Universidad de Valladolid

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

DEPARTAMENTO DE INFORMÁTICA

TESIS DOCTORAL:

Hierarchical Transparent Programming for Heterogeneous Computing

Presentada por **D. Yuri Torres de la Sierra** para optar
al grado de
doctor por la Universidad de Valladolid

Dirigida por:

Dr. Arturo González Escribano
Dr. Diego R. Llanos Ferraris

Mayo 2014

Resumen

La computación paralela y el desarrollo de programas paralelos intentan reducir el tiempo de ejecución de las aplicaciones. Durante años, las optimizaciones de códigos secuenciales fueron diseñadas sin tener en cuenta la paralelizacion de datos ni de las tareas. Actualmente, los dispositivos multi-core se han vuelto omnipresentes en nuestras máquinas de cómputo, haciendo que la paralelización tome un papel aún más importante. La computación paralela está estrechamente relacionada tanto con el hardware como con el software. El objetivo final de la computación paralela es mejorar, en la medida de lo posible, la capacidad computacional de las máquinas.

El constante crecimiento en el rendimiento de las Unidades de Procesamiento Gráfico (GPUs), junto con recientes mejoras en su programabilidad, ha hecho que estos dispositivos sean una buena opción como aceleradores hardware en la Computación de Altas Prestaciones (HPC) para una variedad de aplicaciones.

La noción de la computación heterogénea ha emergido hace muy pocos años. Este concepto se basa en explotar un sistema compuesto por dispositivos de naturaleza diferente. Los sistemas heterogéneos pueden estar formados por procesadores con múltiples núcleos, GPUs y procesadores reconfigurables entre otros. Aunque el uso de sistemas heterogéneos, para aprovechar al máximo toda la capacidad computacional, pueda resultar una idea interesante, su complejidad de programación se encuentra un paso por delante de la programación paralela. Los principales problemas que aparecen son los siguientes: el primer problema es la necesidad de escribir código propios tanto para los núcleos de la CPU como para la GPU y el segundo problema, es la distribución de carga de trabajo entre los diferentes dispositivos, ya que estos sistemas no comparten un espacio de direcciones común. Además, cada dispositivo presenta diferentes capacidades de cómputo, por lo que cada uno es capaz de trabajar a ritmo diferente.

Durante la última década, se han propuesto diferentes modelos de programación con el objetivo de manejar la complejidad de las particiones y mapeo multinivel de datos. Estos modelos de programación pueden caer en dos categorías: aquellos que esconden las comunicaciones subyacentes y aquellos en donde las comunicaciones son impulsadas por las particiones creadas por el usuario. Estos modelos de programación paralela no ayudan al programador a expresar de forma explícita el modelo de comunicación que necesita el algoritmo independientemente de la partición de datos elegida. El Tiling es una técnica conocida que es usada para distribuir datos y tareas en programas paralelos y mejorar la localidad de bucles anidados en códigos secuenciales. El uso de estructuras de datos para soportar tiling, permite explotar eficientemente la jerarquía de memoria ya que a menudo los datos son reutilizados dentro del tile.

Trasgo es un framework de programación que está siendo desarrollado por nuestro grupo de investigación de la Universidad de Valladolid. Trasgo se basa en especifica-

ciones de alto nivel y paralelismo anidado, permitiendo expresar de forma sencilla varias combinaciones de paralelismo de tareas con un esquema común. El back-end de Trasgo es soportado por Hitmap, una biblioteca runtime para el tiling jerárquico y el mapeo de arrays. Hitmap implementa funciones para crear, manipular, mapear y comunicar tiling-arrays jerárquicos.

En esta tesis doctoral estudiamos la posibilidad de desarrollar un sistema de programación portable y transparente que incorpore tiling jerárquico y políticas de scheduling con el objetivo de aprovechar las capacidades de la computación heterogénea.

Para abordar nuestra propuesta de investigación hacemos uso de la biblioteca Hitmap como un framework prototipo. Este framework permite explotar sistemas de memoria distribuida. En este trabajo extendemos de forma conceptual y práctica este framework para aprovechar todos los recursos hardware (CPU-GPU) en entornos heterogéneos. Además, éste permite generar códigos abstractos que a su vez son transparentemente adaptados para sistemas con diferentes tipos de dispositivos.

Para ello es necesario también un estudio de las arquitecturas GPU que ayude a determinar unos buenos valores de los parámetros de configuración GPU (la geometría/tamaño de los bloques de hilo o la configuración de la cache L1) que deberían ser escogidos de otra forma por el programador. El conocimiento obtenido de este estudio es usado para crear políticas de selección de dichos valores. Estas políticas son incluidas en la biblioteca Hitmap.

