimization, the tool will also report back to the user, through the command-line, information related with the tasks finished. For example, after profiling the Telsa K20c device, the tool would print in the command-line the main characteristics associated with the device, as presented in Figure 4.7.

```
<table>
<thead>
<tr>
<th>Device properties:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute capability:</td>
</tr>
<tr>
<td>Number of SMs:</td>
</tr>
<tr>
<td>Max shared memory per block/SM:</td>
</tr>
<tr>
<td>Shared memory allocation unit size:</td>
</tr>
<tr>
<td>Max threads per block:</td>
</tr>
<tr>
<td>Max threads per SM:</td>
</tr>
<tr>
<td>Max thread blocks per SM:</td>
</tr>
<tr>
<td>Max registers per block/SM:</td>
</tr>
<tr>
<td>Max registers per thread:</td>
</tr>
<tr>
<td>Allocation granularity:</td>
</tr>
<tr>
<td>Can execute concurrent kernels:</td>
</tr>
</tbody>
</table>
```

Figure 4.7: Auto-tuning tool example output: device properties.

The developed tool is also able to notify the user, regarding the possible configurations for each of the kernels present in the *.cu file, and which ones achieve high occupancy. During the iterative stage of the procedure, when executing the application kernels, the auto-tuning tool will also print on the command-line the execution duration for each of the kernels, along with the total execution time for the
set of GPU kernels. After iteration 2 of the optimization of 4 concurrent kernels, namely $2 \times \text{MatrixMul}$, 1 $\text{VectorAdd}$ and 1 $\text{MatrixTrans}$, the output of Figure 4.8 would be printed.

**Iteration 2:**

Chosen distribution:
- Kernel 1: (32, 32, 1)
- Kernel 2: (32, 32, 1)
- Kernel 3: (32, 32, 1)
- Kernel 4: (32, 16, 1)

Duration:
- Kernel 1: 114 us
- Kernel 2: 6036 us
- Kernel 3: 5740 us
- Kernel 4: 194 us

Total execution time: 12184 us

Figure 4.8: Auto-tuning tool example output: iteration 2.

Finally, when no more optimizations to the execution parameters are possible, the auto-tuning tool will print in the command-line the best distributions for each optimized kernel (and the operating frequency when optimizing for energy-efficiency), as shown in the example of Figure 4.9 obtained when optimizing for performance the same four kernels mentioned before. Before terminating, the auto-tuning tool will also print the duration of the whole tuning procedure, along with the number of iterations that were required to find the best execution parameters.

No more optimizations possible.

Best distribution:
- Kernel 1: BlockSize (32, 16, 1)
- Kernel 2: BlockSize (32, 32, 1)
- Kernel 3: BlockSize (32, 32, 1)
- Kernel 4: BlockSize (32, 16, 1)

Execution time: 12070 us

Search duration: 5488.561 milliseconds
Number of iterations: 12

All done.
Press ENTER to continue...

Figure 4.9: Auto-tuning tool example output: best execution configuration found.

### 4.3 Power measuring tool

In addition to developing the auto-tuning tool, it was also developed a tool for measuring the power consumption of a GPU device, during a given time interval. This tool takes advantage of the NVML library, in order to have access to the power counters existent in some GPU devices. Although, it is used in the auto-tuning tool (for single- and multi-kernel energy-aware tuning), the developed power...
measuring tool can also be used independently, allowing programmers to easily have measures for the
power consumption of a GPU device, during the execution of a certain piece of code. The developed
power measuring tool takes into account previous work done by Burtscher et al. [20], by applying the
necessary adjustments to the power readings, in order to get the corrected power values values.

The tool was developed in a way to facilitate its use by third-party applications (CUDA applications). It can be used to measure the power and energy consumption of a GPU device, over a certain part of the
code, by calling two special functions right and after that part of the code. In Figure E.11 is the example
of the usage of the power measuring tool.

```
... powerToolStart();
... code to measure the power consumption ...
powerToolFinish(); ...
```

Figure 4.10: Example of usage of the power measuring tool.

With the function call `powerToolStart()`, the tool will launch a thread (on the CPU), which
will sample the power readings of the GPU using the NVML function `nvmlDeviceGetPowerUsage()`,
at constant time intervals of 16 ms (62.5 Hz), as suggested in [20]. Upon the second function call
(`powerToolFinish()`), the tool will signal the thread that it can exit the sampling loop, entering the
finalizing stage, where it will corrections to the power readings, as suggested in [20], according to the
following expression:

$$\hat{P}(t_i) = P_m(t_i) + C \frac{P_m(t_{i+1}) - P_m(t_{i-1})}{t_{i+1} - t_{i-1}}$$  \hspace{0.5cm} (4.1)

where $\hat{P}(t_i)$ is the corrected value, $t_i$ is the time of power sample $i$ and $C$ is a constant representing the GPU capacitance.

The developed tool is also able to determine energy consumption, by computing the integral of the
corrected power over the kernel execution time, which in discrete terms is represented as:

$$E = \sum_i (t_i - t_{i-1}) \hat{P}(t_i).$$  \hspace{0.5cm} (4.2)

The developed power measuring tool returns as output the average power consumption, as well as the
ergy consumed by the GPU device from the first power sample until the last. Additionally, the tool
will also write into two files, the values of all the power readings and the values of the corrected power,
respectively, together with the time instant they are associated with. These files, together with an Excel
file developed along with the tool, allow the programmer to plot the power consumption of the GPU
device over time. In Figure E.11 is presented an example of a power consumption plot, that resulted from
using the developed power measuring tool, allowing an easy understanding of the power consumption
over time, on the level of the GPU device.

