UNIVERSIDAD
DE MÁLAGA

Departamento de Arquitectura de Computadores

TESIS DOCTORAL

Scheduling strategies for parallel patterns on heterogeneous architectures

Antonio Vilches Reina

Julio de 2017

Dirigida por:
María Ángeles González Navarro,
Francisco Javier Corbera Peña

UNIVERSIDAD
DE MÁLAGA

AUTOR: Antonio Vilches Reina

http://orcid.org/0000-0002-9742-3717

EDITA: Publicaciones y Divulgación Científica. Universidad de Málaga

Dra. M. Ángeles González Navarro.
Profesor Titular del Departamento de Arquitectura de Computadores de la Universidad de Málaga.

Dr. Francisco Javier Corbera Peña.
Profesor Titular del Departamento de Arquitectura de Computadores de la Universidad de Málaga.

CERTIFICAN:

Que la memoria titulada "Scheduling Strategies for parallel patterns on heterogeneous architectures", ha sido realizada por D. Antonio Vilches Reina bajo nuestra dirección en el Departamento de Arquitectura de Computadores de la Universidad de Málaga y constituye la Tesis que presenta para optar al grado de Doctor en Ingeniería Informática.

Málaga, Julio de 2017

Dra. M. Ángeles González Navarro.
Codirectora de la tesis.

Dr. Francisco J. Corbera Peña.
Codirector de la tesis.

To my grandmother Rosario,
who passed away whilst this thesis was being reviewed.

Acknowledgments

Ph.D. is a long and arduous journey, which I could not finish successfully without the support from many people. I want to express my gratitude to all those who have contributed to make this thesis possible.

I want to start by giving thanks to my family. My parents have always been a source of motivation for me, they have taught me to persist, insist and never desist from my objectives. Thanks, once more again, for your infinite support and care. I would also like to thank Regina. Thanks for your unconditional love and support, thanks for your affection and enthusiasm, specially when I was abroad. Your patience was stretched to its limits during those stays abroad that I spent working far away from home. Thankfully, we can enjoy future adventures together from now on.

My deepest thanks go to my supervisors Prof. M. Ángeles González and Prof. Francisco J. Corbera. However, in equal measure, I thank Prof. Rafael Asenjo, the research team lead, who I consider my third supervisor. They form an outstanding team with three unique profiles. I was constantly astonished by Rafael's enthusiasm, and his skills to devise the whole picture. Moreover, he was usually hunger to explore all possible solutions to a given problem. In contrast, M. Ángeles was always focusing on the core problem, I want to highlight her attention to detail and her amazing analytical skills. She also exhibits a sharp critical sense that she puts into practice in every meeting. I really appreciate her patience and all the invaluable suggestions she gave me throughout this thesis. Francisco is the third pillar of the team, he has been a never ending source of motivation throughout my thesis, he was always willing to discuss new ideas and solutions. I want to emphasize his endless capacity to envision new solutions. I was constantly amazed by his point of views, and his skills to come up with new solutions.

I would also like to thank Óscar Plata, the head of the Computer Architecture department, for his suggestions and advises about further steps in life, the

technical staff of the department Juanjo and Paco and finally to Carmen for her praiseworthy support.

This thesis has been funded by the following spanish projects: TIN2010-16144 from Ministerio de Economía y Competitividad del reino de España, and P11-TIC-08144 from Consejería de Innovación, Ciencia y Empresa de la Junta de Andalucía. I also want to thank the HiPEAC European Network of Excellence, under the 7th framework programme under grant agreement 287759, whose funds made me enjoy a three month research stay at Codeplay Software Ltd.

I would like to thank Prof. María Garzarán for being a invaluable host during my stay at UIUC in 2014. I really appreciate her suggestions and conversations, about work and life, I learned a lot from them. I want to mention Prof. David Padua as well, a charismatic character full of interesting conversations and experiences. I also want to thank Dr. Konstantinos Karantasis for his welcome and support during my time in Urbana-Champaign.

From my period in Edinburgh, I would like to thank all the nice people that I met in Codeplay. In special to Dr. Ruyman Reyes and the rest of SYCLONYTES. They helped me a lot, and gave me the opportunities to explore on my own. I also thank to Dr. Uwe Dolinsky and Andrew Richards for our interesting talks. You made me enjoy a three awesome months there.

Last but no least, I would like to thank my Lab mates (the Grijanders-Crew), for all the fun that we had, for the awesome work atmosphere, for your support and friendship. I have enjoyed with our lunches, dinners and revelries. Many of them are working out of academy, however I remember you all: Alejandro Villegas, Alex Upton, Ricardo Quislant, Manuel Pedrero, Miguel Ángel González, Antonio J. de Dios, Alberto Sanz, Antonio Manuel Cervilla, Oscar Torreño, Jose Antonio Arjona and Antonio Muñoz.

Abstract

During the last decade, power consumption and energy efficiency have become key aspects in processor design. Nowadays, the power consumption is the principal limitation for further scaling of chip multiprocessors design (CMPs). In general, the research community agrees that current chip multiprocessor technology trends will not scale performance without an increase of power budget. Hardware design innovations as the recent Heterogeneous Architectures and Near Threshold Computing are needed to cope with the performance-power barrier. As a result of this, there has been a shift away from chip multiprocessors to heterogeneous processor architectures. Recently, we have witnessed an explosion in the availability of this kind of architectures. Many hardware vendors have released a number of heterogeneous processors to overcome the aforementioned limitations. However, software also requires changes to allow further performance scaling on these architectures.

With the advent of heterogeneous architectures, hardware manufactures have impose the burden of explicit accelerator management on software developers. In general, programmers are used to sequential programming, but writing high-performance programs for heterogeneous architectures is a complex task. Programming for this kind of platforms requires the understanding of new hardware concepts, orchestration of different parallelism levels, the explicit management of different memory spaces and synchronizations between processing units, and finally the usage of low-level programming models such as OpenCL or CUDA. Moreover, heterogeneous architectures suffer from performance portability, as one program can exhibit unequal performance on different devices.

To help shrink the programmability-performance efficiency gap, we discuss that adaptive runtime systems can be used to facilitate the management of heterogeneous architectures. A runtime system can provide a significant performance boost while reducing the energy consumption, because it is aware of processors' architectures and application's requirements. We analyse how applications map

onto hardware by inspecting built-in processor counters, and therefore build models to describe the observed behaviour.

In this thesis, we discuss how parallel patterns, such as *parallel for* loops and *pipelines*, can be decomposed and efficiently executed on heterogeneous platforms. We propose several scheduling strategies aiming at reducing execution time and energy consumption. We demonstrate how applications can be run faster by mapping the application level parallelism onto the hardware processing units that best fit the application requirements, and by selecting the right task size. First, we devise a load balancing technique, that targets heterogeneous CPU and multi-GPU architectures. It monitors the relative speed of each processing unit, and distributes the remaining workload based on these relative speeds. By making all processing units to finish at same time, we avoid unnecessary waits between processors. Along with this load balancing technique, we propose a *performance-sensitive* partitioner that adapts the amount of computation offloaded to the accelerator for better performance and utilization. We also present an accurate performance model for streaming applications, such as face recognition or object tracking. This model targets pipelined applications, as a series of stages, and performs a scalability analysis of each stage by using coarse and medium grain parallelism. Additionally, it also considers executing the stage on the GPU or not. By applying the model, we always find the best pipeline configuration among all possible, and get substantial performance and energy savings.

All experiments in this thesis have been performed by using state-of-the-art hardware accelerators and benchmarks of the field of HPC. Specifically, we use the Rodinia and SHOC benchmark suites, for the evaluation of the *parallel for* partitioner. Moreover, we use the the ViVid application, along with tracking and SRAD applications from Rodinia Benchmark Suite, all of them are good candidates of vision applications. Finally, we rely on Intel Threading Building Blocks, the core engine of our schedulers; the Intel OpenCL SDK and CUDA SDK to offload computations to the GPU accelerators and Intel PCM library to monitor energy consumption and cache memory metrics.

Contents

List of Figures

List of Tables

1 Introduction

During the last decade, we have witnessed the advent of heterogeneous architectures. Nowadays, we can find heterogeneous processor designs at all levels, from the biggest supercomputer to embedded devices, covering desktop and mobile segments as well. These kind of architectures are becoming predominant in today markets, as they provide significant improvement in performance and energy efficiency over traditional chip multiprocessor (CMP) [64]. Moreover, power consumption is becoming an important performance scaling limitation in new processor designs, and the subsequent dark silicon rise [102]. To help shrink the performance-power consumption gap, we discus the need of adaptive runtime systems (RS) to orchestrate and map the application level parallelism onto the underlying hardware architecture in terms of performance and energy consumption.

We introduce the shift to heterogeneous processors in Section 1.1. They require the use of low-level programming libraries that make the programmer responsible of accelerator management, data partitioning and parallelism orchestration as we spot in Section 1.2. Then, in Section 1.3, we discuss the need of adaptive RS to alleviate the aforementioned complexities when offloading computation to accelerators. Next, Section 1.4 highlights the thesis motivations and elaborates of the main unsolved problems that we target. Section 1.5 states the research question of this thesis and its objectives. Finally, we present the thesis contributions in Section 1.6, and we conclude with Section 1.7, which outlines the thesis structure.

1.1. The era of Heterogeneous Architectures

As Gordon E. Moore predicted around fifty years ago, the number of transistor within a chip is still increasing every two years [78]. But, computing power and energy consumption are not following the same trend. Previous processor generations scaled down the voltage within each new processor generation and increased frequency as a way to get more performance. However, due to physical limitations in current processor technology, nowadays the scaling of frequency does not yield a proportional increment of performance, this fact is known as the Dennard scaling failure [6]. Therefore, the scientific community agrees that more innovations are required to cope with these limitations of power consumption and performance scaling.

During the last decade, we have witnessed a revolution in processor architecture design, where there has been a shift away from CMPs to Heterogeneous Multi/Many Processors (HMPs). As a result of this, there has been an explosion in the availability of heterogeneous architectures. The main reason for this shift is that specially tailored core processors exhibit a higher computation power and less energy consumption than traditional CMPs while executing some specific tasks [64, 76]. Intel Haswell, AMD Fusion, Samsung Exynos and NVIDIA Tegra K1 are examples of these modern architectures, which integrate a graphic processor along with the CPU into the same chip.

The current landscape includes a number of heterogeneous architectures with a wide range of core counts, capabilities per core and an energy budget per core. In this context, GPUs have emerged as the dominant accelerator type on heterogeneous architectures. They are characterized by a large set of simple computational cores, that are hierarchically organized in smaller groups. All cores within a group run in lock-step and share a fast access memory. As mentioned before, there is a range of heterogeneous machines for all segments, from supercomputers to hand-held system, such as mobile phones. Table 1.1 compares performance and energy consumption of several accelerators. The Intel Xeon CPU is the one that offers the poorest performance and the low-end Intel GPU is the accelerator with the minimum energy consumption. However, the high-end Nvidia GPU offers the best ratio in GFlops per watt. Thus, the suitable device for each application would depend on the application performance versus energy consumption requirements.

With the advent of heterogeneous architectures, application developers have a huge amount of available computing power at a low power consumption. However, processor manufacturers have put a big burden on software developers as

Table 1.1: Comparison of accelerator characteristics.

Architecture	N. Cores	Freq.(MHz)	TDP (W)	Perf. (GFlop/s)
Nvidia Geforce Titan X	3072	1000	250	6144
Intel HD Graphics 4600	20	1200	84	432
Intel Xeon Phi 7120P	61	1240	300	2416
Intel Xeon E5-2698v3	16	2300	135	294,4

they are responsible to orchestrate and map the application level parallelism onto the hardware level. To develop applications for heterogeneous architectures, we require the use of low-level programming libraries such as, CUDA [96] or OpenCL [38]. These libraries expose architectural accelerator details that hinder the development of applications on top of heterogeneous architectures. Many Researchers agree that new programming libraries are required to offer an abstraction layer that hides the aforementioned complexities while making the most of these promising platforms. In next section, we explain the limitations of the current programming libraries and identify the future challenges that needs to be solved.

1.2. The complexities of Heterogeneous Computing

Task parallelism is a well known paradigm for parallel architectures, where each application is divided into independent tasks and executed onto a set of threads. Writing multi-threaded applications involves the division of the whole application into independent tasks, assigning each task to a logical thread and orchestrating those threads to map them onto the underlying hardware computing devices. All theses steps make parallel programming difficult for programmers, where orchestration is probably the most complex one, as it comprises threads synchronization that could potentially result in load imbalances and deadlock scenarios.

In a heterogeneous computing context, the execution of an application is even more complex, as the execution time of a given task depends on the kind of core that runs this task. For example, a computing vision application detecting faces or tracking objects executes much faster on GPUs than on a CPU core. On the contrary, sequential algorithms with a certain number of conditional instructions and a random memory access pattern would execute faster on CPUs rather than GPUs. The key idea is to assign each task to the computing device that best fit to it. In this thesis, we focus on heterogeneous CPU-GPU architectures, however

all proposed frameworks and models can be easily extended to work with other accelerators such as, Intel Xeon Phi, FPGAs or DSPs.

Heterogeneous computing is the paradigm which aims to combine the strengths of using heterogeneous architectures to meet the requirements of each application [77]. As a result of the explosion in the availability of heterogeneous architectures, there has also been an increment in the number of heterogeneous programming frameworks and libraries available. The most widely known are OpenCL [38] and CUDA [96], however there are other less common programming models such as, RenderScript [52] or Mare [12], which allow the exploitation of heterogeneous architectures on mobile devices. Additionally, there have been a number of efforts to extend vastly used languages as Java [31, 36], Haskell [16], JavaScript [56] or Python [61]. However, the challenges of harnessing heterogeneity have prevented a successful adoption of heterogeneous computing at the moment. Among these challenges, we can highlight performance issues of heterogeneous processors [59, 85], as for instance the fact that many accelerators are only well-suited to a specific set of applications with coalesced memory access patterns and regular computations [11]. Moreover, application's developers have to decide whether their codes can take advantages from executing the application on a given accelerator or not. In addition, they are responsible to ensure data integrity and task dependencies, they are also required to manually set all the necessary memory operations in order to offload computations.

To help shrink the programmability-performance gap, the research community agrees that more innovations are required. In next section, we discuss that Runtime Systems (RS) can be exploited to reduce this gap.

1.3. Runtimes for Heterogeneous Architectures

RS based on work-staling strategies [7, 89] have been traditionally used as an abstraction layer while dealing with multithreadded applications. RS usually break down the parallelism expressed by the developer into smaller pieces, often called tasks in literature. The RS is responsible of scheduling and orchestrating those tasks by respecting a dependency graph. A *task* is a set of instructions that operates on data and can be run independently of any other task. In general, RS do not guarantee *determinism* or *data races* freedom. Thus, it is the programmer's responsibility to express the existence of dependencies between tasks.

RS for heterogeneous architectures are even more complex [2, 33, 40, 60], as they also have to deal with accelerator management, data integrity and device

synchronization. Hence, these systems have to manage data allocations, data transfers, kernel launches and synchronization points. To ensure the correct application execution, the RS builds up a DAG that represents dependences between tasks. Thus, all independent tasks can be run in parallel. Often, they also integrate workload partition strategies to divide the application according to performance models that propose a work distribution for a given hardware.

1.4. Thesis Motivation

The main problem we address in this thesis is the automatic orchestration and mapping applications coded with high-level parallel patterns when executed on heterogeneous architectures. This is an arduous problem that remains unsolved and proposing a solution to it is further challenging. Even more, if we consider irregular applications that may require different thread orchestration mechanisms depending on the application input data-set and the underlying hardware processors.

In heterogeneous computing, it is expected that applications are divided into tasks and executed on all processing devices (e.g. CPU and GPUs) where the best features of both can be combined to achieve further performance gains or energy savings. Early RS possess an offline training phase to learn how the application's workload should be divided into subtasks [21, 30, 45, 70, 86]. These implementations use a number of machine learning methods and small profiling runs to build performance models that predict the workload partition according to the data-set size. However, these systems may fail in their predictions when; (1) they are used to execute regular applications with bigger data input sets, as they have to extrapolate their initial predictions; (2) they execute irregular applications, as they can potentially exhibit different computation demands at runtime.

More recently, there is a body of RS that take the workload partition decision during application's execution time [3, 4, 58, 84, 97, 111]. In general, these systems have an *online profiling phase* which runs a small portion of the application workload on each available processor. Later, the relative computing speed of each processor is recorded and used to distribute the remaining workload accordingly. Although, these systems exhibit a good performance when executing regular applications, as dense matrix multiplications or Nbody simulations, they may fail when the profiling phase does not have the same computation demands than the remaining workload, and thus the remaining workload partition could be biased. This thesis is motivated by the fact that existing RS cannot easily

cope with irregular applications. In this sense, a number of dynamic approaches are proposed to optimize performance and energy consumption on heterogeneous architectures when targeting this type of applications.

1.5. Thesis Objectives and Research Question

Despite the number of works introduced in the previous section, the design and development of RS for heterogeneous architectures is a research area with ample room for improvement. As explained in the previous section, previous approaches only profile a portion of the workload to reduce overhead. These approaches behaves well with regular applications. However, they may not find the best partition when executing irregular applications, specially on heterogeneous architectures. In any case, in this thesis we aim at bridging the performance-productivity gap, thus we tackle the design and development of RS for heterogeneous architectures in the context of high-level parallel pattern templates. For this reasons along with the introduced complexities in section 1.2, the research question to be answered in this Ph.D. thesis is the following:

Is it possible to develop a generic Runtime System for high-level parallel pattern templates, aimed at heterogeneous systems, such as: (1) it provides a load balance mechanism which reduces resource under-utilization; (2) it offers a partition method which finds the ideal task size for each computing device within the hardware layer; (3) it considers performance and energy trade-offs as part of the criteria to perform the task-device mapping; and (4) it introduces a minimum overhead?

In order to answer this research question, and taking into account that we target heterogeneous systems that consist of CPU multicores and one or more GPUs, the following actions have to be fulfilled during this Ph.D.:

1. Analyse inefficiencies and sources of overhead when splitting, mapping and scheduling workloads from regular and irregular applications that are expressed through high-level parallel pattern templates.

2. Understand application memory access patterns and its impact in the timespan and energy consumption. In particular, analyse hardware performance counters for the CPU and GPU activities as well as memory-related metrics to gain insights on potential bottlenecks in our target heterogeneous architectures.

3. Design performance and energy models to capture and predict observed

behaviours when executing high-level parallel patterns on the heterogeneous architectures of interest.

4. Design new optimal approaches to dynamically split, map, and schedule workloads over heterogeneous computing devices, considering both performance and energy trade-offs for the high-level parallel pattern templates studied.

1.6. Thesis Contributions

This thesis is supported by several conferences and journals contributions in the area of parallel computing. Among all of them, the main contributions of this thesis are summarised in the following points:

- A *scheduling strategy* for heterogeneous systems is proposed as an extension of the TBB strategy for the *parallel for* pattern template [89]. Previous heterogeneous tasks frameworks, such as StarPU [2] or SnuCL [60], aim to assign each task to the computing device that offers the shortest execution time based on their performance models. However, this decision may result in load unbalanced scenarios because it could overfeed some computing devices while others are idle, specially in the case of irregular applications. In contrast, our proposed strategy may turn off some computing devices when they are not needed. For instance, a better overall performance can be yield by deactivating the accelerator, or the CPU cores.

- A *performance-aware partitioner* for *parallel for* pattern template, which is based on a logarithmic model that dynamically finds the task size that is optimal for each accelerator. Previous partition mechanisms consider that over a certain threshold size, there exists a linear relation between the size of the offloaded computation to the accelerator and its timespan. However, there are many applications that exhibit different performance behaviours such as, sparse matrix-vector multiplications or graph traversals which exhibit a random memory access pattern. Furthermore, we have found that offloading a large task to an accelerator may be counter-productive while executing irregular applications. This is due to the fact that threads within a group (or warp) in *SIMT* architectures could not access coalesced memory positions, so the memory system might not cope with the high demand of data requests. Thus, higher demand due to higher task sizes will worsen memory throughput. As a result of this, accelerator's compute units will

be stalled due to a bottleneck in the memory systems. In contrast, our proposed strategy firstly finds an initial task size that fully uses the compute units of the accelerator. Later, it keeps monitoring throughput during execution time in order to react and alleviate the memory data transient. Our strategy uses an heuristic to keep the system working around a near optimal point and avoiding overfeeding the accelerator with unnecessary large task sizes. The heuristic also finds the optimal size for the CPU cores.

- A *performance model* for streaming applications implemented using the *pipeline* pattern template [89]. Streaming applications, such as face recognition or objects tracking are good candidates to run on heterogeneous systems. Previous works [104, 48] propose a producer-consumer strategy that divides the application in two sets of stages. The first group of stages is run in one device and its output is passed to the second group of stages that is run in the other computational device. This approach works well when both group of stages are balanced, it means both group of stages work with a similar throughput. However, they fail when both set of stages are unbalanced, and the overall throughput is limited by the slower set. On the other hand, we propose a strategy that looks for a better solution. Our search space for the best configuration has three variables, the number of threads, the grain size level of the tasks and the device to map each stage on. Thus, we perform a small profiling phase which collects running times and hardware counters while running a few items on the CPU multicore and then on the GPU. As a result, we use this collected data with an analytical model to predict the best possible configuration of the variables for this application and hardware. Our approach is specially favourable, as it is a self-tuned method, whereas previous work need to be setted up manually.

- An *energy consumption* model for streaming applications based on the *pipeline* pattern template, which predict the overall system consumption while executing those applications on heterogeneous processors. This model also requires the same profiling executions as the performance model to estimate the right values of energy consumption.

- An *analysis of overhead sources* is carried out to show performance behaviour differences while executing tasks on a heterogeneous CPU-GPU system in the presence of oversubscription.

The aforementioned contributions have been published in international peer reviewed conferences [80, 92, 93, 107, 109], international workshops [23], and journals [81, 108] ranked by the ISI Journal Citation Report (JCR).

1.7. Thesis Structure

The rest of this thesis is organized in the following way:

- Chapter 2 motivates the need of Runtime Systems for heterogeneous architectures. Then, we present an overview of existing Runtime Systems in literature, and highlight the challenges that remain unsolved.
- Chapter 3 tackles the load balancing problem in the context of heterogeneous multi-CPUs and multi-GPUs systems when considering the *parallel for* pattern template. First, we propose an analytical model and two heuristic functions to deal with the load balance problem. Later, we propose two scheduling strategies that are built on top of the previous model to maximise the use of the GPU host threads.
- Chapter 4 extends the load balancing technique introduced in chapter 3 by adding *a performance-sensitive partitioner*. It considers varying the task size offloaded to the accelerator in order to make the most of it. We identify several issues while executing irregular applications and propose a method to mitigate them.
- Chapter 5 describes the complexities that arise while executing streaming applications based on the *pipeline* pattern template on heterogeneous chips. We define a taxonomy to classify the parallelism degree and possible mappings. Furthermore, we present an analytical model to predict the best possible configuration among all possible in terms of performance and/or energy consumption, and design a dynamic approach to efficiently map the application pipeline stages over the computing devices when performance or energy or any performance-energy metric is considered.

Experimental results are discussed within the scope of each chapter to avoid detracting attention from the overall readability. The last chapter shows the thesis conclusions and potential lines for future work.

2 Background and Related Work

In this chapter, we present a background on Heterogeneous Computing (HC). This area aims to match the requirements of each application to the strengths of the available heterogeneous architecture, by making an effective use of the accelerator and by avoiding idle use of the available processors [77]. These heterogeneous architectures are becoming de facto standard due to their higher performance capabilities and their efficient energy consumption which is key in battery powered devices.

As a consequence, there has been an explosion in the number of different architectures comprised of CPUs and accelerators (GPUs, FPGAs, DSPs). This kind of architectures are ubiquitous, as they are present from large Computing systems, like the top positions of the TOP500 list [87], to embedded and portable devices such as smart-phones and smart-watches. However, these architectures come with several unsolved challenges. Due to the processor architectural differences in a heterogeneous system, developers need to take into account the characteristics of the different computing devices types and consider that traditional optimizations may not be feasible. Thus, novel techniques are required to exploit the potential of heterogeneous architectures and also move toward Exascale Computing.

Furthermore, the programmability of these systems is not trivial for the programmer who also needs to consider aspects such as different programming libraries and the existence of separated memory address spaces. In this thesis we aim for bridging the performance-productivity gap, thus we first analyse the most widely used parallel patterns 2.1. Later, we cover the evolution of processors designs and the advent of heterogeneous architectures 2.2, specifically this

last trend has also forced the software community to develop new programming models for those heterogeneous architectures 2.3. Finally, this chapter gathers the existing runtimes proposals that are related to the use of parallel patterns on heterogeneous architectures 2.4.

2.1. Parallel patterns

During the last decades, software patterns have gained a large adoption in the software community as a way of producing high quality code while keeping productivity. They represent the best practices for software development [37]. In this thesis, we particularly focus on parallel pattern templates and their usage on heterogeneous architectures. In this section, we describe the most widely used parallel patterns in detail, we highlight their functional assumptions, as they are key to understand the issues of accelerating serial applications based on these patterns. Parallel patterns are related to serial patterns but relax their assumptions in several ways.

The **fork–join** pattern forks the control flow of an application into several concurrent flows that join later. Some implementations, like OpenMP or Cilk, fork the control flow into multiples threads that may execute a different amount of operations over a range of data. Once each control flow has finished its assigned body of operations, they synchronize in the join point, until all control flows reach this point. Later, only one control flow continues running with the following operation after the join barrier. Note that each flow can perform a different amount of operations, thus the join function is a requirement to ensure

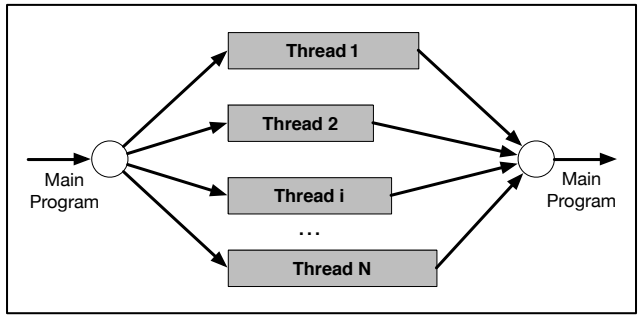


Figure 2.1: Thread workflow of the *fork-join* pattern with n-threads.

the dependency flow [73]. Figure 2.1 shows a running example of a fork-join pattern, where N threads are forked in parallel from the main program thread. Later, each thread executes its work, note that the amount of work of each thread may be different. Finally, each thread synchronise until all threads reach the join function, and only one thread continues executing the main program thread.

The **Map** pattern applies a *function operator* over a range of elements within a collection. In general, the map pattern mimics the loop iteration in serial programs. Specifically, loops in which all iterations are independent and the number of iterations is fixed and known. This requirement means that the computation performed in each iteration can not depend on other iterations results in oorder to ensure correctness. This pattern is often implemented under the name **parallel_for**. Many threading libraries, like OpenMP, TBB and Cilk offer an implementation for this pattern [73]. In Chapters 3 and 4, we extend the TBB's `parallel_for` pattern template and develop a load balancing model to allow its efficient usage on heterogeneous platforms. Figure 2.2 shows an example where a collection of eight elements is computed in parallel. Each element within the input collection is processed with the same function, and produce and output collection of eight elements.

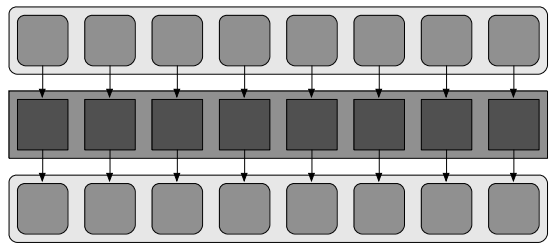


Figure 2.2: Usage example of the parallel *map* pattern with 8-element collections.

The **Stencil pattern** is a specialisation of the map pattern in which a *function operator* is applied to all elements within a collection. Specifically, it applies a function to all element based on the values of a set of *neighbours* elements. This pattern is usually used on matrix structured data, where optimizations like tiling and matrix linearisation can be applied. For the stencil pattern, boundary conditions on element accesses need to be carefully considered. The edges of the matrix require a special handling either by avoiding the indexing for out-of-bounds accesses or by using padding techniques [73]. Figure 2.3 illustrates how each element is computed based on a range of neighbours, horizontal and vertical neighbours within distance≤ 2 in this example. In general, the stencil pattern is

widely used for image filtering, tracking and segmentation. This pattern is used in some of the benchmarks described in Chapter 5.

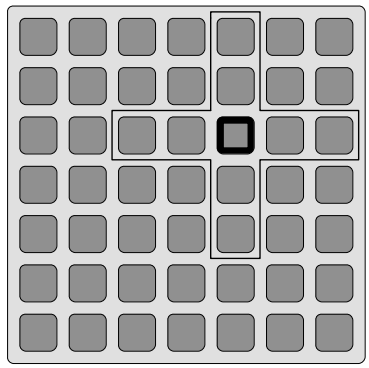


Figure 2.3: Neighbour dependencies when using the *stencil* pattern.

The **reduction** pattern combines all elements within a collection into a single element by using an associative function. Thus, given the associativity of the function operator, many different orderings are possible, but with different spans. If it happens that the function is also commutative, even more orderings are possible. Thus, given n elements in a collection, applying the function operator to any two adjacent elements, we reduce it to $n-1$ elements. This process can be computed in parallel to every pair of adjacent numbers. Thus we reduce the number of elements to its half in each phase of the reduction. We can keep reducing the number of elements until there is only one element. This patterns reduces the amount of available parallelism by two in each iteration of the reduction [73]. The Figure 2.4 shows the reduction of a eight-element array, which reduces the time complexity from $O(N)$ to $O(\log_2 N)$. The reduction pattern is widely used in codes that summarise data like *histograms*.

The **Scan** pattern produces all partial reductions of an input collection. There are two variants: inclusive and exclusive scan. In this point, we explain the inclusive scan, where the nth output element is a reduction over the first n input elements. On the contrary, the exclusive scan performs a reduction on the nth over the first $n-1$ elements. Despite the loop-carried dependence of the inclusive scan pattern, it can be run in parallel. We can perform an ordering similar to the reduction one, we can use the associativity of the reductive function to vary the serial order. However, making scan to run in parallel has a computational cost, because we have to recompute the reduction of some partial reductions.

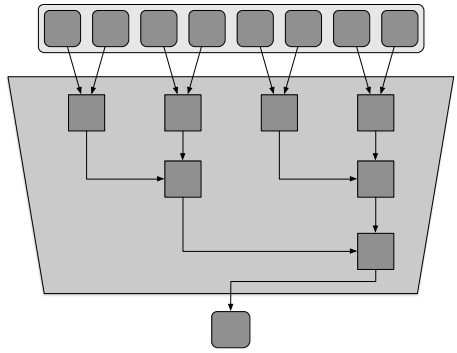


Figure 2.4: The parallel *reduction* pattern.

Thus, we can reduce the time complexity of the code from $O(N)$ to $O(\log_2 N)$ by increasing the amount of calculations [73]. Figure 2.5 shows an inclusive scan example with a collection of eight elements, that requires 3 steps, although the number of reductions operations increases from eight to eleven. This pattern can be used to compute agregations in time-windows based computations.

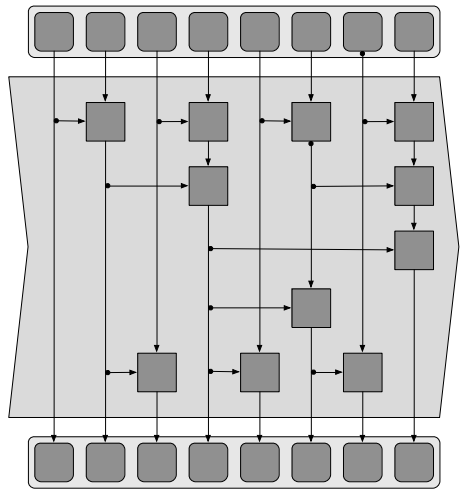


Figure 2.5: The parallel *scan* pattern.

The **Pipeline** pattern that mimics a manufacturing assembly line, where data

flows through several computational stages that process pieces of data producing changes on them. Conceptually, all stages of the pipeline are active, and each stage can keep an internal state that can be updated while data flows through them. The stages that keep an internal state are serial and only can compute a piece of data at a time. On the contrary parallel stages can process many pieces of data at once. Given a pipeline with several stages, the throughput of the slowest serial stage is the one that limits performance. Thus, pipelines are useful for exploiting parallelism in the presence of several serial stages, where the parallelism is reached by executing several pieces of data in different stages at a given time. This kind of parallelism is exploited in applications like video codecs and image processing [73] . Figure 2.6 shows a pipeline with three stages (dark squares) that receives data inputs and produces processed data as output (grey rounded squares). In Chapter 5, we extend the TBB `pipeline` pattern template to be exploited on heterogeneous architectures, and provide a model to predict the best pipeline configuration.

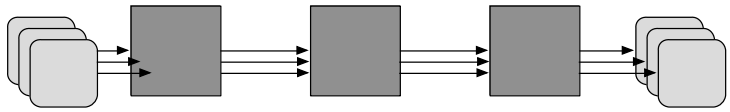


Figure 2.6: The parallel *pipeline* pattern with non serial stages.

Throughout this thesis, we focus on the optimisation of **parallel_for** and **pipeline** patterns. We develop extensions for the templates that implement these patterns to allow their deployment on heterogeneous architectures. Additionally, we analyse the performance limiting factors along with the energy-consumption asymmetries for several heterogeneous architectures.

2.2. Heterogeneous Computing Basics

Parallel computers have been traditionally dedicated to process large amounts of data in big companies, research labs and governmental departments. However, recent trends have led to a new spectrum of devices specially designed for the masses (PCs, and mobile devices). In this section, we dive into hardware architecture evolution during last years and elaborate on the need for parallel programming models for them.

2.2.1. Hardware evolution

As Gordon E. Moore predicted around fifty years ago, in 1965, the number of transistor within a chip is still doubling with each technology generation every two years [78]. Figure 2.7 shows the evolution of several processor features for the last 30 years. The transistor count almost follows a straight line on a logarithmic scale (y-axis). This demonstrates the exponential increment in the number of transistors. In this sense, computer users expect that this increment in transistor count directly translates to a performance improvement. Nevertheless, performance strongly depends on others factors but transistor count.

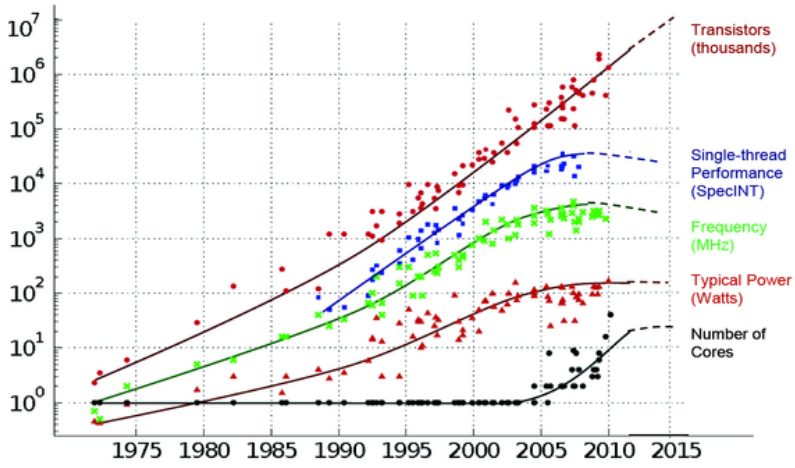


Figure 2.7: Phisycal processor features trends for the last 30 years (The authors of this plot are M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, C. Balten and C. Moore).

As we can see in Figure 2.7, processor clock speed has increased from 1 MHz (in 1973) to 1 GHz (in 2000). It represents an increment of three orders of magnitude. However, as we can see in the graph, the frequency increment has ceased, and now it is stabilised around 3 GHz. In 2005, three factors were limiting performance scaling on processor designs [15]:

- The **Power wall** is a consequence of the non-linear relation between power consumption and the increment of frequency. During the last several years, two key processor design parameters, such as the voltage supply and fre-

quency, have stopped performance scaling. This fact was due to processors with unsustainable power densities. Previous processor generations scaled down the voltage within each new processor generation and increased frequency as a way to get more performance and keeping stabilised the total power consumption. However, due to physical limitations in current processor technology, nowadays the scaling of frequency does not yield a proportional increment of performance (known as the Dennard scaling failure [6]) and exacerbates the power density problem.

- The **ILP wall** reflects that traditional techniques to automatically extract low-level parallelism have reached their limit. Hardware is naturally parallel, and processors usually have a series of mechanisms (pipeline, branch predictors, superscalar instruction issuer ...) to extract the available parallelism from a serial stream of instructions. The illusion of writing serial codes while ignoring the parallel execution does not scales anymore. Figure 2.7 shows that performance of single thread applications are not expected to increase in the upcoming years due to frequency and ILP decrease.

- The **Memory wall** is mainly produced for memory bound codes, this kind of codes exhibit a higher number of memory operations (reads and stores) than arithmetic operations. There are two main factors that affect memory operations: bandwidth (rate at which data is read or stored) and latency (the time between a memory operation request and the time when it is resolved). Bandwidth is scaled with each technology generation, however latency is increased and it is starting to became a limiting factor.

These factors along with the diminishing returns from single-thread architectural improvements, have forced processor designers to shift to multicore and manycores solutions in order to make an effective use of the large number of transistors available. Thus, with the advent of heterogeneous architectures, the current trend in processor design is clear, with each new processor generation more and more core count is available in hardware. Eventually, these cores will be specifically designed for specific computational tasks (signal processing), video processing units, floats units, etc. This specific cores will allow application developers to have a huge amount of available computing power at a low power consumption. However, with the shift to heterogeneous architectures, processor manufacturers have put a big burden on software developers side. Unlike increasing clock frequency, serial applications' performance will not vary without the usage of new programming libraries. The *free lunch* of automatically faster serial applications running on faster processors is over. From now on, program-

mers are responsible to orchestrate and map the application level parallelism onto the hardware level by the usage of explicit parallel programming models. In this case, to develop applications for heterogeneous architectures, we require the use of low-level programming libraries such as CUDA [96] or OpenCL [38]. These libraries expose architectural accelerator details that hinder the development of applications on top of heterogeneous architectures. Researches agree that new programming models are required to offer an abstraction layer that hides the aforementioned complexities while making the most of these promising platforms. In next section, we explain the limitation of the current programming models and identify the future challenges that needs to be solved.

2.3. Programming heterogeneous architectures

In this section, we discuss about the need of new programming models that allows the effective use of multicores and accelerators. These programming models should provide a set of abstractions to reduce the programmability wall and bring productivity to developers worried about performance.

2.3.1. The need of heterogeneous programming models

As a result of the quick evolution in hardware designs, the amount of available heterogeneous architectures has increased [9]. However, many modern programming languages, such as Java, Python, C# and C++ do not offer a direct support for these kind of architectures. The research community has proposed several libraries to offer support for heterogeneous architectures, Java [36, 98], Python [61], C# [83] and C++ [38, 58, 89]. Currently, there are two main programming languages for heterogeneous architectures, CUDA [96], which only runs on NVidia GPUs and OpenCL (Open Computing Language) [38], which is the first standard for parallel programming on heterogeneous architectures. It was first released by Apple in 2008 and developed by the Khronos Group. It has gained a wide support from many hardware vendors such as, Intel, Samsung, Qualcomm, NVidia, ARM and Xilinx among others. It also represents the most significant effort to create a common programming interface for heterogeneous devices (CPUs, GPUs, DSPs, FPGAs and Xeon Phi). However, the main drawback of CUDA and OpenCL is their low level programming model and complexities due to the explicit accelerator management. Developers are responsible for allocating/deallocating accelerator's memory, ensure data integrity among different memory address spaces, launch kernels and synchronise devices.

To alleviate this programming wall inherent to heterogeneous programming, the HSA foundation (Heterogeneous System Architecture) [49] is building a software ecosystem to abstract away the complexities of heterogeneous architectures. They rely on a runtime and a system API that runs on SOCs with cache coherent memory. In this line, the Khronos Group has also published the specification of SYCL [95], which enables code for heterogeneous processors to be written in a completely standard C++. SYCL enables single source development where C++ template functions contain both host and device code to construct complex algorithms that use OpenCL acceleration underneath. Others approaches like OpenACC [91], OpenMP [32] and OmpSs [10] offer a straightforward API to offload computation to accelerators. These models are based in a set of functions and compiler directives, where developers set a pragma directive before the block of code to be executed on the accelerator by specifying the range of computation and memory buffers to be used. Then the runtime is responsible to orchestrate the parallelism and map it to the underlying accelerator. Thus, developers do not need to explicitly manage the accelerator, transfer data between the host and accelerator, or synchronise after the kernel is launched.

Although, all aforementioned approaches allows the execution on heterogeneous architectures and aim at simplifying the computation on this kind of architectures. There are still some challenges that remain open. In next sections, we take a look at the next challenges that heterogeneous computing has to face: *portability of code* and *performance portability*.

2.3.2. Code Portability

Programming for heterogeneous architectures is a tedious and complex task, developers are frequently forced to use platform dependent libraries or language extensions while developing their applications. Thus, on one hand, there are approaches such as, NVidia CUDA [96], Xilinx Vivado [13] and Qualcomm Symphony [53] which provide a private toolchain based on a restricted C++ language that only work on their own platforms, limiting the portability of code among different accelerators and vendors. On the other hand, OpenCL was designed as a standard library for heterogeneous computing. The main idea behind OpenCL is to provide a common layer for a wide range of accelerators (multicore CPUs, GPUs, DSPs, FPGAs and other processor accelerators). However, many of them include specific extensions out of the standard that limit the portability of code among devices.

To overcome this compatibility and portability issues, the Khronos Group re-

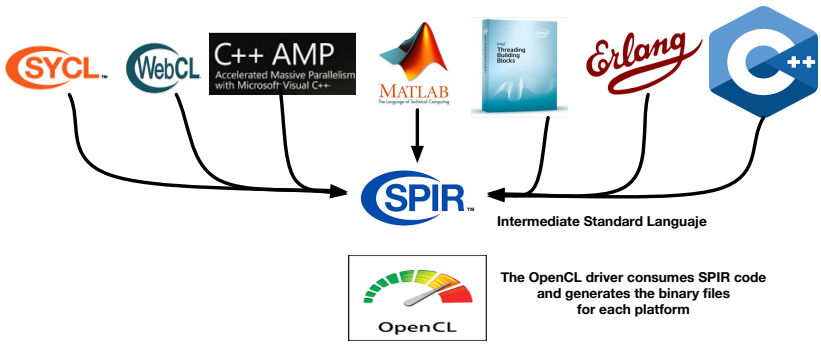


Figure 2.8: SPIR-V separates high-level language processing from binary code production.

leased SPIR-V [46], a Standard Portable Intermediate Representation. SPIR-V is the first standard intermediate language for representing computation (OpenCL) and graphics (Vulkan). Moreover, it has been incorporated as part of the OpenCL 2.1 core specification. Thus, all vendor implementations for OpenCL 2.1 have to be based on SPIR-V. We are currently witnessing a revolution in the compiler ecosystem for heterogeneous architectures, as SPIR-V allows us to separate the generation of IR code from the generation of binaries. It enables the development of a wide range of language front-ends to produce programs in a common and standardised intermediate language (SPIR-V). Figure 2.16 shows how SPIR-V avoids a monolithic compiler approach. In this manner, hardware vendors can avoid the development of high-level language source compiler into device drivers, reducing driver development complexities, and enabling the proliferation of a wide range of languages front-ends to run on heterogeneous architectures.

For application developers, SPIR-V also offers some advantages as it ensures that their applications will run on all SPIR-V compliant devices, avoiding platform dependent issues. Another advantage, is that the source code of OpenCL accelerated applications does not have to be directly exposed to customers, allowing IP protection as they can deploy SPIR-V code within their CPU binaries. In this sense, Figure 2.17 shows how the SPIR-V code is consumed by the OpenCL implementation when `clCreateProgramWithIL()` and `clBuildProgram()` are invoked. First, a set of standard and custom optimisation are carried out on the incoming SPIR-V code, producing a highly optimised SPIR-V code. Later, the SPIR-V code is translated into the specific platform IR and some additional low-level optimisations are performed. Finally, the JIT compiler produces the ex-

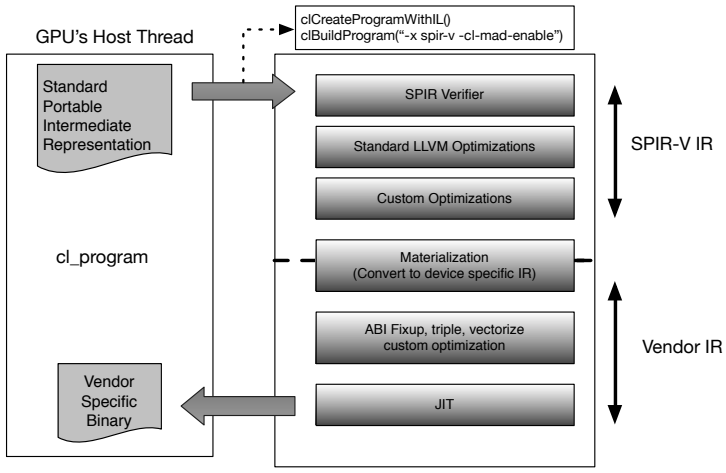


Figure 2.9: SPIR-V compilation process into binary code (based on Ayal Zaks explanation at ACACES'14).

ecutable binary file, and returns it to the user application under a `cl_program` object.

2.3.3. Task-based models

Although OpenCL offers a portable solution for developing platform portable applications for heterogenous platforms, it still does not allow the orchestration and mapping of simultaneous parallel computations over the different devices in the system. Therefore, developers must still tackle this issue. We belive that developers should use high-level abstractions to identify the sources of parallelism, while the compiler and runtime should take care of the mapping of tasks and data over the available hardware resources. In this context, we think that the *task* abstraction model is the right choice for the kind heterogenous architectures that we target in this work (multicore CPUs and one or more GPUs). There are several reasons to use task-based models: (i) they introduce less overhead because they do not deal with explicit threads; (ii) they support optional parallelism given by a distributed load balance mechanism; and, (iii) they support composability and nested parallelism, that allow the development of parallel libraries without exposing internal implementation details [66].

There is a number of libraries and languages that support a model based on tasks, such as: Intel Threading building Blocks (TBB) [89], OpenMP 3.0 [32], Cilk [7], Java 8, Microsoft TPL (Task Parallel Library) and Habanero [71], to name a few. These libraries and languages incorporate an internal engine to distribute the tasks created at runtime and a scheduling policy to assign those task to the underlying running threads.

Next, we explain in detail the Intel TBB library, because it is used as the core engine of our proposed constructs: `parallel_for` and `pipeline`.

Threading Building Blocks

We extensively use TBB through the course of this thesis. It is a C++ library based on a tasking model that provides a set of functions and templates for building parallel applications that run on multicore CPUs. TBB is not limited to Intel CPUs, as it can be executed on ARM based processors as the Qualcomm Snapdragon 800 or the Samsung Exynos. It can also be executed in several operating systems such as Windows, Linux and Mac OS, as long as they include an ISO C++ compiler like GCC, ICC or Clang.

The TBB programming environment encourages developers to express applications in terms of tasks rather than threads. Tasks are functors that can be run in parallel by the TBB runtime when there are more than one available thread. This library allows users to develop portable and scalable applications with the following benefits:

i) TBB enables developers to specify tasks instead of threads, thus developers can avoid the tedious task of directly managing low-level implementation details that makes applications platform dependent.

ii) TBB is also compatible with other threading packages, so it does not limit developers to use an unique threading library. Moreover, this feature provides compatibility with legacy code.

iii) TBB emphasises data-parallel programming, enabling multiple threads to collaborate in the execution of a given parallel workload (in general data-parallel programming scales when increasing the number of processors, as the data is divided into smaller pieces).

iv) TBB relies on generic programming, it is implemented by using interfaces of generic types that can be instantiated by user defined types to adapt the library to the developer needs.

As mentioned before, the performance improvement that TBB provides with respect to other traditional threading libraries (Pthreads, OpenMP) is due to the efficient use of tasks. However, TBB is also faster than other task based libraries thanks to its internal task scheduler which is described in next Section.

Task Scheduler

In TBB applications, tasks are implemented by using a C++ class that extends the `tbb:task` class. This class provides an `execute()` virtual method, which is required to be implemented in the extended class. This method is used to express the implementation of the task. Once a task class has been implemented and instantiated, it is ready to be launched and executed by the TBB's runtime. In TBB, the most basic method for launching a new task is through the invocation of the `spawn(task *t)` method, which receives a pointer to a task class as its unique argument. After invoking this method, the new spawned task gets enqueued in the thread's queue, and the current task continues running, as it is a non-blocking method.

Moreover, the `task` class provides blocking methods as `spawn_and_wait()` and `spawn_and_wait_for_all()` to spawn new tasks and wait for their finalisation. In this manner, the user has to manage the number of spawned child tasks by updating the `ref_count` parameter. Once a task is scheduled for execution by the runtime library, the `execute()` method of the task is invoked in a non-preemptive manner, completing the execution of the task. Tasks are allowed to instantiate and spawn additional tasks by allowing the formation of a direct acyclic graph (DAG), see Figure 2.10. When a task finishes its `execute()` method, it decrements its parent reference counter (`ref_count`) and destroys itself. TBB provides two further mechanisms to enhance task management when possible. In many scenarios with lightweight tasks, task allocation/deallocation operations may introduce a noticeable overhead. Thus to mitigate this overhead TBB provides a *Recycling* mechanism that is used when a task invokes the `recycle_as_continuation()` method. The effects of this method is that the current task is recycled into a child task, by doing that we avoid the allocation/deallocation overheads and also reduce the amount of memory required at runtime. The second mechanism is called *bypass*, it allows developers to specify which is the following task to be executed. Thus the runtime does not have to execute its code to choose the next task, and the following task can start its execution as soon as the current task finishes its execution.

The task scheduler is responsible for evaluating the task graph (see Figure 2.10) and executing tasks in a manner that minimises both memory demands and thread communication [89]. To achieve this, it finds a balance between depth-

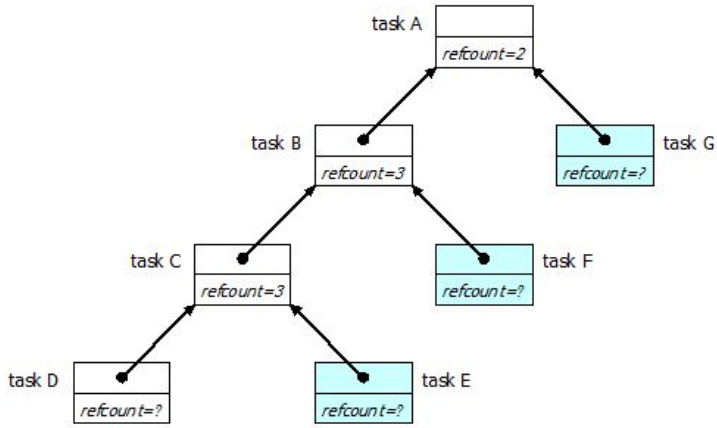


Figure 2.10: Task's dependence graph (DAG) built from a fibonacci TBB code, based on Intel Threading Building Blocks site.

first and breadth-first explorations. On one hand, depth-first approach is best for sequential executions as it reduces the amount of memory demands, as it only spawns a linear number of nodes at a time. Moreover, this approach also takes advantages of the cache memory because the deepest tasks are the most recently spawned tasks, and the hottest in cache. On the other hand, although breadth-first approach has a severe problem with memory consumption as expand the number of spawned tasks, it maximises the available parallelism. Because the number of physical threads is limited, TBB only uses breadth-first to raise enough parallelism to keep all available physical threads effectively working.

The main feature of TBB is the dynamic task scheduler that orchestrates the available application parallelism across the available threads in order to yield further performance scaling. The TBB's task scheduler is initialised by instantiating the `tbb::task_scheduler_init` class, which creates a set of logical threads equal to the number of physical threads by default, however the user can also set the required number of threads by specifying an integer value. Once a worker thread is created, it allocates a local task queue and invokes the method `wait_for_all()`, which implements the TBB's task scheduler, see Figure 2.11. Dequeuing tasks is implicit and carried out by the runtime system.

The main scheduling loop of the TBB runtime is shown in Figure 2.11, it is implemented in the `wait_for_all()` method. It comprises three nested loops

that attempt to obtain work through three different ways: *explicit task passing*, *local task dequeue* and *random task stealing*. The inner loop of the scheduler is responsible for executing the current task by calling the method `execute()` (lines 5-11). Once this method is executed, the *reference count* of the current task's parent is decremented (line 7). This *reference count* allows the parent task to execute once all its children tasks have completed. If this *reference count* reaches one (line 8), the parent task is set as the next task to be executed (line 9). The method `execute()` has the option of returning a pointer to the next task to be executed (*bypass*, line 6). If a `new_task` is not returned, the inner loop exits. Then the middle loop attempts to extract a task pointer from the local task queue (line 4-13), by calling the method `get_task_from_queue())` (line 12). If successful, the middle loop iterates calling the inner loop once again. If `get_task_from_queue())` is unsuccessful, the middle loop ends and the outer loop attempts to steal a task from other existing worker threads (lines 3-17). If the steal is unsuccessful, the worker thread waits for a certain amount of time and attempts to steal a task from a randomly chosen worker thread until all task are computed.

```
1  void wait_for_all(task *child) {
2    task* next_task, task = child;
3    while (task->parent() != NULL){
4      do{
5        while (task){
6          next_task = task->execute();
7          task->parent()->set_ref_count(task->parent()->ref_count()-1);
8          if (task->parent()->ref_count()==1)
9            next_task = task->parent();
10         task = next_task;
11       }
12       task = get_task_from_queue());
13     }while (task);
14     task = steal_task(random());
15     if (task == NULL ) //steal unsuccessful
16       wait(time);
17   }
18 }
```

Figure 2.11: Main skeleton of Intel TBB's scheduler.

As mentioned before, TBB provides a set of high-level templates that are built on top of TBB's task scheduler and allow developers to express their applications without worrying on parallel programming complexities. Through the course of this thesis, we propose several extensions of TBB templates (`parallel_for` and `pipeline`) to allow their execution on heterogeneous architectures while ensuring an effective use of the accelerators. In next section, we deeply explain the details of the TBB templates that are extended in Chapters 3, 4 and 5.

Parallel_for template

Many HPC and user oriented applications are good candidates for exploiting loop level parallelism. In this sense, the library Intel TBB provides a function template, `tbb::parallel_for` to execute `for` loops with independent iterations in parallel. This function template is designed as a C++ STL function template, it receives two parameters, a `Range` object which represents the iteration space with a duple `(begin, end)`; and a user defined `Body` functor which specifies the body of the loop. This abstraction allows developers to express their applications as sequential functors, as it makes easier the development process and avoids users to explicitly handle threads and their synchronization. Thus, these functors are automatically run in parallel when more that one thread is allowed. Note that the execution order of the iterations is not guaranteed, as the internal TBB's scheduler may change the execution order to yield further performance.

Figure 2.12 shows a vector addition example where the `tbb::parallel_for` function template is applied. First, the user is required to include two header files to make the `tbb::parallel_for` function and the class `blocked_range` available (lines 1 and 2). Next, the user has to define a STL functor to compute the addition of two vectors, the class `AddVector` (lines 4-12). This class implements an `operator()` method which receives a `blocked_range` parameter and executes the body of the loop for this range of iterations (`out[i] = a[i] + b[i]`, being $i < n$). Finally, the user has to create an instance of the previous class (line 18) and invoke the `tbb::parallel_for` function template by passing as arguments a valid range of iterations (`(0, n)` in line 19) and the aforementioned functor. The internal TBB's scheduler is responsible for splitting the whole iteration space into chunks of iterations and distributing these chunks across the available threads, thus the users can focus on developing their applications by using a sequential STL like approach.

The internal TBB's engine scheduler recursively splits the iteration space (`(0, n)` in the example) and spawns new tasks with each partition. Figure 2.13 shows a simplified algorithm of the internal `parallel_for` function template implementation. Each task firstly declares a splitter object (`split_obj` in line 3), this object is responsible for splitting the sub-range of iterations assigned to the given task. There are several splitting policies available in TBB and they depend on the selected TBB's partitioner (blocked, guided, uniform, etc ...). Next, it executes the while loop until the range is not further divisible (line 4), in each iteration a new sibling task is spawned with its corresponding range of iterations (line 5). Once, the size of the range of iterations can not be further divided, the task exits the loop and executes the `operator()` function over its range of iterations.

```
#include ``tbb/parallel_for.h''
#include ``tbb/blocked_range.h''

class AddVector{
  const float *a, *b;
  float * out;
  void operator()(const blocked_range<int> &range) const{
    for(int i=range.begin(); i!=range.end(); i++){
      out[i] = a[i] + b[i];
    }
  }
};

int main(int argc, char* argv[]){
  // Setting up NTHREADS
  tbb::task_scheduler_init init(NTHREADS)
  ...
  AddVector addv;
  tbb:parallel_for( blocked_range<int>(0,n), addv);
}
```

Figure 2.12: Usage example of the *parallel_for* construct in TBB.

```
template<typename StartType, typename Range>
void execute(StartType &start, Range &range) {
  split_type split_obj = split();
  while( range.is_divisible() )
    start.offer_work( split_obj );
  start.run_body( range );
}
```

Figure 2.13: Simplified `execute()` method of the `simple_partition_type` class.

All spawned tasks by the `root_task` are allocated on the main thread's task queue, they are ready to be executed after the current task or to be stolen and executed by other idle threads. The splitting process og the iteration space follows a lazy approach, it means that the execution of the tasks are delayed until each task can not be further divided, it is, they reach the minimum allowed size. In Chapters 3 and 4 we extend this `parallel_for` construct to be executed on heterogeneous architectures. However, we implement our extended scheduler by using an eager approach in order to adapt the chunk size of iterations to the relative speed of each device.

Pipeline template

The *pipeline* pattern is widely used when developers have an incoming stream of data that need to be processed in several steps or stages, as it happens in a

traditional manufacturing line. A classical thread-per-stage implementation has two main drawbacks: i) the speed-up does not scale beyond the number of stages and ii) when a thread finishes its computation, it needs to pass the resulting data to the following thread, so there is a communication overhead.