Tras examinar y analizar los resultados experimentales concluimos que es factible la creación de un sistema de programación que incorpore técnicas de particionado de datos, herramientas de comunicación, y selección trasparente para el programador de buenos valores de los parámetros de configuración GPU para sistemas heterogéneos.

Palabras clave

APSP, herramientas de mapeo automático, tuneado automático de código, benchmarking, kernels concurrentes, CUDA, mapeo de datos, particionado de datos, Dijkstra, Fermi, GPU, GP-GPU, dispositivos heterogéneos, sistemas heterogéneos, Kepler, técnicas de mapeo, micro benchmarks, restricciones de tamaño de memoria, MPI, NSSP, OpenCL, OpenMP, algoritmos paralelos, bibliotecas paralelas, medición de rendimiento, modelo poliédrico, SSSP, geometría del bloque de hilos, tiling.

Abstract

Parallel computing and the development of parallel programs is a way to reduce the time of the program execution. During many years, sequential optimization was designed without thinking about parallel tasks. Currently, multi-core devices have arrived, making code parallelization more important. The parallel computing is closely related with both hardware and software point of view, in both cases, many calculations are carried out simultaneously. The final objective of parallel computing is the improvement in computing capacity.

The rapid increase in the performance of Graphics Processing Units (GPUs), coupled with recent improvements in its ease of programming, have made graphics hardware a compelling platform for High Performance Computing field (HPC) in a wide kind of applications. For this impressive processing potential, a single GPU has the sufficient power to compete with many super-scalar CPUs.

The heterogeneous computing notion has emerged few years ago. This concept references to exploit a system composed by multiple mixed compute devices. A Heterogeneous system can be composed by commodity multi-core processors, graphics processors and reconfigurable processors among others. Although the use of heterogeneous systems to take maximum advantage of all computing capabilities may seem a natural idea, their programming complexity is also one step beyond of the intrinsically complex parallel programming. Two main problems appear. First, the need of writing specialized code for both the CPUs cores and GPUs or other accelerators that are present in the system. Second, the problems related with data distribution among devices, and the associated load-balancing problem since heterogeneous systems do not share a common address space and present different computing powers.

During the last decade, different programming models have been proposed to handle the complexity of multilevel data partition and mapping. These programming models roughly falls into two categories: Those that hide the underlying communications, and those where the explicit communication is driven by the partition made by the user. These parallel programming models do not help the programmer to explicitly express the communication pattern needed by the algorithm regardless of the data partition chosen. Tiling is a well-known technique used to distribute data and task in parallel programs and to improve the locality of nested loops in sequential code. The use of data structures to support tiles allows to better exploit the memory hierarchy, since data is often reused withing a tile.

Trasgo is a programming framework that is being developed by our Trasgo research group at the University of Valladolid (Spain). Trasgo is based on high-level and nested-parallel specifications allowing easily express several complex combinations of data and parallelism tasks with a common scheme. One of the most important features is that

this model hides the layout and scheduling details. The Trasgo back-end is supported by Hitmap, a runtime library for hierarchical tiling and mapping of arrays. The Hitmap library implements functions to efficiently create, manipulate, map, and communicate hierarchical tiling arrays.

In this Ph.D. thesis we study the possibility of developing a portable and transparent programming system that incorporates hierarchical tiling and scheduling policies in order to take advantage of heterogeneous computing capabilities.

To accomplish our research proposal we take profit the of Hitmap library. Hitmap is used as a prototype framework that integrates a parallel computation model which takes profit of all available hardware resources (CPU-GPU) in heterogeneous environments. This framework allows to generate abstract codes which are transparently adapted to heterogeneous systems with mixed types of accelerator devices.

We present a study of the GPU architectures to help to determine good values of configuration parameters that should be chosen by the programmer. The knowledge obtained from this study is used to create proper policies of selecting configuration parameters values of GPU devices, such as, threadblock geometry/size and the configuration of L1. These policies are included in previous framework.

After examining and analyzing the experimental results, we consider the feasibility of creating a programming execution containing automatic data partitioning techniques, communication tools, and select transparently to the programmer, good values of GPU configuration parameters for heterogeneous systems.