The previous described behaviour of the developed power measuring tool, works by measuring the
power consumption between the two functions calls mentioned. If programmers want to measure the GPU
power consumption only during the execution of GPU kernels, calling the function `powerToolStart()`, right before launching the kernels, may not be a good solution because of its overhead cause by the required initializations that need to be done. To address this situations, the tool also allows a different working mode, as presented in Figure 4.12.

```
... powerToolStart(1);
...
powerToolKernelsStart();
...Launch GPU kernels...
...Wait until kernels finish...
powerToolKernelsFinish();
...
powerToolFinish();
...
```

Figure 4.12: Example 2 of usage of the power measuring tool.

This way, all the initializations of the power measuring tool are done before the GPU kernels execution. Furthermore, when the kernels are ready to be executed on the GPU, the application signals the power measuring tool, allowing it to start sampling the GPU power levels. It is worth mentioning that although, in example of Figure 4.12 only the power consumption during the kernels execution is being measured, the tool could be applied to a larger time-frame, making it possible to, for instance, also measure the consumption during the data transfers from Host(CPU) to Device(GPU), or from Device to Host, by changing simply the position of the `powerToolStart()` and `powerToolFinish()` function calls. In Figure 4.13 is presented an example of a chart possible to obtain when using the methodology from Figure 4.12.
4.4 Summary

In this chapter was presented an implementation of the procedures proposed in Chapter 3, developed for CUDA applications. The main features of this tuning tool used to tune GPU kernels, were described, as well as the decisions and compromises made during its development. Finally, it was also presented the developed power-measuring tool, which is used in the auto-tuning program, but may also be used by other third-party applications, allowing them to measure the power and energy consumption of GPU devices, and providing the tools to plot in a chart the evolution of the power consumption over time.
5

Experimental results

Contents

5.1 Experimental setup .......................................................... 54
5.2 Optimization considerations .................................................. 57
  5.2.1 Concurrent execution ................................................... 57
  5.2.2 Occupancy ................................................................ 58
5.3 Optimization results ............................................................. 60
  5.3.1 Single-kernel ............................................................... 60
  5.3.2 Multi-kernel ............................................................... 62
5.4 Summary ............................................................................. 70
In order to assess the quality of the performance and energy-aware optimization procedures proposed in Chapter 4 for single- and multi-kernel applications, a set of experiments were devised, on multiple GPUs using different kernels. Accordingly, this chapter starts by presenting the experimental setup in Section 5.1, namely the benchmarks used and the characteristics of the GPU devices where the tests were executed. Afterwards, Section 5.2 presents some interesting results related with the execution of GPU kernels on GPU devices, specifically the scheduling of concurrent kernels and the relation between occupancy and performance. Finally, Section 5.3 presents the results obtained when using the auto-tuning tool described in Chapter 4.

5.1 Experimental setup

To experimental show the benefits of the proposed single- and multi-kernel optimization procedures, a wide set of CUDA kernels from different sources, were executed on different GPU devices, from distinct microarchitectures. This section presents the application kernels chosen to test the proposed procedures, followed by the GPU devices where the experiments were executed.

The GPU applications chosen to evaluate the procedures were extracted from both the NVIDIA SDK and Rodinia Benchmarks suite, and are summarized in Table 5.1. The source codes extracted from the referred application suites were improved, since most of the applications did not provide support for changing the thread block size (the value was hardcoded in the application), which is a requirement of the developed auto-tuning tool.

<table>
<thead>
<tr>
<th>Application</th>
<th>Suite</th>
<th>Problem Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lud</td>
<td>Rodinia</td>
<td>4096×4096</td>
</tr>
<tr>
<td>Hotspot</td>
<td>Rodinia</td>
<td>8192</td>
</tr>
<tr>
<td>ParticleFilter</td>
<td>Rodinia</td>
<td>100000</td>
</tr>
<tr>
<td>Streamcluster</td>
<td>Rodinia</td>
<td>65536</td>
</tr>
<tr>
<td>K-Means</td>
<td>Rodinia</td>
<td>819200</td>
</tr>
<tr>
<td>S-Rad</td>
<td>Rodinia</td>
<td>2048×2048</td>
</tr>
<tr>
<td>BFS</td>
<td>Rodinia</td>
<td>100000</td>
</tr>
<tr>
<td>MatrixTrans</td>
<td>SDK</td>
<td>4096×4096</td>
</tr>
<tr>
<td>MatrixMul</td>
<td>SDK</td>
<td>4096×4096</td>
</tr>
<tr>
<td>VectorAdd</td>
<td>SDK</td>
<td>1000000</td>
</tr>
<tr>
<td>FDTD3D</td>
<td>SDK</td>
<td>373×376×376</td>
</tr>
</tbody>
</table>

The following is the description of the application benchmarks from the Rodinia suite:

- **Lud**: implements a LU Decomposition algorithm to calculate the solutions of a set of linear equations, by decomposing a matrix as the product of a lower triangular matrix and an upper triangular matrix.

- **Hotspot**: estimates processor temperature based on an architectural floorplan and simulated power measurements. The thermal simulation iteratively solves a series of differential equations for block,
where each output cell in the computational grid represents the average temperature value of the corresponding area of the chip.

- **ParticleFilter** is a statistical estimator of the location of a target object given noisy measurements of that target’s location and a rough estimation of the object’s path in a Bayesian framework. It can be used in a wide variety of applications ranging from video surveillance in the form of tracking vehicles, cells, and faces to video compression. This particular implementation is optimized for tracking cells, particularly leukocytes and myocardial cells. After selecting the target object, the application begins tracking it by making a series of guesses about the current frame given what is already known from the previous frame. The application then determines the likelihood of each of those guesses occurring using a predefined likelihood model. Afterwards, the ParticleFilter application normalizes those guesses based on their likelihoods and then sums the normalized guesses to determine the object’s current location. Finally, the application updates the guesses based on the current location of the object before repeating this process for all remaining frames in the video.

- **Streamcluster** is an application that, for a stream of input points, finds a predetermined number of medians so that each point is assigned to its nearest center. The quality of the clustering is measured by the sum of squared differences (SSQ) metric.