To alleviate these scaling issues, TBB proposes other approach. It offers a `pipeline` construct which allows developers to express their applications by exploiting a potential pipeline parallelism. In this implementation, data flow throughout a number of stages, and each stage performs some operations on the arriving data. These stages can be either serial or parallel, serial stages can only process a piece of data at a given time, while parallel stages can simultaneously process several pieces of independent data. These types of stages allow developers to express the behaviour of their applications without dealing with threading complexities. For example, in vision applications, some computations, as blurring effect, do not depend on other frames, so parallel stages can be applied. However, there are other operations, as object tracking, that need information from the previous token, so serial stages are required. With TBB, the user only has to express the stages that can be run concurrently, as it allows more opportunities for load balancing. Thus, with a certain number of CPU cores and concurrent computing opportunities, *the overall throughput of the pipeline is only limited by the execution time of the slowest serial stage.*

```
1  int main(int argc, char* argv[]){
2    // Setting up NTHREADS
3    tbb::task_scheduler_init init(NTHREADS)
4
5    //Declaring pipeline and filter or stages
6    tbb::pipeline pipe;
7    SegmentationFilter sf;
8    ExtractFilter ef;
9    RankFilter rf;
10
11   // Setting functions for each stage
12   pipe.add(sf);
13   pipe.add(ef);
14   pipe.add(rf);
15
16   // Running the pipeline with a maximum number of items in flight
17   pipe.run(numTokens);
18 }
```

Figure 2.14: Usage example of the *pipeline* construct in TBB.

The TBB's pipeline implementation relies on the classes `pipeline` and `filter`. A `filter` object represents one stage of the pipeline, while a pipeline object represents the whole pipeline itself, and it is comprised of one or more filter

objects. Figure 2.14 shows an usage example of the TBB's pipeline construct. As in any threading library, the first step is to initialise the library with the number of threads (in line 3). However, this initialisation is optional, if the user does not initialise the library, the runtime will do it with the default number of threads which is equal to the number of physical threads. Later, between lines 6-9, we declare a `pipeline` instance and the `filter` objects. Assuming that we have previously defined a class for each `filter` (SegmentationFilter, ExtractFilter and RankFilter, in our example). In the next step, we have to add the declared filters to the pipeline by respecting the order between them (lines 12-14). Finally, we have to invoke the method `pipeline::run()` to execute the pipeline (line 17). This method receives an integer parameter (`numTokens` in the figure 2.14) which sets the maximum number of "in flight" items in the pipeline. Thus, the pipeline ramps up the number of items until it reaches the number of `numTokens`, and then holds the "in flight" number of items by limiting the access of new items, as it does not create new input tokens until another item is destroyed at the output stage. It is critical to provide an accurate number of `tokens` because an unlimited number of tokens could lead to a parallel middle stage to keep gaining tokens while the serial output stage can not cope with this pace. Navarro et al. [79] provide a methodology to calculate the right number of items to achieve an optimal performance. Additionally, we also provide a methodology to calculate the optimal number of tokens in the presence of a heterogeneous (CPU-GPU) pipeline, in section 5.3.2.

```
1  task * stage_task::execute(){
2    if(first_stage){
3      if(my_pipeline.end_of_input) // No more data to compute
4          return NULL;
5      if(--my_pipeline.input_tokens > 0) // There are more available tokens
6          spawn(*new(allocate_child_of_(*parent()))stage_task(my_pipeline));
7    }
8    // Execute the filter
9    my_object = (*my_filter)(my_object);
10   // Update the pointer to next filter
11   my_filter = my_filter->next_filter_in_pipeline;
12
13   if(!my_filter){ //There are not more filters
14         my_pipeline.input_tokens++;
15         reset(); // // Recycle as an input stage task
16   }
17   recycle_as_continuation();
18   return this;
19 }
```

Figure 2.15: Simplified method `execute()` of the `stage_task` class.

When the user invokes the method `pipeline::run()`, TBB allocates and

spawns one `pipeline_root_task`. First, this `root_task` is initialised with the list of stages and recycled as a `stage_task` to proceed with the execution of the pipeline stages. It is a decentralised approach that spawns new tasks until a given number of tokens is reached (`numTokens`), Figure 2.15 shows the algorithm that drives the flow and execution of the stages within the pipeline. Thus, when a task is executed, it firstly checks if it has to execute the first stage (line 2). If the previous condition is satisfied it will check whether there are more input data to compute or not (line 3). If it happens that there is no more input data, then the current stage will exit and the number of tokens will start ramping down. On the contrary, if there are more input data and the number of available tokens is bigger than zero (`--my_pipeline.input_tokens>0`) then the current task will allocate and spawn a copy of itself which will be stored in its thread queue. Additionally, this spawned task can be potentially stolen and executed by any other idle thread thanks to the work-stealing mechanism. In any case, the current task executes the `filter::operator()` function given by the instance (`my_filter`), and updates the next filter to be executed, lines 9 and 11 respectively. Note, that the input parameter (`my_object`) of the `filter::operator()` function is overwritten by the output of the filter, as they are variables of the same type (`void *`). Next, we check whether there are more stages to be executed in line 13, if the last filter within the pipeline has been executed by the current task, then the current task will invoke the method `task_info::reset()`, which makes the variable `my_filter` point to the first stage in the pipeline (line 15). Finally, the current task is recycled as continuation (line 17), and it will execute the next filter in pipeline or the first one depending on the value of the variable `my_filter`.

The TBB's task recycling mechanism offers several performance advantages with respect to traditional approaches (Pthreads and OpenMP), as it is exposed by the authors in [22, 89]. This mechanism lets TBB's engine to reduce the task handling overhead, as it reduces the number of allocation/deallocation operations. Moreover, it allows the task to bypass the scheduler, by explicitly indicating to the task's thread what is the next task to be executed, thus we avoid the execution of the scheduler code and the continuation task can start as soon as its predecessor finishes. This mechanism also avoids thread communication overheads, as it reuses in-cache data from the previous task when possible, providing further performance improvements. For this reasons, we use and extend this optimised model to be exploit on heterogeneous architectures in Chapter 5.

To overcome this compatibility and portability issues, the Khronos Group released SPIR-V [46], a Standard Portable Intermediate Representation. SPIR-V is the first standard intermediate language for representing computation (OpenCL)

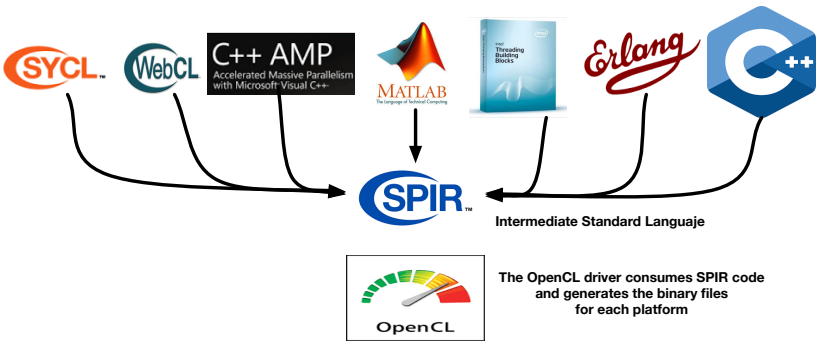


Figure 2.16: SPIR-V separates high-level language processing from binary code production.

and graphics (Vulkan). Moreover, it has been incorporated as part of the OpenCL 2.1 core specification. Thus, all vendor implementations for OpenCL 2.1 have to be based on SPIR-V. We are currently witnessing a revolution in the compiler ecosystem for heterogeneous architectures, as SPIR-V allows us to separate the generation of IR code from the generation of binaries. It enables the development of a wide range of language front-ends to produce programs in a common and standardised intermediate language (SPIR-V). Figure 2.16 shows how SPIR-V avoids a monolithic compiler approach. In this manner, hardware vendors can avoid the development of high-level language source compiler into device drivers, reducing driver development complexities, and enabling the proliferation of a wide range of languages front-ends to run on heterogeneous architectures.

2.3.4. Performance Portability

The next major challenge in heterogeneous computing is that accelerated applications achieve performance portability across a wide range of accelerators (CPUS, GPUs, Xeon Phi, FPGAs). While code portability across different hardware architectures is partially achievable, an optimized application optimised for one accelerator may not execute accordingly fast on a device from a different hardware vendor. Moreover, an optimised application for a given GPU is usually different from the one optimised for a multicore CPU. For these reasons, performance portability can be poor when optimised applications are executed on other target accelerators, being the average efficiency around 40%, according to the works [75, 65].

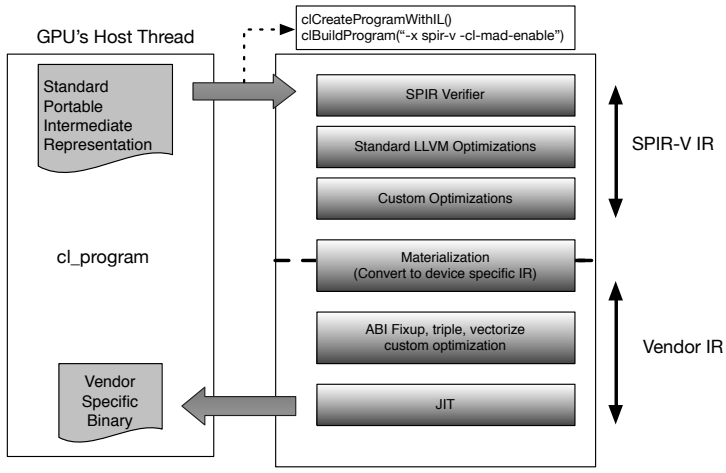


Figure 2.17: SPIR-V compilation process into binary code (based on Ayal Zaks explanation at ACACES'14).

There is a number of works that tackle the challenge of adapting OpenCL applications for heterogeneous architectures. They can be classified into two categories: **Code Transformation** and **Auto-Tuning**. On one hand, there are a few publications that apply code transformations to increase performance. Some works [100, 112] employ a work-item coarsening technique while adapting GPU optimized codes to multicores CPUs, it increases the amount of work performed for each CPU thread and allows an effective use of CPU compute units, it also reduces computation overheads. Other approaches [90, 101] apply tiling and a nested loop scheme to exploit spatial data blocking according to the dimensions of the input data and the capacity of the memory hierarchy. Stratton et al. [100] propose a set of abstract performance extensions in parallel languages to help CPU compilers to understand the logical parallelism while consuming GPU optimized codes. They highlight abstractions like data-parallelism, task-parallelism, spatial locality and temporal locality. On the other hand, other approaches exploit auto-tuning techniques. Cao et al. [14] perform a dynamic division of the workload into tasks and schedule tehm according to a dynamic scheduling strategy that monitors accelerator behaviour. In [65], authors identify a number of tuning knobs that are key for performance portability: the most important are thread-data mapping, memory access pattern, tiling size and data caching. Finally, Phothilimthana et al. [85] propose an approach to combine the above two

methodologies. They introduce the PetaBricks language and its auto-tuner compiler, they also automatically generate OpenCL code, and run it across multiple devices by using a work-stealing technique to balance the workload.

Although there are works that deal with performance portability, none of them achieve a near optimal performance. The research community agrees that more innovations are required to overcome the performance portability issues. The performance behaviour of regular applications, like dense linear algebra, is easy to model or predict. However, it is difficult to predict performance when executing irregular application like sparse matrix-vector multiplication (SpMV) or computing tree-based codes. Irregular applications may exhibit large computation differences between two different regimes of the same application. They also may require a completely different computational needs according to the input date-sets. Furthermore, while executing irregular applications, the relative performance of CPUs and GPUs also varies between different architectures. On one processor architecture (i.e. Intel Ivy bridge), a specific part of an application may run faster on the CPU, while on another processor (i.e. Intel Haswell) it may run faster on the GPU, (see chapter 4).

We tackle the aforementioned problems in chapters 3, 4 and 5, by using runtimes and some adaptive and dynamic scheduling strategies that allow to run parallel patterns by adapting the workload distribution to the processor architecture and applications demand changes. In next section, we review the state-of-the-art of RS and scheduling strategies which are focused on heterogeneous architectures.

2.4. Runtimes for heterogeneous systems

As mentioned in the previous Section, the performance portability issue remains unsolved, although there are a few previous works and approaches targeting performance portability, a general solution is lacking. We strongly believe that RS can be applied to understand applications' needs and to ease an effective mapping of logical parallelism onto the available hardware processors. Through the course of this thesis, we asses that RS can be applied to get a near optimal performance while running applications on several heterogeneous architectures.

In this sense, to make the most out of current heterogeneous architectures a throughput-aware workload partition should be designed. The RS plays a key role in this scenario, which is even more critical in the presence of asymmetric multicore/manycore architectures, where an accurate workload partition has to

be made in order to guarantee an effective utilization of all processors. In Lee et al. [67], the authors prove that GPU throughput is within the range 1.5x-15x with respect to a CPU. In this sense, a CPU-GPU workload distribution may be feasible to achieve optimal performance. In this section, we cover an extensive range of runtime approaches and techniques that tackle the execution issues of running applications on heterogeneous architectures.

2.4.1. Static Approaches for task Scheduling

In general, static workload-partition approaches take the partition decision at compile time. Thus, these methods avoid the overhead of RS and profiling some parts or the application while it runs. In this section, we describe several static partition methods that focus on the workload partition over heterogeneous architectures.

Grewe et al. [45] propose a machine learning based approach to predict a near optimal partitioning mechanism for OpenCL programs. It is merely based on a compiler analysis of the code structure. This static analysis characterises OpenCL programs as a fixed vector of real values, also known as features. They define a function $f()$ that maps a vector of OpenCL code features (c) to the optimal partition of this program, i.e. $f(c) = p$, where p should be as near as possible to the optimal partition point. This approach uses a two-level machine learning method, the first level classifies OpenCL programs that should only be executed either on the CPU or on the GPU. The second level, uses a SVM mechanism to analyse the program features in order to predict the right partition between both devices.

In Kofler et al. [62], the authors also propose a machine learning method, which is based on Artificial Neural Networks (ANN), to find the near optimal partition point between devices. They present a framework with two main phases: training and deployment. The goal of the training phase is to build a workload partition method, which is required for any previously unused architecture. To build the model, a set of OpenCL programs are translated into intermediate representation by the code analyser. From this IR, some static program features are extracted and stored in a database. The intermediate representation of the program is then passed to the back-end compiler which generates multi-device OpenCL code. Once generated, the application are executed with a range of problem sizes. They record some performance metrics and link them with the problem size along with some features of the program which are collected and added to the database as well. Once these steps have been accomplished for

all programs, the trainer uses the features and the performance measurements stored in the database to generate a task partitioning prediction model. In the deployment phase, when a new OpenCL program is executed, the runtime features are extracted and provided to the previously trained model, which combines them with the static program features to predict the best task partitioning for the current program with the selected problem size. Finally, the RS executes the program by using the predicted task partitioning.

Luk et al. [70] propose the Qilin method. It automatically performs a partition of a workload execution across heterogeneous processors at compile-time. To do that, it maintains a database which stores the execution-time prediction for all the programs it has once executed. The first time that a program is run by Qilin, it is used as a training run. It divides the whole data-set (size N_t) into two parts of size N_1 and N_2. The first part (N_1) is mapped to the CPU, whilst the second part (N_2) is mapped to the GPU. Within the CPU part, it further divides N_1 into m sub-parts $N_1^1 ... N_1^m$. Each sub-part N_1^i is run on the CPU and the execution time $TC(N_1^i)$ is measured. Similarly, the N_2 part is further divided into m sub-parts $N_2^1 ... N_2^m$ and their execution times on the GPU $TG(N_2^i)$ are measured. Once all those times are available, Qilin uses a fitting method to construct two linear equations, as projections for the actual execution times TC(N) and TG(N), respectively. The next time that Qilin runs the same program with a different input problem size N_r, it can use the execution-time projection stored in the database to determine the computation mapping.

This kind of methods usually requires a large training phase that may need a prohibitive amount of time in order to perform accurate predictions. To overcome this timespan limitations, the research community relies on dynamic methods that learn how to perform the partition of the applications across heterogeneous processors while keeping the introduced overhead low.

2.4.2. Dynamic Approaches for task scheduling

In this section, we describe a number of dynamic methods that take the decision of distributing the workload at runtime. In general, they are supported by mathematical models, hardware counters and runtime metrics.

Load balance Strategies

As mentioned before, current attempts to provide programming support for heterogeneous systems such as CUDA [96], OpenCL [38] and OpenACC [82] consider

the portability of code across CPUs and GPUs, but do not provide support for simultaneous execution of parallel constructs (*for, scan, reduce, ...*) on CPUs and GPUs. Anyways, the problem of accelerating applications on heterogeneous architectures based on coupled or discrete GPUs by using the aggregate processing power of multicore CPU and multiple GPUs has received some attention lately [2, 10, 40, 70, 88, 106].

Vuduc et al. [106] propose a *wildly asynchronous* implementation that can reduce or even eliminate the synchronization bottleneck between iterations, although for the heterogeneous implementations they did not obtain speed-ups on their platforms. StarPU [2] and XKaapi [40] offer a runtime for scheduling a DAG of tasks on heterogeneous architectures, they also provide programming libraries that include an API to select the preferred scheduling policy. In contrast, the works proposed in this theis are based on the use of higher-level templates. OmpSs [10] is another programming library that provides a set of OpenMP-like pragmas coupled with a runtime system to schedule tasks while preserving dependencies. Although these works include performance results for multi-GPU systems, collaborative work with the CPU multicores is not considered, which is a relevant feature in our work, see chapter 3. Moreover, one distinguishing feature of our research when compared to the aforementioned frameworks is that we explore dynamic CPU block resizing to prevent load unbalance of CPUs and GPUs due to small or large block sizes. Another distinctive feature of our work is that we study the efficiency of non-collaborative and collaborative *host thread strategies* combined with the possibility of using oversubscription to improve CPU cores utilization. Our results, in chapter 3, indicate that oversubscription provides an orthogonal mechanism to increase performance in heterogeneous multi-CPU & multi-GPU systems.

There are other approaches that implement work-stealing techniques to reduce the load imbalances across processors. These are dynamic load balancing methods that extend the use of work-stealing techniques to GPUs and heterogeneous architectures. In this sense, there is a number of articles that propose a global work-stealing mechanism for heterogeneous systems comprised of CPUs and GPUs [1, 34, 39, 85, 97]. Some proposals, like [97] extend the work stealing for heterogeneous CPU-GPU architectures and use a *host thread* to steal taks to feed the GPU. These approaches divide the iteration space lazily, as most work stealing approaches do. In contrast, we perform an eager partitioning to better determine the most appropriate tasks sizes for GPUs and CPUs. Additionally, Chatterjee et al. [17] also propose a method to exploit this technique in single GPU accelerated applications, they reduce the intra-device load unbalances among blocks of threads. However, these methods do not take into account that

the size of the offloaded tasks to GPUs must be accurately selected, as we do in chapter 4.

Partition Methods for parallel_for pattern in heterogenous systems

In this section, we describe a number of dynamic workload partition methods. These methods often incorporate a profiling phase at execution time to adapt the partition according to processors performance while running the application.

Belviranli et al. [4] present a Heterogeneous Dynamic Self-Scheduler (HDSS), it is a dynamic load balancing scheme for loops on heterogeneous architectures. HDSS uses a weighted self-scheduling mechanism to fully use all available processing units in the system during the application execution. This algorithm dynamically resizes blocks to prevent underutilization and load imbalances of GPUs due to small or large block sizes. Unlike static approaches, HDSS does not require an offline training phase. On the contrary, it has an adaptive phase that determines the computational weights of processors. HDSS starts with small block sizes and increase them gradually while continuously monitoring performance. Computational weights are recorded after executing each block of iterations, until they become stable. In this phase, it uses a least squares curve fitting technique to accurately find processor weights and the block of iterations that fully feed the computational units of all processors. In the second phase, completion phase, the remainder and majority of the iterations are computed based on the previously computed weights. This phase uses a modified version of Guided Self-Scheduling (GSS) algorithm to achieve near minimum number of blocks. Moreover, it ensures that processors (CPU and GPUs) finish their execution at nearly the same time.

In [84], Pandit and Govindarajan propose a collaborative method for CPU-GPU computations called *Fluidic*. This method uses a double data buffer as it maintains two copies of the whole data buffer, one on the CPU and one on the GPU. First, the CPU host thread offloads the entire kernel execution to the GPU, which automatically starts its execution. Next, the CPU starts executing sub-ranges of iterations in descending order, from the other end. After executing each sub-range of iterations, the CPU transfers the resulting data plus a status value to the GPU. These status values are checked by the GPU before executing any GPU work-group. Thus, if the GPU tries to execute an iteration that has already been computed by the CPU, then the GPU aborts its execution as the iteration-cross-point has been reached and the whole kernel executed. Finally, the GPU merges the data from both buffers and transfers the resulting buffer to the CPU. This method introduce a certain overhead on the GPU as it is forced to

check for the data being computed on the CPU before starting to compute each work-group. Additionally, the GPU may not notice that a range of iterations has already been computed as the memories from CPU and GPU are different and a data-transfer has to be made, so it will incur in double computed set of iterations and an increment of energy consumption.

Wang et al. [111] propose the Co-Scheduling Based on Asymptotic Profiling method (CAP). This method executes the workload in several steps. In the first step, it executes a small portion of the workload with a static partition and collects the execution time. Then, in subsequent steps, it increases the chunk size. In this method, CAP profiles the GPU performance behaviour for different chunk sizes. Thus, CAP continues profiling and doubles the chunk size in each new step. To find the stable point of performance, CAP comapres the variance of the current and the previous performance ratio[1] in each step. If the variance is smaller than a given threshold, or the chunk size is smaller than the remaining workload, then CAP tries to profile. Otherwise, CAP stops profiling and executes the remaining workload according to the last calculated ratio per device (CPU and GPU). This method considers that larger chunks always get better performance, which may not be true for all applications as we elucidate in Chapter 4. In Concord [58], Kaleem et al. propose a similar method with two stages. In the first stage, it performs a static partition over a small sub-range of iterations and calculates the relative speed of each processor. Next, it shifts to the second phase and performs a distribution of the remaining workload across all processors according to the previously calculate relative speeds. This method updates the relative speed after executing the kernels and uses these values to adapt the partition decision in future executions. Among these works, the ones closer to ours are HDDS [4] and Concord [58]. The main difference between these two works and ours is that they do not take into account the irregularity of the workload. Their main focus is to determine the computational speed of each device and with this information to assign the maximum chunk size to the GPU (and the multicore CPU) to avoid load imbalance. In HDSS [4] the authors compare its proposal with Qilin [70], finding that HDSS always outperforms the later.

In Chapter 4 we compare our proposed method against HDSS [4] and Concord [58]. We elaborate on the main differences between these two systems and ours. We perform a search for a near optimal GPU chunk size, thus we avoid too large or too small chunk sizes that may harm the GPU's throughput; we keep adapting the chunk size of the GPU during the whole execution time in order to adapt to different applications regimes; the CPU chunk sizes are also calculated

[1]The performance ratio is measured as the number of iterations executed in one second.

to work in the same time-window along with the GPU. In this sense, none of the mentioned related approaches change the block size dynamically based on the throughput of the application. Concord is the only one that evaluates irregular applications as we do too. Moreover, it is also the only one that, like us, focus on heterogeneous CPU-GPU chips. All other approaches use more powerful discrete GPUs, but even their best strategy suffers from unbalances when the application is irregular, as they usually stop profiling after 50% of the items have been processed. Also, none of these works evaluate energy consumption.

Pipeline pattern on heterogenous systems

One approach for coding streaming applications is to use a programming language with support for streams, as for example StreamIt [99]. In [50], the authors extend the StreamIt framework to allow its execution on one or more GPUs at the same time. They include several features to increase efficiency by efficiently using on-chip memory and a scheduler to distribute large applications across several GPUs. However, these approaches do not provide support for heterogeneous CPU-GPU executions.

By using both CPU cores and GPUs, simultaneous computation on heterogeneous platforms delivers higher performance than CPU-only or GPU-only executions [113]. In this sense, there are some approaches that provide support for streamming conmputing in heterogeneous architecture such as FastFlow [42] and [104]. In FastFlow [42], Goli et al. present a skeletal framework. It allows developers to easily write OpenCL code inside an heterogeneous algorithmic skeleton and control the allocation of OpenCL kernels. They execute their codes by mapping each stage to one device at a time. As a result, they improve performance over CPU versions by an order of magnitude. In [104], Totoni et al. propose a method to execute pipeline applications on systems with one CPU and one GPU. They recommend a pipeline configuration based on the idea that all pipeline stages has to be mapped to the device where they run faster. They first execute all stages on both the CPU and the GPU. Later, the form two groups of consecutive stages, so they end up running one group on the CPU and the other on the GPU. They select consecutive stages in a way that they balance the timespan of both groups of stages. This work exploits parallelism using an approach similar to software pipelining where two frames are computed at the same time, one on the GPU and another one on the CPU. In this case, the performance is limited by the slower group of stages. In Chapter 5, Totoni et al. approach is used as a baseline and we perform a comparsion with the configuration that our proposed approach finds to be the best.

There are other research efforts, such as [5, 68], that define analytical models to optimize the scheduling of pipeline applications, considering energy and throughput as an objective or a constraint of the problem. However, these works focus on optimizing the concurrent execution of multiprogrammed workloads that consist of independent pipeline applications, whereas we are interested in optimizing one streaming application. In addition, they model energy as a sum of system level components (processor, network, disk, ...) where the energy consumed on each component is the product of the execution time in that component and the dynamic power in the component (measured or estimated using micro-benchmarks and supposed constant for the benchmarks evaluated). On the other hand, our proposed approach, in Chapter 5, uses accurate hardware energy counters available in the underlying architecture. This allows us to measure at runtime the energy consumption on the CPU, GPU and Uncore components for a given application. In contrast with previous static approaches, we use this information to guide the scheduler at runtime to find the optimal mapping that optimize the throughput or the energy (or a trade-off metric) of our application.

3 Parallel_for Pattern: Load Balancing and Scheduling

This chapter introduces a model that deals with the inherent load balancing problem on heterogeneous platforms composed of several CPUs and GPUs [81]. Moreover, we consider the problem of efficiently executing a single application in a heterogeneous environment by allowing the simultaneous execution of work on the accelerators and CPUs. In this context, the Runtime System needs to offer a programming model that considers heterogeneity both in terms of computing power and possibly a disjoint memory address space. Therefore, the effective utilization of resources in GPU-accelerated systems requires a careful partitioning of the workload across CPU cores and GPU accelerators. Designing an adaptive application level work distribution mechanism that is portable across systems with varying node configurations is challenging. The workload distribution is further complex while executing applications that exhibit irregularities in the granularity of parallelism across computational tasks. This chapter focuses on the problem of efficiently partitioning and dynamically scheduling *parallel for* loops on heterogeneous architectures.

In particular, we extend the `parallel_for` function template from Intel TBB task library [89] to allow its exploitation in heterogeneous systems. We have selected TBB because its task scheduler implementation is the most efficient when compared with other task schedulers, it also represents the state-of-the-art on multicore environments. Although we use TBB as the runtime supporting system, our load balancing model and scheduling strategies can also be implemented on top of any other heterogeneous task framework.

The remainder of the chapter is organised as follows: Section 3.1 introduces the internal details of the `parallel for` API that allows the execution on

heterogeneous architectures. In Section 3.2, we deal with the load balancing problem of the iterations of a for loop across the available processors. Next, we propose two scheduling strategies to make an effective use of the GPU host thread, Section 3.3. Finally, Section 3.4 presents the experimental results to end up with conclusions in section 3.5.

3.1. The parallel_for template

The Intel TBB library provides a `parallel_for` function template that performs a parallel run of a for loop over a range of iterations. The default partitioner of this function template recursively splits the range of iterations into sub-ranges, chunks, until a minimum threshold size is reached. Each chunk is then run as an independent task and the internal TBB runtime scheduler employs a work-stealing technique in order to balance the load of task across all CPU cores (for more details, see Section 2.3.3). The original template only allows to run this function on multicores CPUs. In this section we extend this function template to offer a new implementation that allows applications to run on CPUs and GPUs concurrently. In this section we present the library API and describe internal implementation details.

The Figure 3.1 shows in pseudo-code an usage example of our extended `parallel_for` construct for a heterogeneous run. As in any multi-threaded program, the scheduler has to be firstly initialised (see line 20). In this step, the developer sets the number of O.S. threads that the library runtime has to fire up. Once the initialization phase is done, the developer can invoke the `parallel_for` (line 25). Note that some parameters have to be explicitly defined: the iteration space, the range of iterations `itS` that has a pair of integer values to stablish the iteration space limits; the body object of the for loop, `bodyObject()`, which implements the body for CPUs and GPUs; and the partitioner policy object, `NCHT()`, which implements the method to perform the partition of the iteration space across the computational units. This object constructor receives an input parameter `GrainSizeGPU` which is a $n-$tuple of the form (GS^1, GS^2, ..., GS^k), where GS^k indicates the range of iterations that are assigned to the GPU device with $id = k$ during the whole run. Thus, in this approach, we are assuming that the user given $n-$tuple contains the optimal range size for each GPU on the system.

In our proposal, we develop an adaptive partitioning strategy. Although the user provides an optimal chunk size for each GPU device within the `GrainSizeGPU` parameter, it is the responsibility of the partitioner to compute the optimal chunk

```
1  #include ``HScheduler.h''
2  #include ``NCHT.h''
3
4  class bodyObject{
5  ...
6  public:
7    void operatorCPU() (RangeH& r) {
8       for(i=r.begin; i!=r.end; i++)
9       { // CPU compuation }
10   }
11   void operatorGPU() (RangeH& r, Stream& s){
12      hostToDevide_async(r.begin, r.end, s.device, s.streamGPU);
13      launchKernel_async(r.begin, r.end, s.device, s.streamGPU);
14      deviceToHost_async(r.begin, r.end, s.device, s.streamGPU);
15   }
16 };
17
18 int main(){
19   // Start task scheduler with nThreads
20   HScheduler init (nThreads);
21
22   //Allocate GPU buffers
23   AllocatingGPUBuffers();
24   ...
25   parallel_for (RangeH& itS, bodyObject(...), NCHT(GrainSizeGPU));
26   ...
27 }
```

Figure 3.1: Usage example of our proposed `parallel_for` template

size for the CPU cores that will be concurrently computing work with the GPU accelerators. In Chapter 4, we propose a strategy to perform an automatic computation of the optimal chunk size for each GPU device, like [4, 58]. However, in this Chapter we focus on the cooperative work performed by the CPU cores and how to distribute the workload among them and the available GPUs to prevent underutilization and load imbalance between these two types of processors.

To use our library, first, the user has to include the library header files to make the required methods and objects available, lines 1 and 2 in Figure 3.1. The user is also responsible for allocating the GPU buffers (line 23) and defining a class with the implementation of the methods `operatorCPU()` and `operatorGPU()`, these methods will process the chunks of iterations on a CPU core or on a GPU stream device respectively, as shown in lines 4-16. In this sense, two versions of the body must be coded: i) the version for the CPU cores in `C++`, its operator just needs the parameter `r`, the range of iterations to be executed, line 7; and ii) the version for the GPU in `OpenCL`, its operator is shown in line 11, it requires the range of iterations `r` and the parameter `s` which is a struct with the GPU device id (`s.device`) and the stream id (`s.streamGPU`) in the case that such

GPU provides support to more than one concurrent stream. In the example, lines 11-15 show that the user can control one stream to concurrently perform the asynchronous host-to-device, line 12, and device-to-host, line 14, transfers, as well as the kernel launching, line 13, by using the given stream.

In the following section, we describe the inherent load balancing issues of heterogeneous architectures and develop a model and heuristics functions to solve it. In our implementation, this model and heuristics are implemented in the partitioner classes NCHT and CHT, which are described in Section 3.3.

3.2. Load Balancing problem

In this section we first illustrate the load balancing problem and motivate the need of proposing a solution to reduce the effect of load unbalances. We perform an initial comparison between our basic adaptive scheduling strategy and StarPU. Later, we propose an analytical model to deal with the load unbalances of a heterogeneous system. Finally, we propose two heuristics functions to that implement the analytical model.

With the advent of the new heterogeneous programming models such as CUDA, OpenCL or OpenACC, developers are able to develop and run their applications on a wide range of platforms comprised of multicores CPUs and accelerators. However, these programming models do not provide partition strategies to allow the execution of single applications across all available CPUs and accelerators. To overcome that limitation, a few previous proposal are published [2, 10, 40]. However they do not consider varying the size of the range of iterations executed on CPUs according to the relative computational speed of the GPUs. In heterogeneous architectures, it is critical to guarantee an ideal load balance in order to make the most of the platform and avoid unnecessary waiting times between processors. Furthermore, for heterogeneous architectures, three factors are critical to achieve ideal performance:

1. The ideal task size that is offloaded to each device for a computation should be carefully identified and adaptively tuned during execution time.

2. The assignment of chunks to the computational processors, the CPU cores and GPUs, must guarantee minimum load unbalance and overheads.

3. The computational speed of each computing resource should be accurately measured.

The aforementioned state-of-the-art heterogeneous frameworks, do not consider factor 1), whereas in our approach we consider all of them. While scheduling blocks (chunks) of iterations from the iteration space of a parallel *for loop* on heterogeneous architectures, an ideal chunk size for GPU accelerators needs to be large enough to amortize data transfers (host-to-device and device-to-host) and ensure that all theirs computation units are fully utilized. We tackle this issue in Chapter 4. On the contrary, the CPU chunk size is not subjected to these constraints and it can be adaptively re-sized to guarantee that all computational devices finish at the same time.

In order to asses the relevance of a dynamic re-sizing strategy of the chunk of iterations assigned to CPU cores during execution time, we conduct a first experiment in which we compare our basic adaptive partitioning strategy that performs chunk re-sizing, *Non-Collaborative Host Thread*, $NCHT$, against StarPU's fixed chunk size partitioning strategy. Our $NCHT$ strategy has been implemented on top of the TBB task library and follows a greedy scheduling policy (see Section 3.3.1 for more details about this scheduling strategy). Our partitioning heuristic follows this principle when computing the optimal chunk size for a device that requests a chunk of iterations: if there are enough remaining iterations to keep all the devices busy then the chunk size selected is proportional to the resource's effective throughput; otherwise the size is computed from a weighted guided self-scheduling formula, as we explain with more details in Section 3.2.2.

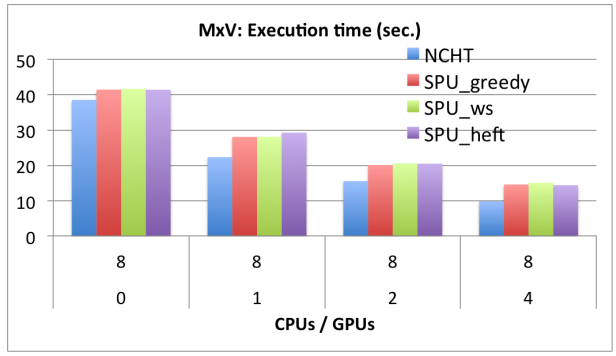


Figure 3.2: Performance comparison for NCHT and StarPU partition strategies (execution time in seconds) while running the Matrix-Vector multiplication benchmark on 8 CPUs and (0, 1, 2, 4) GPUs.

This first set of experiments are carried out in the platform described in Section 3.4.1. Figure 3.2 shows the execution time (y-axis) for the `MxV` Matrix-Vector

multiplication benchmark described in section 3.4.2. A system configuration with 8 CPUs and an increasing number of GPUs, from 0, 1, 2 to 4 (x-axis), with 8 logical (O.S.) threads is used in these experiments. The first set of bars (blue), `NCHT`, represents our adaptive partitioning strategy. The next set of bars represent the execution times for the StarPU fixed chunk size partition strategy. This strategy is evaluated with the best available schedulers in StarPU: `SPU_greedy` (red) that uses a central task queue from which all available workers draw tasks to compute on; `SPU_ws` (green) which is based on a work-stealing scheduler, thus when a worker becomes idle, it steals a task from the most overloaded worker; and `SPU_heft` (purple) that takes a task execution performance model and data transfer times into account to perform a HEFT-similar static scheduling strategy, it schedules tasks where their termination time will be minimal [2]. For StarPU results, different grain sizes are tested (2,000, 200, and 20 matrix rows), obtaining similar results.

In the only CPU execution scenario, the leftmost set of bars (8,0), our strategy outperforms StarPU by an 8%, mainly due to an internal StarPU overhead of the task management mechanism. This difference increases when 1, 2 and 4 GPUs are considered. In heterogeneous executions, the best StarPU strategy, `SPU_heft` (purple), is 25%, 32% and 53% slower than our proposed strategy with 1, 2 and 4 GPUs, respectively. We observe that these increments in the running time differences are explained by a larger load unbalance among the computational devices, mainly between the GPUs and CPU cores, and by an inefficient use of the GPU host threads in StarPU. This result justify our selection of Intel Threading Build Blocks (TBB) as our underlying task framework and the necessity of implementing adaptive chunk re-sizing strategies to avoid load unbalanced scenarios.

3.2.1. Optimization model for load balancing

This section elaborates on the optimization of the partition problem of parallel for loop. Thus, we develop an analytical model that is responsible for adapting the chunk sizes to the processors during execution time to tackle the load unbalances. Specifically, we focus on the problem of scheduling the iterations of a parallel for loop across all the processing devices of a heterogeneous machine, comprised of CPU cores and GPUs. Thus, we aim at modelling the performance behaviour of each processing device to develop an optimization model for this problem. To that end, we envision the execution of a parallel for loop as a sequence of *scheduling intervals*, $[I_0, I_1 ..., I_i, ..., I_N]$, where sub-ranges of iterations, or chunks, are executed on the available processors. Thus, each processing device at its $i-th$

interval, I_{C_i}, for a CPU core and $I_{G_i^k}$, for the $k{-}th$, computes a chunk of iterations given by $Ch(I_{C_i})$ for the CPU and $Ch(I_{G_i^k})$ for the $k-th$ GPU.

We model our partitioning strategy as a simplified optimization problem whose goal is to minimise the system load unbalances subject to the constraint that the system throughput is maximum. For this model, we simply assume that the optimal GPU chunk size, $Ch(I_{G_i^k})$, for the $k-th$ GPU device is known and stationary during the whole program execution. Hence, this value is kept constant for all scheduling intervals, as it happens in regular workloads like linear algebra methods [103]. On the contrary, the optimal chunk size for the CPU cores for all scheduling intervals, $Ch(I_{C_i})$, are values that our optimization problem has to look for. Specifically, the optimal chunk size of each GPU device is given by the user as an input parameter in the `parallel_for()` function template invocation (see line 25 in Figure 3.1). As mentioned before in Section 3.1, these constant sizes are given by the user with the `GrainSizeGPU` parameter, which is a $n-tuple$ of the form $(Ch(I_{G_0^1}), Ch(I_{G_0^2}), \ldots, Ch(I_{G_0^k}))$, where $Ch(I_{G_0^k})$ is the given size of the chunk that the partitioner will constantly assign to the $k-th$ GPU for all scheduling intervals at runtime. The optimal chunk sizes values depend on the kernel specified in the task body and the particular GPU hardware features: the number of registers, size of the memory of each type required in the kernel, maximum number of block size, etc. Let us assume that $Ch(I_{G_i^k})$ and $Ch(I_{C_i})$ represent the range of iterations in the $i-th$ scheduling interval for the $k-th$ GPU and for a CPU core, respectively. Every time that a chunk is run, its processing time is recorded either for the $k-th$ GPU, $T(I_{G_i^k})$, or for a CPU core, $T(I_{C_i})$. These times are needed to compute the effective throughput for the $k-th$ GPU, $\lambda(I_{G_i^k})$, or for a CPU core, $\lambda(I_{C_i})$, again for the $i-th$ scheduling interval. The following expressions allow us to compute the initial throughput and its updates while executing upcoming scheduling intervals:

$$\lambda(I_{G_0^k}) = \frac{Ch(I_{G_0^k})}{T(I_{G_0^k})}, \tag{3.1}$$

$$\lambda(I_{G_i^k}) = \alpha \cdot \frac{Ch(I_{G_i^k})}{T(I_{G_i^k})} + (1-\alpha) \cdot \lambda(I_{G_{i-1}^k}) \text{ and } \quad \forall i = 1:N, \tag{3.2}$$

where $\lambda(I_{G_{i-1}^k})$ represents the effective throughput value for the $k{-}th$ GPU in the previous scheduling interval. Equation 3.2 is the *exponential moving average* of the current throughput sample, $Ch(I_{G_i^k})/T(I_{G_i^k})$, and the previously calculated throughput $(\lambda(I_{G_{i-1}^k}))$, the α parameter is a smoothing constant between 0 and 1 that weights the contribution of the current instant throughput against the his-

torical throughput average. The optimal value for this parameter and its impact on our partition strategy are issues discussed in the experimental Section 3.4. Let us note that for the computation of the CPU throughput in the $i-th$ interval, $\lambda(I_{C_i})$, we replace $Ch(I_{G_i^k})$ and $T(I_{G_i^k})$ by $Ch(I_{C_i})$ and $T(I_{C_i})$ respectively in equation 3.2. Once the effective throughput of the $k-th$ GPU is computed, the current computational speed of this $k-th$ GPU device is calculated as,

$$f^k = \frac{\lambda(I_{G_i^k})}{\lambda(I_{C_i})}, \tag{3.3}$$

where $\lambda(I_{C_i})$ represents the current effective throughput in a CPU core. Note that all CPU cores share the same value, as all CPU cores are homogeneous and exhibit same performance within a scheduling interval. Analogously, when a CPU core executes a task, the effective throughput in a CPU core, $\lambda(I_{C_i})$, is computed by using the equations 3.1 and 3.2. Additionally, as we can observe in Equation 3.3, the parameter f^k represents how many times the $k-th$ GPU is faster than a CPU core within the last interval.

Let us assume that T represents the optimal time span for which the parallel for loop can be executed in the heterogeneous system. In addition, N^k denotes the number of chunks that the kth GPU executes, whereas N^c is the number of chunks per each CPU core. Then, $N = \sum_{nGPUs} N^k + \sum_{nCores} N^c$ is the total number of executed chunks in the whole system. Let us also assume that T^k,or T^c, represents the average time that the kth GPU, or CPU core, needs to execute its chunks of average size R^k and R^c, resulting in an average throughput of λ^k, or λ^c, for the kth GPU and a CPU core, respectively. Our objective is then to minimise the load unbalance in the system. Obviously, the load unbalance due to the $k-th$ GPU can be modelled as $N^k \cdot T^k - T$ or $N^c \cdot T^c - T$ for a CPU core. Consequently, the load unbalance due to all processing units in the system is the sum of the load unbalances due all computing resources. This is the objective function that we want to minimise, as shown in equation 3.4. Furthermore, the constraint shown in equation 3.5 limits the maximum throughput that can be achieved, $\lambda_{max} = \sum_{nGPUs} \lambda(I_{G_i^k}) + \sum_{nCores} \lambda(I_{C_i})$, while the last constraint represented by equation 3.6 ensures the positivity of the variables.

$$\text{Minimize} \quad \left(\sum_{nGPUs} \left|N^k \cdot T^k - T\right|\right) + \left(\sum_{nCores} |N^c \cdot T^c - T|\right), \tag{3.4}$$

$$\text{such that} \quad \left(\sum_{nGPUs} \frac{N^k}{N} \cdot \frac{R^k}{T^k}\right) + \left(\sum_{nCores} \frac{N^c}{N} \cdot \frac{R^c}{T^c}\right) = \lambda_{max} \tag{3.5}$$

$$\text{and} \quad \forall k \quad T^k > 0, T^k > 0. \tag{3.6}$$

This problem has a linear objective function and a non-linear constraint. Thus, to solve that, we make a change of variables. Let $\rho^k = 1/T^k$ and $\rho^c = 1/T^c$. Then the problem becomes as follows,

$$\text{Minimize} \quad \left(\sum_{nGPUs} \left|\frac{N^k}{\rho^k} - T\right|\right) + \left(\sum_{nCores} \left|\frac{N^c}{\rho^c} - T\right|\right), \tag{3.7}$$

$$\text{such that} \quad \sum_{nGPUs} \frac{N^k}{N} \cdot R^k \cdot \rho^k + \sum_{nCores} \frac{N^c}{N} \cdot R^c \cdot \rho^c = \lambda_{max} \tag{3.8}$$

$$\text{and} \quad \forall k \quad \rho^k > 0,\ \rho^c > 0. \tag{3.9}$$

The objective function is non-linear, but convex and separable in its variables. The constraint is now linear. This type of constraint is typically called a *resource allocation* constraint. Thus, the transformation yields a *continuous convex separable resource allocation problem* and the optimal solution occurs when the derivatives of each of the objective function addends ($(N^k/\rho^k) - T$ and $(N^c/\rho^c) - T$) are equal, see [51] for more details. In other words, we have,

$$\frac{-N^1}{(\rho^1)^2} = \frac{-N^2}{(\rho^2)^2} = \ldots = \frac{-N^c}{(\rho^c)^2}. \tag{3.10}$$

Changing the variables again, simplifying, and taking the square roots we have,

$$\sqrt{N^1} \cdot T^1 = \sqrt{N^2} \cdot T^2 = \ldots = \sqrt{N^c} \cdot T^c. \tag{3.11}$$

If we assume that, ideally, the load unbalance in the system is 0, then $N^k \cdot T^k = T$ for the $k{-}th$ GPU, and $N^c \cdot T^c = T$ for a CPU core. Thus, using this assumption for equation 3.11 we obtain the following expression,

$$T^1 = T^2 = \ldots = T^c. \tag{3.12}$$

As $T^k = R^k/\lambda^k$ and $T^c = R^c/\lambda^c$, equation 3.12 can be expressed as,

$$\frac{R^1}{\lambda^1} = \frac{R^2}{\lambda^2} = \ldots = \frac{R^c}{\lambda^c}. \tag{3.13}$$

This expression gives us the key to achieve an optimal strategy for minimizing the load unbalance in the system: *each time that a chunk is partitioned to be assigned to a resource, its size should be selected such that it is proportional to the resource's effective throughput.* Additionally, the CPU cores performance is

not sensible to the size of the chunks assigned to them, it gives a certain leeway when selecting the right CPU chunk size in each scheduling interval. In contrast, GPUs quickly degrade their throughput (iterations/time) when chunks sizes differ from the device's optimal size [4]. Therefore, we have decided to implement a greedy partition algorithm based on the following key observations:

While there are enough remaining iterations, the general partition problem can be approximated in each scheduling interval (I_i) by using the optimal chunk size value assigned to the $k-th$ GPU, $Ch(I_{G_i^k})$, whereas the chunk size assigned to a CPU core should verify equation 3.13, that is,

$$\frac{Ch(I_{C_i})}{\lambda(I_{C_i})} = max\left(\frac{Ch(I_{G_i^k})}{\lambda(I_{G_i^k})}\right), \quad k = 1 : nGPUs, \quad \forall i = 1 : N. \tag{3.14}$$

Using the computational speed definition, f^k, given by equation 3.3, we obtain our optimal goal,

$$Ch(I_{C_i}) = max\left(\frac{Ch(I_{G_i^k})}{f^k}\right), \quad k = 1 : nGPUs, \quad \forall i = 1 : N. \tag{3.15}$$

In next section, we discuss the implementation details of this model and elaborate on the development of our heuristic function that dynamically varies the computing units chunk sizes to achieve a workload balance.

3.2.2. Heuristic functions for the optimization model

In this section, we elaborate on the implementation of the developed model in the previous section. We present two heuristic functions to compute the chunk sizes for GPUs and CPU cores during the whole application execution. Therefore, these heuristics aim at automatically adapting the chunk size of CPUs and GPUs along the iteration space of the parallel for loop. In this sense, the `get_GPU_range()` function is responsible for extracting sub-ranges of iterations from the iteration space of the loop for the GPU devices, this procedure is shown in Figure 3.3.

For the first invocations of the function `get_GPU_range()`, in the scheduling interval I_0, the list of `boolean` values, `first_range[k]`, has all its values initialised as `true`. Thus, the function returns the range of iterations provided

```
get_GPU_range()
// Input: r (the input range)
//        k (the id of GPU^k)
// Output: chunk (the new chunk for GPU^k device)
//           r (the remaining iterations)
```

```
1. If first_range[k] then
2.    chunk = GrainSizeGPU[k];
3.    first_range[k]= false;
4. else
5.    If (GrainSizeGPU[k] / f^k < r−GrainSizeGPU[k] / (Σ_{j≠k} f^j)+nCores) then
6.       chunk = GrainSizeGPU[k];
7.    else
8.       chunk = 0;
9.       f^k = 0;
10.      myStreamGPU.device=NULL;
11.      nCores = min(nCores + 1, nThreads);
12.   endif
13. endif
14. chunk = min(chunk,r);
15. r = r − chunk;
16. return(chunk)
```

Figure 3.3: Pseudo-code for the `get_GPU_range()` function

by the user in the invocation of the parallel for function, `GrainSizeGPU[k]` for the $k-th$ GPU, as shown in line 2 in Figure 3.3. After the first chunk execution and the first throughput computation, $\lambda(I_{G_0^k})$, the computational speed, f^k, is calculated. Next time that `get_GPU_range()` is invoked, a new range of iterations *chunk* for the $k-th$ GPU is calculated by the procedure shown between lines 5-12, in Figure 3.3. Specifically, line 5 checks whether there is a sufficient number of remaining iterations. The condition in that line is equivalent to the next equation, just replace f^k by $\lambda(I_{G_{i-1}^k})/\lambda(I_{C_{i-1}})$ for the $i-th$ scheduling interval,

$$\frac{GrainSizeGPU[k]}{\lambda(I_{G_{i-1}^k})} < \frac{r - GrainSizeGPU[k]}{(\sum_{j\neq k}\lambda(I_{G_{i-1}^j})) + nCores \cdot \lambda(I_{C_{i-1}})}, \tag{3.16}$$

in equation 3.16, we check whether the expected time for the execution of the new chunk of size $GrainSizeGPU[k]$, the optimal size for the $k-th$ GPU is smaller than the expected execution time of the remaining iterations ($r -$

$GrainSizeGPU[k]$) when they are executed by all the other computational units, including all the CPU cores, all GPUs and excluding the $k-th$ GPU. We estimate these times by using the effective throughput in the last scheduling interval $(i-1)$ of the $k-th$ GPU ($\lambda(I_{G^k_{i-1}})$) and the aggregated effective throughput of all computational units excluding GPU_k ($(\sum_{j \neq k} \lambda(I_{G^j_{i-1}})) + nCores \cdot \lambda(I_{C_{i-1}})$). In case that the expected time for the execution of the chunk in the $k-th$ GPU is smaller than $r-GrainSizeGPU[k]$, we can guarantee that such GPU device is not executing a task while the other computational units are idle. In this situation, we assign the optimal chunk size to the device, $chunk = GrainSizeGPU[k]$ (line 6). Otherwise, if the condition does not hold, then this is because there is an insufficient number of remaining iterations to keep all the processing units effectively working. In this situation, it is reasonably to not assign a new range of iterations to the $k-th$ GPU, because it may potentially create load unbalance. For this reason, the chunk size is set to 0 and the `operator_GPU()` method will not be any longer invoked (line 8). In addition, this GPU is disabled as a computational unit in the system, by making $f^k = 0$ (line 9) and nullifying the device id (line 10). Also, the number of computational CPU cores is incremented by 1 (line 11), to indicate that one additional thread (the previous GPU host thread) can perform CPU tasks until all the remaining iterations are computed.

```
get_CPU_range()
// Input: r (the input range)
// Output: chunk (the new chunk for the CPU core)
//         r (the remaining iterations)
```

1. If $(\nexists f^k \neq 0)$ then
2. $\quad chunk = \frac{max^k(GrainSizeGPU[k])}{nCores}$;
3. $\quad$ `first_rangeCPU=` $false$;
4. else
5. $\quad chunk = min\big(max_{f^k \neq 0}(\frac{GrainSizeGPU[k]}{f^k}), \frac{r}{(\sum_k f^k)+nCores}\big)$;
6. $\quad$ If $(chunk < threshold)$ then
7. $\quad\quad chunk = min(threshold, r)$;
8. $\quad$ endif
9. endif
10. $r = r - chunk$;
11. **return**$(chunk)$

Figure 3.4: Pseudo-code for the `get_CPU_range()` function

Furthermore, we present our second heuristic function `get_CPU_range()`, which is responsible for partitioning chunks for the CPU cores, shown in Fig-

ure 3.4. Whilst there is no f^k value different to zero, the function returns the range *chunk* as indicated in line 2. Once the first f^k is calculated, the next time that the function `get_CPU_range()` is invoked, a new range of iterations *chunk* is calculated by following the procedure shown in lines 5-8, in Figure 3.4. In line 5, for the second execution and followings, the algorithm selects the value of *chunk* depending on two options:

(1) $max_{f^k \neq 0}(GrainSize[k]/f^k)$ and

(2) $r/((\sum_k f^k) + nCores)$.

The expression $GrainSize[k]/f^k$ represents the optimal number of iterations that a CPU core should compute to spend the same time as the GPU^k if the device is active (i.e. $f^k \neq 0$), as we established in equation 3.14. In our case, we choose to synchronise a CPU core with the GPU for which the $GrainSize[k]/f^k$ value is the largest. This is done to minimise the number of times that the partitioning function must be invoked when computing the CPU chunk, and therefore to minimise its associated overhead. The term $r/((\sum_k f^k) + nCores)$ represents the number of iterations, from the remaining set of iterations, that a CPU core should execute when considering the computational speed of all the active computing units in the system. In other words, it represents the number of iterations that a CPU core should execute when seeking a weighted *guided self-scheduling* load balancing strategy [94]. When the number of remaining iterations, r, is sufficiently high, or there are a sufficient number of remaining iterations, our strategy will choose the optimal value given by option (1). Eventually, when r is getting small and there is an insufficient number of remaining iterations to feed all the computing unit in the system, then our strategy will choose option (2), the weighted guided self-scheduling approach.

Moreover, line 6 in Figure 3.4 checks whether the computed value for *chunk* is smaller than a *threshold* value, which can be set for each application to guarantee a minimum profitable chunk size for a CPU core. This value will depend on the work per iteration and the overhead of our partitioner for this benchmark. In our experimental section, we elaborate on the right election of this value. Finally, line 7 selects the appropriate chunk size.

3.3. Scheduling strategies

In this section we present two scheduling strategies that build on top of the workload balancing model and the partition heuristics function described in Sec-

tion 3.2.1 and 3.2.2, respectively. Moreover, for our extension of the `parallel_for` function template, we implement a novel engine for greedily scheduling task across all the available computational units. Over that engine, we propose two adaptive partition strategies, $NCHT$ and CHT, which are further described later in this section. As a key function of these strategies, we use the aforementioned heuristics functions that adaptively computes the optimal chunk size for each computational unit that ensures a workload load balance scenario.

Particularly, our engine is internally designed as a two-stage pipeline, as depicted in Figure 3.5. At the top of Figure 3.5, we can see the iteration space with the chunks that have already been assigned (in yellow and orange) and the range `r` with the remaining iterations that have not been assigned yet (in white). As mentioned before, this pipeline consists in two stages (TBB filters): Stage_1 (S1), which performs the selection of the computational unit where the task will be scheduled as well as the election of the number of iterations that are extracted from the set of remaining iterations (chunk size), and Stage_2 (S2), which processes the extracted chunk of iterations on the previously selected computational unit.

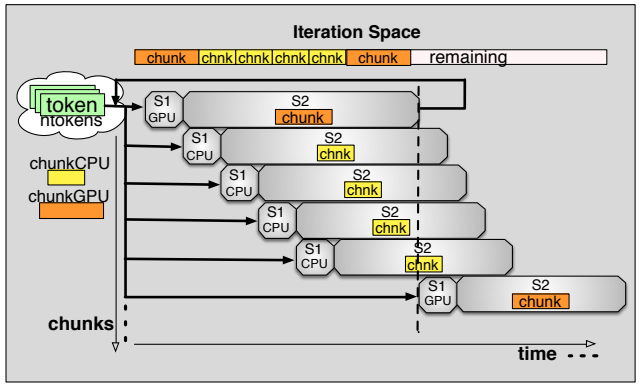


Figure 3.5: Two-stage pipeline engine for scheduling and partitioning chunks of iterations of a parallel for loop in a heterogeneous system.

This parallel for loop engine is implemented on top of the TBB pipeline template [89]. One important feature of a pipeline in TBB is the concept of *tokens*: a token represents a task that has been spawned for each input item of the pipeline and that is going to traverse all the pipeline stages (filters). The left-hand side of Figure 3.5 represents, a cloud of tokens (green boxes), the number of available tokens to the scheduler limits the number of processors (CPUs or GPUs)

that can be concurrently processing chunks of iterations in parallel. Other implementation detail is that we use the CUDA libraries to offload tasks to the GPUs. Once a task, with its corresponding chunk of iterations, is selected to be executed on a GPU, this GPU task ensures the consistency of its input data on the GPU device thanks to its inherent host-to-device memory transfer operation. Then the GPU task enqueues the kernel execution, and finally the GPU task performs the device-to-host memory transfer operation. All these operations are asynchronously executed. Our proposal also supports the usage of CUDA stream features that allows the overlap between memory transfers and kernel launches. Thus, the number of tokens that we consider in our system is equal to the number of GPU streams plus the number of user-allowed CPU cores.

Once the parallel for function is invoked, the internal pipeline is built and executed with a number of tokens equal to the number of CPU cores plus the number of selected GPUs. Thus, we have an available token for each computing processor. All tokens start their execution on Stage_1. However, this stage is serial, it means that only one token can be executed in that stage at a time. For this reason we observe a ramp up process while executing the first chunks of iterations. After a token has been processed in Stage_1 and a computing unit is selected along with its chunk of iterations, it moves towards Stage_2. The second stage is parallel, it means that several tokens can be run in parallel on this stage. Thus, all computing units can be effectively computing tasks at the same time. To achieve this scenario, we ensure that the execution time of the first stage takes less than 0.01% of the whole pipeline execution, and we perform an automatic bypassing of the tasks that finish the second stage to get again into the first stage, as it is shown in Figure 3.5 with a vertical dashed line.

Using the pipeline engine just described, next sections elaborate on the implementation details of the two proposed adaptive partitioning strategies, NCHT and CHT.