Keywords

APSP, automatic mapping tools, automatic code tuning, benchmarking, concurrent kernel, CUDA, data layout, data partition, Dijkstra, Fermi, GPU, GP-GPU, heterogeneous devices, heterogeneous systems, Kepler, mapping techniques, micro-benchmarks, memory-size restrictions, MPI, NSSP, OpenCL, OpenMP, parallel algorithms, parallel libraries, performance measurement, polyhedral model, SSSP, threadblock geometry, tiling.

A mis padres, Ramón y Eva,
y a mi hermano José Ramón,
con gratitud y gran admiración.

Acknowledgments

After all these years of work, insomnia, work, fun, work, challenges, conferences, and..., work, here it is, my Thesis is finally completed. It has been a long, tough way, but if I could come back in time, I would definitely not change it for anything. Becoming a PhD has been my dream during the last four years, and during this long way I have found a lot of people that have helped me to make this dream come true.

First of all, I am deeply indebted to my supervisors, Dr. Arturo González Escribano and Dr. Diego R. Llanos Ferraris, for guiding me throughout the process of becoming a PhD. None of this would have taken place without the help and support of my supervisor, advisor, mentor, and friend Dr. Arturo González Escribano. Thanks a lot for guiding me, for the daily interaction, and for all your commitment, successful advises, and critical questions. I also owe thanks to my supervisor Dr. Diego R. Llanos Ferraris for all the work and fruitful discussions that challenged me, and for showing me the best way to achieve my goals.

I want to thank my colleagues from the Trasgo Group at University of Valladolid (Álvaro, Ana, Héctor, Javier, and Sergio). Sharing my time and experiences with you all has been very rewarding.

I want to thank the Universidad de Valladolid fellowships program (Formación Personal Investigador, FPI-2010, aplicación presupuestaria 180.113-541A.2.01-691) for providing me with the financial support necessary to carry out this work, that has been co-funded by *Banco Santander*. Thanks a lot for making it possible.

I would like to thank all my friends and dear family. *En particular me gustaría dar las gracias a mis padres Ramón y Eva, por todo el amor y cariño, por el esfuerzo y sacrificio, y por responder a todas mis necesidades con un rotundo sí a ojos cerrados. A mi gran amigo David Fernández, nos conocemos de toda la vida, y siempre me ha apoyado en todo de forma incondicional, por ello, le estaré eternamente agradecido. Muchas Gracias.*

Yuri Torres
Valladolid, 2014

Contents

List of Figures

List of Tables

Chapter 0

Resumen de Tesis

Este documento presenta mi tesis doctoral. Me inicié en paralelización en 2009 y desde entonces siempre intento superarme en este complicado pero a la vez apasionante mundo.

0.1 Objetivo de la investigación

0.1.1 Pregunta de investigación

Es posible desarrollar un sistema de programación portable y transparente que incorpore tiling jerárquico y políticas de scheduling con el objetivo de aprovechar las capacidades de la computación heterogénea?

0.1.2 Tareas

Con el objetivo de responder a la pregunta de investigación, hemos llevado a cabo las siguientes tareas:

- A partir de Hitmap, un framework que incorpora tiling y scheduling para sistemas de memoria distribuida, se han estudiado las modificaciones conceptuales y prácticas necesarias para dar soporte a Hitmap en entornos heterogéneos.
- Este estudio muestra la necesidad de nuevos niveles de particionado y sus políticas asociadas. Estas políticas incluyen mecanismos para:
    1. Mover datos de forma transparente entre los diferentes dispositivos.
    2. Realizar subdivisiones de tareas que encajen de forma adecuada en la memoria de cada dispositivo.
    3. Seleccionar tamaños y geometrías para los conjuntos de hilos.

0.1.3 Metodología de investigación

Con el fin de lograr los objetivos propuestos en este documento, seguiremos la metodología de investigación definida por un método de investigación para ingeniería [2]. Este método establece cuatro fases diferentes que el proceso de investigación debe seguir. Cada fase puede repetirse de forma cíclica con el objetivo de redefinir las soluciones propuestas.

- *Observar las soluciones existentes*. Esta fase tiene el propósito de detectar problemas que serán alcanzados durante el proceso de investigación comenzando con la solución existente. Esto conlleva un completo estudio del estado del arte con el objetivo de encontrar trabajos relacionados con nuestro foco de investigación. Este estudio es presentado en este documento.