- **K-Means** is a widely known clustering algorithm used extensively in data-mining and in many other environments. Many data-mining algorithms show a high degree of data parallelism. In k-means, a data object is comprised of several values, called features. By dividing a cluster of data objects into K sub-clusters, k-means represents all the data objects by the mean values or centroids of their respective sub-clusters. The algorithm works as follows: The initial cluster center for each sub-cluster is randomly chosen or derived from some heuristic; afterwards, in each iteration, the algorithm associates each data object with its nearest center, based on some chosen distance metric; the new centroids are then calculated by taking the mean of all the data objects within each sub-cluster respectively; finally, the algorithm iterates until no data objects move from one sub-cluster to another.

- **S-Rad** Speckle Reducing Anisotropic Diffusion (SRAD) is a diffusion method for ultrasonic and radar imaging applications based on Partial Differential Equation (PDEs). This application is used to remove locally correlated noise, known as speckles, without destroying important image features. SRAD can be divided in several stages: image extraction, continuous iterations over the image (preparation, reduction, statistics, computation 1 and computation 2) and image compression. The sequential dependency between all of these stages requires synchronization after each stage (because each one operates on the entire image). In the CUDA version, each stage is a separate kernel (due to synchronization requirements) that operates on data already residing in GPU memory, featuring efficient GPU reduction of sums. In order to improve GPU performance data is transferred to GPU at the beginning of the code and then transferred back to CPU after all of the computation stages are completed on the GPU. Some of the kernels use GPU shared memory for additional improvement in performance.
• **BFS:** Breadth-first search (BFS) is a commonly used graph algorithm, used in many disciplines and application areas, typically requiring the processing of large graphs, thus involving the processing of millions of vertices. This application provides the GPU implementations of BFS algorithm which traverses all the connected components in a graph.

The remaining GPU applications were retrieved from the CUDA Samples included with the CUDA SDK and are the following:

• **MatrixTrans:** GPU application that calculates the transpose of an input matrix.

• **AddVector:** GPU application that implements a element-by-element vector addition.

• **MatrixMul:** GPU application that implements matrix multiplication, taking advantage of the shared memory, by blocking the computation.

• **FDTD3D:** GPU application that applies a finite differences in time domain progression stencil on a 3D surface. It consists in one of the high performance samples provided in the CUDA samples.

Automatic optimization of the application benchmarks was performed on a set of GPU devices of different characteristics, namely: GTX280 (Tesla architecture), GTX580 (Fermi architecture) and GTX680, Tesla K20c and Tesla K40c (Kepler architecture). The characteristics of these devices are summarized in Table 5.2.

<table>
<thead>
<tr>
<th></th>
<th>Tesla K40c</th>
<th>Tesla K20c</th>
<th>GTX680</th>
<th>GTX580</th>
<th>GTX280</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Base architecture</strong></td>
<td>Kepler</td>
<td>Kepler</td>
<td>Kepler</td>
<td>Fermi</td>
<td>Tesla</td>
</tr>
<tr>
<td><strong>Compute capability</strong></td>
<td>3.5</td>
<td>3.5</td>
<td>3.0</td>
<td>2.0</td>
<td>1.3</td>
</tr>
<tr>
<td># SMs</td>
<td>15</td>
<td>13</td>
<td>8</td>
<td>16</td>
<td>30</td>
</tr>
<tr>
<td># CUDA cores</td>
<td>2880</td>
<td>2496</td>
<td>1536</td>
<td>512</td>
<td>240</td>
</tr>
<tr>
<td>SM freq. ranges (MHz)</td>
<td>875, 810</td>
<td>758, 705, 666</td>
<td>1058</td>
<td>1594</td>
<td>1476</td>
</tr>
<tr>
<td>Memory freq. (MHz)</td>
<td>745, 666</td>
<td>640, 614</td>
<td>1594</td>
<td>1476</td>
<td>30</td>
</tr>
<tr>
<td># Registers / SM</td>
<td>64K</td>
<td>64K</td>
<td>64K</td>
<td>32K</td>
<td>16K</td>
</tr>
<tr>
<td>Max. Shared mem. / Block</td>
<td>48K</td>
<td>48K</td>
<td>48K</td>
<td>48K</td>
<td>16K</td>
</tr>
<tr>
<td>Max threads / Block</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>512</td>
</tr>
<tr>
<td>Max. threads / SM</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
<td>1536</td>
<td>1024</td>
</tr>
<tr>
<td>Max. active blocks / SM</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

As mentioned in Chapter 4, the kernels execution time was measured using PAPI and by executing multiple times each considered configuration. Power consumption was measured by using the built-in sensors of Tesla K20c and K40c GPUs, doing the corrections proposed in [20] (see Section 4.3). Since the other GPU devices do not include these built-in sensors, they could only be used to test the performance optimization procedures.
5.2 Optimization considerations

This section presents some interesting considerations regarding the execution of application kernels on GPU devices, which are important to highlight before presenting the results obtained when applying the proposed procedures. Section 5.2.1 presents an example of a case where the serialization of kernels in a multiple kernel environment is evident. Section 5.2.2 presents a different example, where it can be seen the relation between the occupancy of the GPU resources and the performance of the GPU kernels.

5.2.1 Concurrent execution

As it was mentioned in Chapter 2, the current implementation of concurrency on GPUs suffers from a wide variety of serialization issues that prevent true concurrent execution of the workloads [32]. To verify this statement the MatrixMul kernel was executed in a parallel environment under two different scenarios: 1) launching each kernel with a low total number of blocks (16), each composed of 64 parallel threads (arranged in two dimensions 8 × 8), resulting in a low occupancy level; and 2) launching each kernel with the default number of blocks (16384), each also composed of 64 parallel threads, but this time assuming each one computes a single value of the output matrix, resulting in a much higher occupancy level. The results for these two scenarios, both considering the multiplication of two 1024 × 1024 matrices, in either single- or multi-kernel environments, can be seen in Figures 5.1 and 5.2 respectively.