3.3.1. Non-Collaborative Host Thread

Our first partitioning strategy, called *Non-Collaborative Host Thread* ($NCHT$) is represented in Figure 3.6. In this strategy chunks are assigned either to an idle GPU stream or to an idle CPU core. In case of assigning a chunk to a GPU stream, the corresponding GPU host thread will only be responsible for hosting the GPU by transferring the data between devices and launching the kernel on the selected GPU, as it is usually done in related work [2, 10, 40].

In this strategy, Stage_1 (see Figure 3.7(a)) firstly acquires an idle GPU stream

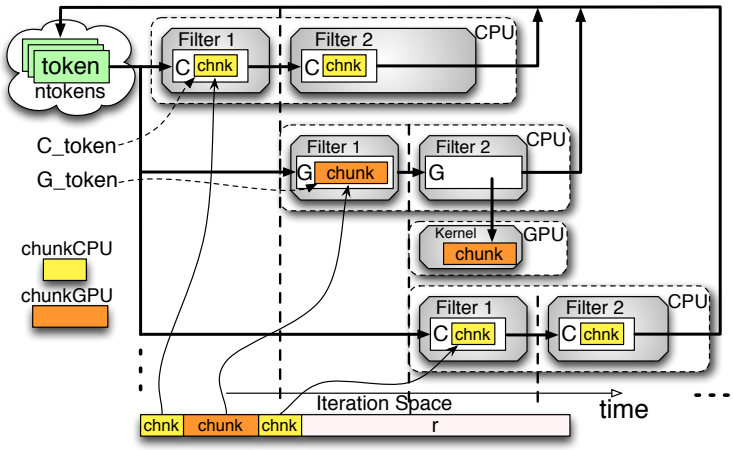


Figure 3.6: Main scheme of the pipeline that implements the $NCHT$ strategy.

and then it checks if the device Id is not null (lines 8-9). In that case, a G_token is created and initialized with information regarding the GPU stream Id of and the range of iterations extracted from the set remaining iterations (r) (lines 10-13). If there is no idle GPU stream or the device Id is null, then a CPU must be idle; thus, a C_token is created and initialized within the range of iterations for the CPU core that the partitioner extracts from the range of the remaining iterations (r) (lines 15-17).

Next, Stage_2 (see Figure 3.7(b)) processes the chunk of iterations in the corresponding computational processor depending on the type of token that arrives, it can be either a G_token (line 10) or a C_token (line 18). In both cases, the time required for the computation of the corresponding chunk is recorded (lines 9-13 and lines 17-19). In the case of a GPU computation, the time recorded[1] is used to update the effective throughput of the corresponding GPU device, and then it is used to compute the factor f^k (line 14). This factor represents the *computational speed* of the *kth* GPU device relative to a CPU core. This computational speed is defined as the ratio of the time per iteration on the GPU device vs. the time per iteration on a CPU core. The factor f^k will be required for the partition function to adaptively adjust the size of the following chunk assigned to a CPU core. In the case of a CPU computation, the time recorded is used to update the effective throughput on a CPU core (line 20). Finally, in the case of a GPU

[1]Let's note that for a GPU, the kernel execution time as well as the data transfer times are taken into account here.

```
1  class GetWork:public tbb::filter{
2  RangeH r;
3  PartitionerH ph;
4  public:
5    GetWork(Range _r, PartitionerH _pH):
         r(_r),pH(_pH){};
6
7    void* operator()(void* myToken) {
8      myStreamGPU=acquire_StreamGPU();
9      if (myStreamGPU != NULL &&
           myStreamGPU.device != NULL){
10       myToken = new G_token();
11       myToken.streamGPU=myStreamGPU;
12       myToken.chunkGPU=pH.
             get_GPU_range(r,myStreamGPU
             .device);
13     }
14     if (myStreamGPU == NULL ||
           myStreamGPU.device == NULL) {
15       myToken = new C_token();
16       myToken.chunkCPU=pH.
             get_CPU_range(r);
17     }
18     return (void*) myToken;
19   }
20 };
```

(a) Stage$_1$

```
1  class ProcessWork:public tbb::filter{
2  BodyObject bO;
3  PartitionerH pH;
4  public:
5    ProcessWork(BodyObject _bO, PartitionerH
         _pH):bO(_bO),pH(_pH){};
6
7    void* operator()(void* myToken) {
8      if (myToken.type == G) {
9        t1=record_time();
10       bO.operatorGPU(myToken.chunkGPU,
             myToken.streamGPU);
11       completion=new_event(myToken.
             streamGPU);
12       waits(completion);
13       t2=record_time();
14       pH.set_factor_GPU(t2-t1, myToken);
15       release_StreamGPU(myToken.streamGPU)
             ;
16     }else{
17       t1=record_time();
18       bO.operatorCPU(myToken.chunkCPU);
19       t2=record_time();
20       pH.set_factor_CPU(t2-t1, myToken);
21     }
22     return NULL;
23   }
24 };
```

(b) Stage$_2$

Figure 3.7: Implementation details of the two-stage pipeline engine that drives the $NCHT$ scheduling strategy.

computation, after the completion of the work and the calculation and update of factor f^k, the GPU stream is released (line 15).

Let us recall that for the GPU devices, we exploit the concurrent nature of the device-memory transfers (host-to-device and device-to-host) and the kernel launches through the use of streams and asynchronous call functions (see Figure 3.1). Specifically, our example of an operator for executing a chunk of iterations on a GPU device is based in the use of asynchronous functions through one stream. After each request is enqueued in the corresponding stream by the `operator()` function, Stage$_2$ is responsible for enqueuing a CUDA event to check the task completion and wait for it (lines 11-12 in Figure 3.7(b)). The `waits()` function can be implemented using either a blocking or a yielding mechanism; both synchronization alternatives are evaluated in the experimental results Section 3.4.

3.3.2. Collaborative Host Thread

Our second partitioning strategy is called *Collaborative Host Thread* (*CHT*), which is the one represented in Figure 3.8. One distinguishing feature of this strategy is that, when assigning a chunk to a GPU stream, an additional chunk of iterations is also assigned to the corresponding GPU host thread. Thus, both the CPU core that runs the GPU host thread and the GPU stream compute two separately chunks of iterations in parallel. As mentioned in the motivation section, this strategy aims to ensure the full utilization of the CPU core that runs the GPU host thread. Figure 3.8 shows two data structures that are required to store the iterations: the range with the remaining iterations of the iteration space (the box at the bottom), and a queue of iterations called `spare_workQueue` that stores sub-ranges of non-executed iterations which were part of a chunk that was assigned to a GPU host thread (the hexagon at the right-hand side).

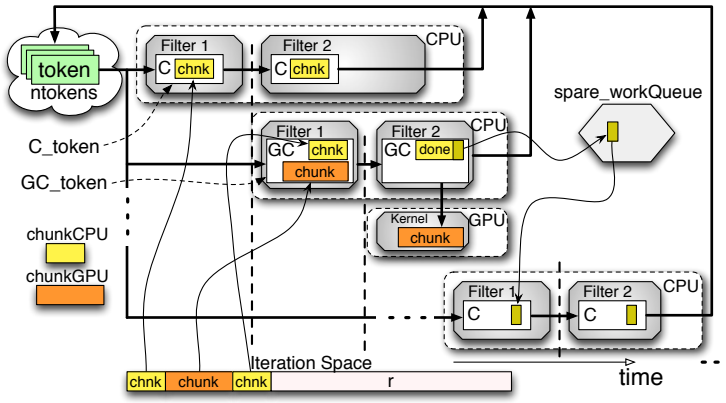


Figure 3.8: Main scheme of the pipeline that implements the *CHT* strategy.

In this strategy, Stage_1 (see Figure 3.9(a)) first acquires an idle stream to one of the GPU devices and checks that the device Id is not null (lines 8-9). Then, it creates a new type of token, a GC_token, a collaborative GPU-CPU token. Thus, this first stage takes a chunk of iterations for the GPU (line 12) and a second extra chunk for the GPU host thread (line 13). In case that all GPUs are busy or that theirs device Id are null, then a C_token is created. Let's note that in this strategy, a CPU chunk of iterations may be extracted from two sources: 1) from the `spare_workQueue` (line 20, see the dark-yellow sub-range in the hexagon in Figure 3.8), it stores subranges of iterations from previously assigned but not completely executed chunks; or 2) if that spare queue is empty, then the chunk of

```
1  class GetWork:public tbb::filter{
2  RangeH r;
3  PartitionerH ph;
4  public:
5    GetWork(Range _r, PartitionerH _pH):
         r(_r),pH(_pH){};
6
7    void* operator()(void* myToken) {
8      myStreamGPU=acquire_StreamGPU();
9      if (myStreamGPU != NULL &&
           myStreamGPU.device != NULL) {
10       myToken = new GC_token();
11       myToken.streamGPU=myStreamGPU;
12       myToken.chunkGPU=pH.
             get_GPU_range(r,myStreamGPU
             .device);
13       myToken.chunkCPU=pH.
             get_CPU_range(r);
14     }
15     if (myStreamGPU == NULL ||
           myStreamGPU.device == NULL) {
16       myToken = new C_token();
17       if (spare_workQueue.is_empty())
18         myToken.chunkCPU=pH.
               get_CPU_range(r);
19       else
20         myToken.chunkCPU=
               spare_workQueue.pop_chunk
               ();
21     }
22     return (void*) myToken;
23   }
24 };
```

(a) Stage$_1$

```
1  class ProcessWork:public tbb::filter{
2  BodyObject bO;
3  PartitionerH pH;
4  public:
5   ProcessWork(BodyObject _bO, PartitionerH
        _pH):bO(_bO),pH(_pH){};
6
7    void* operator()(void* myToken) {
8      if (myToken.type == GC) {
9        t1=record_time();
10       bO.operatorGPU(myToken.chunkGPU,
             myToken.streamGPU);
11       completion=new_event(myToken.
             streamGPU);
12       setOfChunks=split_by(myToken.
             chunkCPU, threshold);
13       while (!setOfChunks.is_empty()) {
14         otherChunk=setOfChunks().pop_chunk
               ();
15         bO.operatorCPU(otherChunk);
16         if (completion.status == COMPLETE)
               {
17           spare_workQueue.push_chunk(
                 compact_by(setOfChunks,
                 threshold));
18           break;
19         }
20       }
21       waits(completion);
22       t2=record_time();
23       pH.set_factor_GPU(t2-t1,myToken);
24       release_StreamGPU(myToken.streamGPU)
             ;
25     }else{
26       t1=record_time();
27       bO.operatorCPU(myToken.chunkCPU);
28       t2=record_time();
29       pH.set_factor_CPU(t2-t1,myToken);
30     }
31     return NULL;
32   }
33 };
```

(b) Stage$_2$

Figure 3.9: Implementation of the two-stage pipeline that implements the *CHT* strategy.

iterations come from the remaining set of iterations (r) (line 18, the white range in the bottom box of Figure 3.8).

Regarding Stage$_2$ (see Figure 3.9(b)), the difference arises when processing the chunks assigned to the collaborative GC_token. The GPU host thread, first enqueues the GPU chunk in the corresponding GPU stream (line 10). And then,

before synchronising in the `wait()` function for the completion of the GPU task, the assigned chunk to the GPU host thread (CPU) is partitioned into a set of sub-ranges of size `threshold` and stored in a temporal queue (line 12). Next, a sub-range is popped from that queue and processed by the GPU host thread on its CPU core (lines 14-15). Later, the status of the GPU stream is polled for completion. In the case that the GPU stream status is `COMPLETE`, then the remaining sub-ranges of iterations stored in the temporal queue are compacted and returned to the scheduler in the `spare_workQueue` (line 17). Otherwise, in the case that the GPU stream status is not `COMPLETE` yet, a new sub-range is popped from the temporal queue and processed by the GPU host thread in the core. This process of polling and computing sub-ranges continues until the GPU stream finish its task or the GPU host thread computes all the sub-ranges stored in the temporal queue. As in the previous partitioning strategy, the time required to compute a GPU or a CPU chunk is recorded and used to compute factor f^k.

3.4. Experimental Results

In this section we conduct a series of experiments to evaluate issues such as the overhead of our framework, the efficiency of the two proposed partitioning strategies, and to what extent their performance is less than optimal. We also explore whether or not it is possible to improve performance by allowing oversubscription and by selecting the appropriate synchronization mechanism.

3.4.1. Experimental setup

We conduct our experiments on a multi-CPU with a quad-socket eight-cores Intel(R) Xeon(R) X7550 2GHz (32 cores). Four decoupled NVidia GPU devices are connected: GPU_1 and GPU_2 are GeForce GTX 480 while GPU_3 and GPU_4 are part of a Tesla S2050. This allowed us to study the scalability of the proposed strategies under different heterogeneous configurations. We refer to the configurations as (no. of CPU cores, no. of GPUs). For instance, for 1 socket (8 cores) and 1, 2 and 4 GPUs we get the configurations (8,1), (8,2) and (8,4), respectively. For 2 sockets (16 cores) and 1, 2 and 4 GPUs: (16,1), (16,2) and (16,4), and finally, for 4 sockets (32 cores) and 1, 2 and 4 GPUs ((32,1), (32,2) and (32,4)). The applications are compiled with Intel C++ compiler (ICC) 11.1, Intel TBB 4.1 and CUDA Development Kit 4.2.

In our experiments, we just consider one CUDA stream per GPU device.

Therefore, every time that a G_token (or GC_token) is selected by Stage_1 in our pipelined engine (see section 3.3), then the corresponding thread would serve as a host thread of the GPU device. In other words, depending on the number of GPUs, there may be at most 1, 2 or 4 threads working as GPU host threads.

3.4.2. Benchmarks

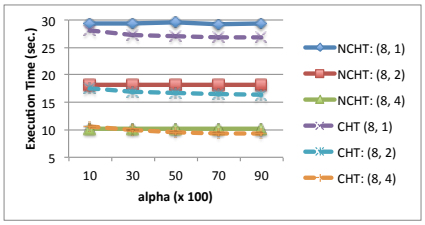
We use a similar benchmark to dense Matrix-vector multiplication `MxV` (although with more operations inside the loop nesting) and the `Barnes-Hut` benchmark for our experiments. The `MxV` is an example of a regular data parallel application. An input matrix of 800,000 x 2,000 elements is considered for the `MxV` benchmark. Specifically, the iterations over the rows of the matrix are computed with the extended `parallel_for` function template. This benchmark can be considered as a fine-grained application (it takes less than 1 ms to process one row, or iteration of the outer loop, on a CPU core). Also, for this problem, the computational speed of the GPUs was within the range $7 \leq f^k \leq 8$, where k represents one of the 4 GPU devices id.

For the `Barnes-Hut` benchmark, we adapt the code proposed by Kulkarni et. al [63], which is part of the Lonestart Benchmarks suite. This code is representative of an irregular application. An input set of 100,000 bodies is simulated in our experiments, where the bodies follow a Gaussian distribution into a 3D space. In this benchmark, the loop that calculates the gravitational force for each particle/body (`computeForce()`) is executed by using our proposed function template. This benchmark can be considered a coarse-grained application (it takes a few seconds to process a body/iteration in a CPU core). For this problem, the computational speed of the GPUs is within the range $3 \leq f^k \leq 4$, again k represents one of the 4 GPU devices id.

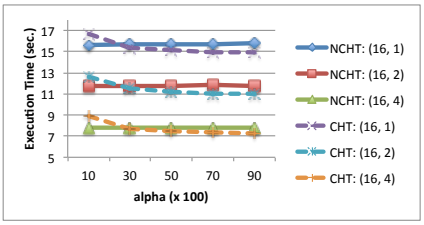
3.4.3. Characterization of the parallel_for template

For all the experiments conducted in this section the number of OS threads considered (the *nthreads* parameter in the initialization of the task scheduler) is equal to the number of CPU cores tested on each machine configuration: 8, 16 and 32, so no oversubscription is allowed. In the case of 1 GPU, we always execute on GPU^1, whereas for 2 GPUs and 4 GPUs, we execute on [GPU^1:GPU^2] and [GPU^1:GPU^4]. In this section, the fine-grained `MxV` benchmark is used as a case study to characterize the `parallel_for` template implementation.

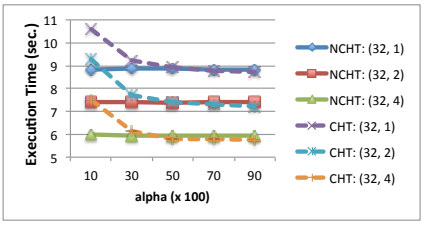
In our first set of experiments we measured the effect that factor α (used to



(a) 8 CPUs and (1, 2, 4) GPUs



(b) 16 CPUs and (1, 2, 4) GPUs



(c) 32 CPUs and (1, 2, 4) GPUs

Figure 3.10: Effect of parameter α on the execution times. The x-axis represents the range of variation of α (from 10 to 90) whereas the y-axis displays the execution time in seconds.

compute the exponential moving average of the throughput) has on performance. Figure 3.10 shows the execution time for $NCHT$ and CHT partition strategies with 8, 16, and 32 threads in Figures 3.10a, 3.10b and 3.10c respectively, and the number of GPUs varies between 1, 2 and 4. These Figures show execution times obtained for both partitioning strategies, $NCHT$ and CHT, when parameter α varies within the range [0.1 to 0.9]. A low α value means that the current throughput sample has less weight than the historic throughput value when computing the new average, whereas a high α value means exactly the opposite.

From Figure 3.10 we can draw an initial conclusion: For both `MxV` and `Barnes-Hut` benchmarks (only MxV is reported), we found that in the case of the $NCHT$ partitioning strategy, the value chosen for parameter α has no effect on performance for any machine configuration, as all $NCHT$ configurations exhibit a plain contour while varying *alpha*. In contrast, in the CHT strategy a low value of α clearly degrades performance, specially when the number of CPU cores is high. In general, a value of $\alpha = 0.5$ guarantees the best performance for all machine configurations (higher values of α tend to give similar execution times). Therefore, a value of $\alpha = 0.5$ is selected for the remaining experiments.

For this value of α, our partitioning heuristic quickly converge to the optimal CPU chunk size (after 3-4 assignments).

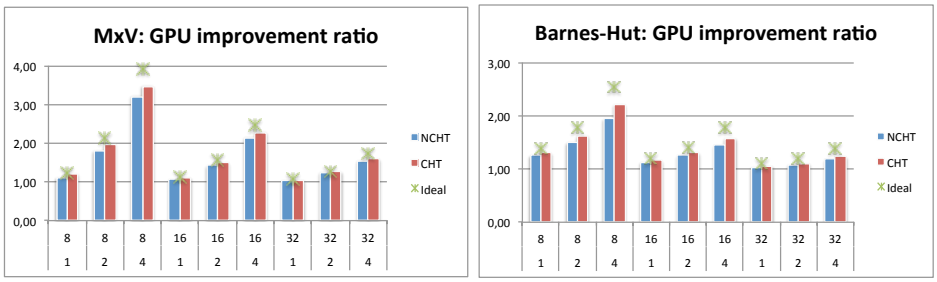
We also measure the overhead introduced by our engine, finding that for the `MxV` benchmark it was between 0,001% (8 cores) and 0,01% (32 cores). For the coarse-grained `Barnes-Hut` benchmark it was even smaller. This allowed us to set the threshold parameter ($threshold = 1$) for all our experiments. This value stablish the minimum amount of iterations that can be assigned to one CPU core.

3.4.4. Efficiency of the scheduling strategies

In this section, we focus on discussing and comparing the performance of $NCHT$ and CHT. In the previous section, the number of OS threads was equal to the number of CPU cores. In this experiment, we start by measuring the improvement achieved by the $NCHT$ and CHT when including the GPU devices on different socket configurations, first we consider a multicore of 8 CPUs, then a multicore of 16 CPUs and finally a multicore of 32 CPUs. In this study, we compute the ratio between the execution time of each benchmark in one multicore configuration ($T(nCores)$) and the time when adding 1, 2 and 4 GPUs ($T(nCores + nGPUs)$) to the same multicore. This ratio is named *GPU improvement ratio* and it is shown in Figure 3.11 for both partitioning strategies. Obviously, this ratio represents the speedup that each partitioning strategy achieves when we incorporate 1, 2 and 4 GPUs to a multicore. We also show the ideal improvement ratios, which are computed as $GIR = (\sum_k f^k + nCores)/nCores$. These ideal ratios represent the maximum computational speed of the heterogeneous system vs the speed of a multicore, or in other words, the maximum speedup we can achieve when we incorporate the acceleration of the GPU accelerators to the multicore. The ideal ratios are depicted as green stars in the figure.

Figure 3.11 shows that the CHT strategy always outperforms $NCHT$, obviously due to a better use of the CPU core where the host thread runs[2]. The results show us that the GPU improvement ratio is more significant when the number of CPU cores is small, for both benchmarks. In both cases, when increasing the number of CPU cores, the relative benefit of CHT over $NCHT$ decreases. Another interesting finding is that when the number of CPU threads is low (8 threads), the relative benefit of CHT is boosted when the number of GPUs increases. For instance, for the `Barnes-Hut` benchmark, the CHT strategy enhances the $NCHT$ performance of the (8,1), (8,2) and (8,4) configurations

[2]For $NCHT$, the host thread just block. Thus, the CPU core can only benefit in oversubscripted scenarios.



(a) MxV (b) Barnes-Hut

Figure 3.11: Ratio of the $NCHT$ and CHT times in a multicore vs the times in a heterogeneous configuration. Note that 1 is the performance in the multicore (only CPUs). The x-axis represents the number of CPU cores (8, 16, 32) and number of GPUs (1, 2, 4) on each heterogeneous configuration

by 4%, 8% and 14%, respectively. Clearly, more GPUs means more host threads that can take advantage of their respective CPU cores in CHT.

Based on the results shown in Figure 3.11, we can also explore another interesting question: how far is our partitioning heuristic from the ideal case? For it, we compare the GPU improvement ratio with the ideal GIR. From the figures we notice that ratios for $NCHT$ and CHT are 5%-20% and 2%-11% below the ideal ratio, respectively. These ranges are valid for both `MxV` and `Barnes-Hut`. The maximum deviation from the ideal value is for the configuration with the highest number of GPUs and lowest number of CPU cores: (8,1). The loss of efficiency in the $NCHT$ strategy is because the CPU core that runs a host thread is underutilized. This is alleviated in part by the CHT strategy, which attempts that the host thread uses the CPU core by collaboratively executing sub-ranges of work while the GPU is processing its assigned chunk. However, in this case, there is still some loss of performance due to the latency in the synchronization mechanism that we study next.

Analysis of oversubscription and synchronization mechanisms

In this section we discuss the effect of oversubscription as well as the different CUDA synchronization mechanisms on our partitioning strategies.

The second challenge we address in this chapter is the effective utilization of the CPU core that manages a GPU accelerator (data transfers and kernel

launches), a GPU host thread. As mentioned before, heterogeneous frameworks assume that the GPU host thread must be kept waiting for the GPU task completion and the reception of processed data on GPU. Instead of waiting (which may result into a waste of a CPU core and energy consumption), we have modified the partition strategy in *NCHT*. So, each time that a GPU device gets a new chunk of iterations, the GPU host thread also gets another proportional chunk to be executed in parallel on the CPU core. Thus, while the GPU host thread is processing its chunk of iterations in batches, it periodically checks the GPU's status until the GPU's driver notifies to the GPU host thread task the completion. We call this strategy Collaborative Host Thread, *CHT*.

One alternative approach to keep the CPU core working, consists in relying on oversubscription and blocking (or yielding) the GPU host thread while waiting for the accelerator to finish. The idea is to have an extra running CPU thread for each GPU accelerator available in the system. These threads will be dispatched to execute chunks of iterations on the CPU cores that remain idle while the GPU host threads block (yield) because of waiting for the GPU completion. More specifically, CUDA offers a function API to allow users to control the synchronization behaviour of the GPU host threads by using the `cudaSetDeviceFlags()` method. We analyse the behaviour of activating the following flags: *Spin*, *Yield* and *Blocking*. The default synchronization mode is *Spin*, in this mode the GPU host thread keeps busy waiting in order to reduce the latency time when the GPU notifies the host thread its task completion. StarPU also uses this strategy by default. By using the *Yield* mode, the host thread periodically runs and checks for the status of the GPU execution in a round-robin fashion. When no oversubscription is allowed (other concurrent ready threads), *Yield* mode behaves like *Spin*, although with some additional overhead due to a more frequent context switching. Finally, with *Blocking* the host thread just blocks until the GPU work is done. As the *Spin* synchronization mode wastes the CPU core that runs the GPU host thread by doing busy-waiting, we have implemented our *NCHT* strategy using a *Blocking* mechanism (although the *Yield* mechanism is also evaluated in Section 3.4.4) and we study the effects of oversubscription under this synchronization mechanism.

In Figure 3.12, we represent the execution time (y-axis) for the `MxV` benchmark in a hardware configuration with 8 CPU cores and 4 GPUs. Moreover, we use 8, 12 and 16 O.S. threads (x-axis). The 8-thread experiment illustrates the scenario of no oversubscription, while the other two stand for scenarios with moderate and high oversubscription.

From the Figure 3.12, we observe that in scenarios with no allowed oversubscription (8 threads, leftmost group of bars), one of our two proposed strategies,

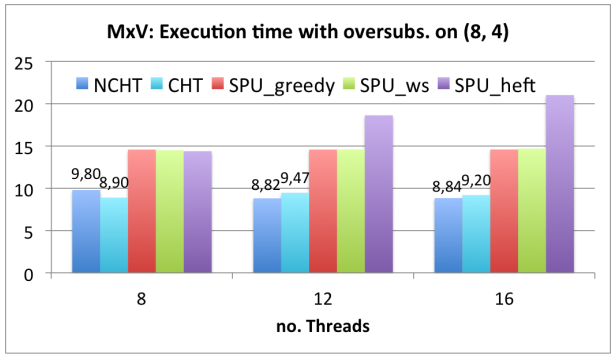
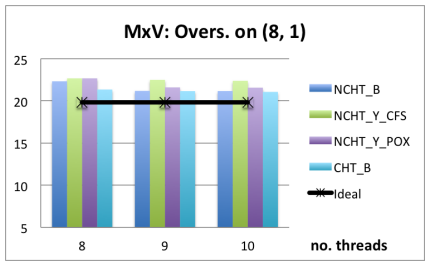


Figure 3.12: Performance comparison of NCHT and CHT against StarPU partition strategies. MxV execution times (seconds) while running on 8, 12 and 16 CPU threads with 4 GPUs.

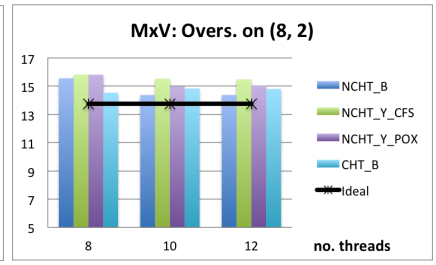
the `CHT` strategy is the most efficient, indeed it is a 10% faster than `NCHT` and a 61% faster than the best StarPU implementation, `SPU_heft`. Under the moderate oversubscripted scenario (12 threads, middle group of bars), `NCHT` along with Blocking synchronization mechanism is the most efficient alternative. In fact, it is the best alternative, as it is a 64% faster than the best `StarPU` strategy. In contrast, strategies that uses polling (`CHT`) or spin mechanisms do not improve their times and they even degrade their performance, as it's the case of `SPU_heft`. This performance degradation is even more important under high oversubscription scenarios (16 threads, rightmost group of bars). In this scenario, under these later types of synchronization mechanisms, the context switching and cache cooling overheads inherent to oversubscripted scenarios are evident. A more detailed analysis is covered in the experimental results Section 3.4. Anyway, these results encourage the development of strategies that fully utilize the GPU host thread, depending on the available synchronization mechanisms provided by the accelerator API and drivers. Either a *NCHT*-like strategy with moderate oversubscription when blocking policy is available, or a *CHT*-like strategy without oversubscription on the contrary.

Oversubscription may improve core utilization, especially in the case of the *NCHT* strategy, but it can also produce more overhead due to higher process of context switch. Also, increasing the number of threads has the potential to increase the duration of synchronization operations due to hardware contention. The user can control how the host thread interacts with the OS scheduler when waiting for results from the GPU device by calling the `cudaSetDeviceFlags()` function. We are not interested in the busy waiting mechanism (*Spin*) because it

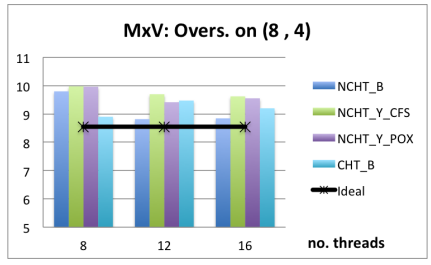
wastes the GPU host thread without performing any effective computation. For this reason, we have studied the *BlockingSync* (from now *Blocking*) and *Yield* flags. *Blocking* differs from *Yield* in that in the latter mechanism, the GPU host thread can still be periodically run by the scheduler and check for the status of the GPU execution, whereas in the former the GPU host thread just blocks until the GPU task is done and this thread receives a notification.



(a) (8,1) configuration



(b) (8,2) configuration



(c) (8,4) configuration

Figure 3.13: Execution time (in seconds) for different numbers of threads in the `MxV` benchmark. The left-most group of bars represents the case of no oversubscription.

Figures 3.13 and 3.14 present the execution time for all the experiments when they are executed on a 8 CPU cores socket and with 1, 2 and 4 GPUs. The ideal time is also represented: it is estimated as $T(nCores = 8)/((\sum_k f^k + nCores)/nCores)$, where $T(nCores = 8)$ is the time of each application in the 8 CPU cores socket without GPUs. The x-axis represents the number of threads and the y-axis the time in seconds. In each figure, the first group of bars always present the 8-threads case, i.e. no oversubscription. The next group of bars present the time for a moderate oversubscription scenario, i.e., 8 threads plus one additional thread per GPU device (i.e. 9, 10 or 12 threads confined in 8

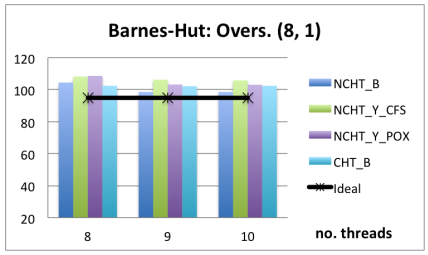
cores). Finally, the last group of bars presents the time for a high oversubscription scenario, i.e., 8 threads plus two additional threads per GPU device (i.e. 10, 12 or 16 threads confined in 8 cores).

For `MxV` and `Barnes-Hut` benchmarks, we evaluate `NCHT_B` and `NCHT_Y`. They represent the times for the *NCHT* strategy in which the *Blocking* and *Yield* mechanisms are evaluated, respectively. By setting the *Yield* flag, the system call `sched_yield` is invoked when reaching the synchronization function. Current Linux distributions allow two different behaviours for this system call. The default behaviour (referred to as Completely Fair Scheduling `CFS`) does not preempt the calling thread until its quantum expires, whereas in the POSIX conforming implementation (referred to as `POX`) the caller immediately relinquishes the CPU. In addition, `CHT_B` represents the times for the collaborative *CHT* strategy[3] in which the *Blocking* scheme is assessed. In this *CHT* strategy, worst times are obtained when a *Yield* mechanism is used.

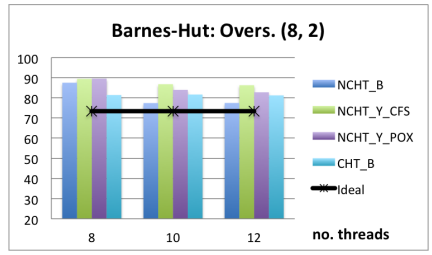
As shown in Figures 3.13 and 3.14, when there is no oversubscription (leftmost group of bars), `CHT_B` presents the best performance when compared with any `NCHT_` version. Also in this scenario of no oversubscription, the *Yield* mechanism performs worse than the *Blocking* one under the *NCHT* strategy. This difference is more evident in the (8, 4) configuration (see Figures 3.13c and 3.14c). We find out that the reason for this performance difference is the TurboBoost feature of the Xeon X7550 processor. This feature enables boosting the frequency of heavily-loaded cores when other cores are idle in the same package. Figure 3.15 shows the average frequency obtained for the `MxV` and `Barnes-Hut` codes for the *NCHT* strategy and the *Blocking* and the *Yield* (default CFS) mechanisms. Clearly, on average, the cores are running at a higher frequency when the *Blocking* strategy is used.

We obtain interesting results when we study the performance of our two partition strategies under oversubscripted scenarios. For instance, the performance of the *CHT* strategy degrades always for any machine configuration when moderate or high oversubscription is allowed in the system. The conclusion in this case is that oversubscription does not improve core utilization, and in fact context switch and cache cooling overheads degrade the execution times. On the other hand, although the non-collaborative strategy, *NCHT*, exhibit worse behaviour in the case of no oversubscription, it improves its performance when oversubscription is enabled, and for both *Blocking* and *Yield* mechanisms. The improvement of *NCHT* with oversubscription is more significant when the number of GPUs

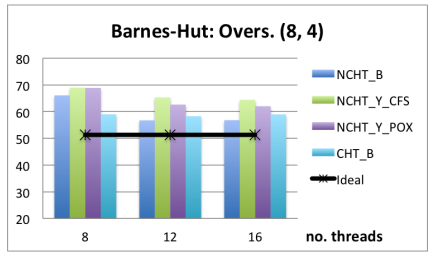
[3]Let's recall that a busy-waiting mechanism is used in the *CHT* strategy when the host thread checks the completion of the GPU work, although eventually also a `wait()` function call is performed, and this is the function affected by the synchronization flag.



(a) (8,1) configuration

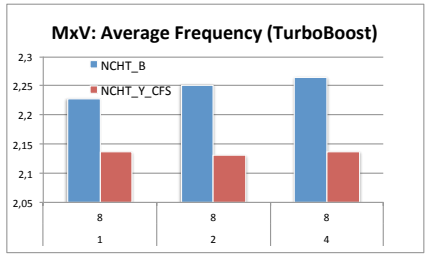


(b) (8,2) configuration

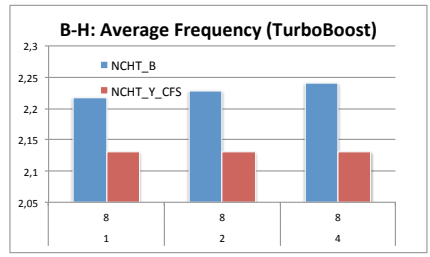


(c) (8,4) configuration

Figure 3.14: Execution time (in seconds) for different numbers of threads in the `Barnes-Hut` benchmark. The left-most group of bars represents the case of no oversubscription.



(a) `MxV`



(b) `Barnes-Hut`

Figure 3.15: Average frequency (GHz) that the Turbostat command report for our benchmarks. No oversubscription case: 8 threads on (8,1), (8, 2) and (8,4) configurations.

increases. This is due to the fact that core utilization is improved by allowing extra CPU threads which help to execute additional work on the CPU core that

is waiting for the completion of the GPU task. We also notice from the figures that the `POX` implementation performs slightly better than `CFS` for *Yield*, but in any case, again the *Blocking* mechanism outperforms the *Yield* one.

In summary, we can see that the overall best performance for MxV and Barnes-Hut benchmarks is achieved with the non-collaborative *NCHT* strategy that uses a *Blocking* synchronization mechanism and in moderate oversubscription scenario. In fact, in this case, the time of *NCHT* is slightly less than 1% above the ideal for MxV and around 3%-5% above the ideal for Barnes-Hut.

3.5. Conclusions

We have explored the possibility of extending a high-level `parallel_for` template that works under the parallel task programming paradigm to enable the effective utilization of accelerators (GPU devices) working in parallel with multicore systems in heterogeneous architectures. The extension of the template is based on a two-stages pipeline engine that is responsible for dynamically scheduling and partitioning the chunks into the computational units. Under this engine, we propose two adaptive partitioning strategies, *NCHT* and *CHT*, that resize chunks to prevent underutilization and load imbalance of CPUs and GPUs due to small or large block sizes. Our partitioning heuristic is based in an analytical model that takes into consideration the effective throughput of the computational resources. *CHT* also tackles the problem of effectively utilizing the CPU core where a host thread operates, by allowing that the host thread gets one chunk to process in parallel each time that launches work on a GPU device.

Using a regular and an irregular benchmark, we have evaluated the overhead introduced by our engine in a heterogeneous platform, finding that is negligible (less than 0.01%). We have also evaluated the behaviour and efficiency of both partitioning strategies, finding that a collaborative host thread strategy implemented at the application level (*CHT*) can be outperformed by a non collaborative host thread (*NCHT*) strategy combined with a blocking synchronization mechanism, when moderate oversubscription controlled by the OS is allowed. Our results encourage the development of strategies that fully utilize the host thread, depending on the available synchronization mechanisms of the host thread: either a *NCHT*-like strategy with moderate oversubscription when blocking policy is available, or a *CHT*-like strategy without oversubscription otherwise.

4 Parallel_for Pattern: Adaptive partitioning

In the previous chapter, we concluded that heterogeneous architectures, such as the ones with GPU accelerators, can be exceptionally powerful in performance, power, and energy efficiency. However, there are various challenges such as the programming complexities of heterogeneous parallel programming.

We are currently seeing a growing variety of heterogeneous processors, characterized by featuring several CPU cores and an accelerator on the same die. In this context, hardware vendors such as Intel, Qualcomm, AMD, Samsung or Altera have developed this kind of heterogeneous chips, where a multicore CPU and one accelerator (GPU, DSP or FPGA) share resources, such as bus memory controllers, cache memory and the energy budget. The success of these systems will rely on the ability to map the application level parallelism to exploit the underlying available devices. In particular, commodity processors are nowadays comprised of several CPU cores and one integrated GPU. These heterogeneous CPU-GPU chip architectures present new opportunities to improve overall application performance and reduce energy consumption. This can be achieved by mapping computation to the CPU cores and the integrated GPU. However, to fully exploit this type of architectures, one needs to automatically determine how to partition the workload between both processors and which is the best chunk size for each processor. This is specially challenging for GPUs while executing irregular workloads, that exhibit variations in the computation needs during execution time and suffers from control and memory divergences.

We consider the problem of efficiently executing the iterations of a parallel `for loop` on heterogeneous CPU-GPU chips by executing the iterations on both, the multicore CPU and the integrated GPU at the same time. This parallel

for loop requires a carefully partition of the iteration space into blocks of iterations (also called chunks) that should be appropriately selected for each processor to guarantee optimal performance. Thus, in this chapter, we present an adaptive partitioning strategy that adapts dynamically to changes in applications throughput. The chunk sizes assigned to the CPU cores are also dynamically computed to avoid load imbalance. Our scheduling strategy finds the right trade off between large and small chunk sizes, maximising the GPU utilisation while balancing the workload with the multicore CPU. We evaluate the performance and energy consumption of our approaches by running on Intel Haswell and Ivy Bridge architectures with OpenCL. In this Chapter, we first introduce an improved version of the `parallel_for` template presented in the previous chapter (Section 4.1). Later, we motivate the need for adapting the size of the offloaded range of iteration to GPU (Section 4.2). Next, we propose the partition strategy to deal with the aforementioned partition problem (Section 4.3) and show our experimental results (Section 4.4). We conclude with conclusions in Section 4.5.

In this chapter, we provide an extension of the `parallel_for` function template of the TBB task framework [89] to allow its exploitation on heterogeneous CPU-GPU chip processors. We have selected TBB because its task scheduler implementation is the most efficient when compared with other state-of-the art multicore task schedulers. However, although we have used TBB as the underlying runtime system, our scheduling and partitioning strategies can be applied to any other task framework. We provide our templated function along with a partition strategy that is able to balance the workload among all available devices (CPUs and GPUs) while dynamically selecting the chunk size that ensures a near optimal throughput on each device.

4.1. The extended parallel_for template

This section proposes an extension of the `parallel_for` template presented in Chapter 3. The main novelty is an adaptive partitioner that automatically finds near optimal chunk sizes for CPUs and GPUs. We integrate this extension within our Heterogeneous Building Blocks (**HBB**) library. It is a C++ template library that takes advantage of heterogeneous processors and facilitates its usage and configuration. HBB aims to make easier the programming for heterogeneous processors by automatically managing memory data buffers and accelerator synchronization. It builds on top of OpenCL and TBB libraries, and it offers a `parallel_for` function template to run on heterogeneous CPU-GPU systems, as it is depicted in Figure 4.1.

We use the *NCHT* scheduler [81] which is presented in Chapter 3, it considers loops with independent iterations and features a dynamic workload balancing policy and an adaptive GPU and CPU chunk partitioner. This partitioner adaptively divides the whole iteration space into chunks or blocks of iterations. The goal of the partition strategy is to evenly balance the workload of the loop among the computational units (GPU and CPU cores) as well as to assign to each device the chunk size that maximises its throughput during execution time. Providing an accurate chunk to each device is extremely important to achieve an optimal performance. Section 4.2 shows that the chunk size can have a significant impact on the performance of heterogeneous CPU-GPU architectures, specially when dealing with irregular applications.

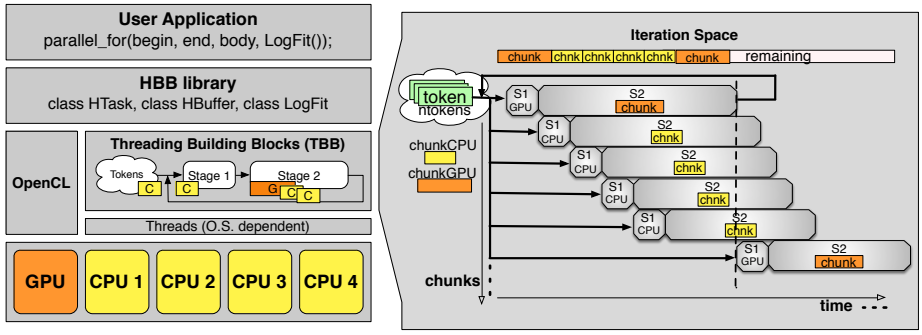


Figure 4.1: Software stack and scheduling approach used in the *parallel_for* template.

As mentioned before, we are extending the *NCHT* scheduler by adding a performance-aware partitioner. Hence, we remember the implementation details of this scheduling strategy. Figure 4.1 shows the software stack that supports user applications. Our library, HBB, offers an abstraction layer that hides the initialization and management details of TBB constructs and OpenCL contexts, command queues, device_ids, etc., thus the user can focus on his own application instead of dealing with thread management and synchronization. The Figure 4.1 also shows the internal engine that drives the `parallel_for` function. Again, it is a two-stage pipeline. This scheduler is internally implemented as a pipeline, comprised of two stages: Stage_1, which selects the computing device (GPU or CPU core) where the work is scheduled and the chunk size (number of iterations) assigned to that device; and Stage_2, which processes the chunk on the selected device. Stage_1 firstly checks if the GPU device is available. In that case, a

G_token is created and initialized with the range of the GPU chunk size that the partitioner returns. If there is no idle GPU device, then a CPU core is idle; thus, a C_token is created and initialized with the appropriate chunk of iterations for the CPU. In both cases, the partitioner extracts a chunk of iterations from the range of remaining iterations. Next, $Stage_2$ processes the chunk in the corresponding device depending on whether the token is a G_token or a C_token and records the time it takes to compute the corresponding. This is necessary to compute the device's throughput, which is used by the partitioner described in Section 4.3.

One of the biggest advantages of this `parallel_for` implementation [81] is the decentralised approach of computing, where each computing device (GPU or CPU core) is represented as a token that traverses the pipeline at its own pace. Thus we avoid unnecessary synchronization points between computational devices with different computing power. In contrast, other state of the art approaches [70, 111] suffer from load unbalance due to the usage of `fork-join` patterns with implicit synchronization points between CPU and GPU. In the rest of this section, we explain the functionality and implementation details of the main HBB components.

4.1.1. HBuffer class

The HBB library provides an `HBuffer` template class that offers an abstraction to avoid the explicit management of memory buffers. This class is accessible by including the `HBuffer.h` header file and by using the namespace `hbb`. Each `HBuffer<T>` instance represents an heterogeneous buffer that can be accessed by CPU and GPU. As we can see in Figure 4.2, this class hides memory data management, the user just needs to call the methods `getHostPtr()` (lines 11 and 12) to get a CPU memory buffer, and `getDevicePtr()` (lines 19 and 20) to get access to the GPU memory buffer, respectively. In line 6 a pointer to an `HBuffer<int>` is declared, this buffer can be accessed later from the methods `operatorCPU()` and `operatorGPU()` by using the previously mentioned methods. The allocation of the `HBuffer` instances is shown in Figure 4.3 (lines 17 and 18), the default constructor of the class takes an argument which represents the number of items to be stored in the buffer (line 17). Moreover, there is an additional constructor definition that takes a second argument to set the zero-copy-buffer (ZCB) mode (line 18). This mode allows developers to allocate the CPU and GPU buffers in the same physical memory addresses, as those platforms have a fused/integrated CPU-GPU processor. However, discrete GPUs can also use this mode to exploit the pinned memory in their platforms. In any case, the user is responsible of choosing the right memory mode (ZCB or default)

because the performance behaviour of the ZCB mode is strongly dependent of the underlying hardware architecture and platform drivers.

4.1.2. HTask class

Before using the `hbb::parallel_for` function, the user must extend the `HTask` abstract class in order to define the body of the parallel for loop (line 5 in Figure 4.2). First, the HTask.h header file has to be included to make its definition available. Figure 4.2 shows a snippet of code with the definition of an arbitrary `Body` class that extends from `HTask` class. This class must implement two methods: one to define the kernel on a single CPU core, and a second one to define the kernel on the GPU device using OpenCL. The `operatorCPU()` method (lines 10-14) defines the CPU function of the kernel in C++, which will be run on a CPU core. The `operatorGPU()` method (lines 15-23) represents the argument setting and kernel launching on the GPU. Note that the user is not responsible of loading and compiling the kernel, as it is automatically done by the HTask constructor when it receives the `KernelInfo` parameter (line 9 in Figure 4.2). The `kernelInfo` struct has two fields: one to store the kernel file path (`KernelFile`), and a second one to store the kernel function name (`KernelName`), as it is shown in Figure 4.3 (line 16).

The `HTask` class automatically manage data memory transfers between devices (CPU-GPU). In this sense, the user only has to set the arguments that are passed to the GPU kernel. There are two methods to set the kernel arguments: the `setKernelArgument()` method for variables of basic types (line 18), and the `setKernelArgumentBuffer()` method for instances of the class `HBuffer<T>` (line 19). This last method, `setKernelArgumentBuffer()`, receives a third parameter with a pointer to the data buffer that is marked with a buffer access mode to the kernel, it can be one of the following values: `BUF_READ_ONLY, BUF_WRITE_ONLY and BUF_READ_WRITE`. Depending on the type of buffer access mode and the internal information of the HBuffer instance the method `setKernelArgumentBuffer()` can apply several optimizations to avoid unnecessary data transfers between devices, as it uses a lazy memory management policy. Additionally, the `HTask` class provides a `launchKernel()` method to execute the kernel on the GPU (line 22). This method only takes two arguments that represent the chunk of iterations to be executed on the GPU. Note that the user does not have to manage the `command_queue` or the `kernel_id` objects inherent to GPU applications.

```
#include ``HTask.h''
#include ``HBuffer.h''
using namespace hbb;

class Body : public HTask{
   HBuffer<int> * b_a;
   HBuffer<int> * b_b;
public:
  Body(KernelInfo k, HBuffer * buf_a, HBuffer * buf_b) : HTask(k){...}
  void operatorCPU(int begin, int end) {
     int * a = b_a->getHostPtr(BUF_READ_ONLY);
     int * b = b_b->getHostPtr(BUF_READ_WRITE);
     for(i=begin; i!=end; i++){ b[i] = a[i] * a[i]; }
  }
  void operatorGPU() (int begin, int end){
     //Setting kernel arguments
     setKernelArgument(0, sizeof(int), &begin);
     setkernelArgument(1, sizeof(int), &end);
     setKernelArgumentBuffer(2, sizeof(BUF), b_a->getDevicePtr(BUF_READ_ONLY));
     setKernelArgumentBuffer(3, sizeof(BUF), b_b->getDevicePtr(BUF_READ_WRITE));
     //Launching kernel
     launchKernel(begin, end);
  }
};
...
```

Figure 4.2: Implementation example of a class *Body* that extends from *HTask*.

4.1.3. Function template: parallel_for

Our scheduler builds on top of an extension of the TBB `parallel_for` template for heterogeneous CPU-GPU systems by Navarro et al. [81]. As in any multi-threading library, the scheduler needs to be initialized with the number of OS threads, that the TBB runtime will create, which can vary from 1 to the number of CPU cores plus one additional thread to host the GPU (the host thread). The developer can invoke our `parallel_for` function, which has the four following arguments: the iteration space (begin and end), the body object of the loop, and the partitioner object (`LogFit()`). The latter argument, effectively overloads the native TBB `parallel_for` function so that the heterogeneous version is invoked. It implements the adaptive partitioning strategy that computes the optimal chunk size for each compute device (CPU cores or GPU). This is described in Section 4.3. The user is also responsible to write a `class` that processes the chunk on the CPU cores or on the GPU. Our function template can work on different types of heterogeneous systems, but in this chapter, we focus on architectures with an on-chip GPU. Thus, our scheme does not constrain the memory management model, so the user can set buffers as default, or pinned-memory host buffers, or zero-copy buffers. While the default model always pays a cer-

tain data communication overhead between CPU and GPU, the pinned memory stores the CPU data in resident memory pages, reducing latency. Alternatively, the zero-copy buffer model allows the GPU to access the CPU memory space directly, so no data movement is needed. Moreover, all experiments conducted in this chapter use the zero-copy buffer capability of the heterogeneous CPU-GPU chip architectures.

```
1  #include ''HInit.h''
2  #include ''HBuffer.h''
3  #include ''parallel_for.h''
4  using namespace hbb;
5
6  class Body : HTask{
7    ...
8    void operatorCPU(int begin, int end){ ... }
9    void operatorGPU(int begin, int end){ ... }
10 };
11
12 int main(int argc, char* argv[]){
13   // Start task scheduler
14   HInit HInit(numcpus, true);
15   ...
16   KernelInfo k(kernelFile, kernelName);
17   HBuffer<int> * a = new HBuffer<int>(N);
18   HBuffer<int> * b = new HBuffer<int>(N, USE_ZCB);
19   Body body(k, a, b);
20   ...
21   parallel_for(begin, end, body, new LogFit());
22   ...
23 }
```

Figure 4.3: Usage example of the *parallel_for* template with *LogFit* partitioner.

Figure 4.3 shows a main function with all the required component allocation and initialisation to make the `parallel_for` function template work. The `parallel_for` function is made available by including the `parallel_for.h` header file. As in any threading library, the first step is to initialise the library with the required number of resources (threads). In the `HBB` library, the class `HInit` abstract the initialization of the underlying libraries (TBB and OpenCL). The constructor of the `HInit` class receives two arguments: the first one indicates the number of active CPU cores, and the second one, which indicates whether the GPU must be initialised or not by setting a `boolean` value (line 14), if the GPU has to be initialised, an extra thread is created to host the GPU accelerator.

Once the library has been initialised, the user can create the `KernelInfo`, `HBuffer` and the `Body` object instances (lines 16-19), which are required to run the `parallel_for` function. As shown in Section 4.1.2, the `Body` constructor class receives a `KernelInfo` instance that will be bypassed to the `HTask`

constructor in order to compile and create the GPU kernel. The other parameters, the `HBuffer` instances, are passed to the `Body` instance to make them accessible to the `operatorCPU()` and `operatorGPU()` methods. Additionally, the `parallel_for` function template receives four parameters (line 21): the first two parameters, `begin` and `end` represent the limits of the iteration space ($[begin, end)$, i.e. the upper limit is not executed). The third parameter is the `Body` instance which have the implementation of the CPU and GPU versions of the body loop. The last parameter is an instance of a partitioner. This partitioner effectively implements the adaptive partitioning strategy that computes the optimal chunk size for each compute device (CPU cores or GPU). In this case, it is an instance of the class `LogFit` that is explained in the Section 4.3.

4.2. The GPU chunk size problem

Developing a dynamic and adaptive work distribution mechanism that is portable across processors is challenging, and even more when the computational needs of the application may change during running time, as it happens in irregular applications. There are several frameworks that offer support for heterogeneous CPU-GPU systems, like StarPU [2], OmpSs [10], XKaapi [40], Qilin [70], HDSS [4], Fluidic [84] and Concord [58]. These task frameworks implement a variety of dynamic scheduling and partition strategies which aim to balance the workload between CPUs and GPUs. In general and assuming that host-to-device and device-to-host transfer times are not an issue, as it happens in chips with integrated accelerators that share the main memory, these strategies would consider that a large chunk size that fully occupies the GPU's computational units, will exhibit a maximum throughput. In this chapter, we demonstrate that not only a small chunk size, but also a large chunk of iterations may exhibit a poor throughput, especially when running irregular applications. In the first case, this is a consequence of underutilization resources, however in the latter, this is due to the large amount of stalled computational units that are waiting for last level cache misses.

Our proposed approach continuously monitors the throughput of each computing device (CPU cores and GPU) during the whole application execution and uses this metric to accordingly resize the chunks assigned to the CPU cores and to the GPU, optimising overall throughput and decreasing load imbalance among the different devices. Particularly, the correct identification of the optimal chunk size for the GPU is key. We have found that correctly determining the chunk size for the GPU is highly important. Let's note that the user has to provide a fixed

chunk size for GPUs on the previous Chapter 3. For instance, if the GPU chunk size is too small, the GPU may not amortize the cost of offloading the task, and may not feed all computational units. On the other hand, if the GPU chunk size is too large, the GPU may suffer from a large memory contention and it may lead to workload unbalances at the end of the iteration space or inefficiencies in the GPU exploitation. For these reasons, dynamically finding the optimal chunk size for each device is critical to minimise execution time and energy consumption.

As a running example, we use an implementation of the n-body problem, called Barnes Hut, which is previously introduced in Chapter 3. We remind the main characteristics of this benchmarks for the sake of readability. It performs a gravitational particle simulation for a number of time-steps. In particular, this algorithm recursively divides the simulation volume into cubic cells by using an `Octree`. Then, only particles from nearby cells directly interact between them, and particles in distant cells can be treated as a single large particle centred at the cell's centre of mass. Thus, this method dramatically reduces the number of required interactions between particles[1], resulting in an application with a time complexity of $O(n \log n)$. We perform several invocations of the `parallel_for` function, each invocation correspond with one simulation step, also called *time-step* throughout this thesis.

Our studies show that, as expected [11], the amount of work performed by each iteration of the `parallel_for` exhibits a high variability. To understand how the range of iterations offloaded to the GPU affects performance in this benchmark, we conduct some experiments. These experiments are carried out on an i7-4770 Intel processor based on a Haswell architecture with four CPU cores and one integrated on-chip GPU. The frequency of the CPU cores varies between 3.4 and 3.9 GHz. The on-chip GPU is an Intel HD 4600, it has 20 Execution Units and a frequency domain between 350 MHz and and 1.2 GHz. The last level cache (LLC) has 8 MB, an it is shared by both the CPU cores and the GPU.

Impact of the chunk size over GPU performance

We find that, in the context of irregular applications as our running example Barnes Hut, offloading large blocks of iterations (chunks) to the GPU does not deliver higher performance. In fact, it can degrade the overall GPU throughput. This is illustrated in Figure 4.4, where we show the evolution of several GPU hardware-based metrics for the first time-step and an input set of 100,000 particles (the iteration space) for Barnes Hut. Each figure represents the evolution of the metric of interest when we offload chunks of fixed size (see chunk sizes legend in Figure 4.4b) to the GPU. We use Intel VTune Amplifier 2015 [55] to collect these

[1] with respect to the brute force N-Body implementation which exhibits $O(n^2)$.

metrics. Figures 4.4a to 4.4b show the ratio of cycles for which all the EUs are in the Active (EU Active), Idle (EU idle) or Stalled (EU Stalled) state respectively. The following expressions show the definition of the three possible EU status:

- The ratio of cycles when EUs are active:

$$\frac{\sum_{all\ EU}\ cycles\ when\ EU\ executes\ instructions}{\sum_{all\ EU}\ all\ cycles}$$

- The ratio of cycles when EUs are stalled:

$$\frac{\sum_{all\ EU}\ cycles\ when\ EU\ don't\ execute\ instructions\ while\ scheduled}{\sum_{all\ EU}\ all\ cycles}$$

- The ratio of cycles when EUs are idle[2]:

$$\frac{\sum_{all\ EU}\ cycles\ when\ no\ thread\ is\ scheduled\ on\ EU}{\sum_{all\ EU}\ all\ cycles}$$

Figure 4.4d represents the LLC cache misses due to GPU memory requests and Figure 4.4e shows the GPU effective throughput, measured as the number of executed iterations per millisecond. Note that data transfer and kernel launch overheads are included in the computation of the throughput and all the other metrics.