- *Proponer mejores soluciones*. En esta fase se propone una solución que aborde las limitaciones encontradas en la fase anterior. Como mostraremos a lo largo de este trabajo, existe una carencia de frameworks que encapsulen técnicas de particionado y mapeo de datos para entornos heterogéneos, que a su vez, exploten de forma automática y transparente los recursos hardware de los dispositivos GPU. Este trabajo está estrechamente relacionado con la arquitectura de las GPUs. Proponemos varias políticas para (a) seleccionar buenos valores para los parámetros de configuración GPU (tamaño/geometría de los bloques de hilos y la configuración de la cache L1 entre otros) para algunos tipos de aplicaciones y (b), explotar de forma eficiente un balanceo de carga sobre los sistemas heterogéneos.

- *Construir o desarrollar la solución*. La solución propuesta en la fase anterior es implementada en la fase actual. Hemos llevado a cabo un estudio sobre las arquitectura de la GPU y como ayudar al programador a determinar unos buenos valores de los parámetros de configuración. También hemos desarrollado un framework de programación para estudiar la factibilidad de las soluciones propuestas.

- *Medir y analizar la nueva solución*. Finalmente, este método ingenieril establece que la solución propuesta tiene que resolver los problemas descubiertos en la primera fase. Hemos evaluado el sistema usando tanto bancos de pruebas sintéticos como aplicaciones reales.

0.2 Contribuciones y conclusiones

En este trabajo de Tesis se ha estudiado la viabilidad de desarrollar un sistema de programación portable y transparente que incorpore tiling jerárquico y políticas de scheduling con el objetivo de aprovechar las capacidades de la computación heterogénea. Para ello

hemos presentado un prototipo de framework que encapsula la elección de (a) buenos valores de los parámetros de configuración para dispositivos heterogéneos, (b) estructuras de datos tile, funciones de mapeo y balanceo de carga y (c), funciones de sincronización/comunicación entre dispositivos heterogéneos CPU-GPU. Finalmente, se realiza una evaluación experimental del prototipo de framework con el objetivo de contestar la pregunta de investigación propuesta en esta Tesis.

0.2.1 Contribuciones

Primera

Hemos contribuido al desarrollado Hitmap. Se trata de una biblioteca software diseñada para desacoplar los patrones de comunicación del particionado de datos gracias al uso de expresiones abstractas de comunicación. Estas abstracciones son automáticamente adaptadas en tiempo de ejecución dependiendo de la partición. Podemos usar las abstracciones desarrolladas para Hitmap en sistemas homogéneos para implementar unas nuevas orientadas a entornos heterogéneos.

1. Arturo Gonzalez-Escribano, Yuri Torres, Javier Fresno, and Diego R. Llanos. An Extensible System for Multilevel Automatic Data Partition and Mapping. Parallel and Distributed Systems, IEEE Trans. on Parallel and Distributed Systems, PP(99):1–1, 2013. [45].

Segunda

Hemos proporcionado nuevos conocimientos sobre la relación entre la ocupación, la geometría y el tamaño de bloque de hilos, la configuración de la jerarquía de la memoria cache (en la arquitectura Fermi), y el patrón de acceso a memoria global que presentan los hilos.

2. Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Understanding the impact of CUDA tuning techniques for Fermi. High Performance Computing and Simulation (HPCS), 2011 International Conference on, pages 631–639, 2011. [113].
3. Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. CUDA Tuning and Configuration Parameters on FermiArchitectures. Advanced Computer Architecture and Compilation for High Performance and Embedded Systems (ACACES 2011), 2011. [112].
4. Yuri Torres, Arturo Gonzalez -Escribano, and Diego R. Llanos. Uso del conocimiento de la arquitectura Fermi para mejorar el rendimiento en aplicaciones CUDA. Actas XXII Jornadas de Paralelismo, 2011. [114].

5. Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Using Fermi architecture knowledge to speed up CUDA and OpenCL programs. Proc. ISPA'12, Leganes, Madrid, Spain, 2012. [118].

Tercera

Hemos introducido una suite de microbenchmarks (llamada uBench) para explorar el impacto sobre el rendimiento de (a) criterios de selección de la geometría y tamaño de los bloques de hilos y (b), las configuraciones y recursos hardware de la GPU. Esta suite de microbenchmarks cubre los detalles hardware de las arquitecturas Fermi y Kepler.

6. Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Measuring the Impact of Configuration Parameters in CUDA Through Benchmarking. The 12th International Conference Computational and Mathematical Methods in Science and Engineering, CMMSE 2012, 2012. [116].
7. Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. uBench: Performance Impact of CUDA Block Geometry. Technical Report IT-DI-2012-0001, Depto. Informática, Universidad de Valladolid, Dec 2012. [117].
8. Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. uBench: exposing the impact of CUDA block geometry in terms of performance. The Journal of Supercomputing, 65(3):1150–1163, 2013. [120].