![Figure 5.1: Execution of MatrixMul kernels with 16 blocks per grid on Tesla K20c.](image)

By analysing the obtained results, it can be seen in Figure 5.1 that the execution time for one kernel or four concurrent kernels is essentially the same. By carefully analyzing the mapping of the kernel launch environment on the Tesla K20c architecture, it can be concluded that this is an expected result, since each kernel results in a very low occupancy of the GPU resources, meaning there are enough available resources for the other kernels to execute concurrently without slowing each other down. In particular, the Tesla K20c architecture features 13 SMs having each at most 16 active thread blocks (see Table 5.2). Thus, on the single-kernel case, the GPU block scheduler tries to divide the 16 thread blocks uniformly between the 13 SMs, resulting in 10 SMs with 1 thread block and 3 SMs with 2 thread blocks. Hence, with this distribution, each SM still has many available hardware resources (using only 2 out 16 thread blocks per SM, and 64 × 2 = 128 out of 2048 thread per SM). When four concurrent MatrixMul kernels are launched, the block scheduler is able to schedule all the kernels between the 13 SMs in a way that they can all execute concurrently, effectively hiding the instruction execution latency of each other. Accordingly, the
execution time does not increase, although leading to an increase in dynamic power, since more resources are being used.

![Figure 5.2: Execution of MatrixMul kernels with 16384 blocks per grid on Tesla K20c.](image)

By analysing Figure 5.2 it can be seen that with two concurrent kernels the total execution time doubles the one achieved with a single kernel, and with four kernels the execution time is four times as high. This seems to indicate that, despite being launched concurrently in the source code, the kernels are executing in a serialized fashion. In fact, since each kernel has 16384 thread blocks, when the block scheduler allocates the resources for the first kernel, it will allocate all the SMs for that kernel alone, and since $16384/13=1260.3$ is higher than the 16 maximum thread blocks per SM, the remaining kernels will need to wait until the first one is finished, before they are allowed to execute. Only then, will the block scheduler allocate resources for the second kernel, and so on. Another interesting result that can be observed by carefully analysing Figure 5.2 is the fact that the consumed power is the same for the three tested environments. This results further supports the conclusion that the kernels are being serially executed. Nonetheless, since the execution time is still smaller than the one obtained in Figure 5.1 it can also be concluded that significant performance improvements cannot be achieved by simply exploiting homogeneous multi-kernel execution environments. However, heterogeneous multi-kernel environments are still a promising approach to help hiding instruction latency beyond what is possible using single-kernel approaches, and to promote overall hardware utilization efficiency.

5.2.2 Occupancy

The proposed procedures use occupancy as a measure of the utilization of the GPU resources, in order to quantify the quality of a certain thread block distribution, with the objective of limiting the search space necessary to explore, until the best execution configuration is found. The FDTD3D kernel is a particularly interesting case, because of its high number of registers required per thread, which limits the maximum number of thread blocks possible to allocate per SM. Table 5.3 presents the amount of registers required by each thread, when executing the FDTD3D on NVIDIA’s Tesla K40c, Tesla K20c, GTX680 and GTX580.

<table>
<thead>
<tr>
<th>Device</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tesla K40c</td>
<td>65</td>
</tr>
<tr>
<td>Tesla K20c</td>
<td>45</td>
</tr>
<tr>
<td>GTX680</td>
<td>63</td>
</tr>
<tr>
<td>GTX580</td>
<td>60</td>
</tr>
</tbody>
</table>

Table 5.3: Number of registers per thread for the FDTD3D kernel, on different GPU devices.
It can be seen that even though the Tesla K20c and the Tesla K40c GPUs have similar characteristics (see Table 5.2), the number of registers used by the kernel is quite different, since different CUDA drivers and compilers are used in each machine. By applying this number of registers per thread in Equation 3.6 (Section 3.1) it is possible to calculate the maximum number of thread blocks for each of the thread block sizes tested. Accordingly, Figure 5.3 presents the occupancy levels achieved with different thread block sizes, when executing the FDTD3D kernel, in each of the four considered GPU devices. It can be seen that different GPU devices will achieve different occupancy levels for the same thread block size, and that the thread block size that corresponds to the maximum occupancy level is not the same across all four GPU devices.

Figure 5.3: Occupancy levels for different thread block sizes for the kernel FDTD3D, on different GPU devices.

To further illustrate the mapping of the previously computed occupancy levels on the actual execution, Figure 5.4 presents the execution time achieved when executing the same FDTD3D kernel, with different thread block sizes on the same four GPU devices previously mentioned. When comparing Figure 5.3 and Figure 5.4 it can be observed that the thread block size associated with minimum execution time for one GPU, always corresponds to a thread block size with maximum occupancy for that kernel on that GPU device. This further confirms and justifies some of the options taken to developed the proposed optimization procedures, in particular that performance is highly correlated with the percentage of GPU resources effectively being used. Furthermore, it demonstrates the advantages of the proposed auto-tuning.

Figure 5.4: Execution time for different thread block sizes for the kernel FDTD3D, on different GPU devices.
procedures and tool, by allowing optimizing the kernel according not only with the GPU architecture, but also with the execution environment that may feature different device drivers.

5.3 Optimization results

To evaluate the performance of the proposed procedures and its implementation, a set of case studies were carefully prepared, which are divided into two different scenarios: single-kernel execution environments, described in Section 5.3.1, and multi-kernel execution environments, described in Section 5.3.2. In order to verify the quality of the proposed procedures in all tested scenarios besides applying the proposed algorithms, exhaustive testing through all the search space was performed, allowing the comparison between the results found by the procedures with the optimal ones. At the end of this section are presented two Tables with the summary of the use of the developed auto-tuning tool in single- and multi-kernel environments, respectively.