As we can see in Figure 4.4e, the chunk size that gets the maximum throughput throughout the iteration space is 640 (see green line in the figure). Note that for irregular applications, the best chunk size may vary through the iteration space (as we discuss in section 4.2). Increasing the chunk size beyond this value degrades the throughput. The hardware metrics indicate that small chunk sizes (i.e. 320) do not effectively feed all the available EUs, as the ratio of EU Idle indicates in Figure 4.4c (see blue line). In contrast, when the chunk size is large enough to feed all the EUs (EU Idle <0.1 for chunks >320), then the EUs utilization improves. However, looking at Figure 4.4a, we find out that chunk sizes higher than 640 decrease the ratio EU Active. Irregular benchmarks like Barnes Hut usually exhibit uncoalesced memory accesses (memory divergences)

[2]Note that the ratio of Idle EUs is equal to (1 − (EU Active + EU Stalled))

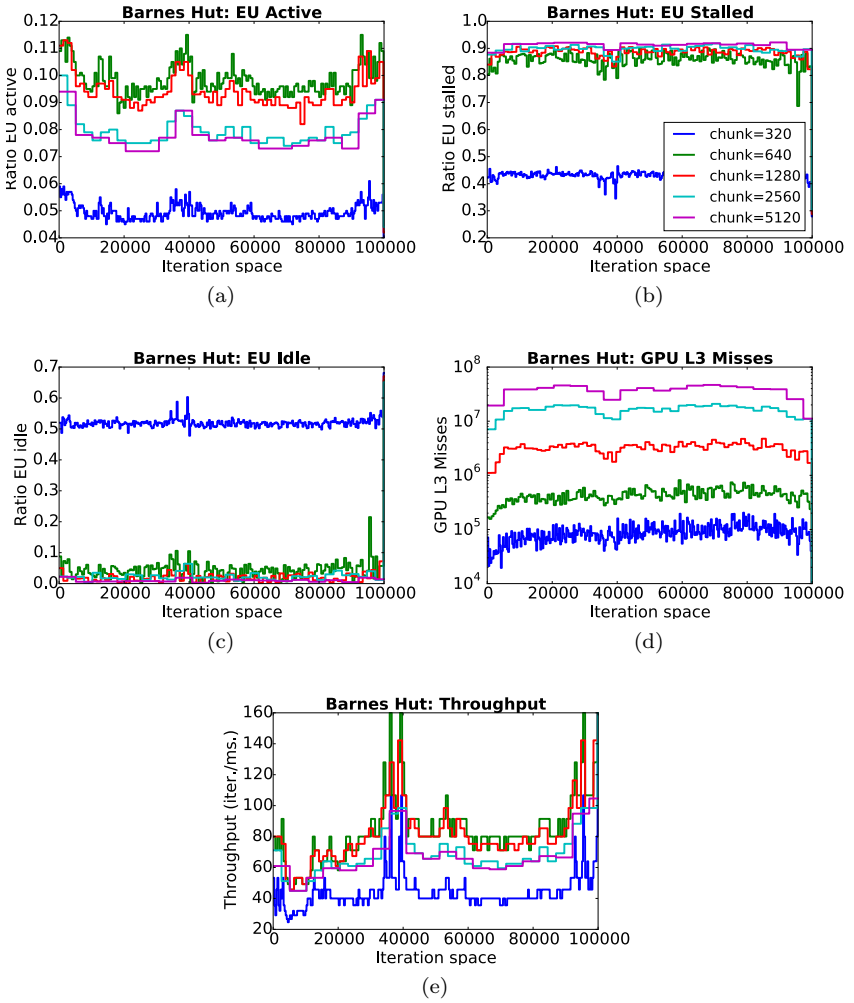


Figure 4.4: Evolution of GPU hardware-based metrics while executing the first time-step of Barnes Hut on the Intel HD Graphics 4600. The legend of subfigure 4.4b applies to all subfigures, it shows several iteration block sizes used to split the iteration space, values start at 320 until 5120.

that can lead to scenarios where most of EUs are stalled due to the contention for the memory bus controllers. As pointed out in a previous work [11], Barnes Hut exhibits an uncoalesced memory access pattern that may represent between 65 to

75% of the total number of issued instructions. Such a pattern is the responsible for the increment in the ratio EU Stalled when the chunk size increases. This is corroborated by the increment in L3 cache misses when the chunk size increases (see cyan and pink lines in Figure 4.4d), which clearly increases the pressure in memory bus. In our case, chunk sizes larger than 1280 dramatically increase L3 misses, which in turn increases the ratio of EU Stalled (> 0.9) and reduces the ratio of EU Active (< 0.08), causing a drop in the effective throughput as we see in Figure 4.4e.

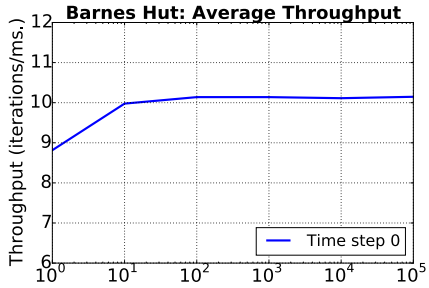


Figure 4.5: Average CPU's throughput (iter./ms.) for different chunk sizes while running with 1 core. Note the log_{10} scale in x-axis.

Therefore, in the quest of finding the optimal distribution of work between the GPU and the CPU, we must also consider that if we assign a large chunk of iterations to the GPU we can end up by not exploiting the EUs optimally. Thus, an efficient partitioning strategy must be aware of the optimal chunk size that must be offloaded to the GPU.

On the other hand, we observe that the effective throughput for one CPU core is not that sensible to the chunk size, as we can see in Figure 4.5. This is because the CPU cores are provided with other architectural features that hide more effectively memory divergences than GPUs. In fact, for our Barnes Hut example, as long as the chunk size is bigger than a certain threshold value[3], the average CPU throughput tends to be constant independently of the chunk size. Figure 4.5 shows a similar chunk size study for Barnes Hut benchmark while computing on CPUs. We corroborate that CPU chunks of 10 or more iterations always obtain the maximum constant throughput, independently of the benchmark regularity.

[3]For instance, Threading Building Blocks library (TBB) [89], recommends to have a CPU chunk size that take 100,000 clock cycles at least.

In next section, we discuss the need to not only adapt the chunk size during the first time-step but performing the adaptation during the whole execution time. We analyse how throughput changes between time-steps.

Chunk size effects over GPU performance across different time-steps of *BarnesHut*

Irregular applications that require a kernel invocation on each time-step, as it is the case of Barnes Hut, can potentially exhibit different performance behaviour on each kernel invocation. To illustrate this fact, Figure 4.6 shows the evolution of the GPU's throughput (measured as iterations per ms.) for some of the fixed chunk sizes studied before and throughout the iteration space of our Barnes Hut running example. At this point, we illustrate the throughput evolution for three time-steps: *time-step=0*, *time-step=5* and *time-step=30*. Again, the measured throughput considers the time due to memory transfers (device-to-host and host-to-device operations), kernel launching plus kernel execution. In any case, the figure shows that the effective throughput not only changes throughout the iteration space, it also changes in different time-steps. For instance, the intervals with higher throughput (or lower number of computations) are not always the same across different time-steps, e.g., the first iterations have a low throughput in time-steps 0 and 5, but a high throughput in time-step 30 (this can be seen at the beginning of the iteration space for the three time-steps). Another observation is that the chunk size that usually obtains high throughput in time-step 0 ($chunk = 640$), obtains low throughput in time-step 5 and 30. Moreover, for a given time-step, there is not a single chunk size that always obtains the highest throughput: this can be noticed, for example, in time-step 5 where for some iterations (between $[0 - 20000]$) the throughput obtained with $chunk = 2560$ is higher than the throughput obtained with $chunk = 1280$, while in the rest of iterations the throughput obtained with $chunk = 1280$ is maximum.

In the following experiment, we compare the behaviour of the previously introduced irregular application, Barnes Hut, and the behaviour of a regular application, Nbody which also computes the forces among a set of particles with a time complexity of $O(n^2)$. Figure 4.7 shows the average GPU throughput for different chunk sizes, for Nbody (time-steps 0 and 30) and Barnes Hut (time-steps 0, 5, and 30). The data is collected by executing all the iterations in a given time-step with a fixed chunk size (x-axis), for chunk sizes in the range $[20, ..., 20 \times 2^i, ..., 81920]$, where i goes from 0 to 12 and 20 is the number of execution units, nEU, on the Haswell's GPU. Figure 4.7(a) shows the behaviour of the regular Nbody application, where $chunk = 640$ obtains an optimal average throughput and larger chunk sizes have a minimum impact on throughput (as it is proved in [4, 111] too). These results are stable across all time-steps (0, 30).

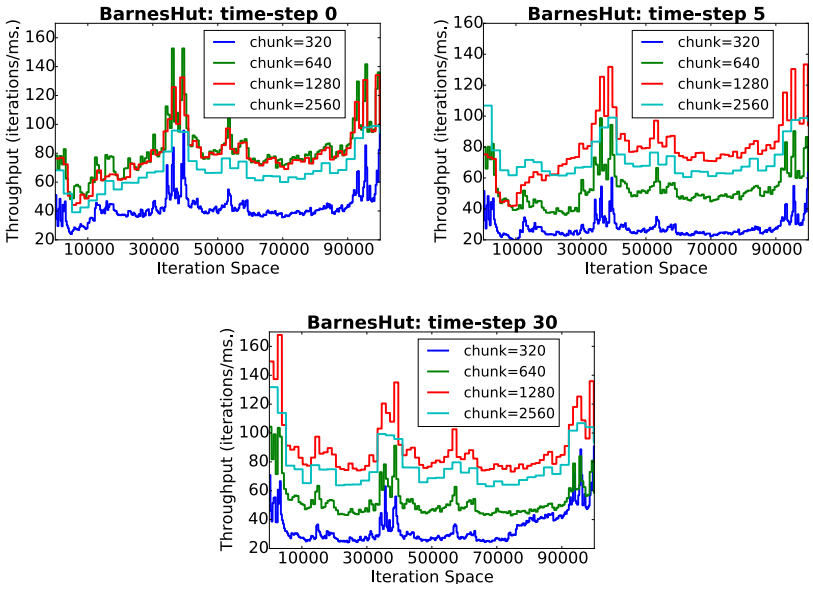
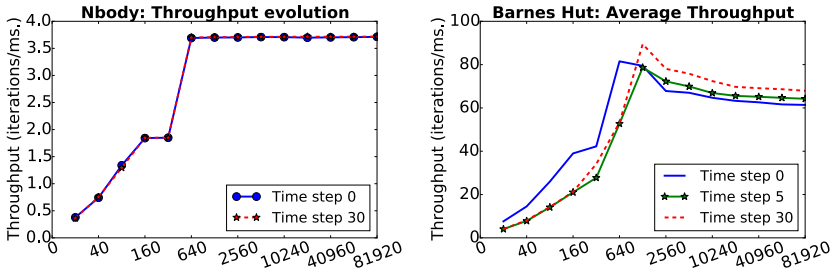


Figure 4.6: GPU's throughput (iter./ms.) for Barnes Hut and different time-steps (0, 5, and 30). The legends show several iteration block sizes used to split the iteration space, values start at 320 until 2560.

However, Figure 4.7(b) shows a very different scenario for the irregular application, Barnes Hut. Here, in time-step 0, $chunk = 640$ obtains the highest average throughput, while in time-steps 5 and 30 the highest average throughput is obtained with $chunk = 1280$. For all time-steps, the application takes an important performance penalty beyond those points.

There are some reasons that explain the GPU throughput behaviour shown in this section. On one hand, while offloading small chunk sizes result in low throughput due to: i) an insufficient amount of work offloaded to the GPU; and ii) the overhead of data transfer and kernel launch. On the other hand, larger chunk sizes must be carefully selected in order to make the most out of the GPU. As previously mentioned, Burtscher et al. [11] illustrate that Barnes Hut accesses memory positions in an uncoalesced manner, with a number of memory operations that oscillates between 65 to 75% of the total number of issued instructions. Since many threads are trying to concurrently access uncoalesced memory addresses, contention for the shared memory controller increases. Therefore, for this kind of codes, a too large GPU chunk size might be counter-productive, as



(a) NBody (b) Barnes Hut

Figure 4.7: Average GPU's throughput (iter./ms.) for fixed chunk sizes and different time-steps. Note the log_2 scale in x-axis. Each point represents the average throughput resulting from splitting and executing the iteration-space with a given fixed block-size.

Figure 4.4e shows. Notice that previous works have not performed a sensitivity analysis as the one shown in Figures 4.4 and 4.6 using irregular applications. Thus, they assume that always offloading large blocks of iterations (chunks) to the GPU is beneficial. They do not impose any restriction on how large the GPU block size should be, as they do not consider that large GPU block sizes can harm performance, because large chunk sizes are not harmful for regular applications like dense matrix multiplications.

These results illustrate that finding the optimal chunk size for irregular applications is challenging, because it might not be a constant, but rather change during the program execution. Clearly, irregular application can benefit from an adaptive mechanism to compute the optimal GPU's chunk size throughout the whole iteration space and all the time-steps. As we are interested in the collaborative execution between GPU and CPU cores, we also consider that offloading large chunk sizes to GPU may result in load unbalance, especially at the end of the iteration space. In any case, our partition strategy, LogFit, which we describe next, tackles the described issues: finding an optimal chunk size for the GPU and the CPU while balancing the load among the devices.

4.3. Partitioning strategy

In this section, we describe the details of `LogFit`, our partition strategy. LogFit targets *parallel_for* loops that run onto heterogeneous processors. It searches

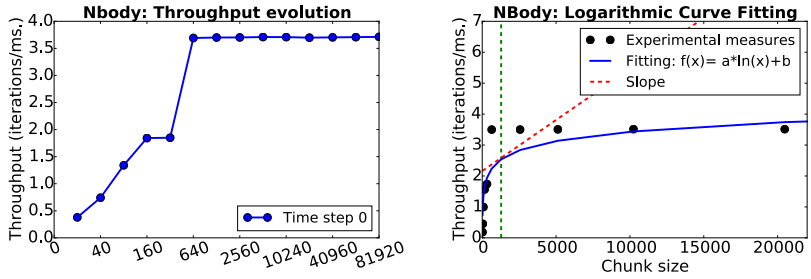
for a chunk size that maximises throughput for each available processor (CPU or GPU), and later it varies the chunk size to adapt it to irregularities in throughput and load unbalances. First, we present an overview of the problem to solve and then we introduce the design of our proposal.

4.3.1. Overview of the partitioning strategy

A previous work [4] states that in the context of regular workloads, the GPU throughput (iter. / ms.) follows a logarithmic curve with respect to the size of the chunk of iterations: $f(x) = a\ln(x) + b$, being x the chunk size and $f(x)$ the throughput (number of iterations / unit of time). We corroborate that finding by executing the NBody [54] benchmark. As introduced in Section 4.2, NBody performs a gravitational particle simulation by using a brute force method, that computes the force exerted on each particle due to its interaction with all the other particles in the system. Thus, all iterations perform the same amount of computations.

Figure 4.8(a) shows the effective throughput while running Nbody on the Intel HD 4600 GPU with 100,000 bodies. For this experiment, we start offloading chunks from a small size (equal to the number of compute units, 20, for our Intel HD Graphics 4600 GPU), and keep multiplying it by 2 for each new point. As we can see, chunk sizes smaller than 640 perform poorly because they do not feed all GPU's Execution (EUs). However, for chunk sizes bigger than 640, the application exhibit a constant optimal throughput around 3,6 (iteration/millisecond). In this sense, the GPU throughput can be approximated with a logarithmic curve while executing regular applications. Figure 4.8(b) shows an example of the logarithmic curve fitting to predict the GPU's throughput for a given chunk. In this case, we can accurately compute an optimal chunk size by collecting some points of the curve (black points in the Figure) that are recorded at runtime for different chunk sizes. Then applying least squares fitting method to the collected data points, we can calculate the values a and b of the expression $f(x) = a\ln(x) + b$ (blue line in the Figure).

After fitting the logarithmic curve, we use a reference value (explained in Section 4.3.2), to determine the point at which the estimated throughput is stabilized. This point represents the chunk size with a throughput that can be considered as near-optimal. By using this fitting procedure, we find that the recommended chunk size for NBody is 700, which is corroborated by inspecting Figure 4.8(a), where chunks of iterations bigger than 640 get optimal throughput. Unfortunately, as Figure 4.7(b) shows, irregular applications do not follow



(a) x-axis Logarithmic scale (b) x-axis Regular scale

Figure 4.8: GPU's throughput (iteration/ms.) while executing regular workloads as NBody and its Logarithmic fitting.

this logarithmic behaviour. However, our proposed LogFit heuristic's assumes that irregular applications can be modelled as a sequence of regular intervals. Thus, the optimal throughput for irregular applications can be approximated by finding the near-optimal throughput for each interval within the iteration space, $[I_0, I_1, \ldots I_{i-1}, I_i, I_{i+1}, \ldots]$. In consequence, the near-optimal throughput of each interval can be estimated by using the previous logarithmic expression, and from that expression we get the near-optimal GPU chunk size as we explain in the remainder of the section. Hence, LogFit is designed to orchestrate the scheduling of both, regular and irregular workloads.

4.3.2. Implementation details of the partitioning strategy

In the design of our LogFit partitioning strategy, we assume that the execution of a parallel for loop can be seen as a sequence of scheduling intervals $\{I_{G_0}, I_{G_1} \ldots I_{G_{i-1}}, I_{G_i}, I_{G_{i+1}} \ldots\}$ for the GPU and $\{I_{C_0}, I_{C_1} \ldots I_{C_{i-1}}, I_{C_i}, I_{C_{i+1}} \ldots\}$ for each CPU core. Each computing device at its $i-th$ interval, I_{G_i} or I_{C_i}, computes a chunk of iterations of size $Ch(I_{G_i})$ and $Ch(I_{C_i})$, respectively. The running times for the assigned GPU chunks, $T(I_{G_i})$, and CPU chunks $T(I_{C_i})$, are recorded. These times are used to compute the throughputs in the corresponding interval, $\lambda(I_{G_i}) = Ch(I_{G_i})/T(I_{G_i})$ for the GPU and $\lambda(I_{C_i}) = Ch(I_{C_i})/T(I_{C_i})$ for a CPU core.

In the manner now being indicated, we keep monitoring the throughput in order to be aware of throughput variations (rises/drops) and adjust the chunk size to avoid device starvation or load unbalance scenarios. In this sense, if there is a

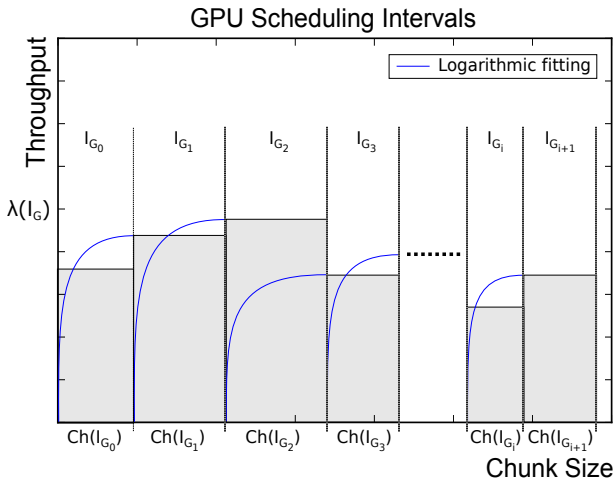


Figure 4.9: GPU scheduling intervals while scheduling an irregular application. LogFit fits a logarithmic curve for each scheduling interval in order to find a near optimal chunk size in each scheduling interval.

change in the throughput, it is because the workload regime has changed and we need to adapt the GPU chunk size accordingly. If the throughput has increased with respect to the previous time interval, this is due to a decrement in the time per iteration to compute this last assigned chunk, or in other words, either the workload per iteration has decreased or the number of cycles that the EU have been stalled has been reduced due to an increment in the number of coalesced memory accesses. In any case, a problem of GPU under-utilization may appear. Thus, we increase the chunk size in order to increase Thread Level Parallelism (TLP), and guarantee the full utilization of GPU's EUs. If the GPU is not under feed, enlarging the chunk size will not worsen the throughput, otherwise the throughput will likely increase in these scenarios. On the contrary, whether the throughput has fallen, then the time per iteration has incremented while computing the last assigned chunk. This issue might be either due to an increment in the computation per iteration or an increment in the number of cycles that the EUs have been stalled, that can be produced by a higher number of uncoalesced memory accesses. In these cases, we reduce the chunk size in order to alleviate a potential problem of over-provisioned GPU and for the sake of better load sharing with the CPU. Figure 4.9 graphically depicts that LogFit applies a logarithmic

curve fitting and recommends new chunk sizes according to throughput variations.

As introduced in Chapter 3, we also compute what we call the factor f, let's remind that it represents the *computational speed* of the GPU device relative to a CPU core in interval I_{G_i}. This computational speed is defined as the ratio of the GPU throughput w.r.t. the throughput of a CPU core, $f(I_{G_i}) = \lambda(I_{G_i})/\lambda(I_{C_i})$. The factor $f(I_{G_i})$ is used to adaptively adjust the size of the next chunk assigned to a CPU core. By doing that, we keep the CPU cores working with chunks sizes that feed all their computational resources and let us balance the workload at the end of the iteration space. Figure 4.10 shows a Barnes-Hut profiling execution where each compute unit (CPU core or GPU) has a sequence of rectangles that represents the time intervals of each device, where we can observe how the CPU intervals closely follow the GPUs ones.

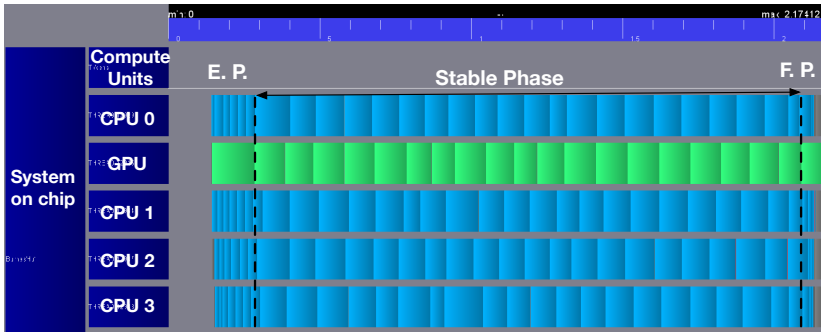


Figure 4.10: Time Profiling: Application BarnesHut with 4 CPU cores (light blue) and 1 GPU (light green).

As we can see in Figure 4.10, LogFit is designed as a three-phase partition strategy. The first phase, called **Exploration Phase (E.P.)** is responsible for finding a chunk size that feeds all the GPU (or CPU) compute units. The second phase is called **Stable Phase (S.P.)**, which monitors device throughput in order to adapt the chunk size according to throughput changes. And the last phase, **Final Phase**, which just distributes the remaining iterations across all CPU cores and GPU at once. Figure 4.11 shows the flow chart of the LogFit partitioning strategy, as we can see, it has two main phases and the Final Phase, which can be executed from Exploration and Stable Phases. LogFit starts its execution in the Exploration Phase, where, it aims to find a GPU chunk size (number of iterations) that fully feeds the GPU Execution Units (EUs.). Once LogFit finds an optimal chunk for the GPU, it moves to the following phase, Stable Phase. In this second phase, LogFit keeps monitoring the throughput of each new interval

and re-fitting the logarithmic curve by updating the chunk size used. It is done to consider potential throughput changes that are intrinsic of irregular codes. Finally, the Final Phase is executed when there are not enough iterations to feed all devices with their respective chunk sizes. Thus a final partition is performed to distribute all remaining iterations among the available processors at once. Note that in both phases the first step is to evaluate whether the **Stop Condition** is satisfied:

$$\frac{Ch(I_{G_i})}{\lambda(I_{G_i})} > \frac{remaining - Ch(I_{G_i})}{n_{cores} \cdot \lambda(I_{C_i})}. \tag{4.1}$$

This condition would be satisfied if the GPU extracts a chunk of iterations from the set of remaining iterations by not leaving enough iterations to feed all CPU cores for a period of time equal to GPU execution time for its chunk. For this reason, it is useful to maintain a work balance at the end of the iteration space. It uses the last GPU chunk size $Ch(I_{G_i})$ and the latest known throughputs $\lambda(I_{G_i})$ (for GPU) and $\lambda(I_{C_i})$ (for CPU), in order to decide if the Final Phase has to be executed. In the following points, we explain in detail the internal decisions of each phase.

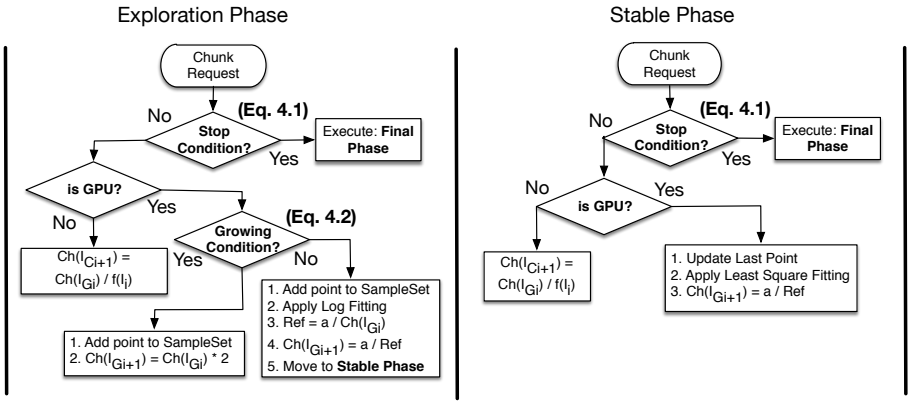


Figure 4.11: Flow chart of the LogFit's partition strategy.

Exploration Phase (E.P.): Initially, we want to determine a GPU chunk size that fully occupies the GPU computational resources, $Ch(I_{G_0})$, for the first GPU's scheduling interval, I_{G_0}. LogFit executes a few chunks with an incremental size by following the next expression: $Ch(I_{G_i}) = nEU \times 2^i, i = 0 : t$,

being nEU[4] the number of Execution Units (EUs) on the target GPU. Basically, LogFit starts from a chunk size equal to the number of computes units and increase the size multiplying it by 2. After executing each one of the chunks on the GPU, we record the execution time, $T(I_{G_i})$ and calculate the effective throughput by using the following expression: $\lambda_G(I_{G_i}) = Ch(I_{G_i})/T(I_{G_i})$. This computed chunk size and its corresponding throughput represent a new sample point that can be used by our fitting procedure to find the GPU chunk size for the next scheduling interval $Ch(I_{G_{i+1}})$. Thus, we obtain a set of sample points, $\{(Ch(I_{G_0}), \lambda_G(I_{G_0})), \ldots, (Ch(I_{G_i}), \lambda_G(I_{G_i}), \ldots, ((Ch(I_{G_n}), \lambda_G(I_{G_n}))\}$ and keep sampling points until the **Growing Condition** is satisfied:

$$\big[\lambda(I_{G_i}) * (1+\theta) > \lambda(I_{G_{i+1}})\big] \wedge \big[\lambda(I_{G_i}) * (1+\theta) > \lambda(I_{G_{i+2}})\big]. \tag{4.2}$$

The condition 4.2 is satisfied when the throughput ($\lambda(I_{G_i})$) of a given chunk ($Ch(I_{G_i})$) is larger than the throughput of the following two sampled points ($\lambda(I_{G_{i+1}})$ and $\lambda(I_{G_{i+2}})$) in more than a certain threshold value (θ), where θ is within the range, $0 \leq \theta < 1$. This condition reveals an early chunk size ($Ch(I_{G_i})$) that fully occupy the computing resources of the GPU. This chunk size value is very important, as we require it to compute the reference [5] (Ref) value, and to determine the optimal chunk sizes of all upcoming scheduling intervals. Once the growing condition is satisfied, we select four (n=4) equidistant points from the set of sample points (from I_{G_1} to I_{G_i}), $\{(Ch(I_{G_{x1}}), \lambda(I_{G_{x1}})), (Ch(I_{G_{x2}}), \lambda(I_{G_{x2}})), \ldots, (Ch(I_{G_{x4}}), \lambda(I_{G_{x4}}))\}$, being $x1 = 1$ and $x4 = i$. Figure 4.12 shows the set of sampled points (black points), and the set of the selected equidistant points (points with a blue circle). Note that the two points which are following the last selected point ($Ch(I_{G_{x4}})$, $\lambda(I_{G_{x4}})$) have a throughput lower than $\lambda(I_{G_{x4}}) \cdot (1+\theta)$ (black horizontal line). Later, we apply the Least Square fitting method to the previously introduced logarithmic function ($f(x) = a \cdot ln(x) + b$, blue line in Figure 4.12) by using the set of equidistant points ($n = 4$), and we obtain the value of the coefficients a and b:

$$a = \frac{n \cdot \sum_{i=1}^{n}(\lambda(I_{G_{xi}}) \cdot ln(Ch(I_{G_{xi}}))) - \sum_{i=1}^{n}\lambda(I_{G_{xi}}) \cdot \sum_{i=1}^{n} ln(Ch(I_{G_{xi}}))}{n \cdot \sum_{i=1}^{n} ln^2(Ch(I_{G_{xi}})) - (\sum_{i=1}^{n} ln(Ch(I_{G_{xi}})))^2}, \tag{4.3}$$

[4] nEU = clGetDeviceInfo(deviceId, CL_DEVICE_MAX_COMPUTE_UNITS)

[5] The reference value allows us to stablish a scale between throughput and chunk-size for each benchmark data-set and GPU.

$$b = \frac{\sum_{i=1}^{n} ln^2(Ch(I_{G_{xi}})) \cdot \sum_{i=1}^{n} \lambda(I_{G_{xi}}) - \sum_{i=1}^{n} (ln(Ch(I_{G_{xi}})) \cdot \lambda(I_{G_{xi}})) \cdot \sum_{i=1}^{n} ln(Ch(I_{G_{xi}}))}{n \cdot \sum_{i=1}^{n} ln^2(Ch(I_{G_{xi}})) - (\sum_{i=1}^{n} ln(Ch(I_{G_{xi}})))^2}. \tag{4.4}$$

Once, we get the first logarithmic curve ($f_1(x)$) that fits the set of equidistant points, we move forward to the Stable Phase. Thus, for each upcoming scheduling interval, we compute a new logarithmic curve fitting (getting $f_i(x)$). However, we also need to compute a reference value (Ref) that relates a near optimum throughput with an appropriate range of chunk sizes values. To that end, we use the slope of the tangent line to $f_1(x)$ in the point $Ch(I_{G_{x4}})$ (see red dotted line in Figure 4.12). This slope reference allows us to determine the near optimal chunk sizes of following $f_i(x)$. Hence, we compute the slope of the tangent line that touches the function ($f_1(x)$) in the point $Ch(I_{G_{x4}})$ by applying the following expression:

$$Ref = f_1'(Ch(I_{G_{x4}})) = \frac{a}{Ch(I_{G_{x4}})}. \tag{4.5}$$

Note that this reference value is only calculated once, as you can see in Figure 4.11. Finally, we calculate the first GPU chunk size with the following expression: $Ch(I_{G_{i+1}}) = a/Ref$ (substituying Ref, we get the first recommended chunk size

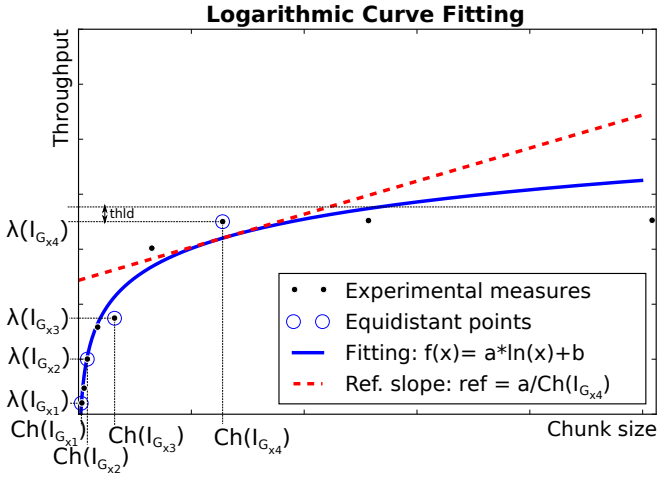


Figure 4.12: LogFit process when performing the first fitting for a benchmark.

$Ch(I_{G_{i+1}}) = Ch(I_{G_{x4}})$ which is the optimal measured value) and we shift to Stable Phase.

Stable Phase (S.P.): This phase is activated once the previous phase finds a chunk that fully occupies the computational resources of the GPU. In this phase, we keep monitoring the throughput of each new scheduling interval and we apply the fitting of the logarithmic curve by adding this last point. Thus, we allow LogFit to adapt the fitting of the logarithmic curve to throughput variations in upcoming scheduling intervals.

To avoid re-taking four samples in the next scheduling interval, we use the previous set of four equidistant points and just update the fourth point by using the new one. Thus, in general, for a given time interval $I_{G_{i+1}}$ for which we want to compute its recommended chunk size $Ch(I_{G_{i+1}})$, we use the set of equidistant points given, where we continuously update the last point ($p(I_{G_{x4}}) = p(I_{G_i})$): $\{(Ch(I_{G_{x1}}), \lambda(I_{G_{x1}})), (Ch(I_{G_{x2}}), \lambda(I_{G_{x2}})), (Ch(I_{G_{x3}}), \lambda(I_{G_{x3}})), (Ch(I_{G_i}), \lambda(I_{G_i}))\}$. We have analytically and experimentally validated that re-sampling 4 points for each time interval by using small chunk sizes is not worthwhile: the penalty incurred by processing small chunk sizes at sub-optimal throughput is not compensated by the throughput improvement obtained due to a more accurate measurements in the scheduling interval. In any case, $(Ch(I_{G_i}), \lambda(I_{G_i}))$ represents an accurate measurement for the next interval $(I_{G_{i+1}})$.

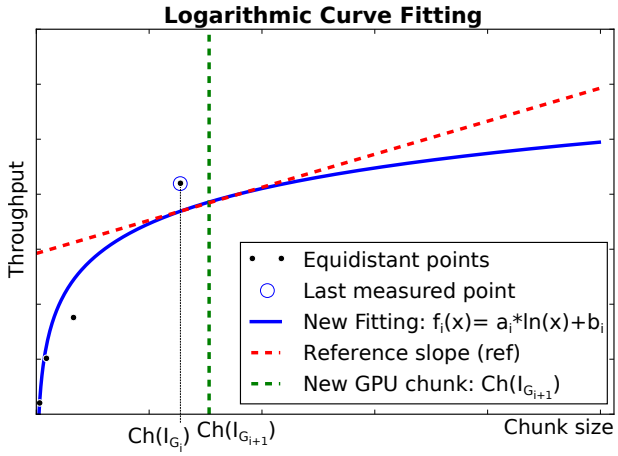


Figure 4.13: LogFit stable phase fitting and adapting the GPU chunk size.

Figure 4.13 shows the set of equidistant points, black points, plus the last updated point $(Ch(I_{G_i}), \lambda(I_{G_i}))$, rounded with a blue circle. Based on these four points, we perform a new logarithmic curve fitting ($f_{i+1}(x)$, blue solid line) in order to compute the new recommended chunk size for the GPU, $Ch(I_{G_{i+1}})$. To obtain the new chunk size, we compute the point where the Ref slope (red dotted line in Figure 4.13) of the first fitting touches the fitted curve (see expression 4.6), and finally we isolate the chunk size variable (expression 4.7). The resulting chunk size $Ch(I_{G_{i+1}})$ is represented by a green dotted line.

$$f_i'(Ch(I_{G_{i+1}})) = \frac{a_i}{Ch(I_{G_{i+1}})} = Ref \tag{4.6}$$

$$Ch(I_{G_{i+1}}) = \frac{a_i}{Ref} \tag{4.7}$$

Figure 4.14 shows how the logarithmic fitting mechanism is used to adaptively change the GPU block size for the next interval $(I_{G_{i+1}})$ based on the last GPU throughput measured in the current interval (I_{G_i}). Note that the solid blue line represents the logarithmic curve that fits the sample set and the blue square represents the optimal block size $Ch(I_{G_i})$ proposed by the fitting method. After executing this chunk, it may happen that the measured throughput $(\lambda(I_{G_i}))$ is actually higher than the value predicted by the last fitting. This is shown in Figure 4.14 with a red circle. By applying the logarithmic fitting to the new

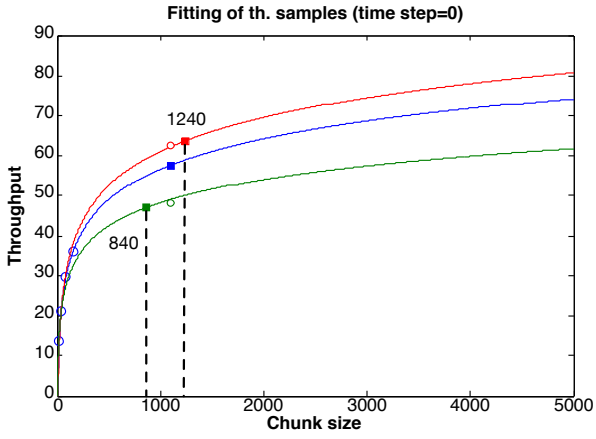


Figure 4.14: LogFit's chunk size variations depending on throughput changes.

set, for the next interval ($I_{G_{i+1}}$), we would get a higher logarithmic curve, as it is shown in Figure 4.14 with the dashed red line. Moreover, the red square represents the new optimal chunk for the next interval ($Ch(I_{G_{i+1}})$), thus by updating the last point and re-fitting the samples set, we get the next chunk size: $Ch(I_{G_{i+1}}) = 1240$. On the contrary, it may happen that the measured throughput $\lambda(I_{G_i})$ is actually lower than the value predicted by the last fitting. This is shown in Figure 4.14 with a green circle. In this case, by applying the logarithmic fitting, we would get a shorter logarithmic curve, as shown in Figure 4.14 with a dashed green line, where the green square represents the new proposed point ($Ch(I_{G_{i+1}}) = 840$), after the update and re-fit of the logarithmic curve.

In this way, LogFit adapts the size of new chunks depending on the throughput variations of previous points. On one hand, regular applications exhibit a constant throughput across the whole iteration space, and accordingly the chunk size is hold. But, on the other hand, irregular applications may exhibit big throughput variations depending on the current scheduling interval. Thus, LogFit will recommend larger/smaller chunk sizes depending on throughput's rises/drops.

Final Phase (F.P.): As mentioned before, the execution of the Final Phase is controlled by the **Stop Condition** shown in the equation 4.1. This condition is satisfied when the number of remaining iterations is smaller than the recommended GPU chunk size plus the CPU chunk size multiplied by the number of CPU cores, it is: $(remaining < Ch(I_{G_{i+1}}) + Ch(I_{C_i}) \cdot n_{cores})$. At this point, we need to find out what is the best possible distribution for the remaining iterations across CPU cores and GPU. Our target is to execute the remaining iterations in the shorter possible time (see expression 4.8). Thus, we devise three possible scenarios: i) **CPU case** when all iterations are executed on the CPU cores (T_{CPU}), ii) **GPU case** when all iterations are executed on the GPU (T_{GPU}) and iii) **HET case** when the set of remaining iterations is split and distributed across CPUs and GPU. Next, We describe these three scenarios in detail:

$$T_{min} = min(T_{CPU}, T_{GPU}, T_{HET}) \tag{4.8}$$

- In the **CPU case**, the entire set of remaining iterations is executed on the CPU cores. In this case, the execution time (T_{CPU}) is computed as the ratio between the number of remaining iterations and the equivalent throughput of all CPU cores (see expression 4.9).

$$T_{CPU} = \frac{remaining}{n_{cores} \cdot \lambda(I_{C_i})} \tag{4.9}$$

Whether this case happens to be the shorter in execution time, it would stop the GPU from executing further chunks and would assign all remaining iterations to the CPU cores, resulting in the following chunk sizes:

$Ch(I_{G_{i+1}}) = 0$ and
$Ch(I_{C_{i+1}}) = \frac{remaining}{n_{cores}}$.

- In the **GPU case**, the whole set of remaining iterations is assigned to the GPU. The GPU execution time is computed in the same way as CPU execution time, by dividing the number of remaining iterations by the estimated GPU throughput. However, as explained in section 4.3.1, the GPU throughput depends on the size of range of iterations to be executed. Thus, we define a function ($\lambda'_i()$) that accurately approximates the GPU logarithmic behaviour by using the previously introduced set of four equidistant points. To simplify calculations, we assume that the GPU exhibit a linear behaviour between each pair of equidistant points, as we can observe in Figure 4.15. In this way, the resulting GPU throughput for a chunk size (c) of iterations ($\lambda'_i(c)$) between two concrete points $\{(Ch(I_{G_{xi}}), \lambda(I_{G_{xi}}),(Ch(I_{G_{xi+1}}), \lambda(I_{G_{xi+1}})\}$ is given by the equation of the line that pass through those points (see equation 4.10).

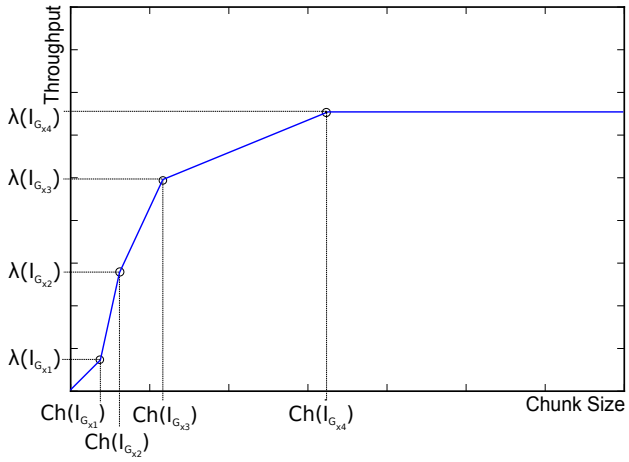


Figure 4.15: Modelling GPU throughput by assuming a linear behaviour between each pair of equidistant points.

$$\lambda'_i(c) = C1_i \cdot c + C2_i \tag{4.10}$$

$$C1_i = \frac{\lambda(I_{G_{xi+1}}) - \lambda(I_{G_{xi}})}{Ch(I_{G_{xi+1}}) - Ch(I_{G_{xi}})} \tag{4.11}$$

$$C2_i = \lambda(I_{G_{xi}}) - C1_i \cdot Ch(I_{G_{xi}}) \tag{4.12}$$

Based on the throughput equation 4.10 between two given equidistant points, we define the general throughput equation for the GPU for any given chunk size ($\lambda_G(c)$), as the equation 4.13.

$$\lambda_G(c) = \begin{cases} \lambda(I_{G_{x4}}) & if \quad c \geq Ch(I_{G_{x4}}) \\ \lambda'_i(c) & if \quad Ch(I_{G_{xi}}) \leq c < Ch(I_{G_{xi+1}}) \wedge (1 \leq i \leq 3) \\ c \cdot \frac{\lambda(I_{G_{x1}})}{Ch(I_{G_{x1}})} & if \quad c < Ch(I_{G_{x1}}) \end{cases} \tag{4.13}$$

Finally, we can determine the GPU execution time that the GPU requires to execute a chunk with a number of *remaining* iterations. Thus, we define the GPU execution time (T_{GPU}) in the same manner that (T_{CPU}), by dividing the chunk of remaining iterations by the general function of the GPU throughput ($\lambda_G(remaining)$), as we can see in the expression 4.14.

$$T_{GPU} = \frac{remaining}{\lambda_G(remaining)} \tag{4.14}$$

Whether the GPU execution time (T_{GPU}) is the overall smallest execution time, all remaining iterations would be executed on the GPU, while the CPU cores are stopped, and the resulting chunk sizes for each device would be the following:

$Ch(I_{G_{i+1}}) = remaining$ and
$Ch(I_{C_{i+1}}) = 0$

- In the **HET case**, we consider that the number of remaining iterations are computed on the GPU and the CPU cores. In order to find an optimal distribution, we need to optimally distribute the remaining iterations between the CPU cores and the GPU. Our target is to make them finishing at the same time, as indicated by the following expression:

$$T_{GPU} = T_{CPU}. \tag{4.15}$$

We can represent the GPU and CPU times by dividing their chunk sizes by their respective throughputs. In this sense, we want to calculate the number of iterations $Ch(I_{G_i})$ that should be assigned to the GPU and to the CPU $(remaining - Ch(I_{G_i}))$. The throughput of the CPU is given by the value of the last scheduling interval, $\lambda(I_{C_i})$. However, the GPU throughput, as shown in equation 4.13, is a piecewise function. Thus, we have to compute the expression 4.15 for each GPU throughput segment, and select the minimum real solution among all of them. For a given segment i of the GPU throughput function, we have to find a feasible distribution of the remaining iterations for GPU (G_i) and the CPU cores $(remaining - G_i)$, which makes both devices finish at the same time, so the GPU's execution time is equal to the CPU cores' execution time, as specified in expression 4.16,

$$\frac{G_i}{\lambda'_i(G_i)} = \frac{remaining - G_i}{\lambda(I_{C_i}) \cdot n_{cores}}. \tag{4.16}$$

Substituting the term $\lambda'_i(G_i)$ in the expression 4.16 by the equality 4.10 gives the following:

$$C1_i \cdot {G_i}^2 + (\lambda(I_{C_i}) \cdot n_{cores} - C1_i \cdot remaining + C2_i) \cdot G_i - C2_i \cdot remaining = 0. \tag{4.17}$$

From the equation 4.17, we can obtain the expression that provides us the GPU chunk size (G_i) for the segment i, this time is equal to the CPU cores' execution time while computing $remaining - G_i$ iterations (see equation 4.17).

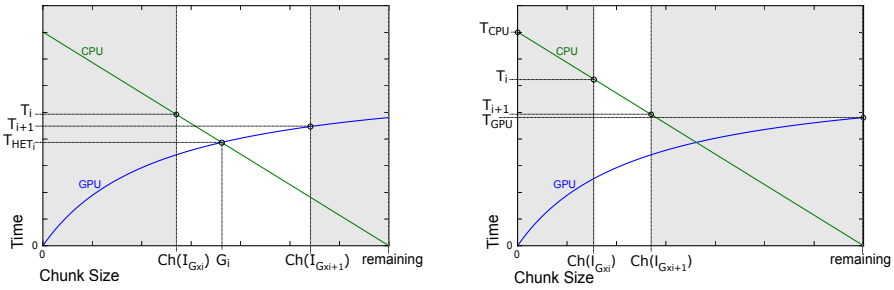
$$G_i = \frac{-B \pm \sqrt{B^2 - 4 \cdot A \cdot C}}{2 \cdot A} \tag{4.18}$$

$$A = C1_i \tag{4.19}$$

$$B = \lambda(I_{C_i}) \cdot n_{cores} - C1_i \cdot remaining + C2_i \tag{4.20}$$

$$C = -C2_i \cdot remaining \tag{4.21}$$

The solution of the equation 4.18 is the intersection point at where both curves (GPU and CPU) intersect. The blue line curve (Figure 4.16) represents the GPU execution time and the green line represents the CPU execution time for a given interval i, $(\{(Ch(I_{G_{xi}}), \lambda(I_{G_{xi}})), (Ch(I_{G_{xi+1}}), \lambda(I_{G_{xi+1}}))\})$.



(a) (b)

Figure 4.16: Possible scenarios while distributing iterations across the GPU and CPU cores for a given interval. Note that the CPU time, green line, goes from a chunk size equal to *remaining* until zero (rightmost value) in descending chunk size order.

The Figure 4.16a shows a scenario with a positive real solution to the equation 4.18 which is inside the limits ($[Ch(I_{G_{xi}}), Ch(I_{G_{xi+1}})]$) of the interval i. Thus, the following condition is satisfied:

$Ch(I_{G_{xi}}) \leq G_i \leq Ch(I_{G_{xi+1}})$.

For this scenario, we find a feasible iteration distribution across the GPU and CPU cores that lets both devices to finish at the same time (there is not load unbalance). Hence, we define the execution time (T_{HET_i}) as follows:

$$T_{HET_i} = \frac{G_i}{\lambda'_i(G_i)} \tag{4.22}$$

whether it happens that the execution time of this heterogeneous distribution is the shortest among all the possible ones, the chunk sizes for each device would be the following:

$Ch(I_{G_{i+1}}) = G_i$ and
$Ch(I_{C_{i+1}}) = remaining - G_i$

On the contrary, it may happen that the equation 4.18 has no real solution between the limits of the segment j, as the figure 4.16b shows. In this scenario, the execution time associated with this segment (T_{HET_j}) is given by the extreme points of such segment. We have to observe which device, GPU or CPU cores, requires more time to process its assigned chunk in

both extremes of the segment in Figure 4.16b. Thus, we get the execution times for T_j, equation 4.23, and T_{j+1}, equation 4.24. Later, we select the minimum between this two values, T_{HET_j} in equation 4.25.

$$T_j = max\left(\frac{Ch(I_{G_{xj}})}{\lambda(I_{G_{xj}})}, \frac{r - Ch(I_{G_{xj}})}{ncores \cdot \lambda(I_{C_j})}\right), \tag{4.23}$$

$$T_{j+1} = max\left(\frac{Ch(I_{G_{xj+1}})}{\lambda(I_{G_{xj+1}})}, \frac{r - Ch(I_{G_{xj+1}})}{ncores \cdot \lambda(I_{C_i})}\right), \tag{4.24}$$

$$T_{HET_j} = min(T_j, T_{j+1}). \tag{4.25}$$

Once we have calculated the T_{HET_j} value for all segments, we select the minimum value for all segments:

$$T_{HET} = min(T_{HET_j}) \quad j = 0 : segments. \tag{4.26}$$

Again, if it happens that the execution time (expression 4.26) is the shortest one among all possible, then the chunk of iteration assigned to the GPU and the CPU cores would be the following:
$Ch(I_{G_{i+1}}) = Ch(I_{G_{xj}})$ and
$Ch(I_{C_{i+1}}) = remaining - Ch(I_{G_{xj}})$.

Finally, once we have calculated T_{CPU} (equation 4.9), T_{GPU} (equation 4.14) and the respective T_{HET} (equations 4.26) for all segments, we substitute these terms in the equation 4.8 and get the overall smaller execution time and its respective chunk sizes for GPU $Ch(I_{G_{i+1}})$ and CPU $Ch(I_{C_{i+1}})$. Resulting in the best possible iteration distribution across the GPU and CPU cores that minimises the execution time.

4.4. Experimental Results

In this section we first describe the architecture on which we conduct the experiments with the selected benchmarks. Next, we present a sensitivity study of the main parameters that drive our partitioner. Later, we analyse how our partitioning strategy follows the throughput of the computational devices and adapts the size of the chunks accordingly. Then we identify the sources of overhead due to our partitioning strategy and propose some optimizations to minimise them. Finally, we compare our proposal with other related state-of-the-art dynamic alternatives to assess the performance and energy efficiency of our approach.

4.4.1. Experimental setup

We run our experiments on two Intel Quad-Core processors: a Core i5-3450 running at 3,1GHz, 77W TDP, based on the Ivy Bridge micro-architecture with an integrated GPU, the Intel HD 2500; and a Core i7-4770, 3.4GHz, 84W TDP based on the Haswell micro-architecture. This processor features Advance Vector Extensions (AVX) and have an on-chip GPU, HD Graphics 4600. A more detailed description of both processors is given in Table 4.1. We rely on the Intel Performance Counter Monitor (PCM) library [28] to access the hardware counters (which also provide energy consumption in Joules). Intel Threading Building Blocks (TBB 4.2) provides the core task engine of the heterogeneous `parallel_for`. The GPU kernels are implemented in OpenCL C language and compiled by using the Intel OpenCL SDK 2014. The host code part of the benchmarks is compiled with Intel C++ Compiler 15.0 and -O3 optimization flag. Table 4.2 shows the numbering version of all software component used on these test-bed machines. We measured time and energy in 10 runs of the applications and report the average.

Table 4.1: Processors details (Ivy Bridge & Haswell) to execute LogFit strategy.

Microarchitecture	Ivy Bridge	Haswell
Processor Number	Core i5-3450	Core i7-4770
Number of cores/threads	4/4	4/8
Clock Speed	3.1 GHz	3.4 GHz
Max Turbo Frequency	3.5 GHz	3.9 GHz
Base CPU peak	99.2 GFLOPs	108.8 GFLOPs
Max CPU peak	112 GFLOPs	124.8 GFLOPs
Cache	6 MB	8 MB
Lithography	22 nm	22nm
Max TDP	77 W	84 W
Intel HD Graphics	2500	4600
Number of Compute Units	6	20
GPU Base Frequency	650 MHz	350 GHz
GPU Max Dynamic Frequency	1.1 GHz	1.2 GHz
Base GPU peak	31.2 GFLOPs	56 GFLOPs
Max GPU peak	112 GFLOPs	192 GFLOPs

4.4.2. Benchmarks

We use five benchmarks whose details can be seen in Table 4.3. It shows the data input, number of invocations, the iteration space size and the execution time per iteration for each benchmark on a Haswell CPU core. These applications come

Table 4.2: Software details to evaluate LogFit partitioner strategy.

Operating System	Windows 7 Build 7601
GPU driver	Intel 9.18.10.3071
Intel OpenCL SDK	3.0.0.64050
Intel TBB	4.2
OpenCV	2.4.5
Intel Performance Counter Monitor	2.5
Intel C++ Compiler	15.0

from several domains and exhibit different behaviour: regular vs. irregular, coarse grained vs. fine grained, single kernel call vs. multiple kernel call. Nbody [54] and Barnes Hut [63] are presented in Section 4.2. However, the table 4.3 introduces three new benchmarks: PEPC [41] which is like a Barnes Hut problem. However, it computes electrical forces instead of gravitational ones. Moreover, there is an important difference with respect to the Barnes Hut implementation. Barnes Hut sorts the particles to better exploit spatial locality, but PEPC does not. The second new benchmark is CFD from the Rodinia Benchmark suite [18], which performs a Fluid Dynamics simulation, and SpMV from the SHOC Benchmark suite [25], which performs a sparse matrix-vector multiplication. For the Nbody benchmark an input set of 100,000 bodies and 1 time-step is simulated. For Barnes Hut an input set of 100,000 bodies and 75 time-steps are computed. PEPC is fed with 100,000 point charges and 75 time-steps. CFD uses the missile.0.2M input data (which simulates 6,000 time-steps) and SpMV processes 200 times the GL7d16 sparse matrix from the University of Florida Sparse Matrix Collection [27] that exhibits a triangular-like profile. Considering the irregular benchmarks, Barnes Hut and PEPC are examples of coarse grained applications (each iteration executes in approximately 32 and 69 microseconds, respectively, on one Haswell core) while CFD and SpMV are fine grained ones (0.13 and 0.1 microseconds per iteration, respectively, on one Haswell CPU core).

Figure 4.17 shows the GPU's throughput throughout the iteration space of the first time step for the three new irregular applications: PEPC, CFD and SPMV. We can observe big differences between the three benchmarks plots. We are profiling one benchmark that presents coarse grain parallelism (PEPC) and two benchmarks with fine grain parallelism, that is the reason of the differences in throughput and chunks size. We can observe how PEPC and SPMV are more stable than CFD, as they make the most with a chunk equal to 1024 and 32768, respectively. While CFD exhibits a higher level of irregularities, as the most performing chunk size varies between 32768 and 131072, depending on the application regime.

Benchmark Name	Benchmark Suite	Description	Input | Invocations	Iteration Space	Total CPU Time (ms)	Time per iteration
NBody	Intel OpenCL [54]	Particle Simulation	100.000 bodies | 1	100.000	102864	1 ms
Barnes Hut	Lonestar [63]	Particle Simulation	runC | 75	100.000	239832	31,27 μs
PEPC	PEPC [41]	Coulombs Simulation	100.000 Q_i | 50	100.000	345696	69,13 μs
CFD	Rodinia [18]	Fluid Dynamics	missile.0.2M | 6000	232536	182519	0,13 μs
SPMV	SHOC [25]	Linear Algebra	GL7d16.mtx [27] | 200	955128	19407	0,1 μs

Table 4.3: Description of the benchmarks used to evaluate LogFit partitioner. The detailed times are the result of executing each benchmark with the described input on a Haswell CPU core.

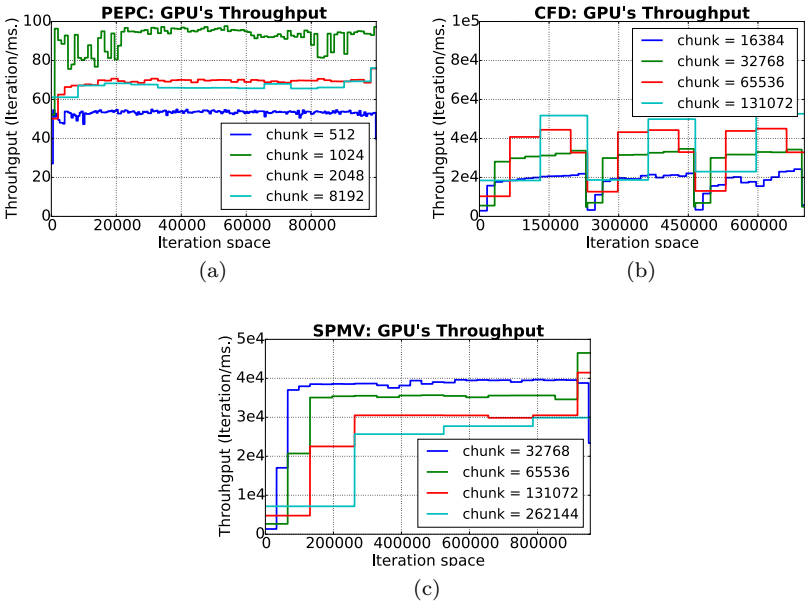


Figure 4.17: Throughput evolution while executing irregular benchmarks on the Intel HD Graphics 4600, just the first time-step. The legends show several iteration block sizes used to split the iteration space of each benchmark.

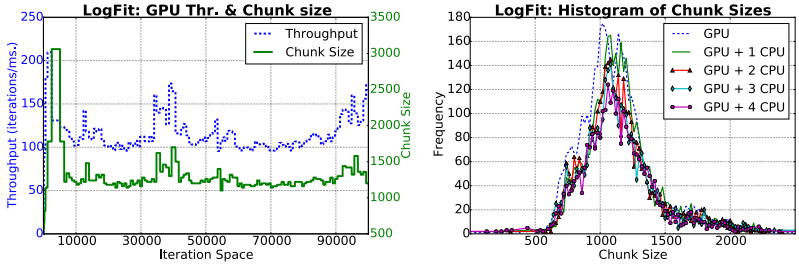
4.4.3. Characterisation of the partitioning strategy

In this section, we elaborate on the efficiency of our partitioner `LogFit`. First, we perform an analysis of its performance while increasing the number of threads in the system to asses its robustness. Later, we carry out a sensitivity analysis of `LogFit's` internal parameters and finally we analyse its sources of overhead and provide optimizations to mitigate them.

Analysis of GPU chunk size variations

In this section, we elucidate the GPU chunk size adaptation according to the changes of GPU's throughput at runtime. To graphically assess how well LogFit adapts the GPU chunk size to each throughput regime, Figure 4.18(a) shows the evolution of the throughput (blue) and the GPU chunk size (green) throughout the iteration space for the first time-step of Barnes Hut while running on the

Haswell GPU. In this Figure, the throughput is indicated in the left y-axis and the chunk size in the right one. It is noticeable that the chunk size curve closely follow the throughput with a small delay, since we use the previous throughput of the previous GPU's scheduling interval ($\lambda(I_{G_i})$) to compute the next GPU chunk size ($Ch(I_{G_{i+1}})$).



(a) GPU Throughput and Chunk size evolution

(b) Histogram of GPU chunks

Figure 4.18: The left hand side figure shows the evolution of the GPU throughput and the chunk size obtained by LogFit for a complete execution of Barnes Hut on Haswell GPU, we can observe how the selected chunk sizes follow the throughput curve . The right hand side figure shows a chunk size histogram when executing BarnesHut on Haswell with different configurations (see legend). Note that the most frequent chunk is the same for all configurations.