Cuarta

Hemos desarrollado dos aplicaciones reales (el problema SSSP para encontrar el camino/ruta más corto desde un punto a todos los demás; el problema APSP para encontrar la ruta más corta entre cualquiera de los puntos al resto) con el objetivo de probar y verificar las conclusiones obtenidas en [112, 113, 114, 116, 117, 118, 120].

9. Hector Ortega-Arranz, Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. A New GPU-based Approach to the Shortest Path Problem. The 2013 International Conference on High Performance Computing & Simulation, (HPCS 2013), pages 505–511, 2013. [92].
10. Hector Ortega-Arranz, Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. A Tuned, Concurrent Multi-Kenel Approach to the APSP problem. The 13th International Conference Computational and Mathematical Methods in Science and Engineering, CMMSE 2013, 2013. [93].

Quinta

Hemos presentado un framework de programación extendiendo la biblioteca Hitmap

con el objetivo de analizar la posibilidad de (a) crear un modelo de programación y un framework que encapsulen la elección de unos buenos valores de los parámetros de configuración de GPU y las estructuras de datos, (b) funciones de mapeo y balanceo de carga, y (c), funcionalidades de comunicación/sincronización entre dispositivos heterogéneos CPU-GPU.

11. Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Automatic Data Layout at Multiple Levels for CUDA. The 10th International Conference Computational and Mathematical Methods in Science and Engineering, CMMSE 2010, 2010. [111].
12. Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Data partition and syn- chronisation in heterogeneous systems. HPC-EUROPA2 project (project number: 228398) with the support of the European Commission - Capacities Area - Research Infras- tructures, 2013. [119].
13. Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Encapsulated Synchronization and Load-Balance in Heterogeneous Programming. Euro-Par 2012 Parallel Processing, volume 7484 of LNCS, pages 502–513. Springer Berlin Heidelberg, 2012. [115].
14. Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Automatic run-time mapping of polyhedral computations to heterogeneous devices with memory-size restrictions. In The 2013 International Conference on Parallel and Distributed Processing Techniques and Applications, 2013. [53].

0.2.2 Conclusiones

Basado en la información, discusiones y resultados que se muestran a lo largo de este trabajo de Tesis, las principales conclusiones son las siguientes:

- El uso de Hitmap promueve programas más abstractos, fáciles de mantener y codificar, manteniendo un buen rendimiento. Esta biblioteca permite lograr rendimientos similares a implementaciones que han sido optimizadas manualmente para kernels y benchmarks conocidos en la comunidad científica. Hitmap reduce significativamente el esfuerzo de programación.

- La elección de los valores de los parámetros de configuración de una GPU está estrechamente relacionada con la implementación particular de la aplicación y la arquitectura del dispositivo. Un análisis combinado del conocimiento de la arquitectura GPU y las características del código implementado (tales como el tipo de patrón de acceso a memoria global, el flujo de carga total por hilo y el ratio de

operaciones lectura-escritura sobre la memoria global) puede ayudar, de forma significativa, a seleccionar unos buenos valores para los parámetros de programación GPU.

- Es posible crear un entorno de programación que contenga técnicas de particionado automático, herramientas de comunicación, y seleccionar de forma transparente al usuario buenos valores de los parámetros de configuración GPU para sistemas heterogéneos.

0.2.3 Trabajo futuro

Existen varias cuestiones que nos gustaría abordar. Estos focos definen la orientación futura de este trabajo:

- Estamos planeando realizar un estudio sobre la influencia de los efectos hardware y los parámetros de configuración en otras arquitecturas de GPU, tales como AMD e Intel.

- Actualmente estamos trabajando sobre políticas de mapeo más sofisticadas que exploten de forma más eficiente los procesadores de la CPU y las arquitecturas GPU. Se desea probar y verificar la aplicabilidad de estas técnicas para un conjunto más amplio de problemas incluyendo otros benchmarks conocidos por la comunidad científica y aplicaciones reales.

- Personalmente, tengo curiosidad por estudiar los dispositivos FPGA (Field Programmable Gate Array) ya que su funcionalidad hardware puede ser reconfigurada tantas veces como se dese. Me gustaría automatizar aquellas decisiones que cualquier programador debería de tomar antes de lanzar cualquier función sobre un dispositivo de estas características. Desearía dar soporte a dispositivos FPGAs en nuestro marco de programación.

Chapter 1

Introduction

This document presents my Ph.D. Thesis. I started working on parallelization in 2009, always trying to move forward in this exciting and complicated world. This document summarizes my research efforts.