5.3.1 Single-kernel

To test the quality of the single-kernel optimization procedures, the developed auto-tuning tool was used with several different GPU applications on distinct GPU devices. Figure 5.5 presents the results of the performance optimization procedure of the Streamcluster kernel, when executed on Tesla K40c, Tesla K20c, GTX680 and GTX580.

As mentioned before, the procedures were developed under the consideration that the maximum performance is attained at the maximum operating frequency. Furthermore, since only thread blocks with a size within the powers of 2 are considerer, the complete search space is limited to a total number of 10 possible combinations. However, by cross comparing the GPU characteristics with the kernel profile information, the whole search space is constrained to 4 possible combinations. Nonetheless, not all possible combinations need to be tested; as an example, for the GTX580 and GTX680 GPUs, only 2 combinations are tested to determine the optimal configuration (which is achieved in all GPUs for the Streamcluster benchmark).
Figure 5.6 presents the evolution of the performance optimization procedure for the *Lud* kernel, when executed on the same four GPU devices, and also on NVIDIA GTX285. On both presented Figures, it can be seen that the configuration found by the proposed procedure depends on the GPU device being used. This confirms the need for a procedure like the one proposed, capable of providing performance portability between different GPU devices.

By using the proposed energy-aware optimization procedure it is also possible to optimize the kernels distributions and GPU operating frequency level such as to minimize energy consumption. As previously mentioned, while the search space of the performance optimization procedure could be constrained to the highest operating frequency, this is not possible for energy-efficiency optimization. As a consequence, the complete search space (number of possible solutions) considers all the combinations of possible blocksize configurations and frequency levels. As an example, for the *Streamcluster* benchmark, presented in Figure 5.7:

![Figure 5.6: Performance optimization procedure for the *Lud* kernel. The full optimization space includes 9 possible kernel configuration on the GTX285 and GTX580 devices, and 10 on the remaining tested GPU devices.](image)

![Figure 5.7: Energy-aware optimization procedure for the *Streamcluster* kernel. The full search space for energy minimization includes 50 and 40 possible configurations for the Tesla K20c and Tesla K40c GPUs, respectively.](image)
Figure 5.7, a total number of 50 (40) possible configurations exist when optimizing for the Tesla K20c GPU (K40c GPU), since there are 10 possible distributions of threads, from $2^1$ to $2^{10}$, and 5 (4) allowed frequency levels.

To minimize the search space, the proposed optimization procedure first constraints the possible solutions to the cases with the highest theoretical GPU occupancy (see Section 3.1), reducing the number of possible solutions to 20 (16), which corresponds to a reduction of 60% of the whole search space. The optimization procedure then starts by iteratively testing possible solutions according to Algorithm 3.2 such as to obtain a configuration with reduced energy consumption. As shown in Figure 5.7, for the Streamcluster kernel, only a set of up to 7 (6) possible configurations are required to be tested in order to achieve the minimum energy consumption case, meaning only 14% (15%) of the initial search space require testing, for the Streamcluster kernel.

Figures 5.8 and 5.9 present the results of applying the energy-aware optimization procedure to the LUD and ParticleFilter kernels, respectively, on both Tesla K20c and Tesla K40c.

Figure 5.8: Energy-aware optimization procedure for the LUD kernel. The full search space for energy minimization includes 25 and 20 possible configurations for the Tesla K20c and Tesla K40c GPUs, respectively.

Again it can be seen that the number of iterations required to find the best kernel configuration is very low when compared with the size of the whole search space. With the results for these two tested kernels when applied the energy-aware optimization procedure, it can also be seen that they always achieve a energy consumption at most 2% above the optimal value, in a low number of iterations (at most 7 distributions required testing, which corresponds to only 17% of the whole search space).

5.3.2 Multi-kernel

While the previous section showed that the proposed single-kernel performance and energy-aware optimization procedures are able to find the optimal solutions using a low number of iterations, this subsection focuses on showing the results obtained when testing the multi-kernel auto-tuning procedures, using the developed auto-tuning tool. Like in the single-kernel experiments, to test the quality of the proposed procedures, the developed auto-tuning tool was used with concurrent GPU kernels on several
Figure 5.9: Energy-aware optimization procedure for the ParticleFilter kernel. The full search space for energy minimization includes 45 and 36 possible configurations for the Tesla K20c and Tesla K40c GPUs, respectively.

different GPU devices. Figures 5.10, 5.11 and 5.12 present the execution results when using the autotuning tool with different kernels on GPU devices with distinct characteristics. Figure 5.10 corresponds to the optimization of two concurrent MatrixMul kernels, Figure 5.11 to the optimization of one MatrixMul and one Lud kernel, and finally, Figure 5.12 presents the optimization of one ParticleFilter and one Lud kernel.

By analysing Figures 5.10, 5.11 and 5.12 it is possible to conclude that the multi-kernel performance optimization procedure is also able to find the best execution parameters using a reduced number of iterations, regardless of the increased dimension of the whole search space. In particular, the worst case regarding the number of iterations, corresponds to the MatrixMul kernel case presented in Figure 5.10 where 2 MatrixMul kernels are optimized. In this case, 3 iterations out of 25 possible combinations are possible, which corresponds to 12% of the whole search space.

Figure 5.10: Multi-kernel performance optimization procedure for the 2 concurrent MatrixMul kernels. The full optimization space includes 25 possible kernel configurations on all tested GPU devices.
Figure 5.11: Multi-kernel performance optimization procedure for the MatrixMul and Lud kernels. The full optimization space includes 50 possible kernel configurations on all tested GPU devices.

Figure 5.12: Multi-kernel performance optimization procedure for the ParticleFilter and Lud kernels. The full optimization space includes 45 possible kernel configurations on all tested GPU devices.