Figure 4.18(b) shows an analysis of the GPU chunk size adaptation throughout all time-steps for Barnes Hut. It shows the chunk size histogram that LogFit selects during several executions where the number of threads varies between 1 (only GPU execution) and 5 (four CPU cores plus the GPU host thread). Notice that for all executions the most frequent chunk sizes are within the range 1000 and 1250. We also see that the frequency of chunk sizes decreases as the number of threads increases, because, as more chunks are assigned to the CPU cores less chunks are computed on GPU. In any case, the increment in the number of threads does not affect the selection of the chunk sizes for the GPU, which demonstrates that our partitioning method is not sensitive to the number of CPU threads, and it works properly for the whole range of threads.

Sensitivity analysis

In this section, we carry out a sensitivity study of LogFit's input parameters to highlight what is the impact of these parameters over LogFit's behaviour. We run a set of experiments on the Intel HD Graphics 4600.

For this set of experiments, we vary the number of points that LogFit needs to perform the logarithmic fitting, so, we analyse the effects that the number of equidistant points ($(Ch(I_{G_{xi}}), \lambda(I_{G_{xi}}))$) has on performance, while using 4, 8 and 12 points. Moreover, we analyse the behaviour of LogFit by using two values for the threshold (θ in equation 4.2) parameter: 0.01 and 0.05. Table 4.4 shows the most frequent *chunk size* (mode) and the respective *execution time* for each benchmark for all possible combinations of the aforementioned variables. In general, we can observe that by forcing LogFit to use more than 4 points, it gets bigger execution times for Barnes Hut, PEPC and SPMV, however the time seems to keep constant in the case of CFD. Thus, it is recommendable that LogFit can decide when it must stop sampling new points, instead of forcing it to record a large number of points. In the case of CFD the number of sampled points do not seem to affect performance because this benchmark keeps increasing the chunk size in *Exploration Phase* until the whole iteration space is processed.

		Samples		
Benchmark	Threshold	4	8	12
Barnes	0,01	**1240 | 872**	3846 | 1059	7692 | 1154
	0,05	1140 | 895	3846 | 1047	7692 | 1164
PEPC	0,01	**735 | 1208**	4000 | 1432	7692 | 1567
	0,05	1149 | 1247	3846 | 1457	7692 | 1565
CFD	0,01	**33219 | 35**	33219 | 35	33219 | 35
	0,05	33219 | 36	33219 | 36	33219 | 36
SPMV	0,01	**36651 | 8791**	37804 | 8838	70671 | 9604
	0,05	38771 | 9027	38755 | 8874	70802 | 9411

Table 4.4: Sensitivity Analysis of the parameters *threshold* and *Number of Samples* for LogFit. Each cell represents the most frequent *chunk size* and *execution time* (in milliseconds).

Note, we are only setting the number of points that are used to perform the logarithmic fitting, however the number of sampled point does not have any limitation. Looking at table 4.4, we highlight in bold the best configuration for each benchmark in term of performance (minimum execution time), which is four points and a threshold value of 0.01 for all tested benchmarks. We also can

observe that by requiring LogFit to use more points it increases the recommended chunk size, and consequently the execution time (second value of each cell), as it can not adapt that easily to changes in throughput. For example, using more than 4 points in Barnes Hut, PEPC or SPMV makes LogFit more rigid and it may not find a near optimal chunk size, thus it may get a poorer performance due to an issue in the uncoalesced memory access pattern and control divergences inside workgroups which leads to serialization and ends up requiring more time. In case of CFD, all tested number of samples get the same results, because CFD is a corner case benchmark that directly shifts from Exploration Phase to Final Phase, as the chunk size keeps growing in Exploration Phase and never gets stabilised. Thus, the GPU chunk size is larger than 1/2 of the iteration space and the `Stop Condition` is satisfied in the fist GPU chunk request of each upcoming time-step.

Sources of overhead in dynamic partitioning

As we have explained before, our framework can be initialized with n threads, from which one of them is called the *host thread* and just offloads work to the GPU. This host thread runs in one of the available CPU cores, it first executes the code associated with the scheduler and then, it calls the functions to feed the GPU (`hostToDevice()`, `launchKernel()`, `deviceToHost()`, and `clFinish()`). The first three calls asynchronously enqueue the memory transfers and the kernel launch on the GPU's command queue, whereas the latter is a blocking synchronous wait. Figure 4.19 shows all the operations that take place each time a chunk of iterations is offloaded to the GPU. We can see that the enqueued operations are sequentially executed on the GPU where we consider the times taken by the "Host-to-Device" memory transfer, "Kernel launch", "Kernel execution" and "Device-to-Host" memory transfer. When this last operation is done, the host thread is notified but some time may be taken by the OS to reschedule the host thread. This time is illustrated in Figure 4.19 with the label "Thread dispatch". Thus, we get a picture of the different phases that have to be performed to offload a chunk of iterations to the GPU.

In order to measure the relevant overheads involved in the execution on GPU's, we take some time-stamps on the CPU (Tc1, Tc2 and Tc3) and on the GPU (Tg1 to Tg5) as depicted in Figure 4.19. To get the CPU time stamps we rely on Intel TBB's *tick_count* class, whereas for the GPU we set up the OpenCL command queue in profiling mode, thus we can read the "start" and "complete" time-stamps of each event of the enqueued commands.

We use the previous times to compute the overhead of Scheduling and Parti-

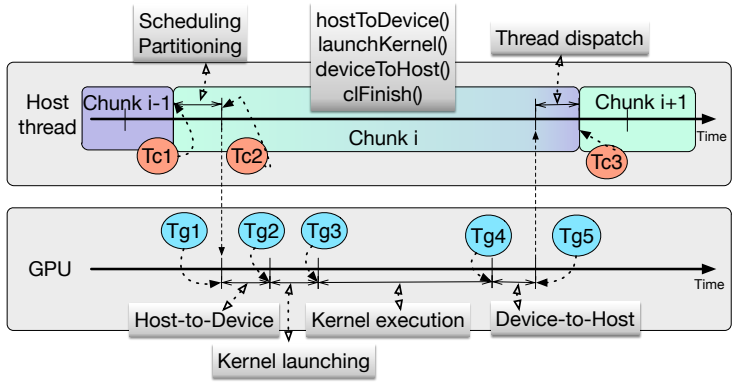


Figure 4.19: Time diagram showing the events across the process of offloading a task to the GPU

tion, O_{sp}, Host-to-Device operation, O_{hd}, KernelLaunching, O_{kl}, Device-to-Host, O_{dh}, and Thread Dispatch, O_{td}, as follows:

$$O_{sp} = \frac{\sum_{\#GPUchunks}(Tc2 - Tc1)}{TotalExecutionTime} \tag{4.27}$$

$$O_{hd} = \frac{\sum_{\#GPUchunks}(Tg2 - Tg1)}{TotalExecutionTime} \tag{4.28}$$

$$O_{kl} = \frac{\sum_{\#GPUchunks}(Tg3 - Tg2)}{TotalExecutionTime} \tag{4.29}$$

$$O_{dh} = \frac{\sum_{\#GPUchunks}(Tg5 - Tg4)}{TotalExecutionTime} \tag{4.30}$$

$$O_{td} = \frac{\sum_{\#GPUchunks}\Big((Tc3 - Tc2) - (Tg5 - Tg1)\Big)}{TotalExecutionTime} \tag{4.31}$$

After identifying the main sources of overhead of our dynamic approach, we discuss the optimizations that can be implemented to tackle them. The overarching goal is not only reduce the impact of the overhead, but also to reduce the energy consumption. In order to study the effects of those overheads, we run two set of experiments in two different scenarios. We consider one scenario without over subscription (4 threads) and one scenario where we allow 1 oversubscripted thread (5 threads). Note that the Intel Core i7-4770 has four CPU cores. With this comparison we aim to study the side effects of having an extra thread working

on heterogeneous contexts. We introduce the following optimizations to reduce the sources of overhead:

- The first optimization technique is the zero-copy-buffer (ZCB) capability of heterogeneous chips. It allows to avoid data movements between CPU and GPU. This optimization has more impact in applications with high data communication times, as it avoid memory transfers by doing a memory address pointer translation.
- The second optimization rises the priority of the GPU host thread (called PRIO), so the GPU host thread has higher priority than any other thread and can be immediately pre-empted. Thus, the host thread can take up a core and start feeding the GPU again. This is key when the GPU processes chunks more efficiently than CPU cores, as it happens in our benchmarks. To boost the host thread priority we rely on the function *SetThreadPriority()* from the Windows API.
- The third optimization is a combination of the previous ones, and we call it, ZCB+PRIO.

Next, we present a set of experiments where we analyse the sources of overhead for all benchmark in two different scenarios, one with no-oversubscription and another one with allowed oversubscription. The first scenario is run with 4 threads (3+1), it means 3 threads for CPU and 1 for GPU, note that we have 4 CPU cores for 4 logical threads. The left hand side of the Figure 4.20 shows an analysis of overheads for all benchmarks while applying the aforementioned optimizations in a scenario without oversubscription. Both graphs represents the ratio of overheads over the total execution time, it shows groups of four bars: the base version (Base, //), a version with zero-copy-buffer (ZCB, empty patch), a version with priority in the host thread (PRIO, \\), and a version combining both optimizations (PRIO+ZCB, :). We can observe that coarse grain benchmarks (Barnes, PEPC and NBody) exhibit an overhead under 7% for the base version (//), first bar. However, fine grain benchmarks are under 4%. This is due to the driver implementation, as it varies its behaviour depending on the kernel duration. When the function `clFinish()` is invoked, the GPU host thread performs a polling technique to check for the completion of the GPU for a period of time. Thus, fine grain benchmarks do not suffer from *thread dispatch overhead*. After a period of time, the GPU host thread starts blocking to reduce the usage of the CPU core that is polling the GPU status. In this manner, if the GPU finishes its task and notifies the GPU host thread while it is blocked, it will have to wait until the O.S. scheduler gives a CPU quantum to it. Thus,

coarse grained applications may suffer from higher *thread dispatch overhead*. On one hand, when we apply the ZCB optimization, we observe how the overheads related to memory transfers *Ohd* and *Odh* are reduced in all cases. On the other hand, when we apply the PRIO optimization we do not get any performance improvement because the GPU host thread has an available CPU core during the whole execution time. However, these applications show a different behaviour while executing on a oversubscripted scenario. The second scenario is run with 5 threads (4+1, it means 4 threads for CPU and 1 for GPU, note there is one oversubscripted thread, as we have 4 CPU cores without hyper-threading). The right hand side of the Figure 4.20 shows an important overhead in the GPU host threads while executing coarse grain applications in this oversubscripted scenario. Here, the base version of coarse grain benchmarks suffer from a high thread dispatch overhead, 36% and 32% for Barnes Hut and PEPC, respectively. In this case PRIO and ZCB+PRIO optimizations reduce the sum of all overheads to a 5% of total execution time. In the presence of oversubscription, the GPU host thread is eventually in ready queue of the S.O. and raising the priority of this thread makes an automatic preemption which reduces the overhead from 35% to a 4% in the case of coarse grain benchmarks.

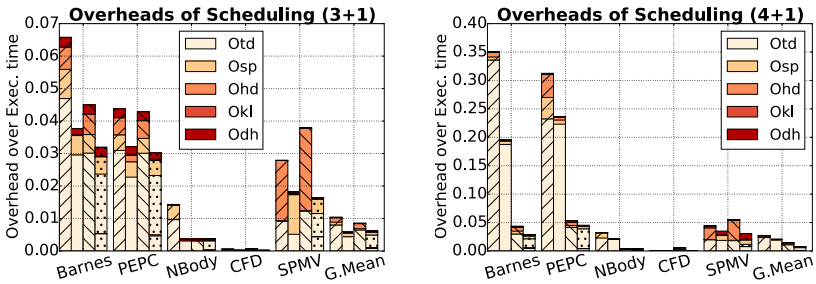


Figure 4.20: Overhead graph results for Haswell in two scenarios: with no oversubscription (3+1) and with oversubscription (4+1). Left to right: Base (//), ZCB ( ), PRIO (\\) and ZCB+PRIO (:).

In addition to the overhead analysis, we also study the impact of the aforementioned optimizations on performance and energy consumption. Hence, Figures 4.21 and 4.22 shows these effects in the two previous scenarios, without oversubscription (3+1) and with oversubscription (4+1), respectively. The left-hand side bar graph of the Figure 4.21 shows the speedup over the base version of each benchmark. We compare the impact of the previously described optimizations, so we can observe that raising the GPU host thread has no effect

on performance for any benchmark in this scenario (3+1). In contrast, applying ZCB or ZCB+PRIO yields to an improvement of 1.5x and 1.6x for CFD and SPMV respectively. Note that coarse grain benchmarks (Barnes, PEPC, NBody) do not benefit from these optimizations, as the computation time is huge compared to the time spent in memory transfers. The right-hand side bar graph of Figure 4.21 shows the energy reduction ratio over the base version. We can observe that the energy consumption follows the speedup behaviour. The energy consumption reduction rises to 32% for CFD and 38% for SPMV when applying ZCB and ZCB+PRIO optimizations. Note that, fine grain benchmarks (CFD and SPMV) get better improvement ratio (speedup and energy reduction) than coarse grain ones (Barnes Hut and PEPC) while applying ZCB because fine grain benchmarks spend most of the time in memory transfers. In Figure 4.22, the left-hand side bar graph shows the speedup over the base version. In this scenario coarse grain benchmarks do not show an important speedup (it is under 1.2x). However, applying ZCB+PRIO to fine grain benchmarks, CFD and SPMV yields a speedup of 2.1x and 1.6x respectively. The right-hand side bar graph shows the energy consumption reduction, and again, the energy reduction strongly follows the speedup shape, this is a common behaviour for both scenarios (3+1 and 4+1). In this case, coarse grain benchmarks show a small energy reduction factor around 10%, while fine grain ones shows a 52% and 40% energy reduction factors for CFD and SPMV, respectively.

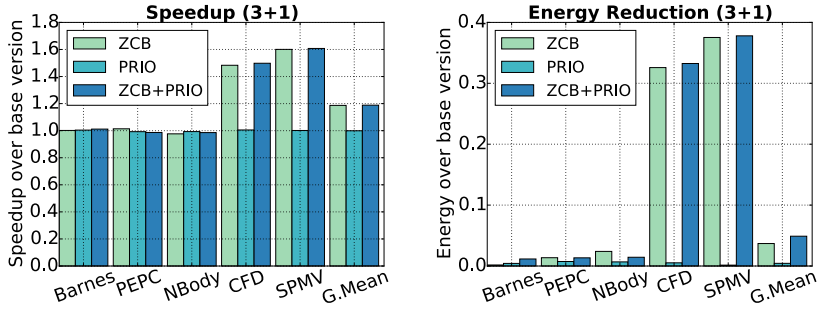


Figure 4.21: Impact of ZCB and PRIO optimizations on performance and energy in a scenario without oversubscription.

Summarizing, we find out that applying the aforementioned optimizations, we can avoid the overheads of continuously partitioning and offloading to GPU the chunks of a parallel loop and make the most of the GPU by selecting the right chunk size depending on the application regime.

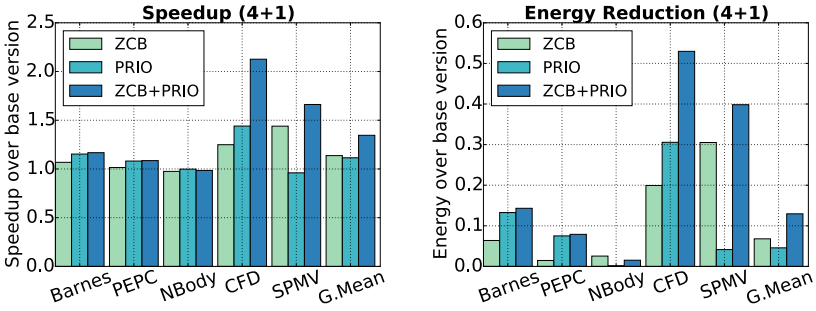


Figure 4.22: Scenario Impact of ZCB and PRIO optimizations on performance and energy in a scenario with oversubscription.

4.4.4. Performance and energy comparison

To validate our partition strategy, LogFit (Logarithmic Fitting), we compare it with other three related partition strategies: Static, Concord [58] and HDSS [4]. As a baseline, we use a Static partitioner (Oracle-like) that assigns one large chunk to the GPU and the rest of the iterations to the CPU cores. The size of this single GPU chunk is computed by a previous offline search phase that exhaustively looks for a partitioning of the iteration space between CPU and GPU that minimizes the execution time. This profiling step runs the application 11 times. For each run, the percentage of the iteration space offloaded as a single chunk to the GPU varies (between 0%, only CPU execution, and 100%, only GPU execution, by increasing in 10% steps). For example, in Figure 4.23, we can see the execution time of Nbody and Barnes Hut on Haswell processor (4 CPU cores + GPU). With this approach, the profiling results for Nbody (left-hand side) and Barnes Hut (right-hand side) show that a 50% and a 70% of the iteration space should be computed on the GPU to get the fastest execution time. Notice that this Static approach does not add any runtime scheduling overheads, but the profiling step requires 11 runs for each possible configuration of the number of threads ($nThreads > 1$), which may result in a time consuming approach. Moreover, while static may work well for regular codes, it does not adapt to changes in irregular codes during execution time and it can be counter productive to offload large chunks of iterations to the GPU, as shown in Section 4.2.

The second considered approach is Concord [58] which also focus on accurately computing the relative speed of the GPU and the CPU. In this case, this is accomplished by assigning a fixed chunk size to the GPU while the CPU cores

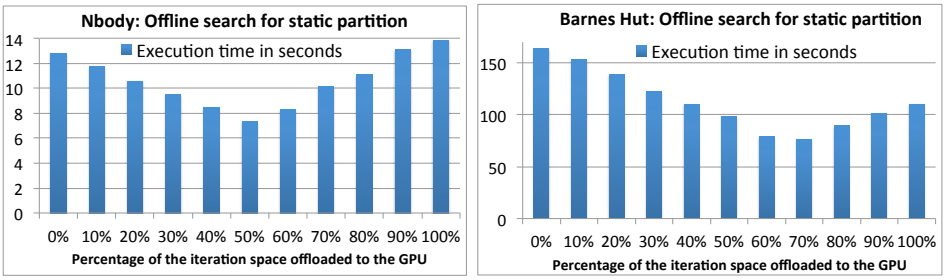


Figure 4.23: Offline search for the near optimal GPU-CPU workload partition on Haswell processor.

pick some chunks until the GPU finishes its assigned chunk. Once the GPU, finishes its assigned chunk of iterations, Concord moves to the execution phase where the computational speed of each device is computed and the remaining iterations are distributed across CPU cores and GPU based on their respective computational speeds. Concord also monitors and updates the computational speed of each processor in each parallel for loop invocation.

The third approach is HDSS [4], which corresponds to a heterogeneous dynamic self-guided scheduler that comprises two phases. The first phase is adaptive and it aims at finding a stable chunk size that fully uses all the GPU's computational resources in order to compute an accurate computational weights among devices. This phase uses a training phase to compute the relative speed of the GPU with respect to the CPU (our computational speed f). It starts with a small chunk size and increases it gradually while recording the corresponding throughput of each sample. With the first four samples, HDSS computes a logarithmic fitting of the curve and computes a first recommended GPU chunk size, as explained in Figure 4.8b. Then, HDSS keeps iterating in the training phase by adding more samples points and recomputing the logarithmic fitting and the relative speed until the slope of the curve in the last point is less than a given threshold (1%) or a fixed percentage (20%) of the iteration space is computed. Then, it moves to a completion phase where block sizes computed in the adaptive phase are no longer used. Instead, HDSS starts assigning chunks to the CPU and GPU relying on a Modified Guided Self-Scheduling (MGSS): it first assigns the largest possible chunk size to each device considering its relative speed and gradually reduces the chunk sizes towards the end of the iteration space to avoid load imbalances.

LogFit departures from these two previously described approaches in two

ways. First, instead of looking for an stable relative speed that may never be found for irregular codes, our main goal is to adaptively select the recommended GPU chunk size that is large enough to fully feed all the GPU's compute units. LogFit is the only alternative that considers variations in the relative speed during the whole application execution. Thus, for each scheduling interval, it also recomputes the CPU chunk sizes to ensure load balance between the CPU and GPU; Second, instead of having a steady phase following the adaptive one, LogFit keeps monitoring and adapting the GPU and CPU chunk sizes, while trying to minimize the overheads of the adaptive mechanisms, as we demonstrate next. Notice that LogFit and Concord remember information from one time-step to the next one: LogFit remembers three points $\{(Ch(I_{x1}), \lambda(I_{x1})), (Ch(I_{x2}), \lambda(I_{x2})), (Ch(I_{x3}), \lambda(I_{x3}))\}$ samples, whereas Concord remembers the CPU-GPU relative speed. On the other hand, HDSS does not re-use previous information.

Next, we present the evaluation of performance and energy consumption of our LogFit approach and the three aforementioned partition strategies on the two Intel processors introduced in section 4.4.1. Figures 4.24 (Ivy Bridge) and 4.25 (Haswell) show the execution time and energy consumption for the five introduced benchmarks in this chapter. These Figures show two different graphs for each benchmark: the left-hand side bar graphs show the execution time (in milliseconds) of executing the experiments from one thread (only GPU execution) to five threads. In general, as we increase the number of threads, and use more computing units, we reduce the total execution time. The right-hand side bar graphs show the total energy consumption (Joules) while increasing the number of threads. We show an energy breakdown which distinguishes the energy consumed on the CPU cores, E_CPU, on the GPU, E_GPU, and on the uncore components of the chip, E_Un. Note that when using only 1 thread, we get the only-GPU execution. However, from 2 to 5 threads, we add one CPU core until 4, plus the GPU.

The study with the regular Nbody application aims at assessing the overhead of the adaptive engine in the three adaptive schedulers with respect to the Static approach. In the Ivy Bridge processor, as the Figure 4.24 shows, Concord and LogFit perform similarly and execution times and energy consumption are close to those of the Oracle-like Static implementation. However HDSS performs poorly in heterogeneous executions (from two to five threads). On the contrary, the results on Haswell processor, Figure 4.25, show that all alternatives performs similar. Nevertheless, for this regular benchmark (Nbody), Concord and LogFit execute faster than Static. In the Figure 4.25, this occurs for 5 threads because Static only evaluates 11 different partitions, while the adaptive approaches may

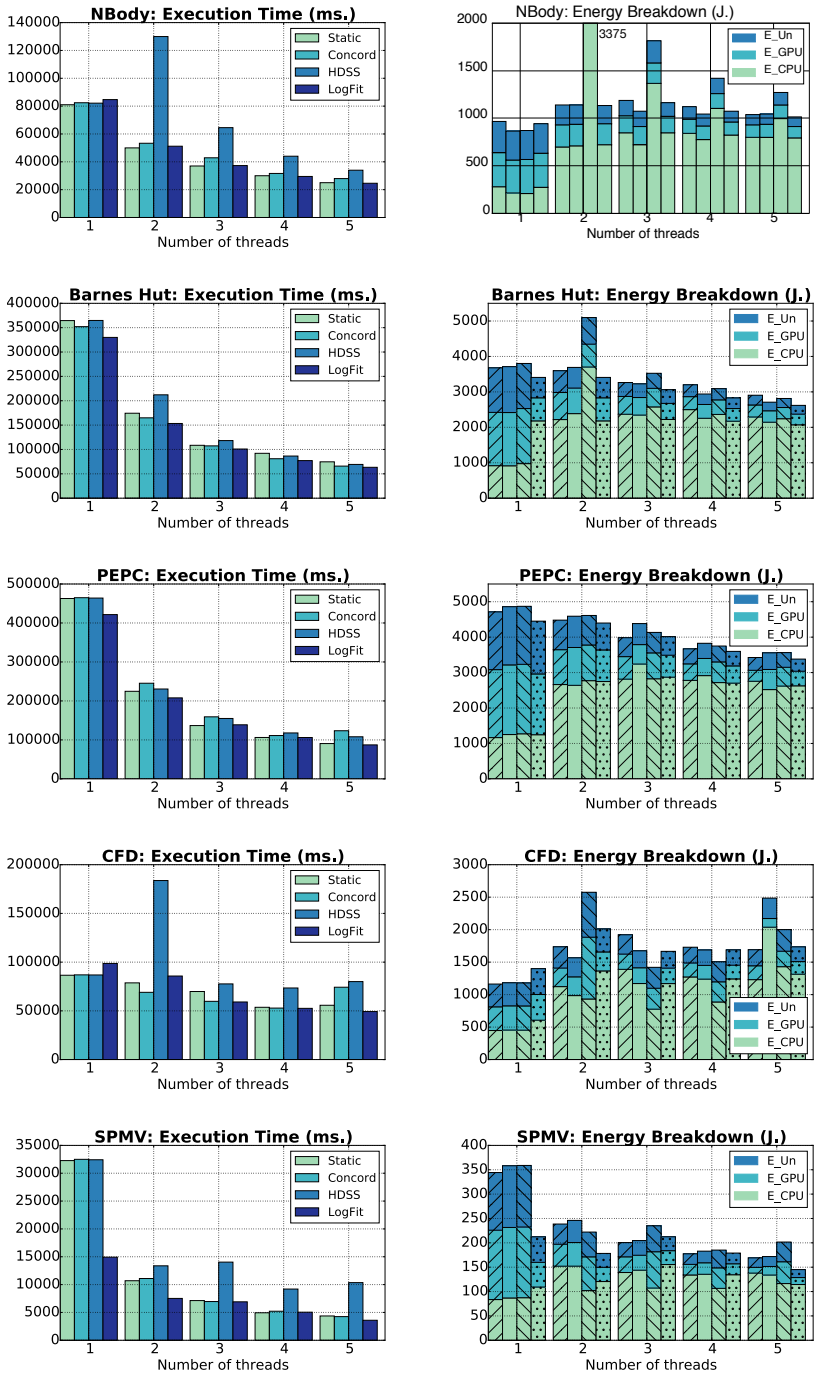


Figure 4.24: Results for Nbody, Barnes Hut, PEPC, CFD and SPMV benchmarks on Intel Ivy Bridge Processor. Time and Energy graphs, left to right: Static (//), Concord ( ), HDSS (\\) and LogFit (:).

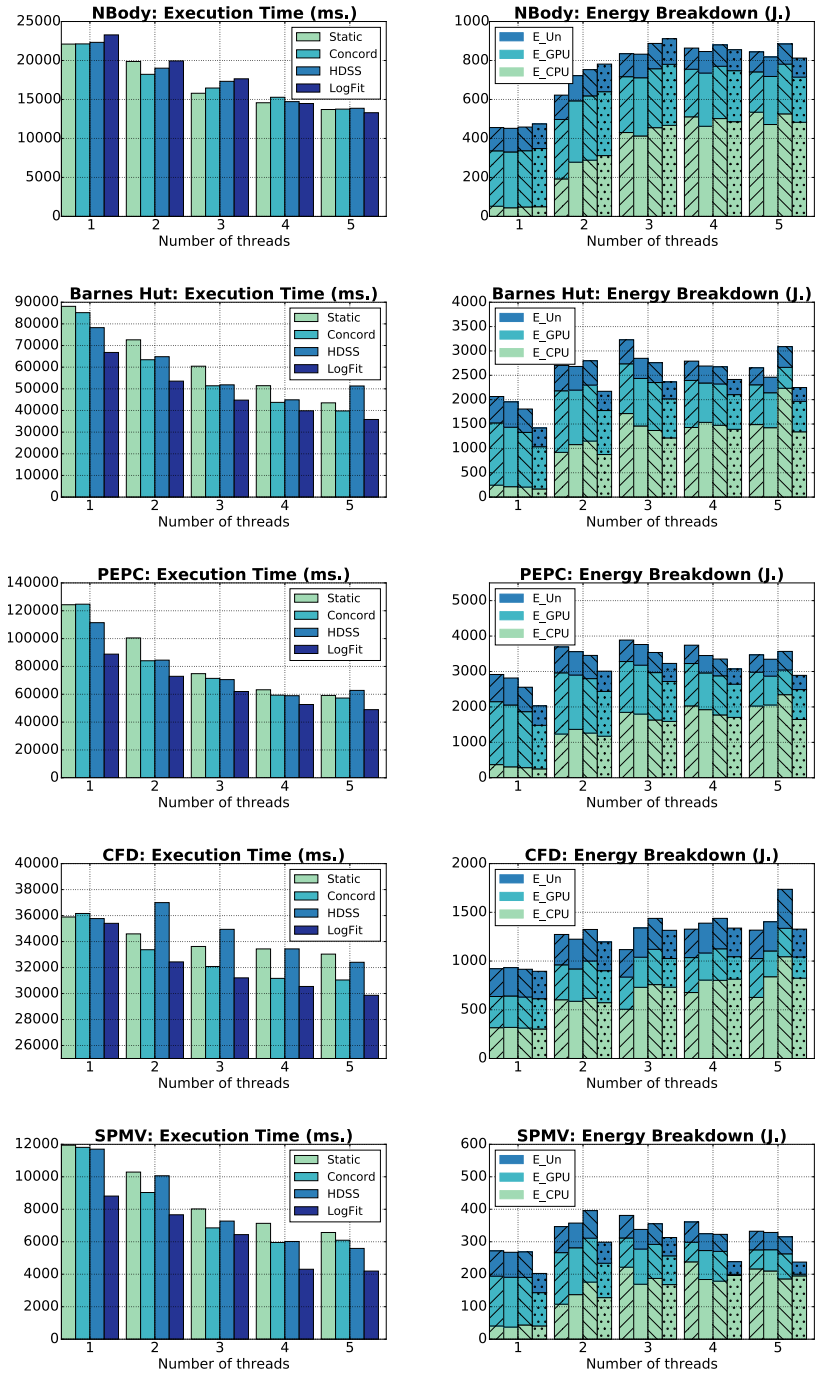


Figure 4.25: Results for Nbody, Barnes Hut, PEPC, CFD and SPMV benchmarks on Intel Haswell Processor. Time and Energy graphs, left to right: Static (//), Concord ( ), HDSS (\\) and LogFit (:).

find a finer distribution of work between CPU and GPU. HDSS pays an additional overhead because it trains at the beginning of each time-step, while Concord and LogFit can use previously computed information. The 1 thread execution (only GPU execution) shows the maximum overhead of dynamic strategies, Concord, HDSS and LogFit are 1%, 2%, and 5% slower than Static, respectively in both architectures. These results show that LogFit has an acceptable overhead in comparison with the Static approach. However, LogFit does not need the offline profiling that Static requires and performs better than any other partition strategy with 5 threads, because it finds a near optimal chunk size by assigning controlled chunks to the GPU and by performing a finer load balance strategy thanks to the LogFit's **Final Phase**.

Figure 4.24 shows that LogFit outperforms the Static and other dynamic alternatives for 1 thread (GPU execution) up to 55% for SPMV, except for CFD and Nbody where it is 10% and 5% slower respectively. In general, all alternatives reduce proportionally the execution time as we increase the number of threads, with the exception of HDSS for Nbody, CFD and SPMV where it performs poorly. Looking at execution time bar graphs, we can observe how LogFit always outperforms the other alternatives with the maximum number of allowed threads (5), it executes between 12%-18% faster for coarse grain algorithms Nbody, Barnes Hut and PEPC and 22%-28% faster for fine grain algorithms, CFD and SPMV. Looking at the Energy bar graphs, we observe that LogFit gets similar results to Static and Concord for coarse grain benchmarks, this is due to the clock frequency domain sharing between CPU cores and GPU. It means that whether the CPU or the GPU is the only device working, the other device will consume a proportional energy as well. This fact does not occur in Haswell micro-architecture as each processor the CPU and the GPU has its own clock frequency domain. In this sense, we only see a noticeable difference in energy consumption when the execution time has big differences, as it happens with CFD and 5 threads, where LogFit is a 30% more energy efficient than Concord and a 13% more efficient that HDSS.

Looking at Figure 4.25, we can analyse how these benchmarks and scheduling strategies perform on the Haswell Architecture. Thus, for the irregular coarse grained benchmarks, Barnes Hut and PEPC, all adaptive approaches outperform the Static one. In terms of execution time, for 1 thread (only GPU), LogFit runs 33% and 40% faster than Static for Barnes Hut and PEPC, respectively. It keeps outperforming all other alternatives with any other number of threads, e.g., for 5 threads LogFit runs 22% and 21% faster than Static for Barnes Hut and PEPC, respectively. However, for 5 threads, Concord just runs 10% (Barnes Hut) and 3% (PEPC) faster, whereas HDSS gets poorer result than Static with 5 threads.

According to the Energy bar graphs, LogFit achieves the minimum energy for 1 thread (GPU execution) while executing Barnes Hut. It consumes 31%, 27%, and 22% less energy than Static, Concord and HDSS, respectively. Moreover, LogFit is more energy efficient than the other partition strategies while executing PEPC: again, for 1 threads, it consumes 30%, 28% and 21% less energy than Static, Concord and HDSS, respectively. As illustrated in the Figure, for both, Barnes Hut and PEPC, for any given number of threads, LogFit always delivers the best performance with the minimum energy consumption.

For fine grained applications, CFD and SpMV, the profiling phases implemented in Concord and HDSS behave worse than the LogFit's exploration phase. Let's recall that in both approaches, the relative speed resulting at the end of the profiling phase is used during the rest of the execution. However, in these two benchmarks, to find the right relative speed we need to perform an exhaustive search by sampling larger chunk sizes. These two fine grained benchmarks also pose and additional challenge as the GPU chunk size required to yield a near optimal GPU throughput is comparable to the whole iteration space. Thus, an exploration phase that allows to profile the whole iteration space will find an optimal chunk size that better exploits the GPU, as LogFit does.

HDSS performs poorly while executing CFD, as this approach executes the profiling phase for each one of the 6000 time-steps of this application. During each run of the profiling phase, HDSS offloads small iteration chunks to the GPU, with sub-optimal chunk sizes. When finally HDSS finishes the profiling phase there are not enough remaining iterations to assign an optimal chunk size to the GPU that obtains the predicted GPU throughput, so, the estimated relative speed can not be guaranteed. In addition, this leads to load imbalance with the CPU. LogFit successfully detects that, for the exhibited granularity, the number of available iterations is not enough to assign to the GPU a chunk sufficiently large to obtain the predicted GPU throughput, so in order to reduce load imbalances the chunk is not assigned to the GPU and the remaining iterations are computed on the CPU cores. By looking at the execution time figures, we can observe that LogFit always outperforms the other alternatives, up to 11%, 5% and 9% faster than Static, Concord and HDSS, respectively. Also for CFD, Concord and HDSS consume 6% and 24% more energy than LogFit, respectively. The improvement of using LogFit is even bigger when executing SpMV. For instance, with 1 thread, LogFit runs 36%, 34%, 33% faster than Static, Concord and HDSS, whereas with 5 threads, it runs 56%, 45% and 33% faster than Static, Concord and HDSS, respectively. According to the energy bar graphs, the most energy efficient configuration is LogFit with 1 thread, it consumes 34% less energy than the other static and adaptive strategies.

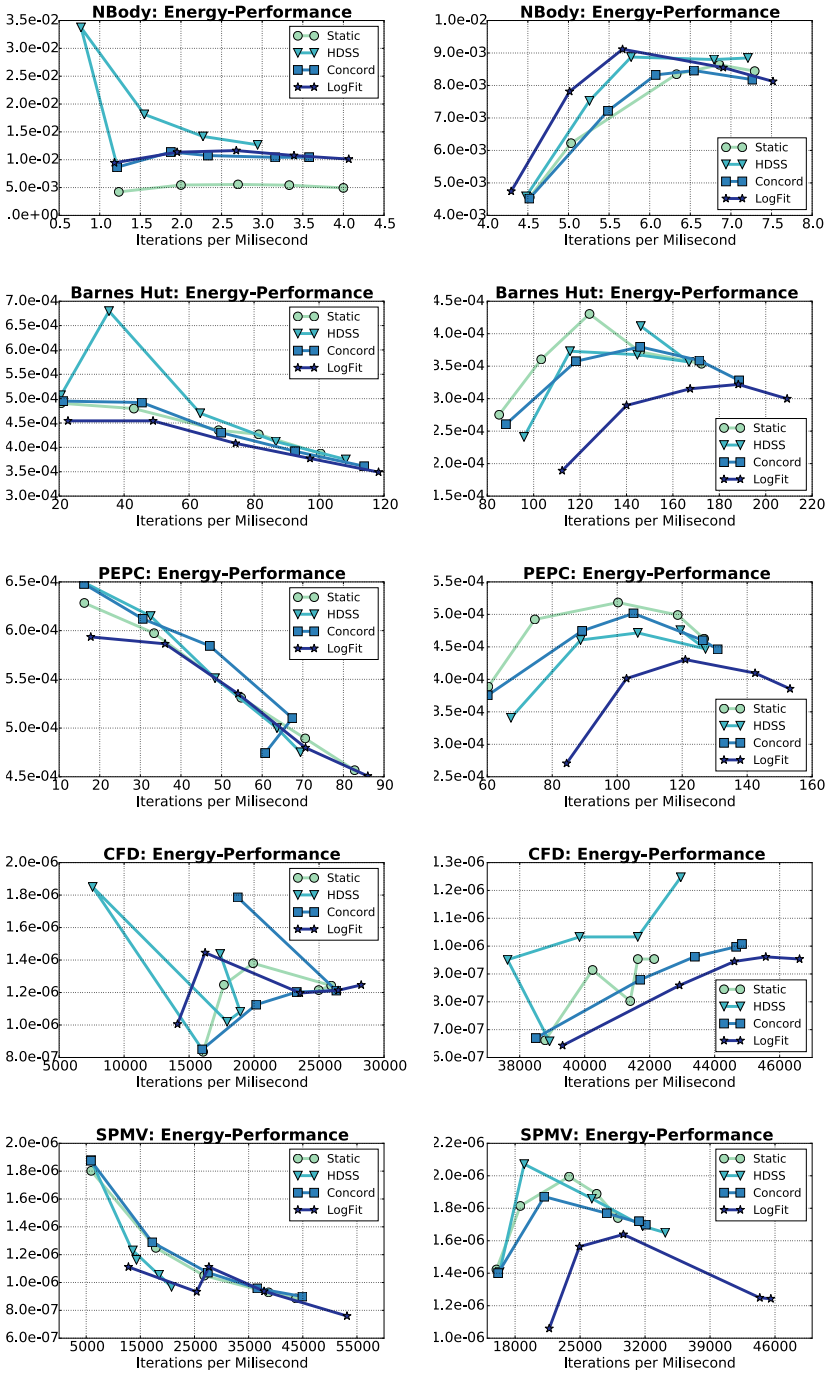


Figure 4.26: Energy-Performance results for Ivy Bridge Processor, left-hand side plots, and Haswell Processor, right-hand side plots. Nbody, Barnes Hut, PEPC, CFD and SPMV benchmark results are plotted from top to bottom.

To help understand how performance and energy consumption are related, we present the Figure 4.26, where all graphs represent the energy consumed by processing each iteration (y-axis) with respect to the number of iterations computed per millisecond (x-axis). Thus, the rightmost values and the bottommost are preferred, as they means that we can compute more iterations with less energy consumption. In these energy versus performance graphs, the left-hand side plots correspond to the Ivy Bridge processor. On the contrary, the right-hand side plots correspond to the Haswell processor. Each point in these lines represent a configuration with a different number of threads from 1 (leftmost) to 5 (rightmost). Note that in these plots, the closer to the right-bottom zone, the better the trade-off between energy consumption and throughput is. Typically, for the Ivy Bridge processor when increasing the number of threads, the curve moves towards the right-bottom corner (better performance and less energy consumption), although there are some exceptions as CFD. In general, for Ivy Bridge, LogFit is always the most performing alternative for all tested benchmarks. However, looking at the energy axis, Static is the most efficient for NBody. The shape of the Ivy Bridge's curves are dominated by the fact that the GPU consumes more energy than the CPU cores. Thus, as we gradually add more CPU cores the energy consumption decreases, as the CPU compute more efficiently. The plots for the Haswell processor are in the right-hand side of the Figure 4.26, they again show that LogFit is the most performing approach for all benchmarks with five threads. As we can observe in these plots, while increasing the number of threads, the curves move towards the upper right corner (higher performance and energy consumption). In this case, the GPU is more energy efficient than the CPU cores, thus the leftmost point in Haswell's lines are the lowest ones. Moreover, while adding CPU cores, we get better performance, but we also increase the energy consumption as the Haswell CPU cores are less energy efficient than the Haswell's GPU.

Summarising, the Energy-Performance plots show that again LogFit is the approach that consumes the least energy (for 1 thread) and the most performing one (for 5 threads). On average, considering the four irregular benchmarks and 5 threads, LogFit runs faster than Static, Concord and HDSS by 27%, 19.5% and 28% and consumes 15%, 14% and 22% less energy on the Haswell Processor.

4.5. Conclusions

In this chapter, we address the problem of finding the appropriate chunk size for GPU and CPU cores in the context of parallel loops in irregular applications running on heterogeneous CPU-GPU chips. We propose LogFit, a novel adap-

tive partitioning strategy that dynamically finds the chunk size that gets near optimal performance for the GPU at any point of the execution, while balancing the workload among the GPU and the CPU cores. LogFit monitors the GPU throughput during the application execution and uses a logarithmic fitting to adaptively partition the iteration space. Using a regular and a set of irregular benchmarks, we have assessed the performance and energy consumption of our partitioner with respect to a Static approach and other adaptive state of the art partitioners. For the studied irregular benchmarks on Haswell with 5 threads, we outperform the Oracle-like Static approach by up to 52% (18% on average) and avoid the exhaustive offline profiling. With respect to the state-of-the-art Concord and HDSS approaches and for 5 threads, we obtain up to 94% and 69% of speedup improvement (28% and 27% on average), respectively. Among all the approaches, LogFit is almost always the solution that results in the minimum energy consumption or the maximum performance. As future work, we will consider the parameter of energy consumption as part of the scheduling decisions.

5 Pipeline Pattern: Optimal pipeline configuration

In the previous two chapters, we propose extensions to efficiently execute the *parallel_for*, a load balancing model and a performance-aware partitioner. However, in this chapter, we tackle a different problem. We focus on efficiently executing streaming applications on commodity processors composed of a multicore CPU and an on-chip GPU [108]. Streaming applications, such as those in vision and video analytic, consist of several pipelined stages that are good candidates to take advantage of this type of platforms. To implement these kind of applications, we extend the TBB's pipeline template to allow its execution on heterogeneous architectures. Thus, each stage of the pipeline can either be executed on the CPU, or on the GPU. We also consider that characteristics of the input stream may change while the application is running. Therefore, we propose a Runtime System (RS) that adaptively finds the optimal mapping of the pipeline stages. The core of the RS is an analytical model coupled with information collected at runtime used to dynamically map each pipeline stage to the most efficient device, taking into consideration both performance and energy. Our RS can be targeted at optimizing performance, energy or a tradeoff metric that considers the ratio throughput/energy. Our analytical model can provide knobs so that the user can specify a desired throughput or power budget. For instance, if the user specifies a throughput of 33 fps (frames per second) for real time video streaming, the model can determine among the possible pipeline configurations, the one that minimises the energy consumption and satisfies the user constraint. Similarly, given a power budget, the model can determine the fastest configuration.

In this chapter, we first motivate the need for solving the pipeline mapping problem and account for all possible configurations in Section 5.1. Next, we in-

troduce the extension of the TBB pipeline template to allow its execution on heterogeneous architectures, Section 5.2. To deal with the pipeline mapping problem, we model it by using queueing theory and propose an heuristic function that finds the best pipeline configuration in Section 5.3. Later, we present the experimental results that include a comparison with the state-of-the-art in Section 5.4. Finally, we sum up with conclusions in Section 5.6.

5.1. Pipeline configuration problem

In this section, we focus on the problem of efficiently scheduling a single streaming application, implemented as a pipeline of stages, that run on heterogeneous chips comprised of several cores and one on-chip GPU, taking into consideration both performance and energy. Streaming applications are very common in current computing systems, specifically in mobile devices [20], where heterogeneous chips are the dominant platforms. Recently, we have seen a significant increment in the number of commodity multicore processors that include an on-chip GPU. Current desktops, ultrabooks, smartphones, tablets, and other embedded devices are also powered by heterogeneous chips that comprise from 2 to 8 CPU cores along with an integrated GPU. The rising number of these platforms is driven by the demand of higher performance and the limitations on power and scalability of multi-core CPUs.

However, developing applications on these architectures is further complex. Most research in frameworks aimed at scheduling tasks on heterogeneous architectures, composed of CPUs and GPUs, has focussed on optimizing execution time without considering energy consumption [2, 10, 40, 70, 104]. However, a CPU and a GPU may exhibit different performance/energy trade-offs, this is, a task (or pipeline stage) can run faster on one device but consume less energy on other. Thus, in order to benefit from the potential energy efficiency that the accelerators can provide in these heterogeneous chips, the runtime scheduler also needs to consider the performance/energy asymmetry when making a scheduling decision [97].

Thus, in this section we consider how to schedule the stages of a pipeline on a heterogeneous architecture by taking into account the performance/energy asymmetry of these platforms and scalability issues. To tackle the aforementioned problem, we study different choices such as:

- The granularity level at which the parallelism of each stage can be exploited (coarse or medium grain).

- The mapping of the pipeline stages onto the different computational devices considering their relative performance and energy.
- The number of threads for which the application scales up.

Our aim is to find the best configuration that considers all these factors. We also consider that the best configuration may change during execution time. This can happen because the number of operations performed on each pipeline stage may change over time. There are several reasons why this can occur. For instance, YouTube, Skype Video [19], or network operated robots [35] adjust the resolution of the video stream according to the available bandwidth of the network connection. Also, the computation of a pipeline stage may depend on the characteristics of the input frame. In this scenario, an off-line training may not be feasible, as the best configuration may depend on the data input.

As a motivating example to demonstrate the benefits of adapting the configuration of a pipelined application, we introduce ViVid, an application that implements an object (e.g., face) detection algorithm [29] using a "sliding window object detection" approach [57]. ViVid consists of 5 pipeline stages from which the first and the last one are the Input and Output serial stages, the three middle stages are parallel, as shown in Figure 5.1. When applications like ViVid run on a heterogeneous chip architectures, many possible configurations are possible. To determine the best configuration one needs to consider the granularity, the number of items that should be simultaneously processed on each stage, the device where each stage should be mapped, and the number of threads that minimise the execution time, the energy consumption, or both.

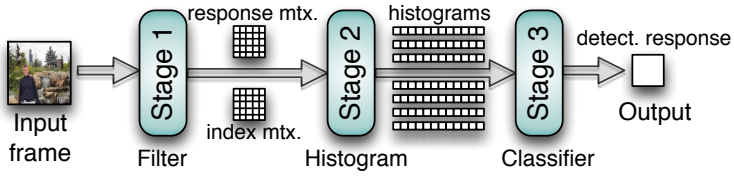
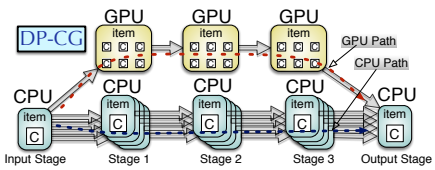


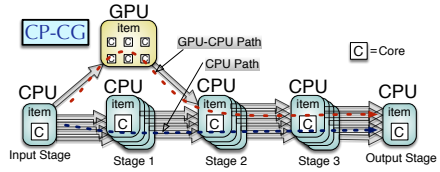
Figure 5.1: Flow of ViVid application divided in 3 parallel stages.

5.1.1. Pipeline configuration alternatives

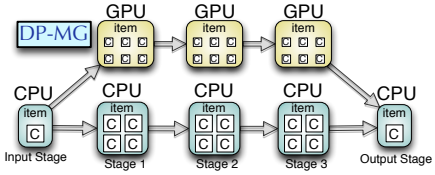
There is a large degree of leeway when it comes to schedule the stages of a pipelined application on heterogeneous architectures. We use two axes to classify the different alternatives: (1) *granularity level*, that represents the level at



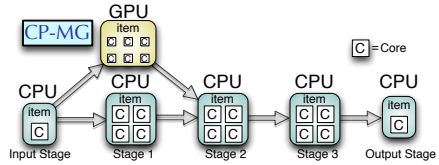
(a) Decoupled mapping, coarse-grain, DP-CG.



(b) Coupled mapping, coarse-grain, CP-CG.



(c) Decoupled mapping, medium-grain, DP-MG.



(d) Coupled mapping, medium-grain, CP-MG.

Figure 5.2: Categorisation of the four main configurations for ViVid.

which the parallelism is exploited on the CPU; and (2) *pipeline mapping*, that represents how the pipeline stages are assigned to the available processors. We use the term *pipeline configuration* for each possible combination of granularity and mapping. Figure 5.2 graphically depicts a pipeline classification which is divided in 4 categories. In deeper detail, this Figure shows 4 possible configurations for the ViVid pipeline on an Ivy Bridge-like architecture with a GPU (6 computing units) and a CPU multicore (4 CPU cores). Moreover, this figure shows the paths that traverse the in-flight items being processed. Each item is processed by a thread, from the first stage until the last one. The pipeline stages are represented as rounded rectangles, while the device (GPU or CPU) on which each stage is processed, is depicted with the number of computing cores (small squares with the letter 'C') that collaborate on the computation of each item.

Granularity level: The vertical axis in Figure 5.2 introduces a pipeline classification based on the granularity level used to exploit parallelism on the CPU. Two levels of granularity are considered: Coarse Grain (CG) and Medium Grain (MG). If different items can be processed simultaneously on the same stage (the stage is parallel or stateless[1]), then CG granularity can be exploited. Additionally, if several items can be processed simultaneously and each CPU thread can

[1]A stage is parallel or stateless when the computation of an item on a stage does not depend on other items.

process one item through all the stages, then CG granularity can be exploited. Furthermore, if a pipeline stage exhibit nested parallelism (which can be exploited by using OpenCL, OpenMP or TBB by using a *parallel_for*), then a single item can be processed in parallel by several cores in the CPU, and MG granularity can be exploited. In the MG, each pipeline stage executes a single item at a given time, and it exploits *nested parallelism*. By nested parallelism, we mean that a set of threads are deployed inside a pipeline stage by following a *fork-join* pattern. CG granularity is shown in Figures 5.2a and 5.2b, while MG granularity is shown in Figures 5.2c and 5.2d.

GPUs are not as flexible as CPU multicores are, according to the granularity level of parallelism they can exploit. They excel at exploiting SIMT (Single Instruction Multiple Threads) type of parallelism. Thus, stages mapped onto a GPU can only process a single item, with all the GPU processing units computing a portion of the item (similar to MG granularity, but at a finer grain).

The MG granularity requires a barrier synchronization at the end of each pipeline stage and before the next pipeline stage can start, to guarantee that all participating threads have finished processing their corresponding part of the item. Therefore, MG can hurt performance when the load is unbalanced or there is not enough computational load per core. With MG, it is like having two devices, a GPU and a CPU, that can only work on two different items at a time. Thus, there is less pipeline parallelism when exploiting MG granularity. On the contrary, with CG granularity, each CPU core (or thread) can process an item throughout all the pipeline stages without intermediate synchronizations, and each item traverses the pipeline at its own pace. Only at the end, when the item reaches the output stage, it synchronise and waits for this stage to sort the order of the output stream. Two drawbacks of the CG approach are that several items are in-flight at the same time, increasing the memory pressure (specially when memory bandwidth is limited), and that only applies to parallel pipeline stages (i.e. stateless pipeline stages). Additionally, notice that CPU cores can also exploit fine grain parallelism, due to the vector units of the processors (AVX, SSE), orthogonally to both, CG or MG granularities.

Pipeline mapping: The horizontal axis in Figure 5.2 classifies the configurations based on whether all the stages execute on the GPU or only a few do. The first pipeline mapping is called *decoupled* (DP), while the other one is called *coupled* (CP). Disregarding the Input/Output stages, DP mappings are illustrated in Figures 5.2a and 5.2c, where we depict two "decoupled" paths: (1) a *GPU path*, in which a thread (the GPU host thread) offloads all stages to the GPU for processing one input item; and (2) a *CPU path*, in which a group of concurrent threads (the CPU threads) process all stages on the CPU. Furthermore, CP

mappings are shown in Figures 5.2b and 5.2d where we see two paths: (i) a *GPU-CPU path* in which a thread (the GPU-CPU thread) offloads some stages to the GPU (stage 1 in the figures) for processing one input item, while the remaining stages are executed on the CPU; and (ii) a CPU path, in which a group of concurrent threads (the CPU threads) process all stages on the CPU multicore. The difference between the CP's GPU-CPU thread and the DP's GPU thread is the following. In a CP mapping, when an item reaches the stage for which it has been decided that it will be processed on the GPU (stage 1 for the ViVid example), we first check if the GPU is idle, and in that case the thread becomes a GPU-CPU thread that launches the item's kernel to the GPU and then waits for the GPU kernel to finish. Then, the same thread also processes the item through the remaining stages (in the example, stages 2 and 3 that are executed on the CPU). However, in DP, when an item reaches the first stage and finds the GPU idle, the corresponding thread becomes the GPU host thread that executes the item throughout all the stages on the GPU. Indeed, when we consider only 1 thread for the DP mapping, that thread becomes the GPU host thread and therefore all the items traverse the GPU path. This is what we call a *GPU homogeneous execution*. In our example, for both CP and DP, if an item on the stage 1 finds that the GPU is already busy, then the item is directed through the CPU path. Although DP could be seen as a particular case of CP where all the stages happen to be mapped to the GPU, we distinguish both mappings because they have to be modelled differently as we will see in section 5.3.2.

CP mappings can be a good alternative when all the stages are not suitable for the GPU, or because it's not advisable to divert the GPU computing power from the stages where it is faster and/or more energy-efficient. This approach also has the advantage that not all the stages need to be implemented for the GPU. However, in the CP mapping, the GPU-CPU thread must orchestrate the "coupling" of the GPU and the CPU devices and the host-to-device/device-to-host communications, which results in some data movement and synchronization overheads. Also, note that DP mappings can be implemented only if all stages are parallel pipeline stages (stateless). On the contrary, if all stages are serial, heterogeneity may be exploited by mapping some stages on the GPU and the rest on the cores, which is a particular case of the CP mapping in which all items are directed through the GPU-CPU path.

Configurations not considered

Some additional alternatives not considered in this classification are the following:

- Splitting an item to be simultaneously computed on the CPU and GPU. As it is demonstrated by Totoni et al. [104], this possibility is not beneficial for

our vision applications due to additional synchronization overheads between both devices.

- Having one stage exploiting both MG and CG granularities on the CPU. For example, a quad-core can be split into two CPU devices with two cores each. This approach would combine CG and MG on the same stage: two CPU devices processing two items in parallel (CG), and each item running on two cores (MG). For that, we explored the OpenCL Device Fission function (`cl_ext_device_fission`) that can divide the CPU device into several subdevices with lower core count. However, we discarded this alternative due to we measured a 14% of overhead (for ViVid on Ivy Bridge) if the `device_fission` is called to change the subdevices configuration from one pipeline stage to the next one. Thus, this approach is beneficial only if the optimum number of cores per device coincides for all the pipeline stages so the fission function is just called once.
- Exploiting stages with both MG and CG granularities on the GPU. The OpenCL fission feature is currently not able to split the GPU on Intel or AMD heterogeneous chips. Therefore, we do not consider this feature to evaluate additional pipeline configurations.
- Hybrid mappings in which some CPU stages exploit MG and the rest CG granularity. This is left for future work.

Accounting for all pipeline alternatives

Assume that we have $nC = 4$ CPU cores (in Figure 5.2), and 1 GPU in a heterogeneous chip, current commercial heterogeneous chips only contain a single GPU, so we overlook configurations with two or more GPUs. In addition to the 4 pipeline configurations, DP-CG, DP-MG, CP-CG and CP-MG, there are two additional factors to consider:

- For CP mappings, we need to find out the stages for which the GPU is more profitable;
- For CG granularity, we also have to explore the optimal number of threads.

For this CG granularity the number of threads in the CPU multicore can go from 1 to nC. Additionally, since the GPU host thread in DP-CG, or the GPU-CPU thread in CP-CG, are mainly hosting the GPU (waiting for the GPU kernel to complete), the total number of threads, n, we explore goes from 1 to $nC + 1$.

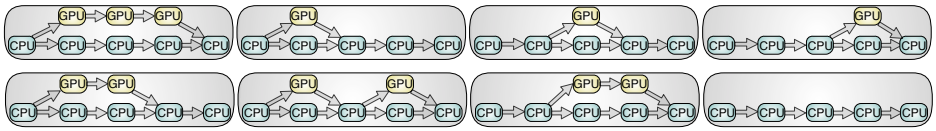


Figure 5.3: All possible mappings of pipeline stages to CPU and GPU for ViVid.

This means that we allow oversubscription of one thread when $n = nC + 1$, and therefore, the GPU (or GPU-CPU) thread eventually interferes with the other nC CPU threads. For the MG granularity, we always configure $nC + 1$ threads because the constructors used to exploit nested parallelism (OpenCL, OpenMP or TBB `parallel_for`) by default use all the threads available in the multicore, nC, plus the GPU (or GPU-CPU) host thread.

With all aforementioned variables, we use ViVid application to illustrate all possible pipeline alternatives. Disregarding the Input and Output serial stages, ViVid has 3 parallel stages that can also exploit nested parallelism. So, there are 2^3 possible CPU/GPU configurations, assuming that the ViVid consists of s parallel stages, there would be 2^s possible GPU/CPU mappings, this is shown in Figure 5.3, for ViVid with $s = 3$. This accounts for all the stages running on the GPU (DP mapping) and the $2^s - 1$ possible CP mappings. These mappings can be combined with $nC+1$ different CG configurations, depending on the number of threads used and 1 MG configuration, i.e., $nC+2$ configurations. Thus, in total we have $2^s \cdot (nC+2)$. That results in 48 alternatives for ViVid with $s = 3$ and $nC = 4$. If the available resources include more GPUs/multicores, these configurations can be seamlessly extended to accommodate the additional computing units. Our goal is to be able to predict the optimal pipeline configuration specifying the granularity, mapping (identifying the stages that should be mapped on the GPU), and the optimum number of threads for a given stream input.