This chapter briefly introduces the reader in the field, showing both the context and motivation of this work and presenting the research question whose answer constitutes this Ph.D. Thesis.

1.1 Context

1.1.1 Parallel computing

The basic idea of parallel computing appeared around 1960. From the hardware point of view, at that point vacuum tubes had been replaced with transistors, and computers started to be more manegeable. That was the era of small-scale shared memory multiprocessors, with commercially-available systems made by Burroghs and IBM [30]. Since then, advances in processor architecture, memory systems, and network topologies, made parallel computing ubiquitous. Nowadays, even modest laptops have two or four processors.

Despite their availability, it is still difficult to make several processors to cooperate in carrying out a common task. Even with the huge number of processors available in modern manycore systems, some fundamental constrains such as Amdahl's Law, and practical issues, such as data partitioning coordination, and sharing, limit the performance of massively-parallel systems.

1.1.2 Multi- and manycore architectures

During the first decade of the new century, power and dissipation problems started to limit the clock frequency of single processors. At the same time, new technologies pushed for-

ward the limits on the number of available transistors per square unit. Faced with both situations, the natural choice of computer architects was to integrate in a single chip several copies of a given processor chip design. These resulting systems, called *multicore*, was comparatively simple to implement and manage, not only from the architectural point of view but also with respect to the operating system. Task schedulers were modified to issue processes to all available chips, quickly putting into production all hardware resources if the system have several or many pending tasks. However, multicore technology does not speed up the execution of a single task. On the contrary, multicore chips usually run at a fraction of the clock frequency of the original design, making sequential tasks slower.

The manycore paradigm also benefited from the number of available transistors. However, their design starting point is different. Manycore architectures were only designed with highly parallel applications in mind. They comprises several hundreds of very simple processors, frequently combining several multiprocessor units that internally work with a SIMD (Single Instruction, Multiple Data) execution model. Although manycore systems are even less useful than multicores to execute a single task, their specialized design made them the default choice for massively parallel applications.

1.1.3 GPUs for parallel computing

The most successful example of manycore architectures are Graphic Processing Units (GPUs). These devices were initially designed to manage floating-point graphical information to be sent to the user's display. Therefore, algorithms that were executed on a GPU had to be mapped into a graphics pipeline.

The search for other uses for the computing power provided by GPU devices, together with the addition of more general processing units per computing element, led in 2007 to the development of the CUDA platform SDK by NVIDIA [78]. The CUDA programming interface allows to express many sophisticated programs with a few, easy-to-understood abstractions. The development of CUDA has boosted the use of GPUs for general purpose (GPGPUs) in the execution of massively-parallel tasks not related with graphic processing.

Despite its popularity, CUDA is not the only choice for GPU programming. There are multiple languages to exploit these devices, such as BrookGPU [19], developed at Stanford University, and, more recently, OpenCL [60], a standard API for programming both GPUs and multi-core CPUs. Other GPGPU languages include Scout [71] for scientific visualization, and Intel Ct [102] for high-throughput architectures. It is interesting to note that both of them, together with Brook, are based on the shape concept introduced by the C* programming language [97] for the Connection Machine.

Although the different benefits of executing general-purpose applications on GPU devices have widely been recognized [61], in order to exploit efficiently the GPU potential,

and to predict the power consumption and performance, it is necessary to study in detail the device architecture. In any case, the programmer needs to specify an appropriate threadblock size-shape value for each GPU function, and to define the sizes of shared- and L1 cache memories in order to improve the applications performance. As we will see in this Thesis selecting good values of these GPU configuration parameters hinders even more the GPU programming tasks.

The scalability has been an attractive feature of graphics systems from the beginning. Although the benefits that multiple GPUs can provide are very hard to predict, the possibilities that even a single GPU offers make multi-GPU programming a natural follow-up. Predicting the execution performance of a system composed by multiple GPUs is still a challenge. Some authors, such as Dana Schaa [101], introduce basic models to predict the combined behavior of multiple GPUs in the execution of different applications, by identifying and classifying the major factors that affect their behavior. Other efforts include the Rigel [62] project, that takes into account the different CUDA constrains (such as shared memory utilization, thread fusing, or static work partitioning) to achieve high performance on NVIDIA GPUs.