Figures 5.13, 5.14 and 5.15 present the results regarding multi-kernel energy-efficiency optimization procedures for the same kernel combinations. In particular, Figure 5.13 shows the iterative optimization for the case with two concurrent MatrixMul kernels on a Tesla K40c GPU, which shows that only 4 iterations are needed to obtain the optimal distribution. It is interesting to notice that the optimal solution according to the energy-efficient optimization procedure is the same as for the performance optimization procedure. Additionally, as expected, the distribution for each of the MatrixMul kernels is the same obtained when executing only one kernel on the GPU device. It is important to mention that the used implementation of the matrix multiplication algorithm is compute-bounded, which is the reason why the optimal configurations use the maximum number of threads available.

Unlike the case with two concurrent MatrixMul kernels, the scenarios represented in Figures 5.14 and 5.15 corresponding to the MatrixMul + Lud and ParticleFilter + Lud kernels cases, result in different kernel distributions than when optimizing for performance (compare with Figures 5.11 and 5.12). It is also worth mentioning that in the worst case between the three scenarios the distributions found result in an energy consumption less than 1% above the oracle value, taking at most 6 iterations out of the 225 necessary for the exhaustive search.
Figure 5.13: Multi-kernel energy-aware optimization procedure for 2 concurrent *MatrixMul* kernels, executed on Tesla K40c. The full optimization space includes 100 possible configurations.

Figure 5.14: Multi-kernel energy-aware optimization procedure for the *MatrixMul* and *Lud* kernels, executed on Tesla K20c. The full optimization space includes 125 possible configurations.

Figure 5.15: Multi-kernel energy-aware optimization procedure for the *ParticleFilter* and *Lud* kernels, executed on Tesla K20c. The full optimization space includes 225 possible configurations.

Finally, Figure 5.16 presents the results of the performance auto-tuning procedure for a multi-kernel environment composed by 4 kernels (*Lud*, *Streamcluster*, *Hotspot* and *ParticleFilter*). In this case there are 2250 possible combinations of kernel distributions when optimizing for the Tesla K40c GPU. After reducing the search space there are 48 possible combinations (2% of the whole search space), however, as it is shown in Figure 5.16 when optimizing for performance the procedure will only need to test 8 combinations out of those 48 (17% of the reduced search space, and 0.3% of the whole search space).

When executing the energy-aware procedure on the Tesla K40c GPU, the whole search space is composed of 9000 combinations, which are reduced to 192 (2%), by making the same consideration mentioned before. Figure 5.17 shows that the procedure will also only need to test 8 combinations of
Figure 5.16: Performance optimization procedure for 4 concurrent kernels, namely *Lud*, *Streamcluster*, *Hotspot* and *ParticleFilter*, executed on a Tesla K40c GPU. The full optimization space includes 2250 possible configurations.

Figure 5.17: Energy-aware optimization procedure for 4 concurrent kernels, namely *Lud*, *Streamcluster*, *Hotspot* and *ParticleFilter*, executed on a Tesla K40c GPU. The full optimization space includes 9000 possible configurations.

A summary of all the results for the performance and energy-efficiency optimization procedures for the full set of benchmarks, under single- and multi-kernel environments is presented in Tables 5.4 and 5.5, respectively. To better analyze the proposed optimization procedures, the tables include: (columns *Blocksize* and *Frequency*) auto-tuned configuration; (column *#iterations*) number of iterations required to achieve the presented solution out of the total number of possible combinations; and (columns *performance trade-off* and *energy savings*) the performance/energy trade-offs between the energy-efficiency and performance optimization procedures.
Table 5.4: Summary of the results for the single-kernel performance and energy-efficiency auto-tuning procedures.

<table>
<thead>
<tr>
<th>BENCHMARK</th>
<th>GPU</th>
<th>PERFORMANCE</th>
<th>ENERGY-EFFICIENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Blocksize</td>
<td>#iterations</td>
</tr>
<tr>
<td>SINGLE KERNEL OPTIMIZATION</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>StreamCluster</td>
<td>Tesla K20c</td>
<td>128</td>
<td>4 (10)</td>
</tr>
<tr>
<td></td>
<td>Tesla K40c</td>
<td>128</td>
<td>4 (10)</td>
</tr>
<tr>
<td></td>
<td>GTX 680†</td>
<td>1024</td>
<td>2 (10)</td>
</tr>
<tr>
<td></td>
<td>GTX 580†</td>
<td>1024</td>
<td>2 (10)</td>
</tr>
<tr>
<td></td>
<td>GTX 285†</td>
<td>128</td>
<td>3 (9)</td>
</tr>
<tr>
<td>Particle Filter</td>
<td>Tesla K20c</td>
<td>128</td>
<td>3 (9)</td>
</tr>
<tr>
<td></td>
<td>Tesla K40c</td>
<td>128</td>
<td>3 (9)</td>
</tr>
<tr>
<td></td>
<td>GTX 680†</td>
<td>512</td>
<td>2 (9)</td>
</tr>
<tr>
<td></td>
<td>GTX 580†</td>
<td>256</td>
<td>2 (9)</td>
</tr>
<tr>
<td></td>
<td>GTX 285†</td>
<td>256</td>
<td>3 (9)</td>
</tr>
<tr>
<td>Lud</td>
<td>Tesla K20c</td>
<td>16×16</td>
<td>2 (5)</td>
</tr>
<tr>
<td></td>
<td>Tesla K40c</td>
<td>16×16</td>
<td>2 (5)</td>
</tr>
<tr>
<td></td>
<td>GTX 680†</td>
<td>16×16</td>
<td>2 (5)</td>
</tr>
<tr>
<td></td>
<td>GTX 580†</td>
<td>16×16</td>
<td>1 (5)</td>
</tr>
<tr>
<td></td>
<td>GTX 285†</td>
<td>16×16</td>
<td>1 (4)</td>
</tr>
<tr>
<td>MatrixMul</td>
<td>Tesla K20c</td>
<td>32×32</td>
<td>2 (5)</td>
</tr>
<tr>
<td></td>
<td>Tesla K40c</td>
<td>32×32</td>
<td>2 (5)</td>
</tr>
<tr>
<td></td>
<td>GTX 680†</td>
<td>32×32</td>
<td>2 (5)</td>
</tr>
<tr>
<td></td>
<td>GTX 580†</td>
<td>32×32</td>
<td>2 (5)</td>
</tr>
<tr>
<td>K-Means</td>
<td>Tesla K20c</td>
<td>512</td>
<td>3 (10)</td>
</tr>
<tr>
<td></td>
<td>Tesla K40c</td>
<td>512</td>
<td>3 (10)</td>
</tr>
<tr>
<td>S-Rad</td>
<td>Tesla K40c</td>
<td>256</td>
<td>2 (8)</td>
</tr>
<tr>
<td></td>
<td>GTX 680†</td>
<td>256</td>
<td>2 (8)</td>
</tr>
<tr>
<td></td>
<td>GTX 580†</td>
<td>256</td>
<td>2 (8)</td>
</tr>
<tr>
<td></td>
<td>GTX 285†</td>
<td>256</td>
<td>2 (8)</td>
</tr>
<tr>
<td>BFS</td>
<td>Tesla K40c</td>
<td>128</td>
<td>4 (10)</td>
</tr>
</tbody>
</table>