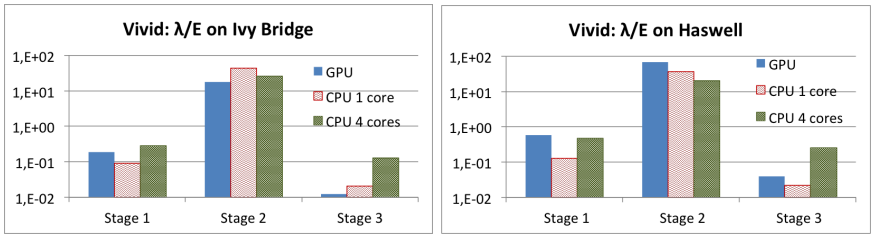
Before undertaking the search of the optimal pipeline configuration, we need a metric that allows us to quantitatively compare the different alternatives. In the context of heterogeneous execution and GPGPU, the *Energy-Delay Product* (EDP) [43] has been widely used as it can measure both performance and energy consumption. However, for streaming applications, in which an unknown number of items have to be processed, the measurement of the delay until execution completes can be an issue. To circumvent that problem we propose a related metric that provides similar information for each item in the pipeline.

The proposed metric is throughput/energy per item, λ/E, where λ is the number of processed items per second, whereas E stands for the consumed Joules per item. We can compute it by measuring the time, t, and energy, E, needed to

process an item. With this, $\lambda/E = \frac{1/t}{E} = 1/(E \times t)$, that is actually $1/EDP$. In summary, λ/E is inversely proportional to the Energy-Delay Product per item, 1/(Joules × sec). Therefore, the larger the metric, the better. The analytical model described in Section 5.3 can be used to find the pipeline configuration that provides the optimal λ/E.

5.1.2. Putting throughput/energy metric to work

In Figure 5.4, we see the resulting throughput per unit of energy (λ/E expressed, axis-y in log. scale) for each stage of the ViVid application and Low Definition (LD) frame resolution. These results are obtained for ViVid on the Ivy Bridge and Haswell processors, presented in Section 5.4, when just one GPU thread carries out the computation on the GPU (blue bar), when just one CPU thread carries out the computation in 1 CPU core (red bar) and when 4 CPU threads exploit MG granularity on four cores (green bar). In all these executions, we execute one stage at a time, thus no pipeline parallelism is exploited.



(a) Ivy Bridge (b) Haswell

Figure 5.4: Throughput / Energy for ViVid without pipeline parallelism.

For the Ivy Bridge processor (Figure 5.4a), we can see that the GPU does not surpass λ/E of the CPU 4 cores. However, on Haswell (Figure 5.4b), stages 1 and 2 are more energy efficient when executed on the GPU. This information can be collected at run-time, by processing a few frames of the input stream to make mapping decisions, such as not using the GPU at all in the Ivy Bridge or mapping stages 1 and 2 to the Haswell's GPU.

However, this type of reasoning can produce suboptimal results. First, these data are obtained with a single frame traversing the whole pipeline, so all hardware resources (a GPU, 1 core or CPU 4 cores) are focused on a single stage at a time, and we avoid side effects measurements due to other stages being concurrently executed. Thus, we do not know what may happen when the number

of in-flight frames increases, or when the CPU cores are working concurrently with the GPU. Second, these numbers are not enough to elucidate which pipeline configuration is desirable. To solve this problem, the analytical model described in Section 5.3.2 can be used to find a near optimal pipeline alternative without testing all the possible ones.

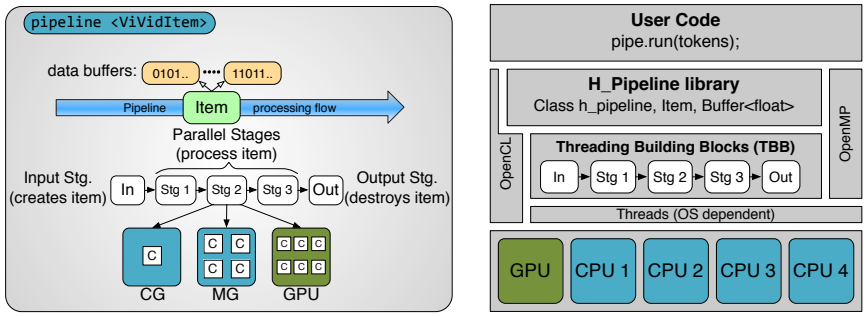
5.2. Pipeline template

In this section we introduce our *pipeline* library API. It provides a C++ template library that facilitates the configuration of a pipeline by hiding the underlying TBB and OpenCL implementations, and by automatically managing the memory data transfers and synchronisation between devices, likewise the *parallel_for* template does in Chapter 4. The pipeline interface has four main components:

- Items: objects that traverse the pipeline stages by carrying pointers to the memory data buffers.
- Pipeline: the pipeline object is composed of $s + 2$ stages. We assume that it contains the following stages: S_{input}, S_1, S_2, ... S_s, S_{output}, being S_{input} and S_{output} the serial Input and Output stages. A Pipeline may use a static configuration or run in a self-adaptive configuration mode. This adaptive configuration mode is the one that uses our proposed model to dynamically compute the best configuration and to change it according to input changes.
- Stage functions: each processing stage needs to be programmed to run on CPU and/or GPU. The pipeline uses the appropriate function for every stage.
- Buffers: n-dimensional arrays that can be used by both, the host code and the OpenCL kernels.

Figure 5.5a shows the components involved in the pipeline construct. The `Item` is the object that traverses the pipeline, it contains the references to the data buffers objects that are required by the pipeline stages as input and output. To create a new `item` instance, the user needs to define a new `Item` subclass (it must extend from a provided `Item` class) that should contain the references to data buffers used by the pipeline stages. For data buffer management, there is a `DataBuffer<T>` template class already defined, it hides all the important operations like allocation, deallocation, data movements, Zero-Copy Buffer mappings, etc. The aim of this data buffer class is to make data accessible to the

device (CPU or GPU) where the item has to be processed. Figure 5.5b shows the environment stack with the heterogeneous devices at bottom. On top of that, the software middle layers (TBB, OpenCL, OpenMP) provide different programming models to exploit the available parallelism on the heterogeneous chip. However, we partially hide the complexities of these low level libraries by providing a simpler interface based on C++ classes and templates.



(a) Pipeline building blocks (b) Software stack

Figure 5.5: Software stack and building blocks of the *pipeline* template.

The programmer can provide up to three different functions for each pipeline stage: one to implement the stage on the GPU device using OpenCL, a second one to implement the stage on a single CPU core, Coarse Grain (CG), and the third one to implement the stage on multiple CPU cores Medium Grain (MG). The implementations not provided by the user (CP, MG, and/or GPU) are not considered when searching for the best pipeline configuration. In the following subsections we cover all components in deeper detail.

5.2.1. Item class

Figure 5.6 shows a code snippet with the `Item` class declaration used as example. First, our pipeline interface is made available by including the `h_pipeline.h` header file (line 1). It also defines the `h_pipeline` namespace which contains all the classes of the interface. As previously mentioned, before creating the pipeline, the programmer has to define an `Item` subclass for all the instances that will traverse the pipeline (line 7). The item class must extend from `h_pipeline::Item` class and declare as many `DataBuffers<T>` members as required for the pipeline execution. The class constructor and destructor meth-

ods must hold the buffers creation and deletion, respectively. Alternatively, to avoid too many buffer allocation/deallocation operations, a pool of buffers can be used. In that case, acquire and release methods can be invoked, so the same buffers are reused by different items. The Input and Output stages (i.e. the first and last serial stages of the pipeline, S_{in} and S_{out}) automatically call the constructor and destructor of the `Item` class, respectively. In the last part of this section we show an example of buffer usage, Figure 5.11.

```
1  #include "h_pipeline.h"          // Required clases defined here
2  using namespace h_pipeline;      // New namespace
3
4  /*********************************************************************
5  * 1.- ITEM Class (holds the data that traverse the pipeline stages
6  *********************************************************************/
7  class ViVidItem : public h_pipeline::Item {
8  public:
9      //Buffer definitions
10     DataBuffer<float> *frame;   // Input buffer
11     ...
12     DataBuffer<float> *out;  // Output buffer
13
14    //Constructor definition. Allocation or buffer acquire
15    ViVidItem() {
16         //Data Buffer allocation
17         ...
18     }
19     //Destructor definition. Deallocation or buffer release
20     ~ViVidItem() { ... }
21 };
```

Figure 5.6: Extending the Item Class defined in *h_pipeline* namespace.

5.2.2. Pipeline class

Figure 5.7 shows a pipeline object instantiation and usage example. After declaring the `ViVidItem` class shown in Figure 5.6, we can create a pipeline instance by using that class (line 6) and by passing as constructor's arguments the number of threads, `numThreads`, that will execute the pipeline in parallel. In this study, we set the maximum number of threads to $nC+1$, being nC the number of CPU cores (see section 5.1.1). Also note that CPU (CG and MG granularity) and GPU functions need to be set up before the pipeline can run (see lines 9 to 11). In case we want to run the pipeline using a static configuration, we can use a specific method to configure some features of the pipeline, such as the stages that should be mapped to the GPU or the granularity (MG or CG) that should be used on the CPU (line 14). In our example, in line 14, the first argument {`1,1,1`}

represents a 3-tuple that represents the mapping of stages to the devices and the second argument, `USE_MG`, which indicates that MG granularity has to be exploited when an item is processed on the CPU.

As mentioned before, we assume that a pipeline consists of S_{in}, S_1, S_2, ... S_s, S_{out} stages (s+2). S_{in} and S_{out}, the serial Input and Output stages are always mapped to the CPU. For any other stage S_i such that $1 \leq i \leq s$, we use a s-tuple to specify all possible stage mappings to the GPU and the CPU devices: {m_1, m_2, ..., m_s}. The $i-th$ element of the tuple, m_i, specifies if stage S_i can be mapped to the GPU and CPU, ($m_i = 1$), or if it can only be mapped to the CPU ($m_i = 0$). If $m_i = 1$, the item that enters stage S_i checks if the GPU is available, in which case it executes on the GPU; otherwise, it executes on the CPU. For instance, for the example of Figure 5.3 (see page 132), we represent the tuples in row major order: {1,1,1}, {1,0,0}, {0,1,0}, {0,0,1}, {1,1,0}, {1,0,1}, {0,1,1}, {0,0,0}.

Once the pipeline is configured for a static configuration, it can be run (line 15) by setting the maximum number of items that are allowed to be simultaneously in flight and traversing the pipeline. Moreover, another option to run the pipeline is to use the adaptive configuration mode, line 18, presented in section 5.3.2. Under this mode, our framework dynamically finds the best configuration, where the user has to select the optimization criterion (`THROUGHPUT, ENERGY, THROUGHPUT_ENERGY`), the maximum allowed overhead (as a ratio between [0,1]) and the threshold `variation` variable which sets a limit to relaunch the training step. This step is required when running in the adaptive mode and it is explained in section 5.3.2. As an advantage, the user does not have to set the *numTokens* variable, as it is automatically calculated by the framework.

Implementation details

In this sub-section, we explain the internal details regarding the `pipeline<T>` class. This class is at the top of the software stack described in Figure 5.5b, so it is designed to work on top of TBB and OpenCL libraries. Figure 5.8 sketches the main implementation decisions we have considered for this `pipeline<T>` class.

Figure 5.8 shows the internal details of the pipeline template. The `pipeline` class is the main component of the library, it is responsible to glue the set of stages and schedule the items in flight to maximise the optimization criteria. In order to make this possible, the `pipeline` class defines a few member variables (lines 7-9). The `gpuStatus` variable shows the current status of the GPU at runtime: it can be ($0 = Idle$ and $1 = Busy$). This class also defines an ordered list of stages, (`l_stage`), that represents the stages of the pipeline. Additionally, there are two integer variables: `nthreads` which is used to set the number of

```
1  /*************************************************
2  * 2.- Pipeline declaration and usage
3  *************************************************/
4  int main(int argc, char* argv[]){
5    int numThreads = nC+1;   // number of threads = nC+1
6    h_pipeline::pipeline<ViVidItem> pipe(numThreads);
7
8    // Set CG, MG and GPU functions for each stage
9    pipe.add_stage(cg_f1, mg_f1, gpu_f1);
10   pipe.add_stage(cg_f2, mg_f2, gpu_f2);
11   pipe.add_stage(cg_f3, mg_f3, gpu_f3);
12
13   //Setting a static pipeline configuration: mapping '111' and MG
14   pipe.set_configuration({1,1,1}, h_pipeline::USE_MG);
15   pipe.run(numTokens);   // maximum number of items in flight
16
17   //Dispatch of the adaptive configuration mode for the pipeline
18   //pipe.run(ENERGY, maxoverhead, variation);
19 }
```

Figure 5.7: Usage and declaration of the *pipeline* template.

logical threads to be created and `num_stages` which has the current count of stages in the list. The class constructor (line 12) initializes the TBB library with the number of threads passed as argument. Then, the constructor creates the OpenCL environment (context, command_queues, ...) and selects the GPU as the target device.

To configure the topology of the pipeline, the pipeline class defines two methods (`add_stage()` and `set_configuration()`). As mentioned before, this class keeps a list of stages, where the Input and Output stages (first and last ones) are serial. For each one of the middle stages (parallel or stateless) the function `add_stage()` is called with three function pointers passed as arguments (CG, MG, GPU), so a new `parallel_stage` instance is created and inserted in the list (line 18). The `set_configuration()` (line 24) method allows the user to set a specific pipeline configuration. This method receives two parameters: an array of zeros and ones, where the i-th element specifies whether the i-th stage can use the GPU (1) or not (0). The second parameter is an enumerated type that sets the type of CPU functor (`USE_MG` for Medium Grain, MG, or `USE_CG` for Coarse Grain, CG) that should be used for all stages.

In order to execute the pipeline, the `run()` method must be invoked. Notice that it is possible to invoke two versions of the `pipeline.run()` method (lines 31 and 36), thus two types of modes are available: a static configuration or an adaptive configuration mode. The former (line 31) has a static behaviour, it means that just one pipeline configuration is used during the whole

```
/************************************************
* pipeline class inner details
************************************************/
template <class Item_T>
class pipeline : public tbb::pipeline {
  //members
  atomic<int> gpuStatus;  //0 GPU is idle, 1 GPU is busy
  list<parallel_stage> l_stage;
  int num_stages, nthreads, tokens;

  //Constructor
  pipeline(int numthreads){
    //Initialize TBB scheduler and OpenCL boilerplate
    num_stages=0; nthreads=numthreads;
  }

  //Adding Stages
  void add_stage(void (*cg_f)(Item_T*), void (*mg_f)(Item_T*), void (*gpu_f)(Item_T
      *)){
    parallel_stage<Item_T> * iStage(cg_f, mg_f, gpu_f);
    l_stage.add(iStage, ++num_stages);
  }

  //Setting Configuration for all stages
  void set_configuration(int mappings[], bool granularity){
    for(int i=0; i<num_stages; i++){
      l_stage.get(i).setConfiguration(mapping[i], granularity);
    }
  }

  //Overloaded pipeline.run(): static configuration mode
  void run(int ntokens){
    //Build the TBB pipeline and run it
  }

  //Overloaded pipeline.run(): adaptive configuration mode
  void run(const int criteria, float overhead, float variation){
    while(/*there are more items*/){
      //Training Phase:
        setConfiguration({1,1,1}, USE_MG); pipe.run_training(1);
        setConfiguration({0,0,0}, USE_MG); pipe.run_training(1);
        setConfiguration({0,0,0}, USE_CG);
        for(int i=1; i<=(nthreads);i++){
          pipe.run_training(i);
        }
        computeModelAndSetBestConfiguration(criteria);

      //Running Phase: It can abort if a change in throughput is detected
      pipe.run_monitoring(ntokens, overhead, variation);
    }
  }
  ...
};
```

Figure 5.8: Implementation details of the *pipeline* class.

execution of the pipeline. In this case, the user is responsible to set the particular pipeline configuration by calling the function `set_configuration()` (i.e. `pipe.set_configuration({1,1,1}, USE_MG)`), (see line 14 in Fig. 5.7). The adaptive configuration mode of the `run()` method (line 36) takes three arguments: the optimization criteria (THROUGHPUT, ENERGY, THROUGHPUT_ENERGY), a float number between 0 and 1 that represents the allowed overhead ratio (see section 5.3.1) and a third argument that allows the user to specify the throughput variation ratio, between 0 and 1, for which the pipeline must execute the training phase again.

The adaptive configuration `run()` method has 2 phases: the *training phase* and the *running phase* (see section 5.3.2). The training phase carries out three experiments. The first one executes one item on the GPU through all stages (line 39). The second one executes one item on the CPU with MG granularity (line 40) through all stages. Finally, the third experiment launches *nthreads* executions (from 1 to *nthreads*) on the CPU with CG granularity (line 43). This is explained with more details in section 5.3.2. Notice that the `run_training()` method is used here. Then, we compute our analytical model (line 45) with the time and energy measurements collected in the previous experiments. Thus, based on the desired optimization criteria, we predict a near optimal configuration. This analytical model returns the best configuration that maximises the optimization criteria passed as argument. This model is explained with deeper details in section 5.3.2.

Once the desired configuration is found, the pipeline enters in the second phase (running phase). In this phase, the `run_monitoring()` method (line 48) always monitors the throughput and energy. Whenever a change (drop/rise) in throughput is detect and the ratio is bigger than the variation argument, the pipeline checks the overhead parameter and the number of items processed in this phase. If the ratio between the total spent time by the previous training phase and the execution time of the current phase is less than the overhead threshold, then the current running phase is aborted and the training phase is executed again. Otherwise the algorithm continues in the running phase until the overhead ratio is less than the overhead threshold. More details about the computation of the overhead are covered in section 5.3.1.

5.2.3. Pipeline stage functions

An important part of the pipeline declaration and usage is the set up of the pipeline stage functionalities. In the API, the `add_stage()` method (Figure 5.7,

lines 9 to 11) is used to add each one of the stages while identifying their associated functions that may be called during pipeline execution to process the items.

```
1  /****************************************
2  * 3.- Functions definition example
3  ****************************************/
4  // Example for filter 3 of ViVid
5  void cg_f3(ViVidItem *item) // Coarse grain CPU version
6  {
7    float * out, cla, his;
8    out = item->out->getHostPtr(BUF_WRITE_ONLY); //buffer on host for WR
9    cla = item->cla->getHostPtr(BUF_READ_ONLY);  //buffer on host for RD
10   his = item->his->getHostPtr(BUF_READ_ONLY);  //buffer on host for RD
11   // do cpu things like  out[XXX] = his[XXX] + cla[XXX];
12 }
13 void mg_f3(ViVidItem *item) // Medium grain CPU version
14 {
15   float * out, cla, his;
16   out = item->out->getHostPtr(BUF_WRITE_ONLY); //buffer on host for WR
17   cla = item->cla->getHostPtr(BUF_READ_ONLY);  //buffer on host for RD
18   his = item->his->getHostPtr(BUF_READ_ONLY);  //buffer on host for RD
19
20     tbb::parallel_for(0, aheight), 1, [&] (size_t i) {
21        // do cpu things like  out[i] = his[i] + cla[i];
22     });
23
24   // #pragma omp parallel for
25   // for (size_t i=0; i<aheight; i++) {
26           // do cpu things like  out[i] = his[i] + cla[i];
27   // }
28 }
29 void gpu_f3(ViVidItem *item)    // GPU OpenCL version
30 {
31   cl_mem out,cla,his;
32   out = item->out->getDevicePtr(BUF_WRITE_ONLY); //buffer on device for WR
33   cla = item->cla->getDevicePtr(BUF_READ_ONLY);  //buffer on device for RD
34   his = item->his->getDevicePtr(BUF_READ_ONLY);  //buffer on device for RD
35
36   // Setting kernel parameters
37   // Launching kernel
38   //...
39 }
```

Figure 5.9: Functions for pipeline stages operations (CG, MG, GPU).

Figure 5.9 shows an example of these stage functions definition. The programmer can provide up to three different versions of the same function. The pipeline will use the appropriate version of the function to map the stage to one CPU core, several CPU cores, or the GPU device. Each function receives as argument a pointer to the item to be processed. From such item we can obtain the pointers to the input/output data buffers by using the method `getHostPtr()` to obtain a host pointer, or `getDevicePtr()` to obtain an OpenCL buffer object usable on the GPU device. In both cases, the access type for that buffer inside

the function must be set by the programmer (options are: `BUF_READ_ONLY`, `BUF_WRITE_ONLY`, `BUF_READ_WRITE`).

Figure 5.9 shows the definition of two functions that can be invoked on the CPU and a third one to process an item on the GPU. First, in line 5 we have the CPU function for CG granularity, that is basically a serial code to process one item on one CPU core. For this granularity, parallelism is exploited at the task level since several cores may be running this function at the same time for different items. Next in line 13, we have the definition of the function for MG granularity, where all the cores collaborate in processing a single item. Now, data parallelism is exploited, and to that end in this example, we rely on `tbb::parallel_for()` (line 20). MG granularity can also be exploited by using OpenMP as shown in commented line 24. Finally we have the GPU code defined in the `gpu_f3()` function (line 29). Note also that pipeline parallelism is exploited because concurrent items traverse the stages of a pipeline at their own pace.

Implementation details

As mentioned before, one of the key components of the `pipeline<T>` class is the `parallel_stage` class. One object of this class is allocated for each `add_stage()` invocation (see Figure 5.7 lines 9 to 11). This class holds important instance variables: three of them are function pointers (see Figure 5.10 lines 9-11) which point to the functions declared in Figure 5.9 (they are initialized in the class constructor in line 14). The other two instance variables, `runOnGPU` and `grain`, are used to decide whether one stage should execute on CPU or on GPU (at runtime) and in the former case, if the MG or CG version should be used to execute the stage on the CPU. The `operator()` function (line 22) is automatically invoked when an item reaches the stage. This functor first receives a pointer to the item that needs to be processed, so it can be passed down to the appropriate function. Then, it decides which function has to be called: if `runOnGPU` is true and the GPU is idle, the item is processed on the GPU (i.e. `gpuFunc()` is called). Otherwise, depending on the `grain` variable, `mgFunc()` or `cgFunc()` function is invoked.

5.2.4. Buffer class

As shown previously in Figure 5.6, thanks to our `DataBuffer<T>` template class, the programmer does not need to manage memory buffers explicitly. The supplied buffer class hides all data buffer management and the programmer just need to ask for the references to the buffers, indicating whether the buffers are read or/and write. Figure 5.11 shows an example of buffer declaration and access

```
1  /*************************************************
2  * Parallel Stage Internal Details
3  *************************************************/
4  template <class Item_T>
5  class parallel_stage : public tbb::filter{
6    //members
7    int runOnGPU;  //1 runs on GPU, 0 runs on CPU
8    bool grain;    //True is MG, False is CG
9    void (*cgFunc)(Item_T*);
10   void (*mgFunc)(Item_T*);
11   void (*gpuFunc)(Item_T*);
12
13   //Constructor
14   parallel_stage(void (*cg_f)(Item_T*), void (*mg_f)(Item_T*), void (*gpu_f
         )(Item_T*)){
15     cgFunc=cg_f; mgFunc=mg_f; gpuFunc=gpu_f;
16   }
17   //Methods
18   void setConfiguration(int mapping, bool granularity){
19     runOnGPU = mapping; grain = granularity;
20   }
21   ...
22   void * operator()(void * item){
23     Item_T *it = (Item_T *) item;
24     if(runOnGPU && h_pipeline::isGPUidle()){
25       gpuFunc(it);
26     }else if(grain){
27       mgFunc(it);
28     }else{
29       cgFunc(it);
30     }
31     return it;
32   }
33 };
```

Figure 5.10: Internal details of the *parallel_stage* class.

setting. In line 5 a data buffer is declared. In the next line a pointer, `*frame`, is declared to access the former data buffer from the CPU.

The `DataBuffer` class offers a way to set up the type of access to a certain OpenCL buffer, so a method to set it up must be used (line 9) and we can choose to use a Zero-Copy Buffer approach (line 10) or copy data from the CPU host to the GPU (and vice versa) when required.

On the creation of the buffer (line 9), we need to set the kind of access that this buffer will take from the OpenCL kernel (read or/and write). The actual allocation of the buffer (on host or device memory) is delayed until the first usage. To use the buffer, the programmer has to invoke the right method: `getHostPtr()` to obtain a host pointer, or `getDevicePtr()` to obtain a device memory object. In the example of Figure 5.11 (line 14), the host pointer,

```
/*************************************************
* 4.- BUFFER usage
*************************************************/
int main(int argc, char* argv[]){
  DataBuffer<float> * global_frame; // data buffer
  float *frame;                     // pointer to the buffer from CPU

 //Specify access mode for OpenCL kernel: BUF_READ  BUF_WRITE
     BUF_READ_WRITE
  global_frame = new DataBuffer<float>(size, BUF_READ);
  global_frame->set_ZCB(true);  //Set Zero Copy Buffer usage
  global_frame->use_Pool(true);  //Use a pool of buffers

  //acquire the buffer reference to write it on the CPU
  frame = global_frame->get_HOST_PTR(BUF_WRITE);
  frame[XXX] = XXX;   // fill the buffer on the CPU
  //Pipeline definition and usage
  ...
}
```

Figure 5.11: Usage example of the *DataBuffer* class.

`frame`, is initialized using `getHostPtr()` with write intent. In the next line, the buffer is initialized with the appropriate data.

5.3. Optimal pipeline configuration strategy

Our proposed framework is particularly suitable for streaming applications that may exhibit a variation in the streaming characteristics. In these cases, we can adjust the pipeline configuration to optimise the desired metric, raw throughput, energy, or a tradeoff.

Our framework is designed as a two phase engine: first, a *training phase* followed by a *running phase*. The training phase carries out two steps: 1) *a measurement collection step*, where we measure time and energy on CPU and GPU; and 2) *an evaluation step*, where our model (see section 5.3.2) finds the optimal pipeline configuration using the collected data by computing a few frames. During the training phase, a few frames are used, so no off-line training is necessary. Also, the runs to collect the measurements are conducted on one processor at a given time, either the CPU or GPU (homogeneous runs). Once the evaluation step finds the optimal configuration, the framework enters the running phase. In order to adapt to variations in the computation needs of the applications, throughput is monitored during running phase, so that any throughput change bigger than the variation ratio, which is given by the user, can return the frame-

work to the training phase. However, to limit the overhead of the the training phase, training is only performed when its associated overhead is less than a threshold value provided by the user (more details in subsection 5.3.1).

Let's assume that a streaming application has s parallel stages S_1, S_2, .., S_s and that all items are executed through all these stages. Additionally, we consider two more stages, Input and Output stages are serial (S_i, S_o), although this is not a pre-requisite in our model. Our model is based on a set of equations that allow us to estimate the throughput and energy consumption per item for all possible pipeline alternatives. Assume that our system consists of nC CPU cores and 1 on-chip GPU. Then, our framework invokes the model's equations for the $2^s \cdot (nC + 2)$ possible pipeline configurations, and for each one computes the effective throughput, λ_e, and the effective energy per item, E_e. From the estimations, it selects the pipeline configuration for which the optimal is found: highest λ_e or lowest E_e, depending on the metric considered. We can also use any combination of these metrics to define a tradeoff metric and look for the configuration which obtains the optimal value.

5.3.1. Measurement Collection step

As mentioned, the equations of our model use the collected data recorded in the measurement collection step. In this step, we carry out $nC + 3$ experiments to obtain all the values needed by the model. Moreover, for each experiment, we use a few input items (from 1 to 5) to compute average time and energy per item.

Note that this number of experiments is usually much smaller than the $2^s \cdot (nC+2)$ possible alternatives. Thanks to the model we do not need to experimentally assess all of them. For time measurements, we use the clock ticks hardware counter, while for the energy measurements, we use the energy counters available on the Ivy Bridge and the Haswell architectures [26, 28]. Let's remind that these counters measure three domains: P (or total), C (or $PP0$) and G (or $PP1$). P or Package means the consumption of the whole chip, including CPU, GPU, memory hierarchy and auxiliary units. C is CPU cores domain and G is the GPU domain. In our model we consider C, G and $U = P - C - G$. Therefore, this last component represents the Uncore energy consumption. For other architectures, energy information can be estimated by either relying on performance counters that can be read by using a library, such as PAPI [74], or by using a power sensor, like the Texas Instruments INA231 power monitor integrated with the Exynos 5 on the Odroid XU3 platform [47]. Current trends point out that energy counters will be widely available in the near future. Anyway, even if energy information

is not accessible, our framework is still useful to minimise execution time.

The experiments and measurements we collect are always from homogeneous runs (only GPU or CPU execution). These experiments are the following:

- CG experiments: we perform 1 experiment in which all stages are executed by one thread in one CPU core. We collect time and energy per stage (see T_k^{CG} and $\left(E_{C_k}^{CG}, E_{G_k}^{CG}, E_{U_k}^{CG}\right)$, $k = 1 : s$, in Table 5.1). For energy measurements we collect three components (C, G, U) as previously explained. Next, we carry out nC additional experiments in the CPU multicore: on each one, n threads (with n changing from 2 to $nC+1$[2]) process n items (each thread processes one item) throughout all the pipeline stages, i.e. homogeneous CG executions. We collect the total time and energy per item (see $T^{CG}(n)$ and $\left(E_C^{CG}(n), E_G^{CG}(n), E_U^{CG}\right)$, $n = 2 : nC + 1$, Table 5.1). Note that the case for one thread was already considered in the first experiment described above. Actually, $T^{CG}(1) = \sum_{k=1}^{s} T_k^{CG}$ and $E_*^{CG}(1) = \sum_{k=1}^{s} E_{*_k}^{CG}$, where $*$ takes the value C, G and U. With these measurements we implicitly incorporate to the model the impact that n threads processing n items have in the memory system as well as the scalability behaviour in the CPU. To carry out these $nC + 1$ experiments, $(nC + 2) \cdot (nC + 1)/2$ items of the stream are processed.
- MG experiments: we conduct 2 additional experiments in which all stages are executed first by one thread on the GPU, and next by nC threads on the CPU multicore, i.e. homogeneous MG execution, where nC is the number of CPU cores. Again, we collect time and energy per stage (see T_k^G and $\left(E_{C_k}^G, E_{G_k}^G, E_{U_k}^G\right)$, $k = 1 : s$, for the GPU and T_k^{MG} and $\left(E_{C_k}^{MG}, E_{G_k}^{MG}, E_{U_k}^{MG}\right)$, $k = 1 : s$, for MG on the CPU, in Table 5.2). Now, 2 additional items of the stream are processed to carry out these 2 MG experiments.

In Tables 5.1 and 5.2, the third column, Time Col., represents the time to collect the corresponding parameters in the same row. For example, to collect $T^{CG}(1), \ldots, T^{CG}(s)$ and $\left(E_{C_k}^{CG}, E_{G_k}^{CG}, E_{U_k}^{CG}\right)$ $k = 1 : s$, we need t^{CG} time. This time is required to compute the total overhead of the collection step.

Note that in practice, $T^{CG}(1) = \sum_{k=1}^{s} T_k^{CG}$, $E_*^{CG}(1) = \sum_{k=1}^{s} E_{*_k}^{CG}$, where $*$ takes the value C, G and U, and $t^{CG}(1) = t^{CG}$.

Although not reported in Table 5.2, when we collect the GPU execution data, T_1^G, ..., T_s^G, and energy measurements, $\left(E_{C_1}^G, \ldots, E_{U_1}^G\right)$, ..., $\left(E_{C_s}^G, \ldots, E_{U_s}^G\right)$, we

[2]We increase the number of threads until $nC + 1$ in order to simulate the interference from the GPU host thread.

Table 5.1: Collected data (Measured time and energy per item) for CG experiments. This third column has the times needed to collect the corresponding parameters.

Parameter	Device	Time col.	Description
$T_1^{CG}, \ldots, T_s^{CG}$	CPU	t^{CG}	time per item (and stage) on the CG exec. (1 thread)
$\left(E_{C_1}^{CG}, E_{G_1}^{CG}, E_{U_1}^{CG}\right)$ $\ldots$ $\left(E_{C_s}^{CG}, E_{G_s}^{CG}, E_{U_s}^{CG}\right)$	CPU		(C,G,U) components of the energy per item (and stage) on the CG exec. (1 thread)
$T^{CG}(1), \ldots, T^{CG}(n_m)$	CPU	$t^{CG}(1)$ $\ldots$ $t^{CG}(n_m)$	total time per item on the CG exec. (1, 2, $\ldots$, $n_m = nC + 1$ threads)
$(E_C^{CG}(1), E_G^{CG}(1), E_U^{CG}(1))$ $\ldots$ $(E_C^{CG}(n_m), E_G^{CG}(n_m), E_U^{CG}(n_m))$	CPU		(C,G,U) comp. of the total energy per item on the CG exec. (1, 2, $\ldots$, $n_m = nC + 1$)

Table 5.2: Measured time per item, T, and energy per item, E, and per stage for GPU and for MG. Also time to collect them.

Parameter	Device	Time col.	Description
$T_1^G, \ldots, T_s^G$	GPU	t^G	time per item (and stage) on the GPU exec. (1 thread)
$\left(E_{C_1}^G, E_{G_1}^G, E_{U_1}^G\right)$ $\ldots$ $\left(E_{C_s}^G, E_{G_s}^G, E_{U_s}^G\right)$	GPU		(C,G,U) components of the energy per item (and stage) on the GPU exec.
$T_1^{MG}, \ldots, T_s^{MG}$	CPU	t^{MG}	time per item (and stage) on the MG exec. (nC threads)
$\left(E_{C_1}^{MG}, E_{G_1}^{MG}, E_{U_1}^{MG}\right)$ $\ldots$ $\left(E_{C_s}^{MG}, E_{G_s}^{MG}, E_{U_s}^{MG}\right)$	CPU		(C,G,U) components of the energy per item (and stage) on the MG exec.

also include in the measurement for each stage the time and energy due to the host-to-device and the device-to-host data transfers. These transfer times are negligible for the applications and the on-Chip GPUs used in the experiments, as they communicate through the Last Level Cache (LLC).

Notice that the model can also be used when hyperthreading is enabled. To consider the use of hyperthreading, the collecting measurement step needs to run $nC * 2 + 3$ (instead of $nC + 3$) experiments. Hyperthreading was not beneficial for our applications and we did not consider it in our experimental results.

Controlling the overhead

The cost of the training phase is mainly due to the measurement collection step, where items are computed in an inefficiently way, because of the homogeneous runs (only one device is used at a time) carried out during this step. After the measurement collection step and the subsequent model instantiation, we can control the time when a new training can be performed to guarantee that the overhead due to the training is kept under certain threshold.

If after the training phase and the subsequent model instantiation, no change of configuration is recommended, this training phase completely translates into overhead. To limit this training overhead in the worst case, we can control how frequently our scheduler enters this phase. Suppose that after performing the training step, λ_c is the throughput of the current configuration and that $N_s = (nC + 2) \cdot (nC + 1)/2 + 2$ is the number of items processed during the measurement collection step (see Tables 5.1 and 5.2). Then, Δt can be defined as the time penalty due to the training. It is computed as the time needed to carry out the collection step minus the time it takes to compute N_s items with the current λ_c throughput:

$$\Delta t = \left(t^{CG} + \left(\sum_{n=2}^{nC+1} t^{CG}(n) \right) + t^{MG} + t^{G} \right) - N_s/\lambda_c \tag{5.1}$$

The *overhead* of the last training with respect to the current throughput can be computed as,

$$ov = \frac{\Delta t}{t + \Delta t} \tag{5.2}$$

We can keep this overhead below a threshold value, ov_{thl}, if $\Delta t/(\Delta t + t) < ov_{thl}$, or in other words:

$$t > \frac{(1 - ov_{thl})}{ov_{thl}} \cdot \Delta t \tag{5.3}$$

This expression gives us a lower bound to control the time t between training phases that guarantees that the overhead of training is less than a certain threshold. Notice that equation 5.3 is not based on the real overhead of the training phase, as computing this overhead would require a) the throughput before training, b) the time to perform the training, and c) the throughput after training.

However, b) and c) are only known after the training is performed. Thus, equation 5.3 is based on the overhead incurred by the last training that is processing items at a throughput lower than that of the current configuration.

For the ViVid application on the Ivy Bridge chip presented in section 5.4, 5% of overhead is paid when the training takes place every 3.7 sec. for low resolution input video. Thus, the training overhead can be amortized after processing a few items with the new recommended pipeline configuration, as we discuss in section 5.4. The result 3.7 seconds is obtained by substituting in equation 5.3, $ovthl = 0.05$, and $\Delta t = 0.46 + 17/65 = 0.198$, this is, 0.46 seconds is the time needed to collect the measurements using 17 items ((4+2)(4+1)/2+2) and the current throughput in LD is 65 fps. Δt is a positive number because the training time is larger than the time to process the 17 items at the current throughput.

5.3.2. Model for finding the optimal pipeline configuration

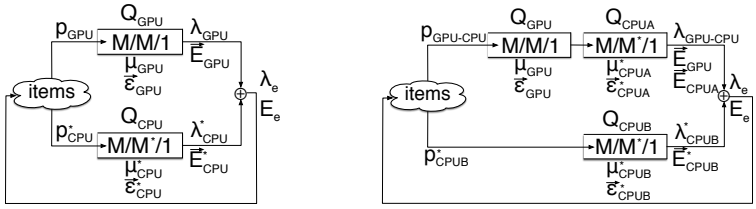
We model the heterogeneous pipeline configurations as a closed network of logical queues where items arrive following a *Poisson process* [44]. This is pertinent in the context of streaming applications [79] where item arrivals can be considered independent and inter-arrival time can be viewed as following an exponential distribution. In these closed systems, items can be viewed as circulating continuously and never leaving the network of queues, because a new item can not enter until a previous one leaves. Figure 5.12 shows our models for the Decoupled and Coupled configurations, where we can see that an item can follow one of two alternative paths before entering again in the system. In our models, we can find one or more queues on each path. In particular, any sequence of consecutive stages mapped to one device (the GPU or the CPU) is represented as a M/M/1 queue. This stands for a logical queue where a single server serves items that arrive according to a Poisson process and have exponentially distributed service times. Although there may be several concurrent threads on a device processing the sequence of stages represented by the queue, we have found that assuming one logical server on each queue still provides accuracy while keeping the equations of the model simple. In a closed network of queues, the following expressions define the *flow balance conditions* [8] at equilibrium,

$$\lambda_e = \sum_{path_j} \lambda_j \tag{5.4}$$

$$\sum_{path_j} p_j = 1 \tag{5.5}$$

$$p_j \cdot \lambda_e = \lambda_j \tag{5.6}$$

These equations allow us to relate the relative throughput of a path in a configuration with the effective throughput in that configuration. A path $path_j$ refers to one of the two possible paths defined in Section 5.1.1 for each configuration: for DP configurations, it is either the GPU path or the CPU path (note the subindices for the parameters on each path of the model (GPU, CPU) in Figure 5.12a); For CP configurations, it is either the GPU-CPU path or the CPU path (note the sub-indices for the parameters on each path of the model (GPU-CPU, CPUB) in Figure 5.12b). In particular, equation 5.4 establishes that given the relative throughputs of the paths in the system, λ_j, then the effective throughput, λ_e, may be obtained as a sum (i.e. combining independent Poisson processes leads to a Poisson process). Equation 5.5 states that splitting a Poisson process probabilistically leads to Poisson processes, being p_j the probability of taking $path_j$. Equation 5.6 states that, in a M/M/1 queue at equilibrium, the average flow rate leaving the queue will also be the same as the average flow rate entering the queue.



(a) Model for DP configurations

(b) Model for CP configurations

Figure 5.12: Closed network of queues.

We define two parameters for each queue Q_i: the *service rate*, μ_i, or average rate at which an item is processed, and the *energy rate*, $\overrightarrow{\varepsilon}_i$, or average energy consumed by an item in the corresponding device (GPU or CPU) where the queue works. This last parameter represents a vector with three components, one for each energy domain: $(\varepsilon_{i_C}, \varepsilon_{i_G}, \varepsilon_{i_U})$. They can be seen as the components of the average energy consumed by an item on a device due to the stages represented by the queue Q_i, when the device is the only one working in the system (homogeneous execution).

In any case, as the network is in equilibrium, each individual queue must be in equilibrium. This means that the utilization on the queue, ρ_i, is less than 100%, that is, the ratio between the relative throughput of the corresponding path, λ_j, and the queue's service rate, μ_i, is at most 1 [44].

$$\rho_i = \frac{\lambda_j}{\mu_i} \leq 1 \tag{5.7}$$

Regarding the energy, as each individual queue Q_i is in equilibrium, we assume that the energy utilization on the corresponding device, $\rho_i^{\overrightarrow{E}}$, is proportional to the probability of items serviced on the corresponding queue, p_j, or in other words:

$$\rho_i^{\overrightarrow{E}} = p_j \leq 1, \quad \overrightarrow{E}_i = \rho_i^{\overrightarrow{E}} \cdot \overrightarrow{\varepsilon}_i \tag{5.8}$$

This expression allows us to estimate the *relative energy per item* consumed by queue Q_i on the corresponding device (GPU or CPU), $\overrightarrow{E}_i$. This parameter is also a vector that consists of three components: $(E_{i_C}, E_{i_G}, E_{i_U})$. In the case there are several logical queues mapped on a device, from Q_1 to Q_d, then the relative energy per item consumed by the queues in the device is the sum of the relative energy per item for all the queues working in the device: $\sum_{i=1}^{d} \overrightarrow{E}_i = \left(\sum_{i=1}^{d} E_{i_C}, \sum_{i=1}^{d} E_{i_G}, \sum_{i=1}^{d} E_{i_U}\right)$. These components can be seen as the components of the energy consumed by the items that a device processes when the device is the only one working in the system (homogeneous execution).

However, the effective energy consumed by the GPU and CPU when both devices are working in the system (heterogeneous execution), is not the sum of the relative energies of the queues on each device. Let's define the *effective energy per item* consumed in the system, E_e, as the sum of three components:

$$E_e = min\left(\frac{TDP}{\lambda_e}, E_{e_C} + E_{e_G} + E_{e_U}\right) \tag{5.9}$$

where TDP is the power budget of the chip, λ_e the effective throughput, and $E_{e_C} + E_{e_G} + E_{e_U}$ the effective energy consumed when the TDP is not reached. For the heterogeneous chips studied, we have found that in case the TDP is not reached, then each component of the effective energy is given by the dominant component of the relative energy computed for each device. This is what we call the *energy balance condition*. Let's suppose that the relative energy per item for all the queues in the GPU device is given by $\overrightarrow{E}_{GPU} = (E_{GPU_C}, E_{GPU_G}, E_{GPU_U})$, and the relative energy per item for all the queues in the CPU device is given by $\overrightarrow{E}_{CPU} = (E_{CPU_C}, E_{CPU_G}, E_{CPU_U})$. The rationale for the energy balance condition is that the C-component of the effective energy is typically dominated by the C-component of the relative energy of the CPU device, E_{CPU_C}, while the C-component of the GPU device, E_{GPU_C} is just a "residual" or standby consumption when the CPU is idle. Remember that this last C-component of

the relative energy of the GPU device is obtained with homogeneous runs (runs on the GPU where the CPU is idle) during the measurement collection step. On an heterogeneous run, however, the CPU is not idle, and so the standby consumption measured during the homogeneous run is already included in the C-component of the relative energy of the CPU, E_{CPU_C}, and does not need to be included again. A similar argument can be made for the G-component of the effective energy. With respect to the U-component, we have observed that the effective energy consumed is determined by the device (CPU or GPU) that processes a higher load.

Next sections explain how we model *Decoupled* and *Coupled* configurations, respectively, and how we incorporate the granularity to the models.

Model for Decoupled pipeline configurations

These configurations are shown in Figures 5.2a and 5.2c (DP-CG and DP-MG, respectively). Figure 5.12a depicts our model for them. As explained in section 5.1.1, in these configurations there is a GPU path in which a thread processes an item through all stages in the GPU, and also there is a CPU path in which n concurrent threads process other item/s through all the stages in the CPU device.

The GPU device is modelled with Q_{GPU} which is the M/M/1 queue that serves all the stages for the items that go through the GPU path. This queue is characterized with two parameters: μ_{GPU}, the service rate of the GPU, and $\overrightarrow{\varepsilon}_{GPU}$, the energy rate consumed by the queue in the GPU device. The latter parameter represents a vector with the three components of the energy per item rate consumed in the GPU: $(\varepsilon_{GPU_C}, \varepsilon_{GPU_G}, \varepsilon_{GPU_U})$. These parameters are computed from the time and energy measurements taken in the collection step, as we show in Table 5.3. For both parameters we consider the time and the energy per item of all the stages S_k that are mapped to the GPU (k from 1 to s, see Table 5.2).

The CPU device is modelled with Q_{CPU} which is the M/M*/1 queue that serves all the stages for the items that go through the CPU path. The * stands for the different instantiations of the queue, depending on the granularity exploited. For the CG granularity, the queue is characterized with two parameters: $\mu^{CG}_{CPU}(n)$, the service rate of the CPU under CG granularity, and $\overrightarrow{\varepsilon}^{CG}_{CPU}(n)$, the energy rate consumed by the queue in the CPU device under CG granularity. Note that under the CG granularity the CPU device can run from 1 to $nC + 1$ concurrent threads. The $n = 1$ case represents in fact the GPU homogeneous

execution, while the $n = nC + 1$ represents the maximum number of threads in the CPU path. Therefore, for the CG granularity, both the service rate and the energy rate are computed for each possible number of threads. Table 5.3 shows how these parameters are computed, where we see that time and energy are taken from the measurements in Table 5.1. Regarding the MG granularity, the queue is defined by μ^{MG}_{CPU} and $\overrightarrow{\varepsilon}^{MG}_{CPU}$. In Table 5.3 we show these parameters, where we notice that time and energy are taken from the measurements in Table 5.2.

Since our queues are in equilibrium, and we assume maximum utilization on each queue, by applying equation 5.7 we get $\rho_{GPU} = 1$ and $\rho^*_{CPU} = 1$. From this assumption, we find that the relative throughput for each path is given by the corresponding queue's service rate, that is, $\lambda_{GPU} = \mu_{GPU}$ and $\lambda^*_{CPU} = \mu^*_{CPU}$. Also, the flow balance conditions at equilibrium (equations 5.4-5.6) allow us to compute the effective throughput of the system, $\lambda_e = \lambda_{GPU} + \lambda^*_{CPU}$, and the probability that an item goes through the GPU path, $p_{GPU} = \lambda_{GPU}/\lambda_e$, or the probability that it goes through the CPU path, $p^*_{CPU} = \lambda^*_{CPU}/\lambda_e$.

On the other hand, by applying equation 5.8, we get that the energy utilization of each queue on the corresponding device is proportional to the probability of items serviced on the queue, or in other words, $\rho_{\overrightarrow{E}_{GPU}} = p_{GPU}$ and $\rho_{\overrightarrow{E}^*_{CPU}} = p^*_{CPU}$. This assumption allows us to estimate the relative energy per item consumed by Q_{GPU} in the GPU device, $\overrightarrow{E}_{GPU} = p_{GPU} \cdot \overrightarrow{\varepsilon}_{GPU}$ and by Q_{CPU} in the CPU device, $\overrightarrow{E}^*_{CPU} = p^*_{CPU} \cdot \overrightarrow{\varepsilon}^*_{CPU}$ (for CG or MG granularities), respectively.

The effective energy per item consumed in the system, E_e, can be computed as the minimum of TDP/λ_e and the sum of three components, as defined in equation 5.9. If the TDP is not reached, then each component can be computed by the energy balance condition that establishes that each component of the effective energy is given by the dominant component of the relative energy computed for each device. In particular, this condition in the DP-* configurations means:

$$(E_{e_C}, E_{e_G}, E_{e_U}) = max\left(\overrightarrow{E}_{GPU}, \overrightarrow{E}^*_{CPU}\right) = \tag{5.10}$$
$$\left(max(E_{GPU_C}, E^*_{CPU_C}), max(E_{GPU_G}, E^*_{CPU_G}), max(E_{GPU_U}, E^*_{CPU_U})\right)$$

Model for Coupled pipeline configurations

These configurations are shown in Figs. 5.2b and 5.2d (CP-CG and CP-MG, respectively). Figure 5.12b depicts our model for them. In these configurations there is a GPU-CPU path in which a thread processes a item through some stages on the GPU and other stages on a CPU core, and there can also be a

Table 5.3: Set of Parameters for DP-CG and DP-MG configurations (* stands for both).

Parameter	**Device / Gr.**	**Value**	**Description**
μ_{GPU}	GPU	$\frac{1}{\sum_{k=1}^{s} T_k^G}$	service rate for the stages mapped to the GPU
$\overrightarrow{\varepsilon}_{GPU}$	GPU	$\left(\sum_{k=1}^{s} E_{C_k}^G, \sum_{k=1}^{s} E_{G_k}^G, \sum_{k=1}^{s} E_{U_k}^G\right)$	energy rate consumed by the stages mapped to the GPU
λ_{GPU}	GPU	μ_{GPU}	relative throughput of the GPU path
$\overrightarrow{E}_{GPU}$	GPU	$p_{GPU} \cdot \overrightarrow{\varepsilon}_{GPU}$	relative energy per item consumed by Q_{GPU}
$\mu_{CPU}^{CG}(n)$	CPU / CG	$\frac{1}{T^{CG}(n)}, \ n = 1 : nC$	service rate for the stages mapped to the CPU under CG and n threads
$\overrightarrow{\varepsilon}_{CPU}^{CG}(n)$	CPU / CG	$\left(E_C^{CG}(n), E_G^{CG}(n), E_U^{CG}(n)\right), n = 1 : nC$	energy rate consumed by the stages mapped to the CPU under CG and n threads
μ_{CPU}^{MG}	CPU / MG	$\frac{1}{\sum_{k=1}^{s} T_k^{MG}}$	service rate for the stages mapped to the CPU under MG
$\overrightarrow{\varepsilon}_{CPU}^{MG}$	CPU / MG	$\left(\sum_{k=1}^{s} E_{C_k}^{MG}, \sum_{k=1}^{s} E_{G_k}^{MG}, \sum_{k=1}^{s} E_{U_k}^{MG}\right)$	energy rate consumed by the stages mapped to the CPU under MG
λ_{CPU}^{*}	CPU / *	μ_{CPU}^{*}	relative throughput of the CPU path
$\overrightarrow{E}_{CPU}^{*}$	CPU / *	$p_{CPU}^{*} \cdot \overrightarrow{\varepsilon}_{CPU}^{*}$	relative energy per item consumed by Q_{CPU}
λ_e	GPU + CPU	$\lambda_{GPU} + \lambda_{CPU}^{*}$	effective throughput of the system
E_e	GPU + CPU	$min\left(\frac{TDP}{\lambda_e}, E_{e_C} + E_{e_G} + E_{e_U}\right)$	effective energy per item consumed in the system. See eq. 5.11

CPU path in which other concurrent threads process items through all the stages on the remaining CPU cores. To model the service provided by a sequence of stages mapped to each device on each path, we use a logical queue. Thus, in the GPU-CPU path we can find at least a Q_{GPU} which is the M/M/1 queue that represents the sequence of consecutive stages that service an item on the GPU device, and at least a Q_{CPUA} which is a M/M*/1 queue that represents the rest of stages that serve the item on the CPU device (* stands for the granularity studied). For simplicity, in the figure we have represented the case in which the item is first processed by some consecutive stages on the GPU, and later by the rest of stages on the CPU. In case of a mapping where the item is first processed by consecutive stages mapped to the CPU, then to the GPU, then to the CPU, and so on, the model would include more logical queues in the GPU-CPU path: first a Q_{CPUA}, followed by a Q_{GPU}, then another Q_{CPUA}, and so on.

Each Q_{GPU} queue is characterized with two parameters: μ_{GPU}, the service rate due to the consecutive stages mapped to the GPU, and $\overrightarrow{\varepsilon}_{GPU}$, the energy rate consumed by those stages in the GPU device. These parameters are computed from the time and energy measurements taken in the collecting step, as we show in Tables 5.4 and 5.5. For both parameters, we just consider the time and the energy per item of the corresponding consecutive stages S_k that are mapped to the GPU ($S_k \in Q_{GPU}$). Also, each Q_{CPUA} queue is characterized with two parameters, depending on the granularity. For the CG granularity, the parameters are: μ^{CG}_{CPUA}, the service rate due to the consecutive stages mapped to the CPU under CG granularity, and $\overrightarrow{\varepsilon}^{CG}_{CPUA}$, the energy rate consumed by those stages in the CPU device under CG granularity. Tables 5.4 and 5.5 show how these parameters are computed, where time and energy come from measurements in Table 5.1. Regarding the MG granularity, the Q_{CPUA} queue is defined by μ^{MG}_{CPUA} and $\overrightarrow{\varepsilon}^{MG}_{CPUA}$. In Tables 5.4 and 5.5 we show these parameters, where we notice that time and energy are taken from measurements in Table 5.2.

On the other hand, the stages mapped to the CPU in the CPU path, are modelled with Q_{CPUB} which is a M/M*/1 queue. Again, * stands for the different instantiations of the queue, depending on the granularity. For the CG granularity, the queue is characterized with: $\mu^{CG}_{CPUB}(n)$, the service rate of the CPU under CG granularity, and $\overrightarrow{\varepsilon}^{CG}_{CPUB}(n)$, the energy rate consumed by the queue in the CPU device under CG granularity. With CG, the CPU can run from 1 to $nC+1$ concurrent threads, in addition to the coupled GPU-CPU thread that serves the GPU-CPU path. Therefore, for CG, the service rate is computed taking into account this additional coupled thread and we model it assuming that the GPU-CPU thread is interfering with the threads that are working concurrently on the CPU. We model this interference by subtracting to the service rate of $n+1$

concurrent threads running in the CPU (because the CPU consists of Q_{CPUA} and Q_{CPUB}), a virtual service rate of 1 thread that is executing in the GPU-CPU path (Q_{CPUA}, the coupled thread). The energy rate is computed for the n concurrent threads working on the queue. In any case, for CG, both the service rate and the energy rate are computed for each number of threads. Tables 5.4 and 5.5 show how these parameters are computed, where time and energy come from measurements in Table 5.1. Regarding the MG granularity, the queue is defined by μ^{MG}_{CPUB} and $\overrightarrow{\varepsilon}^{MG}_{CPUB}$. Under this granularity, all the CPU threads will be serving the Q_{CPUA}. Therefore, we assume that Q_{CPUB} will have a very low probability of serving new items, and so, $\mu^{MG}_{CPUB} = 0$ and $\overrightarrow{\varepsilon}^{MG}_{CPUB} = 0$.

In this configuration, we assume optimistic utilization on each queue. By applying equation 5.7 we get $\rho_{GPU} \leq 1$, $\rho^*_{CPUA} \leq 1$ and $\rho^*_{CPUB} \leq 1$. From these expressions we find that a solution for the relative throughput for each path is given by, $\lambda_{GPU-CPU} = min(\mu_{GPU}, \mu^*_{CPUA})$ and $\lambda^*_{CPUB} = \mu^*_{CPUB}$. In general, if there were more logical queues in the GPU-CPU path, then a solution for $\lambda_{GPU-CPU}$ could be the minimum of the corresponding service rates in the path. Again, the flow balance conditions at equilibrium (equations 5.4-5.6) lead to computing the effective throughput of the system as $\lambda_e = \lambda_{GPU-CPU} + \lambda^*_{CPUB}$, and the probability that an item goes through the GPU-CPU path as $p_{GPU-CPU} = \lambda_{GPU-CPU}/\lambda_e$, or through the CPU path as $p^*_{CPUB} = \lambda^*_{CPUB}/\lambda_e$.

Similar to the DP-* configurations, we assume that the energy utilization of each queue on each device is proportional to the probability of items serviced on the corresponding queue, as defined in equation 5.8. This means $\rho^{\overrightarrow{E}}_{GPU} = p_{GPU-CPU}$, $\rho^{\overrightarrow{E}^*}_{CPUA} = p_{GPU-CPU}$ and $\rho^{\overrightarrow{E}^*}_{CPUB} = p^*_{CPUB}$. These expressions allow us to estimate the relative energy per item consumed on the GPU device, $\overrightarrow{E}_{GPU} = p_{GPU-CPU} \cdot \overrightarrow{\varepsilon}_{GPU}$ and on the CPU device, $\overrightarrow{E}^*_{CPUA} + \overrightarrow{E}^*_{CPUB}$ $= p_{GPU-CPU} \cdot \overrightarrow{\varepsilon}^*_{CPUA} + p^*_{CPUB} \cdot \overrightarrow{\varepsilon}^*_{CPUB}$ (for CG or MG granularities), respectively. As we see, in the CP-* configurations we estimate the relative energy per item consumed in the CPU from the activity in Q_{CPUA} and in Q_{CPUB}. In general, if there were more logical queues in the GPU-CPU path, then all the resultant $\overrightarrow{E}_{GPU}$ for the different Q_{GPU} should be added to estimate the relative energy per item consumed in the GPU device. Similarly, the $\overrightarrow{E}_{CPUA}$ terms should be added to estimate the relative energy per item consumed in the CPU in that path. Finally, as in DP-* configurations, the effective energy consumed in the system, E_e, can be computed as the minimum of TDP/λ_e and the sum of three components (see eq. 5.9). If the TDP is not reached, then following the energy balance condition we get for CP-* configurations that,

Table 5.4: Set of Parameters of the CP-CG and CP-MG configurations (* stands for both).