1.1.4 Heterogeneous computing

The concept of heterogeneous computing [17, 76] is related to the combined use of different kinds of computing resources. It is a step further with respect to what we may call "homogeneous computing", where all processing elements share a common set of characteristics, regardless of the memory organization used (shared or distributed). In heterogeneous computing, systems with different processing capabilities are combined to collaborate in carrying out a given task. A classic example of such computing systems are shared or distributed- memory multiprocessors with GPU devices attached to them.

Although the use of heterogeneous systems to take maximum advantage of all computing capabilities may seem a natural idea, their programming complexity is also one step beyond of the intrinsically complex parallel programming. Two main problems appear. First, the need of writing specialized code for both the CPUs cores and GPUs or other accelerators that are present in the system. Second, the problems related with data distribution among devices, and the associated load-balancing problem since heterogeneous systems do not share a common address space and present different computing powers. All these problems, along with others, make programming for heterogeneous system a piece of craftsmanship, since there is a lack of general-purpose programming frameworks that handle this complexity. This Ph.D. Thesis aims to advance the knowledge needed to solve this problem.

1.1.5 Parallel programming and tiling models

During the last decade, different programming models have been proposed to handle the complexity of multilevel data partition and mapping. These programming models roughly falls into two categories: Those that hide the underlying communications (e.g. Chapel [27], UPC [22]), and those where the explicit communication is driven by the partition made by the user (e.g. MPI). These parallel programming models do not help the programmer to explicitly express the communication pattern needed by the algorithm regardless of the chosen data partition. Parallel programming tools and frameworks presented in the last years do not establish clear boundaries between virtual topologies, domain partitions, and tile management (such as HTA [13] or UPC), or clear boundaries between data management and communications (such as Chapel, UPC or HTA). Such a division of duties would allow the programmer to decouple the communication structures, that depend on the algorithm characteristics, from the data partition mechanisms.

Tiling is a well-known technique used to distribute data and tasks in parallel programs [125] and to improve the locality of loop nests in parallel and sequential code [124]. Although originally presented as a loop transformation technique, the use of data structures to support tiles for generic arrays allows to better exploit the memory hierarchy, since data is often reused within a tile. Tiling can be applied to multiple levels, to distribute work among processors at the outermost level, while locality are enhanced at the innermost level. In the context of distributed-memory, tiles can also make explicit communication, since computations involving elements from different tiles result in data movement [5, 18].

1.1.6 The Trasgo programming framework

Trasgo [43] is a programming framework that is being developed by the Trasgo research group at the University of Valladolid (Spain). Trasgo is based on high-level and nested-parallel specifications allowing easily express several complex combinations of data and parallelism tasks with a common scheme. One of the most important features is that this model hides the layout and scheduling details.

The Trasgo programming system supports its own programming model based on a simple process-algebra model and exploiting data-distribution algebras. Trasgo provides a high level parallel language. The main features of this new language are the following: (1) It is a nested parallel coordination language; (2) it uses extended function interfaces to allow the compiler detect data-flow; and (3), it uses an abstract and unified parallel primitive for coarse- or fine-grain logical tasks. The parallel expressions combine topology and data-distribution plug-in modules to automatically create data-domain partition and mapping.

This high level programming language is translated to an intermediate language called *SPC-XML* to allow the use of powerful XML tools to manipulate it. The framework includes a plug-in system with several modules that generate partition and mapping information, which may be used to create abstract communication codes independently of the mapping details. The framework uses a back-end to translate the internal representation to a parallel source code using the MPI message passing interface as a communication and synchronization layer.

1.1.7 The Hitmap run-time library

Hitmap [45], a runtime library for hierarchical tiling and mapping of arrays, gives support to Trasgo as its back-end. The Hitmap library implements functions to efficiently create, manipulate, map, and communicate hierarchical tiling arrays. Hitmap offers generic mapping functionalities that could be used to implement other tiling arrays solutions (such as HTA) as a special case. The Hitmap topology and layout plug-in system is flexible, extensible, composable at different levels, and supports irregular or load-balancing data partitions with a common interface. These features go beyond HTA functionalities [31, 41]. Hitmap also has a generalized hierarchy system, where a given branch of the hierarchy can be independently and recursively refined, in a dynamic way, to an arbitrary level.

I have participated in the development of Hitmap, and we decided to use it as testbed for our heterogeneous development. In this work we derive a new prototype of Hitmap that includes our proposals to take profit of the computational capability of heterogeneous environments, hiding to the programmer the details about the machine structure and thread management. As we will see, this allows to easily generate programs with multiple levels of parallelism in heterogeneous systems.