† Energy optimization not possible since there is no built-in power sensor.
(a) The optimal configuration (16×8 @ 810 MHz) was not found, which would use 1% less energy.
Table 5.5: Summary of the results for the multi-kernel performance and energy-efficiency auto-tuning procedures.

<table>
<thead>
<tr>
<th>BENCHMARK</th>
<th>GPU</th>
<th>PERFORMANCE</th>
<th>ENERGY-EFFICIENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Blocksize</td>
<td>#iterations</td>
</tr>
<tr>
<td>MULTIPLE KERNEL OPTIMIZATION</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lud &amp; ParticleFilter</td>
<td>Tesla K20c</td>
<td>16 × 16</td>
<td>5 (45)</td>
</tr>
<tr>
<td></td>
<td>Tesla K40c</td>
<td>16 × 16</td>
<td>5 (45)</td>
</tr>
<tr>
<td></td>
<td>GTX 580†</td>
<td>16 × 16</td>
<td>4 (45)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>128</td>
<td>6</td>
</tr>
<tr>
<td>Lud &amp; MatrixMul</td>
<td>Tesla K20c</td>
<td>16 × 16</td>
<td>4 (25)</td>
</tr>
<tr>
<td></td>
<td>Tesla K40c</td>
<td>32 × 32</td>
<td>3 (25)</td>
</tr>
<tr>
<td></td>
<td>GTX 580†</td>
<td>32 × 32</td>
<td>3 (25)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32×32</td>
<td>0%</td>
</tr>
<tr>
<td>2× MatrixMul</td>
<td>Tesla K40c</td>
<td>32 × 32</td>
<td>7 (2500)</td>
</tr>
<tr>
<td></td>
<td>Tesla K40c</td>
<td>32 × 32</td>
<td>10 (625)</td>
</tr>
<tr>
<td></td>
<td>Tesla K40c</td>
<td>128</td>
<td>8 (2250)</td>
</tr>
<tr>
<td>Streamcluster &amp; ParticleFilter &amp; Lud &amp; Hotspot</td>
<td>Tesla K40c</td>
<td>128</td>
<td>8 (2250)</td>
</tr>
</tbody>
</table>

† Energy optimization not possible since there is no built-in power sensor.

(a) The optimal configuration (32×32, 128 @ 640 MHz) was not found, which would use 0.3% less energy.

(b) The optimal configuration (512, 32×32, 32×32, 64 @ 745 MHz) was not found, which would use 8% less energy.
By analysing the results in the tables, it can be observed that the proposed optimization procedures are able to find a near-optimal solution using a reduced set of iterations, with a maximum of 10 iterations out of 625, achieved in one of the tested 4 concurrent kernels scenarios. This significantly contrasts with an exhaustive search procedure that can require testing a large set of possibilities, especially for the energy-aware and multi-kernel cases. As mentioned before, for a 4-kernel configuration the total set of possible combinations can go up to 2500 and 10000 for the performance and energy-efficiency cases, respectively. Furthermore, not only the number of iterations is significantly reduced, but also the obtained configurations are very close to the optimal ones. In particular, the proposed algorithms were able to find the oracle solution in most considered cases. Nonetheless, the obtained solutions consume at most <1% more energy for the 2-kernel cases and <4% for the 4-kernel case when compared with the oracle solutions.

From both Tables 5.4 and 5.5 it can also be confirmed that not only is the optimal kernel configuration dependent on the GPU device, but it can also change when multiple kernels are concurrently executed, which further illustrates the need for the procedures proposed in this thesis. As an example, for the ParticleFilter kernel, the best distribution ranges from 128 threads/block to 512 threads/block between the Tesla K20c and GTX580 GPU, respectively. Furthermore, by comparing the kernel distributions and operating frequency of the concurrent 4-kernel cases and the corresponding cases when the kernel is individually executed, different solutions may be observed. This reinforces the advantages of integrating an automatic tuning tool with the applications source-code such as to allow optimizing the source code to the users GPU.

Figure 5.18 presents the summary of the energy-savings and performance trade-offs when comparing the found distributions after applying the auto-tuning tool for performance and energy-efficiency optimization. This figure reinforces that significant energy-savings can be achieved by applying an energy-efficient optimization procedure. In particular, in the experiments an average of 5.90% of energy savings was observed, with 6.40% energy savings in the single-kernel scenarios, and 5.40% in the multi-kernel.