Param.	**Dev. / Gr.**	**Value**	**Description**
μ_{GPU}	GPU	$\frac{1}{\sum_{S_k \in Q_{GPU}} T_k^G}$	service rate of stages mapped to Q_{GPU} in the GPU-CPU path
$\overrightarrow{\varepsilon}_{GPU}$	GPU	$\left(\sum_{S_k \in Q_{GPU}} E_{C_k}^G, \sum_{S_k \in Q_{GPU}} E_{G_k}^G, \sum_{S_k \in Q_{GPU}} E_{U_k}^G\right)$	energy rate consumed by stages mapped to Q_{GPU} in the GPU-CPU path
μ_{CPUA}^{CG}	CPU / CG	$\frac{1}{\sum_{S_k \in Q_{CPUA}} T_k^{CG}}$	service rate of stages mapped to Q_{CPUA} in the GPU-CPU path under CG
$\overrightarrow{\varepsilon}_{CPUA}^{CG}$	CPU / CG	$\left(\sum_{S_k \in Q_{CPUA}} E_{C_k}^{CG}, \sum_{S_k \in Q_{CPUA}} E_{G_k}^{CG}, \sum_{S_k \in Q_{CPUA}} E_{U_k}^{CG}\right)$	energy rate consumed by stages mapped to Q_{CPUA} in the GPU-CPU path under CG
μ_{CPUA}^{MG}	CPU / MG	$\frac{1}{\sum_{S_k \in Q_{CPUA}} T_k^{MG}}$	service rate of stages mapped to Q_{CPUA} in the GPU-CPU path under MG
$\overrightarrow{\varepsilon}_{CPUA}^{MG}$	CPU / MG	$\left(\sum_{S_k \in Q_{CPUA}} E_{C_k}^{MG}, \sum_{S_k \in Q_{CPUA}} E_{G_k}^{MG}, \sum_{S_k \in Q_{CPUA}} E_{U_k}^{MG}\right)$	energy rate consumed by stages mapped to Q_{CPUA} in the GPU-CPU path under MG
$\lambda_{GPU-CPU}^{*}$	GPU-CPU / *	$\min(\mu_{GPU}, \mu_{CPUA}^{*})$	relative throughput of the GPU-CPU path
$\overrightarrow{E}_{GPU}$	GPU	$p_{GPU-CPU} \cdot \overrightarrow{\varepsilon}_{GPU}$	relative energy per item consumed by stages mapped to the Q_{GPU} in the GPU-CPU path
$\overrightarrow{E}_{CPUA}^{*}$	CPU / *	$p_{GPU-CPU} \cdot \overrightarrow{\varepsilon}_{CPUA}^{*}$	relative energy per item consumed by stages mapped to Q_{CPUA} in the GPU-CPU path

Table 5.5: Set of Parameters of the CP-CG and CP-MG configurations (II).

Parameter	**Device / Gr.**	**Value**	**Description**
$\mu_{CPUB}^{CG}(n)$	CPU / CG	$\frac{1}{T^{CG}(n+1)} - \frac{1}{T^{CG}(1)}, \;\; n = 0 : nC$	service rate for the stages mapped to Q_{CPUB} in the CPU path under CG and n threads
$\overrightarrow{\varepsilon}_{CPUB}^{CG}(n)$	CPU / CG	$\left(E_C^{CG}(n), E_G^{CG}(n), E_U^{CG}(n)\right), n = 0 : nC$	energy rate consumed by the stages mapped in CPU path under CG and n threads
μ_{CPUB}^{MG}	CPU / MG	0	service rate for the stages mapped to Q_{CPUB} in the CPU path under MG
$\overrightarrow{\varepsilon}_{CPUB}^{MG}$	CPU / MG	0	energy per item rate consumed by the stages mapped in the CPU path under MG
λ_{CPUB}^{*}	CPU / *	μ_{CPUB}^{*}	relative throughput of the CPU path
$\overrightarrow{E}_{CPUB}^{*}$	CPU / *	$p_{CPUB}^{*} \cdot \overrightarrow{\varepsilon}_{CPUB}^{*}$	relative energy per item consumed by the stages mapped in the CPU path
λ_e	GPU + CPU	$\lambda_{GPU-CPU} + \lambda_{CPUB}^{*}$	effective throughput of the system
E_e	GPU + CPU	$min\left(\frac{TDP}{\lambda_e}, E_{e_C} + E_{e_G} + E_{e_U}\right)$	effective energy per item consumed in the system. See eq. 5.11

$$
\begin{aligned}
(E_{e_C}, E_{e_G}, E_{e_U}) &= max\left(\overrightarrow{E}_{GPU}, \overrightarrow{E}^*_{CPUA} + \overrightarrow{E}^*_{CPUB}\right) \\
&= \begin{bmatrix} max(E_{GPU_C}, E^*_{CPUA_C} + E^*_{CPUB_C}) \\ max(E_{GPU_G}, E^*_{CPUA_G} + E^*_{CPUB_G}) \\ max(E_{GPU_U}, E^*_{CPUA_U} + E^*_{CPUB_U}) \end{bmatrix}
\end{aligned} \tag{5.11}
$$

In our estimations, the transference time/energy between GPU and CPU devices are not explicitly included because they were negligible for the applications studied.

Model extensions

Notice that in our throughput and energy estimations, the transfer time and energy between GPU and CPU devices have not been explicitly stated for the sake of readability. However, they can be easily incorporated into our models: the measurement collection step can collect the host-to-device and device-to-host time and energy for each stage mapped to the GPU during the GPU homogeneous run[3]. Then, the service rate of each Q_{GPU} queue, μ_{GPU}, would need to add the host-to-device time of the first stage and the device-to-host time of the last consecutive stage mapped to the corresponding Q_{GPU}. Similarly, the energy rate of the queue $\overrightarrow{\varepsilon}_{GPU}$, would also consider the energy consumed during the transfers on those stages. In the integrated GPUs that we use in our experiments and for our benchmarks, the transfer times are negligible and can be ignored without affecting the accuracy of the model. For discrete GPUs, we expect transfer times to have a higher impact, though.

Our model can also be extended to include the alternatives not considered in section 5.1.1. For instance, the splitting of an item on each stage would be modelled with a GPU-CPU path, where each stage i would be represented by a Q_{CPUi} and a Q_{GPUi} queue, and the service rates of the corresponding queues should be the time to process the portion of the item in the corresponding device and stage (similarly for the energy rate). The model for the alternative in which one stage can have items exploiting both MG and CG granularities in the CPU multicore can be modelled as two independent paths, with a CPU queue on each one: one path with a Q_{CPUA} queue should consider in its service rate the time to compute the item on the stage under one type of granularity (for example the MG granularity) and the other path with a Q_{CPUB} queue should consider

[3]If Zero-Copy-Buffer approach is used then this information can not be measured easily, but in this case time and energy due to communication operations can be disregarded.

the time to compute the item on the stage under the other type of granularity (the CG granularity in the example). Similar considerations should be taken for computing the energy rate on each queue. For this alternative, the GPU could be incorporated as a Q_{GPU} queue to one of the paths in the case of a Coupled Configuration (the GPU-CPU path as shown in Figure 5.12b), or in the case of a Decoupled Configuration the GPU would be incorporated to one independent path with a Q_{GPU} queue (the GPU path as shown in Figure 5.12a). The model for the alternative in with some stages exploit MG while others exploit CG granularity would be similar to the ones studied in this chapter, but in these cases the service rates of the Q_{CPU} queues should consider the time to process the item under MG or CG granularities in the corresponding stages (similarly for the energy rate). In any case, due to the constraints commented in section 5.1.1 we do not explore these alternatives further.

Effect of serial stages

The serial stages can become the bottleneck of the pipeline. Our model can detect when a configuration has reached this bottleneck. Assuming that T_i and T_o represent the times of the serial Input and Output stages, then the maximum throughput that can be achievable is,

$$\lambda_{max} = \frac{1}{max(T_i, T_o)}$$

In the case that our estimated throughput for one of the possible configurations, has a value higher than the previous maximum, i.e. λ_e^{DP-CG}, λ_e^{DP-MG}, λ_e^{CP-CG}, $\lambda_e^{CP-MG} > \lambda_{max}$, then we say that we have reached the input/output bottleneck in that configuration.

The output stage can also produce a serialization of items in the TBB implementation if we do not size the TBB queues carefully. Thus, the variable $nTokens$ which defines the maximum number of tokens in flight and therefore the size of the internal queues, must be correctly defined. For the Decoupled configurations, the optimal value for $nTokens$ is,

$$nTokens^{DP-CG} = \left\lceil \frac{max\left(\sum_{S_k \in Q} T_k^G, \sum_{S_k \notin Q} T_k^{CG}\right)}{min\left(\sum_{S_k \in Q} T_k^G, \sum_{S_k \notin Q} T_k^{CG}\right)} \right\rceil \cdot (1 + nC) \qquad (5.12)$$

whereas for the Coupled configurations, the optimal value for $nTokens$ is,

$$nTokens^{CP-CG} = \left\lceil \frac{max\left(\sum_{S_k \in Q} T_k^G + \sum_{S_k \notin Q} T_k^{CG}, \sum_{S_k \in Q} T_k^{CG}\right)}{min\left(\sum_{S_k \in Q} T_k^G + \sum_{S_k \notin Q} T_k^{CG}, \sum_{S_k \notin Q} T_k^{CG}\right)} \right\rceil \cdot (1 + nC) \quad (5.13)$$

For computing $nTokens^{DP-MG}$ and $nTokens^{CP-MG}$ we just replace T_k^{CG} by T_k^{MG} in equations. 5.12 and 5.13, respectively. In any case, these equations ensure that the number of items that can arrive to the serial output stage is enough to keep the computational resources busy. We compute the value of the parameter $nTokens$ by dividing the maximum execution time of the GPU path, GPU path in DP and GPU-CPU path in CG, by the minimum execution time of the CPU path. Thus, we get the number of tokens that are required to avoid load unbalanced with 1 GPU and 1 CPU. Later, we multiply this value by the maximum number of allowed threads to ensure a valid number of tokens when all computational units are collaborating.

In the next section, we study the accuracy of our models in two different heterogeneous chips by using a set of real applications as well as the benefits of adapting to changes in the input stream.

5.4. Experimental results

In this section, we present our experimental results. More precisely, Section 5.4.1 introduces the processors evaluated; and Section 5.4.2 presents the list of benchmarks used in the evaluation section. Next, Section 5.4.3 shows the benefit of the analytical model by comparing its performance with a state-of the art baseline approach as well as a study of the overhead due to the training phase and the profit due to the adaptive nature of our framework. Section 5.4.4 discusses our experimental results in detail and compares the throughput and energy predicted by the model with the measured values.

5.4.1. Experimental setup

In this section we present two Intel Quad-Core processors which are used in our experiments: a Core i5-3450, 3.1GHz, 77W TDP based on the Ivy Bridge architecture, and a Core i7-4770, 3.4GHz, 84W TDP based on the Haswell one,

Table 4.1 shows a detailed processors description (see page 103). Both processors feature Advance Vector Extensions (AVX) and have an on-chip GPU, the HD-2500 and HD-4600, respectively. Although the Core i7 supports hyperthreading, we found that hyperthreading is not beneficial for our applications, maybe because our benchmarks implementations use the AVX vector units and they fully utilize the computational resources. Thus, only one thread per core is considered for all experiments, and so the upper value for n is 5 threads (4 cores plus 1 GPU). We rely on Intel Performance Counter Monitor (PCM) tool [28] to access the HW counters (energy, clock ticks, L2 and L3 misses, etc).

Intel TBB 4.2 provides the core template to implement the pipeline [89]. Inside each pipeline stage, we use Intel OpenCL SDK 2014 for the stages that can be executed on the GPU, or AVX intrinsics for the computations conducted on the cores. For the MG results, we implement nested parallelism on each stage using TBB `parallel_for` or OpenCL (it depends on the benchmark, as we will note for each code in the next sections). All versions have been compiled using Intel C++ Compiler 14.0 with -O3 optimization flag. Table 4.2 shows a more detailed list of all libraries, SDKs, compiler an OS used in the reported experiments. We measured time and energy in 10 executions of the applications and compute the average. The reported metrics are throughput, λ, energy per item, E, and as a tradeoff metric, throughput/energy, λ/E. Therefore, λ is the number of frames per second, fps, E stands for the Joules per frame, and λ/E is the fps/Joule.

For our measurements, we run each application with a set of video frames that makes the applications to be running for around one hundred seconds, later we calculate the average of running times and energy measurements.

5.4.2. Benchmarks

We validate our framework on Ivy Bridge and Haswell heterogeneous chips using four real applications: ViVid [29], with Low Definition (LD) videos (600×416 pixels) and High Definition (HD) videos (1920×1080 pixels), SRAD [18], Tracking [105] and Scene Recognition [24]. For all the benchmarks and the heterogeneous chips evaluated, the transfer times between GPU and CPU are negligible. Thus, we do not specify them in our model equations, as they communicate through the Last Level Cache (LLC). Next, we describe the purpose of each benchmark application.

The **Vivid** application is previously introduced in section 5.1. ViVid is comprised of 5 stages, being the first and last ones the serial Input and Output stages,

while the three middle ones are parallel. These middle stages are: **Stage 1** that finds the maximum response of 100 filters, **Stage 2** that summarizes the low level information collected by the previous stage and **Stage 3** that computes the actual detection step.

The **SRAD** (Speckle Reducing Anisotropic Diffusion) application is part of the Rodinia benchmark suite [18]. This benchmark implements a diffusion method for ultrasonic and radar imaging applications based on partial differential equations (PDEs) [114]. This method is able to remove locally correlated noise (speckles) while maintaining important image features. SRAD has 8 pipeline stages: a serial Input and Output stages, and 6 parallel stages. In our experiments we feed these stages with a stream of 200 low-definition (LD) images.

The **Tracking** application calculates the movement of a set of features over the image-flow of a video stream. The implementation is based on the Kanade Lucas Tomasi (KLT) [69] algorithm of the San Diego Visual Benchmark Suite [105]. This implementation comprises three phases: i) image processing, ii) feature extraction and iii) feature tracking. The algorithms in the first two stages exhibit pixel-grained parallelism over the complete frame to be processed, but also they are parallel at frame granularity. The third phase, feature tracking, calculates movements of each feature over two consecutive frames of the video stream. Therefore, there does not exist frame-grained parallelism, but in exchange, the movement calculations for each feature are independent each other (feature-grained parallelism).

The **Scene-Recognition** application performs generic visual categorization, i.e., it identifies the object content of natural images while generalizing across variations inherent to the object class (view, imaging, lighting, occlusion, etc). This code is based in the algorithm proposed in [24]. That algorithm presents a *Bag of Keypoints* approach to visual categorization based on the Bag of Words (BoW) model. Under this model, an image is represented by a histogram of the number of occurrences of particular image patterns. The main advantages of the method are its simplicity, its computational efficiency and its invariance to affine transformations, as well as variations. This application is implemented as a four-stage pipeline. The first and last stages are serial. The second stage can be executed on CPU and GPU, however the third stage can only be executed on the CPU cores.

5.4.3. Baseline comparison and impact of adaptation

To assess the benefit of using our framework, we compare the pipeline configuration that our model predicts as best with the baseline configuration recommended by a previous work [104]. This approach recommends a configuration based on the intuition that pipeline stages should be mapped to the device where they run more efficiently. This work also recommends exploiting parallelism using an approach similar to software pipelining where two frames are computed at the same time, one on the GPU and another one on the CPU. Therefore, only MG granularity is exploited on the CPU cores by this baseline approach. This approach firstly executes 1 item throughout all stages on the GPU and 1 item on the CPU (only MG granularity is exploited on the CPU cores). Later, each stage is then mapped to the device where it obtains a higher throughput. Once a stage is mapped to a device, it only runs on that device, as it is less efficient to run it on the other device. The recommendation is also that the execution times of the stages running on CPU and GPU should be balanced, because with this configuration the fastest device has to wait for the slowest one. This policy may or may not result in the same pipeline mappings that the one found by our model: it is the same for ViVid, but not for SRAD and Tracking. In this work, the stages mapped to the CPU are not mapped to the GPU, and vice versa. For instance in [104], the pipeline configuration used for ViVid is stage 1 on GPU and 2 and 3 on CPU.

Table 5.6 shows the throughput in terms of frames per second (fps), and throughput/energy (fps/Joule), for homogeneous executions, where only the CPU (with MG and CG granularities) or only the GPU is used: "CPU MG", "CPU CG" and "GPU". Moreover, two heterogeneous executions are compared, where CPU and GPU are used: "Baseline" that identifies the results of the aforementioned baseline configuration [104], and "Best" which correspond to the best configuration found by our framework. For both, performance metrics (fps) and (fps/Joule), the higher the value, the better. The "Improv." column shows the percentage of improvement of "Best" with respect to "Baseline" (computed as $(Best - Baseline)/Best$). The last column shows the best pipeline configuration and the optimum number of threads, between parenthesis, for the CG cases. This table also shows that the best configuration obtained using our model significantly outperforms the baseline, specially when energy is also considered. These data show that the intuition can result in the selection of a suboptimal configuration, whereas the model can evaluate all configurations and select the best. Also, the baseline only considers "CP-MG"-like mappings, whereas the model considers more alternatives. As the table shows, in 10 out of 16 cases, the best configuration is not CP-MG. In 6 cases (all appear in ViVid) the baseline uses

Table 5.6: Comparison of our pipeline alternatives against baseline. For both λ and λ/E the higher the better.

			Homogeneous Results			Heterog. Results			
Bench.	Architect.	Metric	CPU MG	CPU CG	GPU	Baseline	Best	Improv.	Best conf.
ViVid LD	Ivy Bridge	λ (fps)	40	62	10	51	65	27%	CP-CG (5)
		λ/E (fps/J)	46	83	8	66	92	40%	CP-CG (5)
	Haswell	λ (fps)	59	47	22	80	91	13%	CP-MG
		λ/E (fps/J)	61	43	24	116	134	15%	CP-MG
ViVid HD	Ivy Bridge	λ (fps)	3.7	3.1	1.1	5.6	5.9	5%	CP-MG
		λ/E (fps/J)	0.3	0.2	0.1	0.6	0.7	15%	CP-MG
	Haswell	λ (fps)	5.4	2.8	2.7	6.5	7.2	10%	CP-MG
		λ/E (fps/J)	0.5	0.1	0.3	0.78	0.9	12%	CP-MG
SRAD	Ivy Bridge	λ (fps)	82	62	72	114	132	16%	DP-MG
		λ/E (fps/J)	212	100	362	403	523	30%	DP-MG
	Haswell	λ (fps)	95	64	93	147	170	15%	DP-MG
		λ/E (fps/J)	182	79	673	499	673	34%	DP-CG (1)
Tracking	Ivy Bridge	λ(fps)	6.2	10	6.8	13	16	23%	CP-CG (4)
		λ/E (fps/J)	1.3	3.2	2.8	4.0	6.7	67%	CP-CG (4)
	Haswell	λ (fps)	6.3	11	9.2	13	19	46%	DP-CG (5)
		λ/E (fps/J)	1.1	2.8	4.0	3.7	8.4	127%	DP-CG (5)

the same mapping as the best (Stage 1 is mapped on the GPU). In these 6 cases, λ and λ/E of baseline and best differ because in the baseline the stages mapped to the GPU can only execute on the GPU (Stage 1 only runs on the GPU), while in our implementation the stages mapped to the GPU can also execute on the CPU (Stage 1 runs on both GPU and CPU). The items that follow the CPU path in our implementation slightly increase the throughput with respect to baseline [104]. Notice that even if we only consider CP-MG mappings, the approach we use as baseline may not find the best mapping of stages to CPU and GPU. For instance, for Tracking the best CP-MG mapping would be to map stages 1 and 3 to the GPU, whereas the baseline approach would map stages 1 and 2 to the GPU.

In addition, as the number of possible configurations increases, relying on the intuition to find the best one becomes increasingly difficult. In this situation, our model can reduce the number of runtime tests that are needed to determine which one is the best configuration.

Furthermore, Table 5.6 shows that, overall throughput improvement ranges from 5% to 46% (20% on average), whereas the improvement in throughput/energy ranges from 12% to 127% (43% on average). Energy improvement ranges from 1% to 55% (18% on average). Interestingly, for ViVid on Ivy Bridge, the best pipeline configuration depends on the resolution. CP-CG is the best configuration for LD, while CP-MG is the best for HD. Also, the best configuration can change based on whether the metric to be optimized is λ or λ/E. For instance, SRAD on Haswell obtains maximum throughput with a DP-MG configuration, whereas the maximum throughput/energy is obtained using DP-CG with a single thread, where this single thread is the GPU thread. So, a GPU homogeneous execution obtains the highest λ/E. We will discuss each benchmark in more detail in section 5.4.4.

The previous work that we have considered as a baseline, can not adapt to changes in the input stream. We experimented with changes in the video stream feeding the ViVid application, from Low Definition to High Definition and vice versa. For instance, on Ivy Bridge, when changing from LD to HD, a change in the pipeline configuration from CP-CG to CP-MG results in an improvement of 81% in λ (204% in λ/E). Also, when changing from HD to LD, reconfiguring the pipeline from CP-MG back to CP-CG results in 30% improvement in λ (40% in λ/E). Since the training time for LD is 0.46 sec, and for HD is 6.47 sec, we can determine using equation 5.3, that the training is amortized (from the throughput point of view) when changes from LD to HD happen at most every 7.8 sec ($\sim$25 HD frames) and when changes from HD to LD happen at most every 0.85 sec ($\sim$42 LD frames). However, when throughput/energy is considered, these values

are 0.11 sec (∼1 HD frame) and 0.66 sec (∼43 LD frames), respectively.

The training overhead can be reduced if we keep a knowledge database were we record for every used pipeline configuration and a given throughput change, the optimum configuration that has to be adopted. That way, when detecting a throughput change that has already been recorded for the current configuration, we can directly read the recommended configuration from the database. In case, the configuration is not yet recorded, we can use the Equation 5.3 to control the period between trainings to guarantee that the training overhead is never above a certain threshold.

The scene recognition benchmark was not used on this section, as only one pipeline mapping is possible. Nevertheless, it is used in the following section 5.4.4 to validate the accuracy of the model.

5.4.4. Performance and Energy discussion

In this section we validate the accuracy of the model. Figures 5.13 to 5.17 show the results for all applications. In all of them we follow the same convention. On the left of each figure we see the CG evaluation (lines and marks) as the number of threads increases from 1 to 5, as shown on the x-axis. On the right side of each figure, we show the MG evaluation (three bars and two marks). The homogeneous CPU measurements collected in the training phase are represented by a dashed orange line for the CG execution (see Table 5.1) and by a patterned orange bar for the MG execution (see Table 5.2). Solid lines and bars represent model estimations for heterogeneous runs and marks represent experimental results. For CG predictions we use solid lines: in light-blue for the DP-CG configuration and in dark-brown for the CP-CG one. The square marks are the measurements obtained for both CG mappings: solid for DP-CG and hollow for CP-CG. The solid bars represent the model prediction for MG granularities for heterogeneous executions: in light-blue for the DP-MG configuration and in dark-brown for the CP-MG one. The x marks are the experimental results obtained for the CP-MG configuration whereas the solid triangles are the results for the DP-MG one. By comparing the measurements with the model estimates we can asses the accuracy of the model.

We have experimentally assessed all the evaluated pipeline configurations (48, 384, 48, and 5 for ViVid, SRAD, Tracking, and Scene Recognition, respectively). To facilitate readability, instead of cramming the results of all these experiments on a single chart, amongst all the possible CP mappings, Figures 5.13 to 5.17 only show the configuration that achieves the highest λ/E result. As we dis-

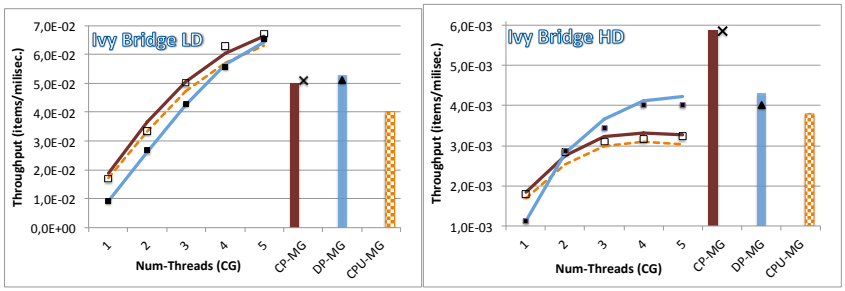
cuss in the next sections, for all the applications and architectures studied, the estimations of the model reasonably match the measured metrics. For all the cases, the model needs less than 10 microseconds to instantiate the equations for all the possibilities and determine the optimal granularity, mapping and number of threads. During the rest of the section, we discuss that in all the cases the estimations of the model accurately match the measured metrics.

ViVid

In this benchmark, the MG results are obtained exploiting nested parallelism with TBB `parallel_for`. Figures 5.13 and 5.14 depict the estimated and measured Throughput (λ), Energy per item (E) and Throughput/Energy (λ/E) for LD and HD on Ivy Bridge and Haswell, respectively. Amongst all the CP mappings we only show the most performing one: when stage S_1 is the only one mapped on the GPU (as illustrated in Figure 5.2 b) and d)), both for the CG and MG granularities.

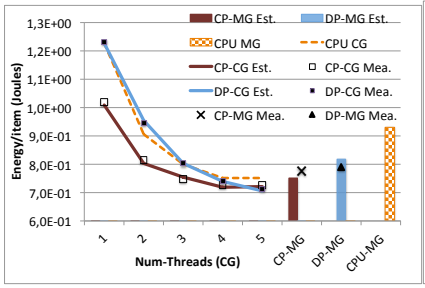
As Figures 5.13 and 5.14 show, our model is able to give a near optimal estimation of λ, E and λ/E. In general, it tends to slightly overestimate the throughput in the CP configurations, because for CP we always consider the ideal contribution of all threads without considering their associated overheads. These overheads account for the synchronization costs of the GPU-CPU threads in the coupled GPU-CPU path, that we do not consider in our equations. The results show that our model fits the measured throughput reasonably well, specially on the Ivy Bridge architecture for which the estimated values are within 2% of the measured ones. On Haswell, our overestimation of the throughput is under 9%. On the same platform, our model tends to slightly underestimate the throughput (both for LD and HD inputs) on the DP configuration, but the model predictions are always within 6% of the measured ones. We have verified in our experimental results that this underestimation occurs because the actual number of items that end up going through the CPU is smaller than the number that we estimate (using equation 5.7).

Regarding the energy results, we can also see that, our equations tend to slightly underestimate the energy, although deviation is always within 5% of the measured values. The deviation is more noticeable for Haswell, where our model predicts that CP-MG is better than DP-MG, though measures tell the contrary. Anyway, this imprecision is not significant because the differences between CP-MG and DP-MG are small, so there is not a big penalty to be paid by this error. In any case, the best configuration for energy optimization is DP-CG with 1 thread, that our model correctly predicts. Let's recall that our equations always ignore the residual values of the energy components of each device, although deviation

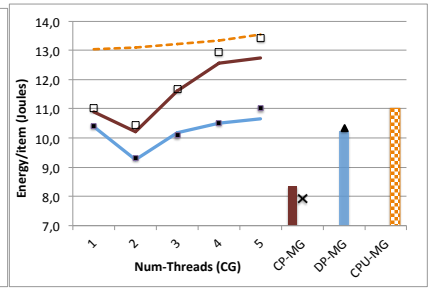


(a) Throughput

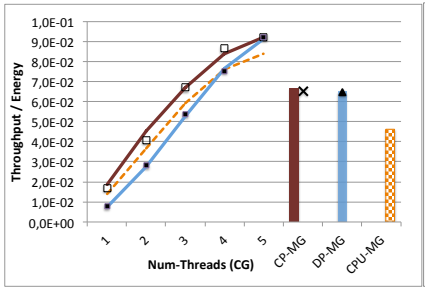
(b) Throughput



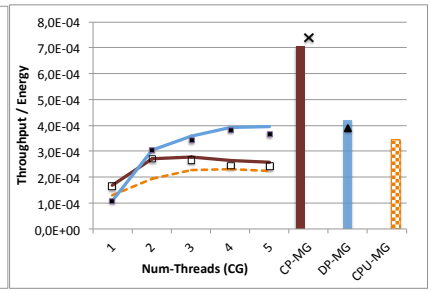
(c) Energy per Item



(d) Energy per Item



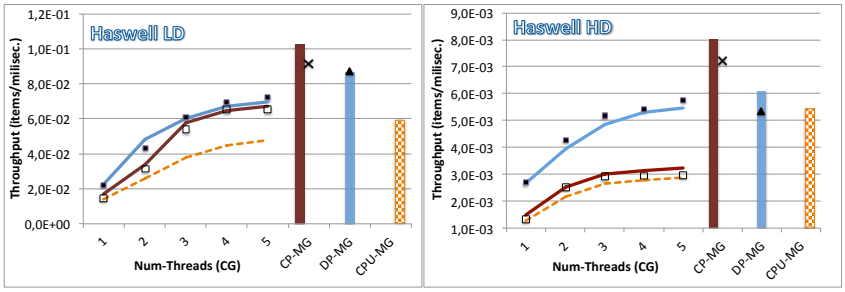
(e) Throughput/Energy



(f) Throughput/Energy

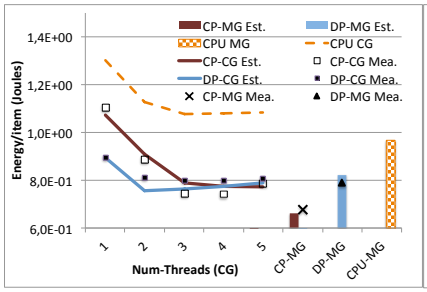
Figure 5.13: Performance metrics for ViVid when processing LD (left) or HD (right) video on Ivy Bridge. (Solid lines/bars represent model predictions and Marks are the experimental results).

is always within 5% of the measured values. In general, these results validate our initial assumption when deriving the simplified model for the energy consumption, which we introduced in section 5.3.2. Also, we can mention that the accuracy of

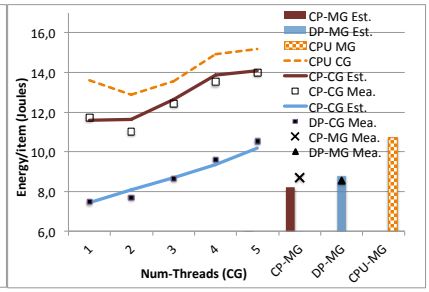


(a) Throughput

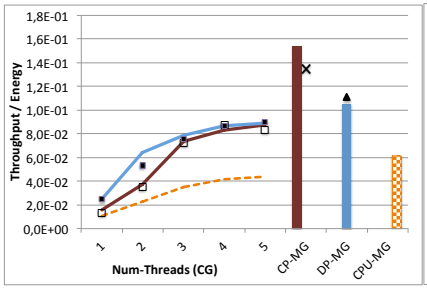
(b) Throughput



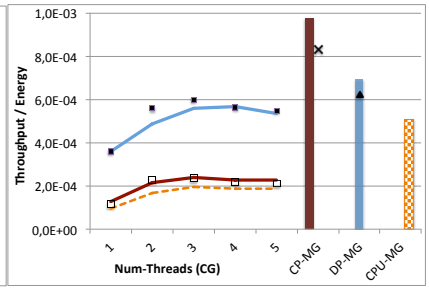
(c) Energy per Item



(d) Energy per Item



(e) Throughput/Energy



(f) Throughput/Energy

Figure 5.14: Performance metrics for ViVid when processing LD (left) or HD (right) video on Haswell. (Solid lines/bars represent model predictions and Marks are the experimental results).

the predicted values for our other metric of interest, Throughput/Energy (λ/E), are within -5% to 10% with respect to the measured values.

Overall, and despite these small inaccuracies, the model successfully predicted

the best pipeline configuration (granularity, mapping) as well as the appropriate number of threads, for each type of input and architecture. On Ivy Bridge, for LD videos, the optimal is found with the CP-CG configuration and 5 threads (although DP-CG is very close), whereas for the HD input the optimal is provided by the CP-MG configuration (remember that MG configurations are evaluated under $n = 1 + nC$, i.e. 5 threads in this architecture). However, on Haswell, the best option for LD and HD is always CP-MG.

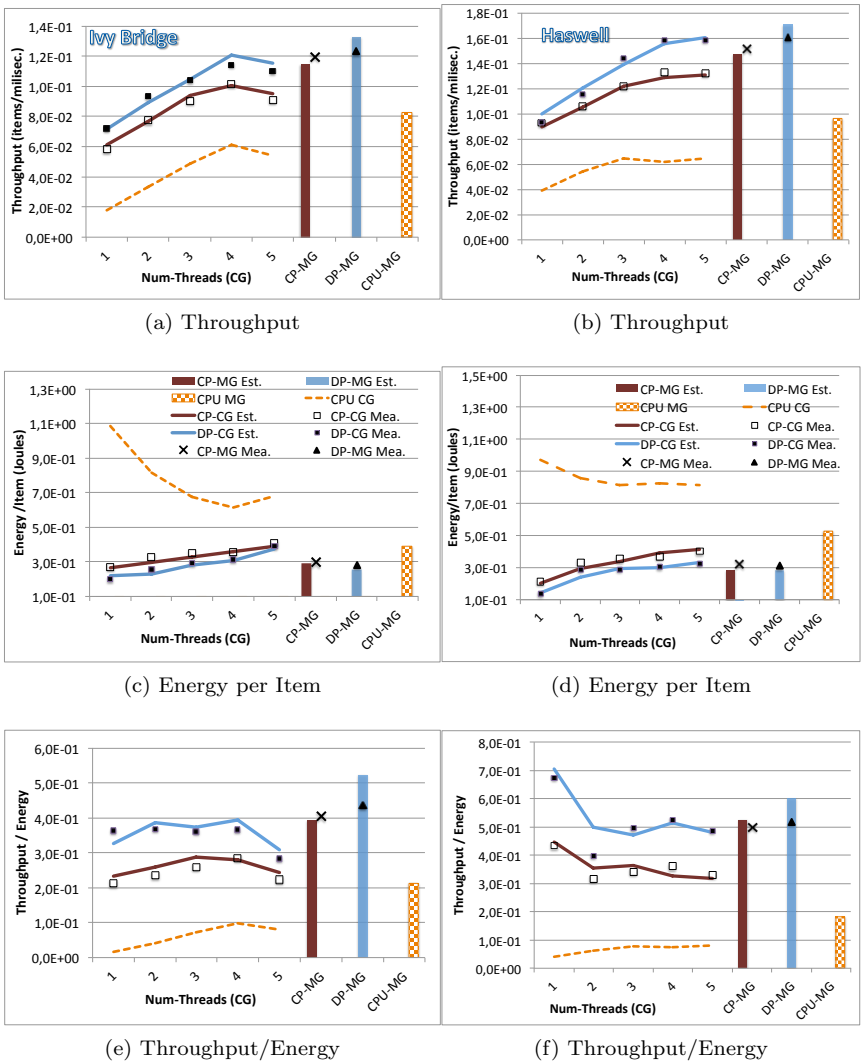
The figures also show an important result: a configuration with a higher throughput can consume more energy than a lower throughput one. This can be observed on Haswell, in Figures 5.14b and 5.14d, for the HD input and the CP-CG configuration, for which the highest throughput is obtained with $n = 5$ threads. However, the energy consumption is also higher for that number of threads. In fact, for this configuration the optimal λ/E for HD is found for $n = 3$, solution that our model correctly predicts as we see in Figure 5.14f.

Finally, notice that on Haswell our model tends to slightly underestimate the throughput (both for LD and HD inputs) on the DP configuration. This underestimation results on smaller estimated values for λ^{DP}/E^{DP} than the measured ones, although the variation is always below 6%.

SRAD

In this benchmark, each stage can implement a CG or MG granularity. Moreover, MG granularity is exploited using OpenCL. So there are $2^6 \cdot (4 + 2) = 384$ pipeline alternatives. However, just instantiating our model equations for all these possibilities we can find out the best configuration on each architecture. Figure 5.15 shows throughput, Energy per Item and Throughput per Energy results (from top to bottom) for Ivy Bridge and Haswell architectures. From all the CP mappings, we only show the most efficient one, that happens to be when the GPU is mapped on all but the second stage, for both the CP-CG and CP-MG configurations, in both machines.

The six parallel stages take 94% of the total execution time and it runs 2.6x faster on the GPU than in one CPU for the CG case, and 1.44x faster on the GPU for the MG case (similar speedups for Ivy Bridge and Haswell). These results explain the performance improvement of the heterogeneous versions over the homogeneous multicore one. Our model tends to overestimate the λ/E metric for both CG and MG granularities (due mainly to slight overestimation of the throughput and underestimation of the energy consumed, as explained in section 5.4.4). In any case, the deviation is always below 12% of measured values. Again, the model predictions are accurate enough to assess that the appropriate granularity is CG, and the optimal number of threads is 4, for both architectures.



(a) Throughput (b) Throughput

(c) Energy per Item (d) Energy per Item

(e) Throughput/Energy (f) Throughput/Energy

Figure 5.15: SRAD evaluation on Ivy Bridge (left) and Haswell (right).

Results in Figure 5.15 show that for SRAD our model also provides a reasonable estimation of the throughput and energy on each pipeline alternative. For this application, our equations tend to overestimate the throughput of the DP configurations, especially for the DP-MG configuration, where 7% of deviation

over the measured throughput was found. Also, as pointed out for ViVid, a slight underestimation of the energy consumption was registered, in this case always below 8%. These inaccuracies are the reason of the 16% of overestimation for λ/E in the DP-MG configuration. All in all, our model correctly predicts that the optimal configuration for Ivy Bridge is DP-MG, whereas for Haswell is DP-CG with $n = 1$ if we optimize λ/E. Notice, that DP-CG with $n = 1$ implies that the only thread is the GPU one, which corresponds to an homogeneous execution on the GPU. This is another example of a case in which the highest throughput does not result in the lowest energy. For instance, here we find the maximum λ with DP-CG for $n = 5$ (see Figure 5.15b for Haswell). However, since the minimum energy consumption is found for $n = 1$ (Figure 5.15d), the optimal λ/E is also for DP-CG $n = 1$ (Figure 5.15f for Haswell). Our model correctly captures this fact.

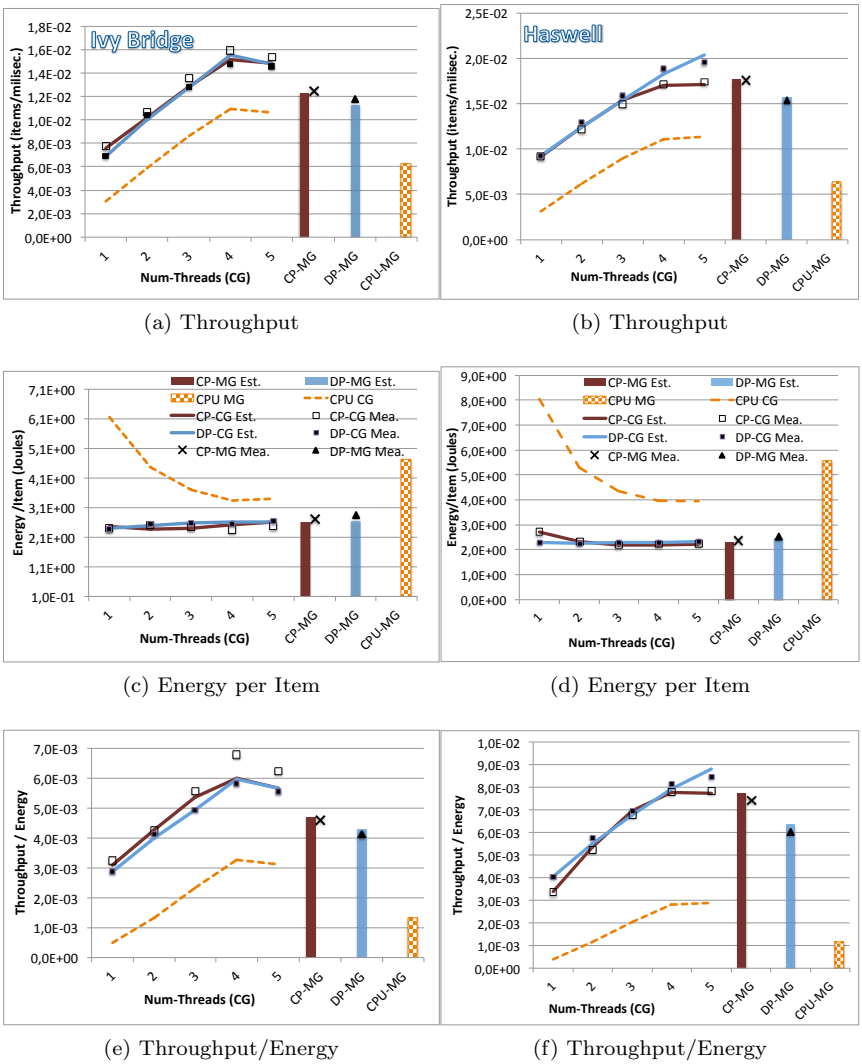
Tracking

The pipeline of this application has 5-stages: first and last ones are Input and Output serial stages, whereas the middle ones are parallel and can be mapped on both CPU and GPU. Each parallel stage can implement a CG or MG granularity. So there are $2^3 \cdot (4 + 2) = 48$ pipeline alternatives. In this benchmark, the MG granularity was exploited using TBB `parallel_for`. Again, from all the CP mappings we only show the most efficient one: stages 1 and 3 on the GPU for both the CP-CG and CP-MG configurations for Ivy Bridge and Haswell. In the experiments for tracking, we have used a video stream with 200 frames (1080x1920) HD.

Figure 5.16 shows the computed and estimated Throughput (λ), Energy (E) and Throughput per Energy (λ/E) on Ivy Bridge and Haswell architectures. The model's deviation for both platforms is always below 5%, 7% and 11% of measured throughput, energy and throughput/energy, respectively. Again, the model predictions are accurate enough to assess that the appropriate configuration is CP-CG with 4 threads with the GPU used on stages 1 and 3 for Ivy Bridge and DP-CG with 5 threads for Haswell.

Scene Recognition

This application is implemented as a 4-stage pipeline. The first and last stages take care of the sequential Input, and Output. The two middle stages are parallel. `ImageRead()` generates independent work items from an input stream of images. The next stage, `KeypointsExtraction()` extracts the keypoints for each image. This stage has been implemented to run on CPUs as well as on GPUs (using OpenCL). Next stage, `FeatureExtraction()` computes a normalized histogram of vocabulary words encountered in the image and carries out



(a) Throughput (b) Throughput

(c) Energy per Item (d) Energy per Item

(e) Throughput/Energy (f) Throughput/Energy

Figure 5.16: Tracking results on Ivy Bridge (left) and Haswell (right).

a Support Vector Machine (SVM) classification. The output of this filter is the higher ranked class identified for the image. The last stage, `OutputWrite()`, append the pair < image, class > to an output file. The input to this code are 200 images of 256×256 pixels from a database containing images from 8 different

classes (forest, street, coast, etc). In this benchmark, only the CP-CG configuration is feasible. DP mapping is not an option because the nature of the second parallel stage has loop dependences and it is not suitable for nested parallelism. However, the first parallel stage can execute on both the CPU and GPU. It has a throughput per energy on the GPU of 11% and 8% higher than that of the CPU on the Ivy Bride and Haswell, respectively. However, this first stage just represents the 18% of the pipeline execution (on both architectures).

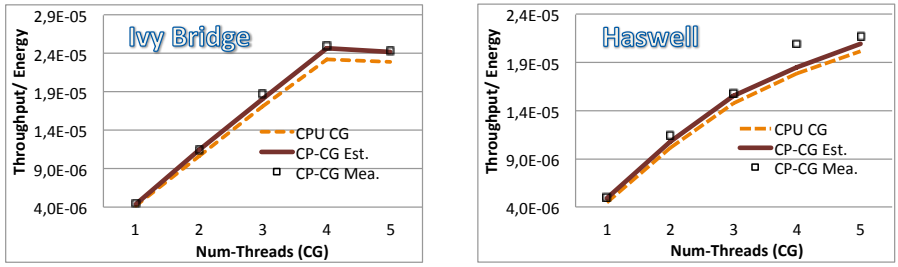
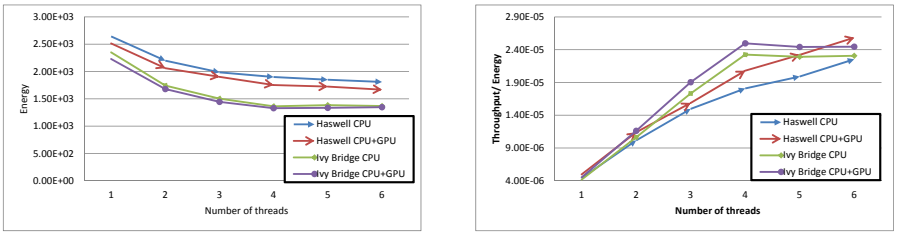


Figure 5.17: Throughput / Energy for Scene Recognition.

Figure 5.17 shows the computed and estimated λ/E for the CP-CG configuration when executing the application with 1 to 5 threads on Ivy Bridge and Haswell. Our model accurately predicts the measured values. The higher difference between predicted and measured values is found in Haswell with 4 threads. In this case, λ is underestimated 7% whereas the energy is overestimated 5.5% which turns into 11% of underestimation for λ/E. We can note that the improvement of the CP-CG execution with respect to the homogeneous multicore is small, as we are just affecting the 18% of the application, being the improvement factors (the ratio of the stage's throughput/energy on the CPU vs the GPU) also small. Anyway, one interesting finding is that the Ivy Bridge CP version reaches the point of diminishing returns with 4 threads. Here we find a similar case as with Vivid for HD: although the throughput is slightly higher with 5 threads, the energy is also higher, so the solution with the highest throughput is not the optimal for the Throughput/energy metric.

Regarding energy consumption, Ivy bridge versions are a 31% less consuming than Haswell when running 4 threads, partly because the Ivy Bridge processor has 77W of TDP whereas the Haswell one has 88W, and partly due to Haswell's GPU having more cores. On other hand, taking a look at Figure 5.18b we observe that with less than 5 threads Ivy Bridge versions achieve better throughput/energy, whilst executing with 6 threads both versions almost achieve the same value due to oversubscription renders more throughput. On Haswell, our model is a little bit

more pessimistic because in this problem we slightly overestimate the consumed energy. Anyway, the optimal solution is for $n = 5$, and our model finds it. This is due to the already discussed differences in the frequency domains of these two architectures.



(a) (b)

Figure 5.18: Scene Recognition result with Energy and Throughput per Energy on Ivy Bridge and Haswell architectures (higher the better).

5.5. Lessons learned

One relevant result of our model is that it helps us to identify the appropriate granularity for each problem. For instance, for Ivy Bridge and the LD input for ViVid, the CG granularity configurations are always more performing than the MG ones, whereas for the HD input of ViVid is the contrary. On Haswell, both for LD and HD, MG granularities perform better than CG ones. In the quest of choosing the right granularity for each problem, we have found one important piece of experimental evidence that helps us to understand how the granularities affect performance: the throughput and energy values reported by the multicore homogeneous execution for the two type of granularities (in the figures of Section 5.4, the dashed orange line for the CG execution and the patterned orange bar for the MG execution) are key to predict when one type of granularity will perform better than the other.

For instance, for ViVid on Ivy Bridge-LD (see Fig. 5.13), the CPU-MG throughput (and energy) is outperformed by the CPU-CG one when $n > 3$. However, on ViVid for Ivy Bridge-HD, the CPU-MG throughput (and energy) outperforms the CPU-CG for any n. Therefore, configurations that exploit CG

granularity seem more suitable for LD, while configurations exploiting MG will be the ones for HD, as finally the heterogeneous executions prove. This is also valid for all the other benchmarks and architectures, as We can see this result in Table 5.6 by comparing the "CPU MG" and "CPU CG" columns of the homogeneous results: if λ or λ/E is larger for "CPU MG" than for "CPU CG", then the recommended granularity for the best configuration (see "Best conf." column in that table) is MG, and vice versa.

One hardware counter quite correlated with the expected scalability for each type of granularity is the `L3CLK` ratio (ratio of CPU cycles lost due to L3 cache misses) [28]. We have observed in our codes, that heterogeneous configurations tend to have slightly higher values of `L3CLK` than the homogeneous CPU-CG and CPU-MG counterparts. But nonetheless, the work performed in the CPU multicore is the dominant source of memory traffic in the heterogeneous configurations, being the homogeneous CPU-CG and CPU-MG executions good indicators about when any type of granularity is putting more memory pressure. For instance, on IvyBridge-LD, we get `L3CLK` ratios of 0.19 and 0.7 for the CPU-CG and CPU-MG homogeneous executions respectively (hinting that CG granularity is beneficial), whereas with HD we get ratios of 1.30[4] and 0.77 respectively (hinting now that MG granularity can be the best option). Similarly, on Haswell we always obtain higher ratios for CPU-CG executions (both LD and HD inputs) than for CPU-MG ones, indicating that MG granularities can potentially outperform CG ones, as finally it occurs in the heterogeneous configurations.

In addition to granularity, the mappings also play an important role. DP mappings work well if the GPU thread obtains better values for the metric of interest in all the stages of the pipeline than a CPU thread for CG granularities (or better efficiencies in all stages than nC threads for MG granularities). If this is not the case, then CP can potentially exploit better the heterogeneity of the system, as long as the CP mapping ensures that the pipeline stages are mapped to the device where they execute most efficiently. However, CP requires synchronization between a GPU stage and a CPU stage, while DP does not require synchronization. In any case, DP and all possible CP configurations must be analysed to find the optimal configuration.

Other interesting result is that higher throughput does not always imply a lower energy consumption. This is most noticeable in the CG plots of previous figures, mainly for SRAD and ViVid HD. For example, in Figure 5.15, the CG configurations for SRAD have the highest throughput for 4-5 threads, whereas

[4]When `L3CLK` >1.0 it indicates higher memory latency that can not be hidden, hinting that the memory bus becomes a bottleneck.

the minimum energy consumption is achieved for one thread.

As summary, we have discussed some of the main trade-offs that affect throughput and energy in the DP and CP mappings under different granularities, and how our reasonable simple model is able to correctly predict all these trade-offs.

5.6. Conclusions

To the best of our knowledge, this is the first work proposing an analytical model that can be used to efficiently map the different stages of a pipeline application onto an heterogeneous chip (integrated CPU-GPU processor). The model can use throughput, energy, or a tradeoff metric such as throughput/energy to predict the best pipeline setting. The model was validated with four applications, finding that the accuracy of our estimations are within 2% to 16%, that suffices to find out the optimal pipeline configuration.

We have also compared the best configuration predicted by the model with a state of the art approach. Our results show that the configurations selected by the model produce, on the average, 20% higher λ and 43% higher λ/E. We have measured improvements in λ and λ/E of up-to 82% and 204%, respectively, when the model is used to adapt to an input video that changes its resolution. Our framework guarantees that the runtime overhead due to the training required to adapt to a changing input is always kept below a user-defined limit.

6 Concluding Remarks

This thesis settles in the context of Heterogeneous Computing where we propose several runtimes and scheduling strategies for parallel patterns with the goal of bridging the performance-programmability gap. Specifically, our proposals focus on the problem of efficiently scheduling high-level parallel patterns on heterogeneous architectures.

6.1. Contributions

In the era of Heterogeneous Computing, with the proliferation and fast evolution of heterogeneous architectures, the complexities for developing applications for these architectures increase in equal measure. Thus, harnessing the available computing power in these architectures has turned out to be a complex task, specially when programmers are used to the sequential programming paradigm. In this thesis, a few contributions are proposed to bridge the gap between performance and ease of use. In particular, we have focused on the optimisation of two parallel patterns, *parallel_for* and *pipeline*. Next, we elaborate on our proposals.

We have extended the implementation of the TBB's *parallel_for* template to allow its execution on heterogeneous architectures comprised of CPUs plus GPUs. Furthermore, we have also designed and implemented an optimization model for the inherent load unbalance problem of heterogeneous architectures, as it is further important on the presence of several accelerators with different computing power. More specifically, we propose two partitioning functions, one for the CPU cores and another for the GPU accelerators, to extract iterations from the iteration space of a *parallel_for* loop. These functions are aware of the computing power of all computing devices in the system, thus we can take better scheduling

decisions when there are not enough remaining iterations to feed all computing devices. Additionally, we propose two scheduling strategies, NCHT and CHT, that are built on top of the load balance mechanism. These two schedulers aim at maximising the effective use of the GPU host thread. Our NCHT strategy applies oversubscription to the CPU cores, thus the CPU core that holds the GPU can be kept working when waiting for GPU results. On the contrary, in the CHT strategy, the GPU host thread computes a chunk of iterations after asynchronously queueing a task on the GPU and before synchronizing in the waiting function. Finally, we find out that best performance is achieved when selecting the NCTH strategy with a number of oversubscripted threads that is equal to the number of accelerators. We compare our load balancing and scheduling strategies with the related approaches proposed in StarPU and corroborate that our NCHT proposal outperforms all the StarPU's load balance strategies. In general, nor StarPU neither other state of the art approaches pay attention to the usage of the GPU host thread that is key for optimising performance. Further details are given in Chapter 3 and in the following published articles.

➲ A case study of oversubscription on multi-CPU & multi-GPU heterogeneous systems. Antonio Vilches, Angeles Navarro, Francisco Corbera and Rafael Asenjo. In *International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE)*, Cabo de Gata, Spain, June 2013.

➲ Adaptive partitioning strategies for loop parallelism in heterogeneous architectures. Angeles Navarro, Antonio Vilches, Francisco Corbera and Rafael Asenjo. In *International Conference on High Performance Computing & Simulation (HPCS)*, Bologna, Italy, July 2014.

➲ Strategies for maximizing utilization on multi-CPU and multi-GPU heterogeneous architectures. Angeles Navarro, Antonio Vilches, Francisco Corbera and Rafael Asenjo. In *Journal of Supercomputing, 70 (2), pp 756-771*, 2014.

The next contribution of this thesis is also in the line of the *parallel_for* pattern. We propose an adaptive partition strategy that automatically finds out a near optimal task size for GPU accelerators. This contribution is key to achieve better performance, as the GPU's performance may depend on the size of the

offloaded task. Our approach has a profiling phase at the beginning, where it samples several chunk sizes and builds a model to recommend a near optimal chunk size. Later, our approach moves to a monitoring phase where it adapts the GPU chunk size according to changes in GPU's throughput. The core of this partition method is a logarithmic curve fitting that relates the profiled GPU's throughput with the offloaded chunk size. Other state of the art approaches contain a profiling phase as well, however they do not monitor throughput variations at runtime, thus they are not aware of throughput variations. In contrast, our approach resizes the GPU chunk with each throughput variation. This approach builds on top of the load balance mechanism explained before, so it is designed to work in heterogeneous systems. We compare performance and energy consumption of our approach against other three state of the art approaches and find out that we always outperform the other alternatives. Our proposal gets better results as it is always adapting to application throughput changes during runtime, and because it performs a better decision in the last part of the iteration space, when there are not enough iterations to feed all computational devices. More details area available in Chapter 4 and in the following articles.

➲ Adaptive Partitioning for Irregular Applications on Heterogeneous CPU-GPU Chips. Antonio Vilches, Rafael Asenjo, Angeles Navarro, Francisco Corbera, Ruben Gran and Maria Garzaran. In *International Conference on Computational Science (ICCS)*, Reykjavík, Iceland, June 2015.

➲ Heterogeneous parallel for template based on TBB. Antonio Vilches, Angeles Navarro, Francisco Corbera, Andrés Rodríguez, Rafael Asenjo. In *International Symposium on High-Level Parallel Programming and Applications (HLPP)*, Valladolid, Spain, July 2017.

We have also explored the sources of overhead when applying an adaptive partition to the iteration space of a *parallel_for* loop. We consider scenarios with and without oversubscription, where we profile the memory transfer operations, the internal status of the GPU queue and the behaviour of the GPU host thread when waiting for GPU completion. We get interesting insights in our results that impact performance, for example for coarse grain applications with oversubscription, the bigger overhead comes from the GPU host thread, as it is blocked and it need a period of time for the O.S. scheduler to assign a new quantum of CPU. In the case of fine grain applications, the bigger overheads comes from the memory transfer times. In any case, we propose two type of optimizations that succeed in

reducing these overhead under 1%, namely PRIO and ZCB. In particular, when using PRIO, we raise the execution priority of the GPU host thread, so it can get automatically relinquished in the list of ready threads. The second optimization is ZCB (Zero-Copy-Buffer), that avoids the copy between CPU and GPU address space. By applying these optimizations we get a performance improvement up to 2x. These results are published in the following article.

➲ Reducing overheads of dynamic scheduling on heterogeneous chips. Francisco Corbera, Andres Rodriguez, Rafael Asenjo, Angeles Navarro, Antonio Vilches and Maria Garzaran. In *Workshop on High Performance Energy Efficient Embedded Systems Co-Located with HiPEAC'15 (HIP3ES)*, Amsterdam, The Netherlands, January 2015.

In this thesis, we have also extended the TBB's *pipeline* pattern to allow the execution of streaming applications on heterogeneous systems. We provide an API that facilitates the mapping and orchestration of a series of pipeline stages onto heterogeneous architectures. Moreover, this API automatically manages memory buffers and GPU kernels. The core of this API is an optimisation model that finds the optimal *pipeline configuration* among all possible. As a pipeline configuration depends on the number of stages that can be mapped onto the GPU, the number of CPU threads and the type of parallelism exploited on the CPU cores, the number of possible configurations could be very high and might not be feasible to explore all possible configurations at runtime. Thus, we propose a model that is able to predict the performance and energy consumption of each pipeline configuration and find the optimal configuration. Our pipeline implementation has two phases: i) the *exploration phase*, where it collects some time and energy consumption data when executing a few items; and ii) the *exploitation phase*, where it executes the input stream of data with the pipeline configuration found by executing the analytical model. Additionally, we also consider that the characteristics of the input stream may change. Hence, we also monitor the throughput of the application for changes during the second phase. If it happens that we detect a variation in throughput bigger than a user given threshold, we re-execute the profiling phase and evaluate the model with the new collected data to get a new optimal configuration. We perform a comparison with other related approach [104], and show that we always outperforms their performance, as they only consider executing a stage either on the CPU or on the GPU. More details are given in Chapter5 and in the next published articles.

➲ Mapping streaming applications on commodity multi-CPU and GPU on-chip processors. Antonio Vilches, Angeles Navarro, Rafael Asenjo, Francisco Corbera, Ruben Gran and Maria Garzaran. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2015.

➲ Pipeline Template for Streaming Applications on Heterogeneous Chips. Andres Rodriguez, Angeles Navarro, Rafael Asenjo, Francisco Corbera, Antonio Vilches, Maria Garzaran. In *International Conference on Parallel Computing (ParCo)*, Edinburgh, United Kingdom, September, 2015.

➲ Parallel Pipeline on Heterogeneous Multi-Processing Architectures. Andrés Rodriguez, Angeles Navarro, Rafael Asenjo, Antonio Vilches, Francisco Corbera and Maria Garzaran. In *International Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms (RePara) Co-Located with ISPA'15*, Helsinki, Finland, August 2015.

6.2. Limitations

Whereas this thesis provides several contributions, it also has some limitations. The first limitation is related to code portability. For the two parallel patterns that have been extended in this thesis, we require that the user provides two versions of the kernel, one for the CPU cores and one for the GPU accelerators. However, it would be beneficial to express a unique version of the kernel that is compiled and run on each computational device within the system. In this sense, we have published a preliminary work [110] based on SYCL that aims at helping to overcome this limitation. It is a library that only requires a sequential version in C++ that is later translated to SPIR-V and then compiled for all computational devices in the system.