Furthermore, in the single-kernel environment up to 13% energy savings could be observed for the Lud kernel. In multi-kernel environments the observed energy-savings were as high as 11%, for the case with 2 MatrixMul kernels concurrent with 1 MatrixTrans and 1 VectorAdd kernels. Moreover, in this latter scenario, the energy savings were achieved with a performance degradation lower than 1%, proving that with the proposed procedures, in certain cases significant energy savings can be achieved with a reduced performance degradation.

Finally, it is interesting to notice that the application of the proposed adaptive search procedures, take a considerable less amount of time to obtain near-optimal solutions. For example, the case with 4 concurrent kernels, namely 2× MatrixMul, 1 VectorAdd and 1 MatrixTrans, the performance optimization procedure requires only 7 iterations to find the output configuration corresponding to a total of 1.475 seconds for the whole algorithm to execute. In comparison, if one would use an exhaustive search procedure, through a brute-force method it would take around 264 seconds to reach the optimal solution. Furthermore, the adaptive search finds a solution that has a performance only 1.3% worse than the one found through the exhaustive search, and in fact, in all tested cases, the solutions of the proposed
procedures are very close to the optimal solutions.

5.4 Summary

This chapter presented the experimental results obtained throughout the course of this thesis, regarding the execution of concurrent kernels on GPU devices, and the implementation of the auto-tuning procedures for GPU kernels, proposed in Chapter 3. Accordingly, the chapter started with the description of the GPU application benchmarks and GPU devices used during the experiments. Furthermore, the results obtained when applying the auto-tuning tool, were presented for both optimization goals (performance and energy-efficiency), on both possible GPU environments (single-kernel and multi-kernel). To show the benefits of the developed auto-tuning tool, charts with the execution of the iterative procedure were presented, showing step-by-step the variations that each change to the kernels configuration or op-
Operating frequency can provoke. Finally, it was shown that the proposed multi-kernel energy-efficiency optimization procedures can in certain cases achieve energy-savings as high as 11%, requiring only 7 iterations, which corresponds to 0.06% of the whole search space.
6 Conclusions

Contents

6.1 Future work ................................................. 75
The work developed during the six month period corresponding to this masters thesis was focused on the investigation of novel auto-tuning procedures for single- and multi-kernel optimization in what concerns performance and energy-efficiency, by considering kernel launch environment (and in particular, the number of threads per block) and GPU frequency scaling. This leads to a discrete and non-linear optimization problem, which is herein tackled by means of iterative algorithms.

Exhaustive search approaches are unfeasible for this kind of problems, especially when considering the natural evolution of GPU architectures, which is moving towards supporting increased number of threads per block and blocks per SM and also supporting increased frequency levels both at graphics core and at the memory levels. Accordingly, search space reduction techniques were investigated by relying on theoretical evaluations of kernel occupancy in the GPU. The employment of such techniques is fundamental by significantly reducing the number of possible solutions and thus allowing achieving near-optimal solutions in a feasible amount of time and iterations.

The developed single- and multi-kernel optimization procedures were further integrated into an auto-tuning tool that allows profiling GPU kernels and monitoring their execution such as to obtain an important set of parameters, which are fundamental for the optimization process. The tool is thus able to collect all the necessary information in order to apply the proposed optimization procedures and obtain near-optimal execution parameters.

Furthermore, to provide the means for energy optimization, a power-measuring tool was further developed that relies on GPU internal power counters to monitor both GPU power and energy consumption. To improve the quality of the obtained measurements, a set of corrections were also applied, by following state-of-the-art recommendations. This power measurement tool was further integrated into the auto-tuning tool such as to provide the means for energy-aware optimization. Notwithstanding, the developed power measuring tool can also be used independently such as to be used by third-party applications.

Experimental results indicate that, in some kernels, there is indeed a difference between optimizing for performance and optimizing for energy, in terms of the optimal values for the execution parameters. All experimental results, also demonstrate that the mechanisms proposed for the reduction of the search space are effective, since the procedures are always able to find near-optimal solutions, in a very reduced number of iterations. Furthermore, the proposed auto-tuning procedures can lead to significant energy savings, sometimes as high as 13%, with a cost of degrading the performance by at most 19%. However, other situations were observed where energy-savings of 11% could be achieved, with a performance trade-off under 1%.

Therefore, it can be concluded that the development of such procedures is a fundamental path in the investigation of GPGPU solutions, given that they can facilitate application developers in the optimization process of their code, since these procedures can provide the insights to chose the kernel parameters that have the desired trade-off between performance and energy consumption, and also by providing the means for efficient migration of application kernels across device architectures and GPU generations.
6.1 Future work

While the application of the developed procedures sets the path for automatic kernel optimization procedures, this work could be complemented by using state-of-the-art simulators, such as GPGPU-sim \cite{19} and GPUWattch \cite{30}. This should allow the creation of simulated GPU environments, where new execution conditions could be tested. For example, it would be possible to test the execution of concurrent kernels with a larger range of GPU operating frequencies, allowing a better understanding on the behaviour of the performance and energy-efficiency metrics, when changing the operating frequency. Furthermore, by relying on the simulator tools, one could set different values for the memory subsystem frequency, which, as already stated, is not possible in current GPUs. This way, it would be possible to also study the influence of the memory subsystem, which would most likely lead to further energy-aware optimization techniques.

Using these same state-of-the-art simulators, the proposed multi-kernel procedure for energy-efficiency could be updated, in order to find different optimal frequency levels for each of the concurrent application kernels. Since the simulators provide the ability to study a controlled environment it is possible to find the instants between the kernels execution where the frequency should be changed, which is not possible to do in real on-line systems.

Finally, a different implementation of the proposed procedures could be developed, with full support for OpenCL programs. This would allow to test the proposed procedures in a wider range of compute devices, since OpenCL applications can execute on all GPUs (NVIDIA, AMD, etc.), on multi-core CPUs (Intel Haswell, etc.) and also on coprocessors (Xeon Phi, etc.).
References