Another limitation of this thesis is that our scheduling strategies and runtimes consider the application of interest as the only one running on the system. Thus, if it happens that two or more applications are executing in the system, our models will fail in their performance predictions, specifically those related to the estimation of energy consumption. Our analytical models and schedulers make this assumption since nowadays it is not possible to isolate the energy

consumption of an application when there are several applications running at the same time.

6.3. Future work

Whilst in this thesis we have presented a number of contributions in the field of *task scheduling* to support efficient execution of parallel patterns on heterogeneous architectures, there are a number of potential future work that may be worth exploring. In this sense, we conclude this thesis by pointing out several research continuations.

- Our partition strategy follows a centralized approach, that may suffer from memory contention when hundred or thousands of threads work in parallel. However, with the advent of heterogeneous shared virtual systems (SVM), CPUs and GPUs can access the same memory positions, thus we may lessen the impact of synchronisation between devices by using atomic operations. Furthermore, SVM will greatly simplify the development of distributed lock-free algorithms to solve the issue of distributing the partitioning of work.
- One line of research worth exploring is the implementation of our schedulers on top of HSA (*Heterogeneous Systems Architecture*). This technology would further reduce the overheads due kernel launching and GPU completion signaling overheads.
- Our results show that many innovations are required in the memory subsystem, as the GPU computational units are mainly stalled when waiting for memory operations being resolved. Hopefully, the upcoming abstraction in 3D stacked memory and the processing in memory paradigm will help to reduce the memory wall, specially in memory bound applications.
- We may also find interesting to extend and adapt our models and schedulers in order to offer support to heterogeneous systems comprised of CPUs plus a FPGA, or even a more disruptive combination comprised of CPUs plus a GPU plus a FPGA. For instance, the latest Xilinx heterogeneous platform, UltraScale+ MPSoC, features a quad core CPU, plus a Mali 400 GPU plus a FPGA. We strongly believe that our models and schedulers can be extended to offer support for these new heterogeneous platforms, with the goal of increasing productivity of programmers and providing performance without pain.

Heterogeneous Computing is the most promising paradigm for accelerating applications and reducing energy consumption. However, there is no free lunch, this architectural heterogeneity and the availability of different computational units increases the complexity of developing applications for these architectures. Thus, this thesis aims at reducing the programming wall of these architectures and provide a number of contributions to Heterogeneous Computing field, with the hope that it will help to develop further abstractions that reduce programming complexities and improve the overall performance.

Apéndice A
Resumen en castellano

En los últimos años se ha experimentado una revolución en el área de arquitectura de procesadores. El final de la era de los uniprocesadores ha dado paso a los procesadores *multicore* y *manycores* con arquitecturas heterogéneas. Estos cambios han sido debidos a varios factores tencológicos entre los que destaca el elevado consumo de potencia, que degeneraba en situaciones en las que el calor generado era tan grande que no podía ser disipado del encapsulado del chip. Como solución a este problema, la industria ha apostado por arquitecturas *multicore* como nuevo paradigma para asegurar que se siguen aumentando las prestaciones con cada nueva generación tecnológica. Actualmente, las empresas líderes de fabricación de procesadores van más allá de las arquitecturas *multicore*, siguiendo una evolución hacia las llamadas arquitecturas *manycore* y heterogéneas con grandes números de núcleos por chip. Estas arquitecturas están emergiendo como una alternativa que ofrece un alto rendimiento, con un bajo consumo de potencia. Sin embargo, la explotación de este tipo de arquitecturas presenta un importante desafío a los desarrolladores de aplicaciones y de herramientas software. Para acelerar la ejecución de las aplicaciones, el programador se enfrenta a una tarea compleja y tediosa en la que se ve forzado a extraer y expresar el paralelismo de la aplicación lidiando con detalles de bajo nivel de la arquitectura del procesador, lo que limita la portabilidad y la productividad. De hecho, se ha llegado a comparar la paralelización con una actividad de artesanía, cuando debiera ser un proceso de ingeniería software. Para maximizar la productividad del programador es necesario que se diseñen nuevos modelos de programación que dispongan de patrones paralelos de alto nivel que permitan expresar el comportamiento paralelo de los algoritmos. Lo ideal es que estos patrones actúen como una capa de abstracción, que oculte los complejos detalles de implementación inherentes a la programación paralela. En esta línea surge esta tesis doctoral, que se centra en el estudio e

implementación de patrones paralelos de alto nivel que expresen intuitivamente las oportunidades de paralelismo presentes en las aplicaciones.

A.1. Introducción

La evolución reciente en diseño de procesadores parece indicar que las futuras generaciones contendrán cientos de cores. Más aún, estudios de la eficiencia energética y del rendimiento por vatio [72], indican que el óptimo se consigue en arquitecturas en las que los cores no son simétricos: hablamos de sistemas heterogéneos que se componen de *aceleradores* muy potentes (como GPUs o el coprocesador Xeon Phi) con numerosos, pero más simples cores [64]. De hecho, muchas plataformas HPC están diseñadas de manera que los nodos integrantes consisten en un multicore con una (o más) GPUs. Ejemplos de este tipo de arquitecturas heterogéneas son los 2 mayores supercomputadores del mundo, así tenemos, el *Tianhe-2* que ocupa la primera posición del top 500[1]. Este supercomputador ofrece 33.86 petaflop/s gracias a que en cada uno de sus 16.000 nodos dispone de 2 CPUs Intel Ivy Bridge y tres coprocesadores Intel Xeon Phi. De la misma forma, *Titan*, número 2 del top 500 alcanza unos 17.59 petaflop/s gracias a las 18.688 GPUs Nvidia Tesla, que son responsables del 90 % de esa capacidad de cómputo.

Los sistemas heterogéneos basados en GPUs, pueden incorporar GPUs discretas conectadas a través del bus PCI-Express, o bien pueden estar integradas con el sistema multicore formando parte del mismo circuito integrado. Ejemplo de este último tipo de arquitecturas son los procesadores AMD Fusion (APUs), Intel Haswell o NVIDIA Jetson K1. Claramente, y por restricciones en el consumo de potencia, estos sistemas que combinan distintas arquitecturas se están convirtiendo en las arquitecturas dominantes. Incluso para dispositivos móviles como smartphones y tablets, los últimos procesadores comercializados ya incluyen 4 u 8 cores y una GPU integrada. De este tipo de procesadores embebidos podemos destacar el procesador Qualcomm Snapdragon 800 (que contiene una GPU Adreno 320), o el procesador de Samsung Exynos Octa con una GPU PowerVR. Independientemente del grado de acoplamiento de las GPUs en el sistema heterogéneo, el éxito de estas arquitecturas dependerá de la capacidad del software del sistema para adaptar el paralelismo definido a nivel de aplicación al paralelismo disponible en el nivel hardware. El objetivo de esta tesis consiste en proporcionar mecanismos para incrementar el rendimiento de arquitecturas heterogéneas de una forma productiva y efectiva para el desarrollador.

[1] www.top500.org

En este contexto, cuando hablamos de acelerar aplicaciones, necesitamos que el software del sistema ofrezca un modelo de programación que permita expresar todas las oportunidades de paralelismo de la aplicación, mientras que el *runtime* será el responsable de mapear el paralelismo a la heterogeneidad del sistema, tanto en términos de potencia de cálculo como en términos de consumo de energía. Es decir, el *runtime* debe asegurar la utilización eficiente de los recursos en estas arquitecturas a través de un preciso particionado de la carga de trabajo entre los *cores* de CPU y las GPUs del nodo. Diseñar una estrategia o mecanismo de partición del trabajo en estos sistemas no es trivial: se ha de considerar que cierto tipo de cálculos pueden ser ejecutados más eficientemente en CPUs que en GPUs, debido a la sobrecarga de movimiento de datos entre las diferentes memorias, o a ineficiencias de los kernels que ejecutan las GPUs en los que aparecen demasiadas divergencias de control, o accesos aleatorios a la memoria de la GPU, etc. De forma inversa, también existen aplicaciones que se ejecutan mucho más eficientemente en GPUs, como aplicaciones de algebra matricial. Otro problema, es determinar la cantidad de trabajo que se asigna a cada GPU, dependiendo de la aplicación, puede afectar en mayor o menor medida al rendimiento que se obtiene en dicha GPU. Hasta el punto de que dicho rendimiento puede ser inferior al que se consigue en un core, en cuyo caso puede no ser rentable asignar dicha computación a la GPU. Además dependiendo de las características *hardware* de cada una de las GPUs, la cantidad de trabajo óptima para cada una de ellas puede ser diferente. Todo ello se complica en el caso de aplicaciones irregulares, para las que la carga computacional puede variar a lo largo de la ejecución de la aplicación, por lo que un mecanismo de partición adaptativo es altamente recomendable [40].

A.2. Motivación

El problema general de la planificación de tareas en este tipo de sistemas heterogéneos basados en multicores que incorporan una o más GPUs, ha recibido una importante atención últimamente. Sin embargo, las librerías CUDA [96], OpenCL [38] y OpenACC [82] que representan el estado del arte para la programación de GPUs permiten especificar que una aplicación se ejecute en la GPU (CUDA, OpenACC, OpenCL) o bien en los núcleos de CPU (OpenCL), pero no en las dos unidades de computación simultáneamente. Además, los compiladores actuales son poco efectivos en la optimización de este tipo de aplicaciones para su ejecución en las modernas arquitecturas heterogéneas, debido a que cada unidad computacional dispone de un conjunto de instrucciones ISA diferente. De hecho, en arquitecturas heterogéneas con aceleradores, el problema es aún más complicado, porque sus entornos de programación requieren que el programa-

dor explícitamente maneje importantes operaciones de bajo nivel como son la organización de los grupos de threads que se mapean en los *cores* físicos de los aceleradores, el alojamiento de memoria en el acelerador, o la comunicación de datos entre CPU y aceleradores. Todos estos factores hacen que la aproximación actual sea tediosa cuando se trata de resolver problemas complejos. Este tipo de aproximaciones redunda en poca productividad del programador, y nuestra visión es que hay que automatizar y optimizar el proceso tanto como sea posible.

La planificación de trabajo entre CPUs y GPUs ha recibido mucha atención recientemente. Existen propuestas [2, 10, 40, 45, 58, 70] que permiten la ejecución concurrente de tareas en *cores* de CPU y GPUs. Se basan en planificadores que gestionan la planificación de tareas de manera concurrente en ambos tipos de recursos manteniendo el tamaño de grano (o carga computacional) de tareas sin cambiar durante todo el tiempo de ejecución. Por ejemplo, OmpSs [10] permite especificar el tipo de planificación entre CPUs y GPU utilizando directivas similares a las de OpenMP, mientras el planificador utiliza un modelo basado en data-flow análisis para asegurar que las tareas que se planifican no son dependientes de las que se están ejecutando. StartPU [2] y XKaapi [40] exponen al programador una API a partir de la cual éste puede especificar distintas estrategias de planificación. En [40] el planificador utiliza un modelo de coste para planificar por adelantado todas las tareas disponibles. Esta solución presenta la desventaja de que la asignación puede dar lugar a desbalanceos de la carga, en caso de que la ejecución real no se adapte a lo predicho por las funciones de coste. En [2] el runtime usa un heurístico para decidir cuando hay localidad entre tareas que se pueden planificar en la misma GPU, o en caso de que haya que robar trabajo para balancear la carga, tener en cuenta esta información de localidad. Sin embargo, esto obliga a que el usuario sea responsable de especificar el rango de direcciones y el tipo de acceso (read, write, reduction, exclusive), a través de la API que proporciona el framework para definir las tareas. En [58] se presenta un modelo de programación que permite la ejecución de bucles paralelos sobre la CPU o la GPU pero nunca en las dos unidades de procesamiento al mismo tiempo, de esta forma se presenta un mecanismo que permite hacer códigos portables que se pueden ejecutar tanto en la CPU como la GPU. Por lo tanto la asignación del tamaño de trabajo a cada dispotivos es estática, decidida a priori por el programador tras analizar empíricamente el tamaño de bloque óptimo. De hecho, en todos los trabajos descritos, el programador es el responsable de determinar el tamaño de grano de cada tarea, es decir no se aborda el problema de la partición dinámica del trabajo, que el objetivo de nuestro estudio.

Por otro lado, recientes propuestas como [4, 58, 84] realizan un ajuste automático del tamaño de grano que se asigna a cada dispositivo. En [4] se toman

4 muestras, comenzando con un tamaño igual al número de núcleos de GPU y se multiplica por 2 sucesivamente en los 3 siguientes pasos. Una vez tomadas las 4 primeras muestras se realiza un ajuste de mínimos cuadrados de una curva logarítmica, y a partir de aquí se mantiene el tamaño fijo. En [84] se presenta una estrategia en la que la GPU empieza a procesar toda la carga computacional mientras que la CPU computa pequeños subconjuntos empezando por el final, hasta que se cruzan las computaciones. El problema que presenta esta estrategia es que todos los datos tienen que poder alojarse en la memoria de la GPU y esto no siempre es posible. En [58] se presenta otro esquema adaptativo en el que solo se realiza un ajuste del tamaño de grano en el primer 10 % de la carga computacional por lo que si hay variaciones en el resto de la carga el sistema quedará desbalanceado, como es el caso de aplicaciones irregulares que presenten cargas de trabajo triangulares. En nuestro caso, puesto que también estudiamos aplicaciones irregulares, el grano óptimo dependerá de los datos de entrada y variará durante todo el tiempo de ejecución de la aplicación, por lo que el planificador debe calcularlo dinámicamente, siendo ésta una de las premisas que diferencia nuestra investigación de trabajos anteriores. Otro punto diferenciador de nuestro trabajo es que pensamos usar la asimetría entre rendimiento y consumo de energía de las diferentes unidades de procesamiento para proporcionar soluciones en las que en función de la métrica de interés se consiga el mejor rendimiento, o bien el mejor rendimiento para la energía consumida o simplemente la solución que consuma menos energía (ver Capitulos 4 y 5 para más información).

Las contribuciones que resultan de la realización de esta tesis han sido publicados en conferencias internacionales con revisión por pares [80, 92, 93, 107], workshops internacionales [23], y revistas [81, 108] que se encuentran clasificadas en el ISI Journal Citation Report (JCR), a continuación resumimos las contribuciones en las secciones de este anexo.

A.3. Balanceador de tareas para bucles paralelos

Cuando ejecutamos aplicaciones sobre arquitecturas heterogéneas con varios tipos de procesadores, nos encontramos con el problema del reparto de tareas entre procesadores heterogéneos del sistema, esto es que cada tipo de procesador tiene unas características propias y ofrece un rendimiento diferente. Para explotar al máximo las capacidades de cómputo que ofrecen este tipo de arquitecturas, es necesario alimentar a todos los procesadores del sistema con un tamaño de tareas acorde a los recursos hardware de cada procesador. En este contexto, primero nos centramos en la ejecución eficiente de bucles *for* con un conjunto de iteraciones

independientes sobre arquitecturas heterogéneas con varias CPUs y GPUs.

En esta primera aproximación asumimos que el usuario proporciona un tamaño de tarea óptimo para cada acelerador, GPU en nuestro caso. De esta forma podemos reducir la complejidad del problema y centrarmos en el problema del balanceo de carga de trabajo entre todos los procesadores (CPUs + GPUs), dado que en el caso de arquitecturas heterogéneas tres factores son críticos para alcanzar un rendimiento ideal: 1) el tamaño de las tareas ejecutadas debe de ser cuidadosamente seleccionado y adaptado durante eñ tiempo de ejecución; 2) La asignación de las tareas a CPU y GPU debe garantizar unos desbalanceos de carga mínimos; 3) el rendimiento de cada unidad computacional tiene que ser medido con precisión.

En esta sección presentamos un modelo de optimización para resolver analíticamente el problema de balanceo de carga en un sistema heterogéneo. En este contexto, nosotros observamos que la ejecución de un bucle *for* paralelo se puede descomponer en intervalos de ejecución, donde en cada uno son ejecutados subrangos de iteraciones. Así tenemos que el tamaño óptimo de los *cores* de CPU en el intervalo i-ésimo viene dado por la siguiente función,

$$Ch(I_{C_i}) = max\left(\frac{Ch(I_{G_i^k})}{f^k}\right), \quad k = 1 : nGPUs, \quad \forall i = 1 : N, \tag{A.1}$$

donde f^k representa la velocidad relativa de la GPU k-ésima respecto de un CPU *core*. Mientras que el término $Ch(I_{G_i^k})$ representa el tamaño de tarea que el usuario ha proporcionado para la GPU k-ésima. De esta forma, garantizamos que los *cores* de CPU recibirán tareas de un tamaño que necesitará un tiempo de cómputo equivalente al de la GPU más lenta.

De esta forma, todos las unidades computacionales del sistema tendrán suficiente trabajo disponible mientras que se cumpla la siguiente condición,

$$\frac{Ch(I_{G_{i-1}^k})}{\lambda(I_{G_{i-1}^k})} < \frac{r - Ch(I_{G_{i-1}^k})}{(\sum_{j \neq k} \lambda(I_{G_{i-1}^j})) + nCores \cdot \lambda(I_{C_{i-1}})}, \tag{A.2}$$

esta condición se satisface cuando el tiempo que require la k-ésima GPU para ejecutar su tarea sea menor que el tiempo que requieren el resto de unidades computacionales del sistema para computar el número de iteraciones restantes (r) menos el tamaño de la k-ésima GPU. Cuando alcanzamos el tramo final del espcio de iteraciones y la ecuación A.2 no se cumple, la k-ésima GPU que es

desactivada, y el *thread* de GPU pasa a computar bloques de iteraciones como una CPU más. Una vez que todas las GPUs han sido desactivadas, los *cores* de CPU terminan de ejecutar las iteraciones restantes siguiendo una estrategia de planificación *guided*.

Además de este modelo analítico que pretende solucionar el problema del balanceo de carga en arquitecturas heterogéneas, se proponen dos strategias de planificación, NCHT and CHT, que tienen como objetivo mejorar el uso del *thread* de CPU que sirve a la GPU. En la estrategia de planificación NCHT el *thread* de CPU que gestiona a la CPU se bloquea mientras que espera a que la GPU termina el cómputo de su tarea. Por otro lado, en la estrategia de planificación CHT, este *thread* de CPU computa una tarea justo después de encolar trabajo para la GPU y antes de bloquearse en el punto de sincronización. Aunque pudiera parecer que esta la estrategia CHT debe rendir mejor, hay que tener en cuenta que con códigos irregulares es complicado hacer que la CPU y GPU acaben al mismo tiempo, esto sumado a que estamos estimando el tamaño de CPU, hace que cometamos pequeños errores que producen esperas entre los dispositivos. Por otro lado, la estrategia NCHT simplemente bloquea el *thread* de CPU en el punto de sincronización haciendo que esta versión desperdicie un *core* de CPU. Sin embargo, la estrategia NCHT consigue el mejor rendimiento cuando se aplica junto con la técnica de *oversubscription* con un número de *threads* adicionales igual al número de GPUs en el sistema.

A.4. Particionador adaptativo para bucles paralelos

Para extender el trabajo anterior, proponemos un particionador, LogFit, que es sensible al rendimiento de la GPU. Este particionador se encargará de partir y distribuir bloques de iteraciones de un bucle *for* paralelo entre las unidades computacionales del sistema. Este particionador es complementario al modelo de balanceo de carga anterior. Además, este particionador está diseñado para monitorizar las aplicaciones durante todo el tiempo de ejecución para ser capaz de adaptarse a cambios en las demandas de cómputo.

Este algoritmo dispone de tres etapas: muestreo, explotación y final. En la primera etapa o etapa de muestreo, este modelo empieza muestreando pequeños tamaños de tareas que son multiplos del número de unidades de ejecución de la GPU, a medida que la GPU ejecuta estas pequeñas tareas también se duplica el tamaño de éstas. Este proceso continúa hasta que el rendimiento (medido en

iteraciones por segundo) deja de crecer, en dos muestras consecutivas. En ese momento se escogen 4 tamaños de tareas equidistantes entre todos los muestreados y se realiza un ajuste logarítmico con esos cuatro puntos.

Posteriormente, el algoritmo se mueve a la fase de explotación, el objetivo de esta fase es adaptar el tamaño de las tareas que se asignan a la GPU de acuerdo al rendimiento del último intervalo de ejecución. Para ello, el algortimo fija los tres primeros puntos y actualiza el cuarto punto por el resultado de ejecutar el último tamaño de tarea. Con este procedimiento, el ajuste logarítmico se ajusta a la demanda de cómputo del bucle paralelo en todos sus intervalos de ejecución. El algoritmo reduce el tamaño de tareas de GPU cuando cae el rendimiento, con el objetivo de reducir los accesos no alineados a la memoria de GPU o la carga computacional que se asigna a cada unidad de ejecución de la GPU. Por el contrario, cuando el rendimiento de GPU aumenta, esto se debe a que hay un menor tráfico en el bus de memoria o se ha reducido la carga computacional por iteración, por tanto se aumenta el tamaño de tareas para sacar un mayor partido a la GPU.

De nuevo, cuando se cumple la condición de parada, $remaining < Ch(I_{G_{i+1}}) + Ch(I_{C_i}) \cdot n_{cores}$, el algoritmo se mueve a la fase final. En esta fase, el algoritmo contempla tres posibles escenarios, que deba de ejecutar todas las iteraciones restantes en la CPU, en la GPU, o que deba de encontrar un punto intermedio para repartir las iteraciones entre la CPU y la GPU dependiendo del rendimiento relativo de cada procesador.

A.5. Planificación de tareas para el patrón *pipeline*

Para resolver el problema que surge a la hora de ejecutar aplicaciones de tipo *streaming* sobre arquitecturas heterogéneas nosotros proponemos un modelo para estimar el rendimiento y otro para estimar el consumo de energía. Las aplicaciones de tipo *streaming* se implementan típicamente como una serie de etapas en *pipeline* y son ampliamente usadas en cualquier sistema de cómputo, pero especialmente en los sistema portátiles como *smartphones* y *tablets*, donde los *chips* heterogéneos dominan el mercado. Nuestra propuesta extiende la plantilla *pipeline* que ofrece la librería *Intel Threading Building Blocks* (*TBB*) para permitir la ejecución de este patrón sobre sistemas heterogéneos. Con esta extensión de la plantilla se alcanzan tres objetivos: i) simplificar el trabajo de programación de arquitecturas con *GPU* y *multicores*; ii) optimizar la ejecución mediante un

reparto balanceado de la carga entre los distintos dispositivos; y iii) reducir el consumo de energía permitiendo al programador elegir fácilmente la configuración del *pipeline* más adecuada a tal efecto.

Como caso de estudio y para mostrar un ejemplo de uso de nuestra plantilla y modelos, vamos a usar la aplicación ViVid[2], una aplicación que implementa un algoritmo de detección de objetos (caras), que se basa en una aproximación del algoritmo de ventana deslizante para la detección de objetos. ViVid consiste en un *pipeline* de 5 etapas. Las etapas primera y última son las de entrada y salida respectivamente (son etapas serie), mientras que las tres etapas intermedias son etapas paralelas (sin estado[3]).

Cuando aplicaciones como ViVid son ejecutadas en arquitecturas heterogéneas integradas en *chip*, existen muchas posibles configuraciones. Por ejemplo, el usuario tiene que considerar la granularidad, el número de *items* del *pipeline* que se deben de computar simultáneamente, el procesador al que debe de ser asignada cada etapa y el número de *cores* de *CPU* a usar, para minimizar el tiempo de ejecución, el consumo de energía, o ambas (dependiendo de la métrica de interés). La granularidad determina el nivel de paralelismo que es explotado en la *CPU*. En nuestra aproximación, el usuario puede especificar dos niveles de granularidad: grano grueso (*Coarse Grain*, CG) y grano medio (*Medium Grain*, MG). Si diferentes *items* pueden ser ejecutados simultáneamente en la misma etapa (sin estado) en la *CPU*, entonces la granularidad CG puede ser explotada. Por otro lado, si el cuerpo de una etapa es paralelizable, entonces un único *item* puede ser procesado en paralelo por varios *cores* de *CPU* en dicha etapa, y la granularidad MG puede ser explotada. En nuestra propuesta, la granularidad CG implica que hay tantos *items* procesándose simultáneamente en el *pipeline* como *threads* disponibles, mientras que en la granularidad MG hay como máximo dos *items* siendo procesados a la vez, uno por el *multicore* y otro por la *GPU*.

El mapeo del *pipeline* determina los procesadores en los que se pueden ejecutar las diferentes etapas del mismo. La Fig. 5.3 muestra todos los posibles mapeos que se pueden dar para las tres etapas paralelas de la aplicación ViVid.

Asumamos que un *pipeline* está compuesto de S_1, S_2, ... S_n etapas paralelas. Usamos una n-tupla para especificar todos los posibles mapeos a la *GPU* y a los *cores* de *CPU*: $\{m_1, m_2, \ldots, m_n\}$. El i-ésimo elemento de la tupla, m_i, especifica si la etapa S_i puede ser mapeada en la *CPU* y *GPU*, $(m_i = 1)$, o si ésta puede ser mapeada únicamente en la *CPU*, $(m_i = 0)$. Si $m_i = 1$, cuando entra un

[2]http://www.github.com/ mertdikmen/vivid

[3]Una etapa es paralela o sin estado cuando el cómputo sobre un elemento en esa etapa no depende del cómputo de elementos anteriores.

item en la etapa S_i se comprueba si la *GPU* está libre. Si la *GPU* está libre, la etapa S_i procesa dicho *item* en la *GPU*; en otro caso, es procesado en la *CPU*. Si $m_i = 0$, el *item* sólo podrá ser procesado en la *CPU*. Por ejemplo, las posibles configuraciones de la aplicación ViVid se corresponden con las siguientes tuplas: {1,1,1}, {1,0,0}, {0,1,0}, {0,0,1}, {1,1,0}, {1,0,1}, {0,1,1}, {0,0,0}. En nuestra implementación, el mapeo {1,1,1} representa un caso especial: si la *GPU* está libre cuando un nuevo *item* entra al *pipeline*, entonces todas las etapas para este *item* se mapean en la *GPU*.

A.6. Conclusiones

En la era de computación heterogénea, existe una necesidad por encontrar nuevos paradigmas de programación que faciliten el desarrollo de aplicaciones para estas nuevas arquitecturas sin comprometer la productividad de los programadores. Esta tesis tiene como objetivo principal proporcionar patrones paralelos de alto nivel que son capaces de ejecutarse eficientemente en arquitecturas heterogéneas. Para apoyar este objetivo se han propuesto varias optimizaciones que permiten explotar arquitecturas heterogéneas de forma eficiente. En esta tesis nos hemos centrado en la optimización de dos patrones paralelos: *parallel for* y *pipeline*, a continuación detallamos las contribuciones de esta tesis.

1. Nuestra primera contribución consiste en un modelo análitico para minimizar el desbalanceo de cómputo entre las diferentes unidades computacionales del sistema. Este modelo asume que los tamaños óptimos de las tareas que son asignadas a las GPUs son estáticos y no varían durante el tiempo de ejecución. Sin embargo, este modelo si considera la variación del tamaño de las taras asignadas a los *cores* de CPU para hacer que las CPUs y las GPUs trabajen a un ritmo de trabajo cercano. De esta forma, el modelo permite que todas las unidades computacionales trabajen concurrentemente hasta que se alcanza la *condición de parada*. Esta condición se activa cuando no quedan suficientes iteraciones para mantener trabajando a todas las unidades funcionales, entonces las funciones heurísticas que implementan el modelo desactivan la GPU que estaba intentando obtener una tarea para favorecer el balanceo de carga del resto del sistema. Cuando ya no queda ninguna GPU trabajando, los *cores* de CPU siguen una planificación de tipo *guided*. Junto con este modelo de balanceo de carga, se presentan dos estrategias de planificación de trabajo que intentan maximizar el uso del hilo que mantiene a la GPU, NCHT y CHT. La estrategia NCHT usa la técnica de *oversubscription* para maximizar el uso de la CPU que gestiona

la GPU

2. Otra contribución consiste en un particionador adaptativo de bucles *for* paralelos que tiene en cuenta el rendimiento de las GPUs. Este particionador se basa en un modelo logarítmico que encuentra un tamaño de tareas que alimenta todas las unidades de ejecución de cada procesador. Particionadores del estado del arte consideran que bloques de iteraciones mayores a un determinado número tienen una ralación lineal con el tiempo de ejecución que requiren para ser computadas. Sin embargo, existen aplicaciones como la multiplicación dispersa matriz por vector o aplicaciones de grafos que no siguen esa relación debido a sus irregularidades. En este sentido, nosotros hemos encontrado que asignar tareas de gran tamaño a los aceleradores puede ser contraproducente cuando se ejecutan aplicaciones irregulares. Esto es debido a que los *threads* de un mismo grupo deben acceder a posiciones alineadas en memoria, o en otro caso el sistema de memoria podría verse afectado ante una alta demanda de accesos a posiciones de memoria aleatorias. Como resultado de esto, las unidades de ejecución de la GPU podrían quedar bloqueadas mientras que esperan a que el controlador de memoria les sirva sus peticiones. Por el contrario, nuesto algoritmo primero encuentra un tamaño de tarea que es capaz de alimentar todas las unidades de ejecución de la GPU. Después, éste monitoriza el rendimiento de la GPU durante todo el tiempo de ejecución para adaptar el tamaño de las tareas de GPU al rendimiento y el tráfico de datos.

3. En esta tesis también proponemos un modelo para estimar el rendimiento y el consumo de energía para aplicaciones de tipo *streamming* que son implementadas como una serie de etapas. Aplicaciones de tipo *streamming*, como las de reconocimiento facial o seguimiento de objetos, son apropiadas para ser ejecutadas en arquitecturas heterogéneas. Trabajos previos en la planificación de este tipo de aplicaciones proponen un enfoque productor-consumidor que dividen las etapas en dos conjuntos. El primer grupo de tareas es ejecutado en un procesador y su salida es pasada al segundo grupo de tareas que es ejecutado en el otro dispositivo. Este enfoque funciona bien cuando el tiempo de ejecución de ambos grupos de etapas es similar, esto es que ambos grupos de tareas se computan a una velocidad similar. Sin embargo, este enfoque no producirá buenos resultados cuando los conjuntos de tareas están desbalanceados. Por otro lado, nosotros proponemos una estrategia que busca una solución más fina. Nuestro espacio de búsqueda para la mejor configuración del *pipeline* tiene tres variables: el número de *threads*, el tamaño de grano para CPU y la asignación de las etapas a las unidades de computación. De esta forma, nosotros ejecutamos una pequeña

etapa de muestreo donde capturamos los tiempos de ejecución y los contadores *hardware* mientras que ejecutamos unos cuantos *items* en la CPU y un *item* en la GPU. Como resultado, utilizamos todos los datos recolectados para alimentar nuestro modelo analítico y así poder predecir la mejor configuración posible para una aplicación dada y su entrada. Además, nuestro enfoque tiene la ventaja de que se autoorganiza y realiza la toma de decisiones automaticamente, mientras que en los otros trabajos necesitan ser especificados por el usuario manualmente.

A.7. Trabajos futuros

En esta tesis hemos presentado varias contribuciones para optimizar la ejecución de dos patrones paralelos sobre arquitecturas heterogéneas. Sin embargo, hay un número de líneas de trabajo que serían interesantes de explorar, dado que todavía no existe un método automático y generalista para repartir la carga de trabajo en este tipo de arquitecturas. A partir de esta limitación, hay varias líneas de investigación para extender el trabajo presentado en esta tesis.

- Nuestro sistema de partición de trabajo sigue un enfoque centralizado, este hecho podría hacer que apareciesen problemas de contención de memoria cuando se ejecuten cientos o miles de *threads*. Sin embargo, con la aparición de los sistemas heterogéneos con sistemas compartidos de memoria virtual (SVM), los *cores* de CPU y los *cores* de GPU pueden acceder a las mismas posiciones de memoria. Este avance evita las sincronizaciones explícitas entre dispositivos mediante la incorporación de instrucciones *atomic*. Además, posibilita el desarrollo de algoritmos paralelos sin *locks*, dado que las dependencias de acceso a la memoria pueden ser resueltas por el protocolo de coherencia de la memoria cache.
- Nuestros resultados muestran que más optimizaciones e innovaciones son necesarias en el sistema de memoria, ya que como analizamos en el capítulo 4, las unidades computacionales de la GPU están paradas esperando datos de memoria durante la mayor parte del tiempo de ejecución. Las nuevas tecnologías de memorias 3D y el emergente paradigma de procesamiento en memoria son unos enfoques prometedores para paliar el bajo rendimiento de los sistemas de memoria actuales, especialmente en aplicaciones limitadas por los lentos accesos a memoria.
- Otra interesante línea de trabajo sería la extensión de nuestros modelos analíticos para soportar arquitecturas heterogéneas compuestas de CPUs y

una FPGA, o incluso de combinaciones mucho más rompedoras compuestas de CPUs, una GPU y una FPGA. Por ejemplo, la última plataforma heterogénea de Xilinx, la UltraScale+ MPSoC, integra una CPU con cuatro *cores*, una GPU Mali 400 y una FPGA. Nosotros pensamos que nuestros modelos y planificadores pueden ser extendidos para ofrecer soporte para esas nuevas arquitecturas, con el objetivo de incrementar la productividad de los programadores a la vez que se aumenta el rendimiento en sus aplicaciones.

Para concluir, me gustaría decir que el paradigma de Computación Heterogénea es el más prometedor para acelerar aplicaciones y reducir su consumo de energía. Sin embargo, en esta vida todo tiene un coste, la heterogeneidad de los tipos de procesadores incrementan la complejidad del desarrollo de aplicaciones para esas arquitecturas. De esta forma, esta tesis tiene como objetivos la reducción de la complejidad de programación para estas arquitecturas de procesadores y proporcionar varias contribuciones al área de Computación Heterogénea, con la esperanza de que ayuden a desarrollar más abstracciones para reducir la complejidad de desarrollo y aumentar el rendimiento.

Bibliography

[1] H. Arafat, J. Dinan, S. Krishnamoorthy, P Balaji, and P. Sadayappan. Work stealing for GPU-accelerated parallel programs in a global address space framework. *Concurrency Computation Practice and Experience*, 22(6):685–701, 2015. (Cited on page 37)

[2] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency Computation Practice and Experience*, 22(6):685–701, 2010. (Cited on pages 4, 7, 37, 46, 48, 57, 80, 126 and 190)

[3] Peter E. Bailey, David K. Lowenthal, Vignesh Ravi, Barry Rountree, Martin Schulz, and Bronis R. De Supinski. Adaptive Configuration Selection for Power-Constrained Heterogeneous Systems. In *43rd International Conference on Parallel Processing*, pages 371–380, 2014. (Cited on page 5)

[4] Mehmet E. Belviranli, Laxmi N. Bhuyan, and Rajiv Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Transactions on Architecture and Code Optimization*, 9(4):1–20, jan 2013. (Cited on pages 5, 38, 39, 45, 52, 80, 85, 88, 114, 115 and 190)

[5] Anne Benoit, Paul Renaud-Goud, and Yves Robert. Performance and energy optimization of concurrent pipelined applications. *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*, 2010. (Cited on page 41)

[6] Emily Blem, Hadi Esmaeilzadeh, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon And The End Of Multicore Scaling. In *International Symposium on Computer Architecture*, pages 122–134, 2011. (Cited on pages 2 and 18)

[7] Robert Blumofe, Christopher Joerg, and Bradley Kuszmaul. Cilk : An Efficient Multithreaded Runtime System. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, 1995. (Cited on pages 4 and 23)

[8] Sanjay K. Bose. Open and closed networks of M/M/m type queues. Technical report, Indian Inst. of Technology Guwahati, 2002. (Cited on page 149)

[9] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. State-of-the-art in heterogeneous computing. *Sci. Program.*, 18(1):1–33, January 2010. (Cited on page 19)

[10] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M. Badia, Xavier Martorell, Eduard Ayguade, and Jesus Labarta. Productive programming of GPU clusters with OmpSs. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IPDPS 2012*, pages 557–568, 2012. (Cited on pages 20, 37, 46, 57, 80, 126 and 190)

[11] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. IEEE Xplore - A quantitative study of irregular programs on GPUs. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151, 2012. (Cited on pages 4, 81, 83 and 86)

[12] Cascaval Calin, Fowler Seth, Montesinos-Ortego Pablo, Piekarski Wayne, Reshadi Mehrdad, Robatmili Behnam, Weber Michael, and Bhavsar Vrajesh. ZOOMM: a parallel web browser engine for multicore mobile devices. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 271–280, 2013. (Cited on page 4)

[13] F. Callaly, D. O'Loughlin, D. Lyons, a. Coffey, and F. Morgan. Xilinx Vivado High Level Synthesis: Case studies. *25th IET Irish Signals & Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communities Technologies (ISSC 2014/CIICT 2014)*, pages 352–356, 2014. (Cited on page 20)

[14] Chongxiao Cao, Mark Gates, Azzam Haidar, Piotr Luszczek, Stanimire Tomov, Ichitaro Yamazaki, and Jack Dongarra. Performance and Portability with OpenCL for Throughput-Oriented HPC Workloads across Accelerators, Coprocessors, and Multicore Processors. *Proceedings of ScalA 2014: 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems - held in conjunction with SC 2014: The International Conference*

for High Performance Computing, Networking, Storage and Analysis, pages 61–68, 2015. (Cited on page 33)

[15] Marc Casas, Miquel Moreto, Lluc Alvarez, Emilio Castillo, Dimitrios Chasapis, Timothy Hayes, Luc Jaulmes, Oscar Palomar, Osman Unsal, Adrian Cristal, Eduard Ayguade, Jesus Labarta, and Mateo Valero. *Runtime-Aware Architectures*, pages 16–27. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. (Cited on page 17)

[16] M M T Chakravarty, G Keller, S Lee, T L McDonell, and V Grover. Accelerating Haskell array codes with multicore GPUs. *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14, 2011. (Cited on page 4)

[17] Sanjay Chatterjee, Max Grossman, Alina Sbîrlea, and Vivek Sarkar. Dynamic Task Parallelism with a GPU Work-Stealing Runtime System. In *Languages and Compilers for Parallel Computing*, pages 203–217, 2013. (Cited on page 37)

[18] S Che, M Boyer, J Meng, D Tarjan, S Lee, J W Sheaffer, and K Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE International Symposium on Workload Characterization*, pages 44–54, 2009. (Cited on pages 104, 105, 162 and 163)

[19] Luca De Cicco, Saverio Mascolo, and Vittorio Palmisano. Skype Video Responsiveness to Bandwidth Variations. In *Nossdav*, pages 81–86, 2008. (Cited on page 127)

[20] Jason Clemons, Haishan Zhu, Silvio Savarese, and Todd Austin. MEVBench: A mobile computer vision benchmarking suite. *Proceedings - 2011 IEEE International Symposium on Workload Characterization, IISWC - 2011*, pages 91–102, 2011. (Cited on page 126)

[21] Alexander Collins, Christian Fensch, Hugh Leather, and Murray Cole. MaSiF: Machine learning guided auto-tuning of parallel skeletons. *20th Annual International Conference on High Performance Computing, HiPC 2013*, pages 186–195, 2013. (Cited on page 5)

[22] G. Contreras and M. Martonosi. Characterizing and improving the performance of intel threading building blocks. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 57–66, Sept 2008. (Cited on page 31)

[23] Francisco Corbera, Andres Rodriguez, Rafael Asenjo, Angeles Navarro, Antonio Vilches, and Maria Garzaran. Reducing overheads of dynamic scheduling on heterogeneous chips. In *International Workshop on High Performance Energy Efficient Embedded Systems*, 2015. (Cited on pages 8 and 191)

[24] Gabriella Csurka et al. Visual categorization with bags of keypoints. In *ECCV*, pages 1–22, 2004. (Cited on pages 162 and 163)

[25] Anthony Danalis, Gabriel Marin, Collin Mccurdy, Jeremy S Meredith, Philip C Roth, and Jeffrey S Vetter. The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite Categories and Subject Descriptors. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units Pages*, pages 63–74, 2010. (Cited on pages 104 and 105)

[26] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. RAPL: Memory power estimation and capping. In *ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 189–194, 2010. (Cited on page 145)

[27] Timothy A Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1–25, 2011. (Cited on pages 104 and 105)

[28] Roman Dementiev, Thomas Willhalm, Otto Bruggeman, Patrick Fay, Patrick Ungerer, Austen Ott, Patrick Lu, James Harris, Phil Kerly, and Patrick Konsor. *Intel Performance Counter Monitor - A better way to measure CPU utilization*, 2012. (Cited on pages 103, 145, 162 and 177)

[29] Mert Dikmen, Derek Hoiem, and Thomas S Huang. A data driven method for feature transformation. In *Computer Vision and Pattern Recognition (CVPR)*, pages 3314–3321. IEEE, 2012. (Cited on pages 127 and 162)

[30] Grewe Dominik, Zheng Wang, and Michael Boyle. OpenCL Task Partitioning in the Presence of GPU Contention. In *26th International Workshop Languages and Compilers for Parallel Computing*, pages 87–101, 2013. (Cited on page 5)

[31] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. Compiling a high-level language for GPUs: (via language support for architectures and compilers). *PLDI: Programming Languages Design and Implementation*, 47(6):1–12, 2012. (Cited on page 4)

[32] Alejandro Duran, Julita Corbalan, and Eduard Ayguade. Evaluation of openmp task scheduling strategies. In *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*, IWOMP'08, pages 100–110, Berlin, Heidelberg, 2008. Springer-Verlag. (Cited on pages 20 and 23)

[33] J Enmyren and CW Kessler. SkePU: a multi-backend skeleton programming library for multi-GPU systems. *Hlpp*, pages 5–14, 2010. (Cited on page 4)

[34] Naila. Farooqui, R. Barik, B. Lewis, T. Shpeisman, and k. Schwan. Affinity-Aware Work-Stealing for Integrated CPU-GPU Processors. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 30, 2016. (Cited on page 37)

[35] J. A. Fernández-Madrigal, E. Cruz-Martín, A. Cruz-Martín, J. González, and C. Galindo. Adaptable web interfaces for networked robots. *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS*, pages 2164–2169, 2005. (Cited on page 127)

[36] Juan Jose Fumero, Toomas Remmelg, Michel Steuwer, and Christophe Dubach. Runtime Code Generation and Data Management for Heterogeneous Computing in Java. In *International Conference on Principles and Practices of Programming on the Java Platform*, pages 16–26, 2015. (Cited on pages 4 and 19)

[37] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. (Cited on page 12)

[38] Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011. (Cited on pages 3, 4, 19, 36 and 189)

[39] Thierry Gautier, Joao Vicente Ferreira Lima, Nicolas Maillard, and Bruno Raffin. Locality-Aware Work Stealing on Multi-CPU and Multi-GPU Architectures. In *6th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG)*, Berlin, Germany, January 2013. (Cited on page 37)

[40] Thierry Gautier, João V F Lima, Nicolas Maillard, and Bruno Raffin. XKaapi: A runtime system for data-flow task programming on heterogeneous architectures. *Proceedings - IEEE 27th International Parallel and*

Distributed Processing Symposium, IPDPS 2013, pages 1299–1308, 2013. (Cited on pages 4, 37, 46, 57, 80, 126, 189 and 190)

[41] Paul Gibbon, Wolfgang Frings, and Bernd Mohr. Performance analysis and visualization of the n-body tree code pepc on massively parallel computers. In *PARCO*, pages 367–374. Citeseer, 2005. (Cited on pages 104 and 105)

[42] M. Goli and H. González-Vélez. Heterogeneous algorithmic skeletons for fast flow with seamless coordination over hybrid architectures. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 148–156, Feb 2013. (Cited on page 40)

[43] Ricardo Gonzalez and Mark Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1283, 1996. (Cited on page 132)

[44] William J. Gordon and Gordon F. Newell. Closed queuing systems with exponential servers. *Operations Research*, 15(2):254–265, 1967. (Cited on pages 149 and 150)

[45] Dominik Grewe and Michael F P O Boyle. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In *CC'11/ETAPS'11 Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software*, pages 286–305, 2011. (Cited on pages 5, 35 and 190)

[46] Khronos Group. Spir-v white paper, July 2016. https://www.khronos.org/registry/spir-v/papers/WhitePaper.pdf. (Cited on pages 21 and 31)

[47] HardKernel. Odroid-XU3, Aug. 2014. http://www.hardkernel.com/main/. (Cited on page 145)

[48] Joel Hestness, Stephen W Keckler, and David A Wood. GPU Computing Pipeline Inefficiencies and Optimization Opportunities in Heterogeneous CPU-GPU Processors. In *2015 IEEE International Symposium on Workload Characterization*, pages 87–97, 2015. (Cited on page 8)

[49] HSA. Hsa foundation. harmonizing the industry around heterogeneous programming, July 2016. http://www.hsafoundation.com/. (Cited on page 20)

[50] H. P. Huynh, A. Hagiescu, O. Z. Liang, W. F. Wong, and R. S. M. Goh. Mapping streaming applications onto gpu systems. *IEEE Transactions on Parallel and Distributed Systems*, 25(9):2374–2385, Sept 2014. (Cited on page 40)

[51] T. Ibaraki and H. Katoh. *Resource Allocation Problems: Algorithmic Approaches*. MIT Press, Cambridge, Mass., 1988. (Cited on page 51)

[52] Google Inc. RenderScript for Android, 2016. (Cited on page 4)

[53] Qualcomm Inc. Qualcomm mare: Parallel computing sdk, July 2016. https://developer.qualcomm.com/software/mare-sdk. (Cited on page 20)

[54] Intel. *Intel OpenCL N-Body Sample*, 2014. (Cited on pages 88, 104 and 105)

[55] Intel. *Intel VTune Amplifier 2015*, 2014. https://software.intel.com/en-us/intel-vtune-amplifier-xe. (Cited on page 81)

[56] Norman Rubin Jin Wang and Sudhakar Yalamanchili. Paralleljs: An execution framework for javascript on heterogeneous systems. In *Seventh Workshop on General Purpose Processing Using GPUs (GPGPU-7)*, March 2014. (Cited on page 4)

[57] M Jones and P Viola. Fast Multi-view Face Detection. Technical Report July, Mitsubishi Electric Research Lab, 2003. (Cited on page 127)

[58] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T. Lewis, Chunling Hu, and Keshav Pingali. Adaptive heterogeneous scheduling for integrated GPUs. In *Proceedings of the 23rd international conference on Parallel architectures and compilation - PACT '14*, pages 151–162, New York, New York, USA, aug 2014. ACM Press. (Cited on pages 5, 19, 39, 45, 80, 114, 190 and 191)

[59] Christoph Kessler, Usman Dastgeer, Samuel Thibault, Raymond Namyst, Andrew Richards, Uwe Dolinsky, Siegfried Benkner, Jesper Larsson Träff, and Sabri Pllana. Programmability and performance portability aspects of heterogeneous multi-/manycore systems. In *Design, Automation and Test in Europe (DATE)*, pages 1403–1408, 2012. (Cited on page 4)

[60] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. *Proceedings of the 26th ACM international conference on Supercomputing - ICS '12*, page 341, 2012. (Cited on pages 4 and 7)

[61] Andreas Kloeckner. Opencl for python, July 2016. https://documen.tician.de/pyopencl/. (Cited on pages 4 and 19)

[62] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An Automatic Input-Sensitive Approach for Heterogeneous Task Partitioning

Categories and Subject Descriptors. *Proceedings of the 27th international ACM conference on International conference on supercomputing - ICS '13*, pages 149–160, 2013. (Cited on page 35)

[63] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 65–76, April 2009. (Cited on pages 63, 104 and 105)

[64] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *IEEE Computer*, 38(11):32–38, 2005. (Cited on pages 1, 2 and 188)

[65] Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer, editors. *Improving Performance Portability in OpenCL Programs*, pages 136–150. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. (Cited on pages 32 and 33)

[66] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006. (Cited on page 22)

[67] Victor W. Lee, Per Hammarlund, Ronak Singhal, Pradeep Dubey, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, and Srinivas Chennupaty. Debunking the 100X GPU vs. CPU Myth. In *ISCA*, volume 38, pages 451–460, 2010. (Cited on page 35)

[68] Chih Sheng Lin, Chao Sheng Lin, Yu Shin Lin, Pao Ann Hsiung, and Chihhsiong Shih. Multi-objective exploitation of pipeline parallelism using clustering, replication and duplication in embedded multi-core systems. *Journal of Systems Architecture*, 59(10 PART C):1083–1094, 2013. (Cited on page 41)

[69] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *IJCAI*, pages 674–679, 1981. (Cited on page 163)

[70] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture - Micro-42*, page 45, 2009. (Cited on pages 5, 36, 37, 39, 76, 80, 126 and 190)

[71] Deepak Majeti and Vivek Sarkar. Heterogeneous Habanero-C (H2C): A Portable Programming Model for Heterogeneous Processors. *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 708–717, 2015. (Cited on page 23)

[72] Ami Marowka. Analytical modeling of energy efficiency in heterogeneous processors q. *Computers and Electrical Engineering*, 39(8):2566–2578, 2013. (Cited on page 188)

[73] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012. (Cited on pages 13, 14, 15 and 16)

[74] Heike Mccraw, James Ralph, Anthony Danalis, and Jack Dongarra. Power monitoring with PAPI for extreme scale architectures and dataflow-based programming models. In *2014 IEEE International Conference on Cluster Computing, CLUSTER 2014*, pages 385–391, 2014. (Cited on page 145)

[75] S. McIntosh-Smith, J. Price, R. B. Sessions, and a. a. Ibarra. High performance in silico virtual drug screening on many-core processors. *International Journal of High Performance Computing Applications*, 29(2):1094342014528252–, 2014. (Cited on page 32)

[76] Sparsh Mittal and Jeffrey S Vetter. A Survey of Methods for Analyzing and Improving GPU Energy Efficiency. *ACM Computing Surveys*, 47(2):1–23, 2014. (Cited on page 2)

[77] Sparsh Mittal and Jeffrey S. Vetter. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Computing Surveys*, 47(4):1–35, 2015. (Cited on pages 4 and 11)

[78] Gordon E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998. (Cited on pages 2 and 17)

[79] Angeles Navarro, Rafael Asenjo, Siham Tabik, and Calin Caşcaval. Analytical modeling of pipeline parallelism. In *Conference Proceedings Parallel Architectures and Compilation Techniques, PACT*, volume 2, pages 281–290, 2009. (Cited on pages 30 and 149)

[80] Angeles Navarro, Antonio Vilches, Rafael Asenjo, and Francisco Corbera. Adaptive Partitioning Strategies for Loop Parallelism in Heterogeneous Architectures. In *International Conference on High Performance Computing & Simulation (HPCS)*, pages 120–128, 2014. (Cited on pages 8 and 191)

[81] Angeles Navarro, Antonio Vilches, Francisco Corbera, and Rafael Asenjo. Strategies for maximizing utilization on multi-CPU and multi-GPU heterogeneous architectures. *Journal of Supercomputing*, pages 756–771, 2014. (Cited on pages 8, 43, 75, 76, 78 and 191)

[82] OpenACC Working Group. *The OpenACC Application Programming Interface*, 2013. (Cited on pages 36 and 189)

[83] Opencl for .net platform, July 2016. http://openclnet.codeplex.com/. (Cited on page 19)

[84] Prasanna Pandit and R. Govindarajan. Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 273. ACM, feb 2014. (Cited on pages 5, 38, 80, 190 and 191)

[85] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. Portable performance on heterogeneous architectures. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems - ASPLOS '13*, volume 41, page 431, 2013. (Cited on pages 4, 33 and 37)

[86] Ashwin Prasad, Jayvant Anantpur, and R. Govindarajan. Automatic compilation of MATLAB programs for synergistic execution on heterogeneous processors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 47, pages 152–163, 2012. (Cited on page 5)

[87] TOP500 project. TOP500 List, 2016. (Cited on page 11)

[88] V. T. Ravi and G. Agrawal. A dynamic scheduling framework for emerging heterogeneous systems. In *2011 18th International Conference on High Performance Computing*, pages 1–10, Dec 2011. (Cited on page 37)

[89] James Reinders. *Intel Threading Building Blocks: Multi-core parallelism for C++ programming*. O'Reilly, 2007. (Cited on pages 4, 7, 8, 19, 23, 24, 31, 43, 56, 74, 84 and 162)

[90] Toomas Remmelg, Thibaut Lutz, Michel Steuwer, and Christophe Dubach. Performance Portable GPU Code Generation for Matrix Multiplication. *GPGPU: Workshop on General Purpose Processor Using Graphics Processing Units*, pages 22–31, 2016. (Cited on page 33)

[91] Ruymán Reyes, Iván López-Rodríguez, Juan J. Fumero, and Francisco De Sande. accULL: An OpenACC implementation with CUDA and OpenCL support. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7484 LNCS(228398):871–882, 2012. (Cited on page 20)

[92] Andres Rodriguez, Angeles Navarro, Rafael Asenjo, Francisco Corbera, Antonio Vilches, and Maria Garzaran. Pipeline Template for Streaming Applications on Heterogeneous Chips. In Gerhard R Joubert, Hugh Leather, Mark Parsons, Frans Peters, and Mark Sawyer, editors, *Parallel Computing: On the Road to Exascale*, number 27 in Advances in Parallel Computing, pages 327–336, Amsterdam, Berlin, Tokyo, Washington DC, 2015. IOS Press. (Cited on pages 8 and 191)

[93] Andres Rodriguez, Angeles Navarro, Rafael Asenjo, Antonio Vilches, Francisco Corbera, and Maria Garzaran. Parallel Pipeline on Heterogeneous Multi-Processing Architectures. In *International Symposium on Parallel and Distributed Processing with Applications*, pages 166–171, 2015. (Cited on pages 8 and 191)

[94] David C Rudolph and Constantine D Polychronopoulost. An Efficient Message-Passing Scheduler Based on Guided Self Scheduling. In *Proceedings of the 3rd international conference on Supercomputing*, pages 50–61, 1989. (Cited on page 55)

[95] Victor Lomüller Ruyman Reyes. Sycl: Single-source c++ accelerator programming. *Parallel Computing: On the Road to Exascale*, 27:673–682, 2016. (Cited on page 20)

[96] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010. (Cited on pages 3, 4, 19, 20, 36 and 189)

[97] Alina Sbîrlea, Yi Zou, Zoran Budimlíc, Jason Cong, and Vivek Sarkar. Mapping a Data-flow Programming Model Onto Heterogeneous Platforms. In *International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, volume 47, pages 61–70, 2012. (Cited on pages 5, 37 and 126)

[98] Oren Segal, Martin Margala, Sai Rahul Chalamalasetti, and Mitch Wright. High Level Programming for Heterogeneous Architectures. In *International Workshop on FPGAs for Software Programmers*, volume 1, pages 49–54, 2014. (Cited on page 19)

[99] Robert Soulé, Michael I. Gordon, Saman Amarasinghe, Robert Grimm, and Martin Hirzel. Dynamic expressivity with static optimization for streaming languages. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 159–170, 2013. (Cited on page 40)

[100] John A Stratton, Hee-seok Kim, Thoman B Jablin, and Wen-mei W Hwu. Performance Portability in Accelerated Parallel Kernels. Technical report, University of Illinois at Urbana-Champaign, 2013. (Cited on page 33)

[101] Huayou Su, Nan Wu, Mei Wen, and Chunyuan Zhang. On the GPU-CPU Performance Portability of OpenCL for 3D Stencil Computations. In *IEEE International Conference on Parallel and Distributed Sys*, pages 78–85, 2013. (Cited on page 33)

[102] Michael B. Taylor. A landscape of the new dark silicon design regime. *IEEE Micro*, 33(5):8–19, 2013. (Cited on page 1)

[103] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, 2010. (Cited on page 49)

[104] Ehsan Totoni, Mert Dikmen, and María Jesús Garzarán. Easy, fast, and energy-efficient object detection on heterogeneous on-chip architectures. *ACM Transactions on Architecture and Code Optimization*, 10(4):1–25, 2013. (Cited on pages 8, 40, 126, 130, 164, 166 and 182)

[105] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. SD-VBS: The san diego vision benchmark suite. *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009*, pages 55–64, 2009. (Cited on pages 162 and 163)

[106] S. Venkatasubramanian and R. W. Vuduc. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *International. Conference on Supercomputing*, 2009. (Cited on page 37)

[107] Antonio Vilches, Rafael Asenjo, Angeles Navarro, Francisco Corbera, and Maria Garzaran. Adaptive Partitioning for Irregular Applications on Heterogeneous CPU-GPU Chips. In *International Conference on Computational Science (ICCS)*, volume 51, pages 140–149, 2015. (Cited on pages 8 and 191)

[108] Antonio Vilches, Angeles Navarro, Rafael Asenjo, Francisco Corbera, Ruben Gran, and Maria Garzaran. Mapping streaming applications on

commodity multi-CPU and GPU on-chip processors. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2015. (Cited on pages 8, 125 and 191)

[109] Antonio Vilches, Angeles Navarro, Francisco Corbera, Andres Rodriguez, and Rafael Asenjo. Performance Portability of a GPU Enabled Factorization with the DAGuE Framework. In *10th International Symposium on High-Level Parallel Programming and Applications, HLPP'17*, 2017. (Cited on page 8)

[110] Antonio Vilches and Ruyman Reyes. Syclparallelstl: A parallel stl library for heterogeneous systems. In *1st SYCL Programming Workshop*, 2016. (Cited on page 183)

[111] Zhenning Wang, Long Zheng, Quan Chen, and Minyi Guo. CPU+GPU scheduling with asymptotic profiling. *Parallel Computing*, 40(2):107–115, feb 2014. (Cited on pages 5, 39, 76 and 85)

[112] Mei Wen, Da-fei Huang, Chang-qing Xun, and Dong Chen. Improving performance-specific OpenCL kernels on multi-core / many-core CPUs by analysis-based transformations. *Frontiers of Information Technology & Electronic Engineering*, 16(1):899–916, 2015. (Cited on page 33)

[113] Canqun Yang, Feng Wang, Yunfei Du, Juan Chen, Jie Liu, Huizhan Yi, and Kai Lu. Adaptive optimization for petascale heterogeneous CPU/GPU computing. *Proceedings IEEE International Conference on Cluster Computing, ICCC*, pages 19–28, 2010. (Cited on page 40)

[114] Yongjian. Yu and Scott T. Acton. Speckle Reducing Anisotropic Diffusion. *IEEE Transactions on Image Processing*, 11(11):1260–1270, 2002. (Cited on page 163)