1.2 Purpose of this research

1.2.1 Research question

> *Is it possible to develop a portable and transparent programming system that incorporates hierarchical tiling and scheduling policies, in order to take advantage of heterogeneous computing capabilities?*

1.2.2 Tasks

In order to answer to our research question, we have carried out the following tasks:

- We have contributed to the development of Hitmap, a runtime library for hierarchical tiling and mapping of arrays in distributed-memory systems. Starting with the

tiling and scheduling framework for distributed-memory systems, we have studied the modifications needed to allow Hitmap to support heterogeneous environments.

- As we will see, this study shows the need for new partitioning levels and their associated policies. This include mechanisms to:

  1. Transparently move data across devices for GPUs.
  2. Perform task subdivisions to fit into the devices capabilities.
  3. Select appropriate shapes and sizes for thread sets.

- The final task is to integrate the proposed mechanisms and policies into an heterogeneous version of Hitmap, evaluating the resulting programming framework with existing benchmarks.

1.2.3 Research methodology

In order to accomplish the objectives proposed in this document, we have followed a research methodology defined by a research method for engineering [2]. This method establishes four phases that the research process has to follow. These phases can be cyclically repeated with the aim of refining the proposed solutions.

1. *Observe existing solutions*. This phase has the purpose of detecting the problems that will be addressed during the research process, starting with the existing solutions. It leads to a complete study of the literature, in order to find works related with our research. This study is presented in this dissertation.

2. *Propose better solutions*. In this phase, a solution that overcomes the limitations found in the previous step is proposed. As we will see, there is a lack of frameworks that encapsulate data partition and mapping techniques for heterogeneous environments, automatically and transparently, squeezing the underlying GPU hardware resources. These tasks are closely related to the GPU architecture. We propose several policies to (a) select good values of GPU configurations parameters for some kind of applications, and (b) exploit a good load balancing on an heterogeneous system.

3. *Build or develop the solution*. The solution proposed in the previous phase is implemented in this step. We have carried out a study about the GPU architectures and how to help the programmer to determine good values of configuration parameters. We have also developed a prototype programming framework to study the feasibility of the proposed solutions.

4. *Measure and analyze the new solution*. Finally, the research method for engineering establishes that the proposed solution has to solve the problems discovered in the first phase. We have evaluated the system using both synthetic benchmarks and some real-world applications.

1.3 Outline

This document is organized as follows. Chapter 2 shows the state of the art through citations and discussions. Chapter 3 presents a runtime library, named Hitmap, whose distributed-memory version will be used to implement new abstractions that support heterogeneous systems composed by CPU and GPU devices. In Chapter 4 we analyze the possibility of creating a programming model to encapsulate the selection of good values of GPU configuration parameters and tile data structures, mapping and load balancing functions, and synchronization/communication functionalities between CPU-GPU heterogeneous devices. As we will see in that chapter, to implement the desired abstraction levels for heterogeneous computing it is needed to arrive to a good data and task partitioning, that it turn leads to the need of an optimal choice of GPU parameters. This issue is covered in Chapter 5. The experimental evaluation of the solutions proposed is carried out in Chapter 6. Finally, Chapter 7, aims to answer our research question, summarizing our results and enumerating the works published during the development of this Ph.D. Thesis. Appendix A shows the fundamentals of the CUDA programming model used to develop the GPU codes, and Appendix B contains some algorithms used as benchmark in our experiments.

Chapter 2

State of the Art

This chapter presents the state of the art related to this Ph.D. Thesis. The citations and discussions included in this chapter is the pillar that supports the research work presented in this dissertation.

As we described in the previous chapter, the purpose of this work is to improve programability aspects of heterogeneous systems. Thereby, a broad study of these architectures and their related optimization techniques will be presented. In addition, we will describe tiling arrays supporting tools, that will be the foundation of abstractions to hide the parallel management of data structures in heterogeneous systems. Finally, we discuss the properties and features of some scientific community benchmarks we have selected for our experimental work.

2.1 Programming tools for heterogeneous systems

The notion of heterogeneous computing emerged many years ago [17, 76]. This concept is associated with exploiting a system composed by multiple mixed computing devices. We have to highlight the significant growth in the use of the scalable heterogeneous computing system composed by commodity multi-core processors, graphics processors and reconfigurable processors, among others [110]. In these years, heterogeneous computing are gaining more acceptance, while problems, like their energy management, are being gradually solved.

Several programming models can be jointly used to take profit of heterogeneous systems. For example, several works, such as [58, 65] use MPI and CUDA parallel programming models in order to exploit the GPUs devices present in an heterogeneous environment. However, these works do not abstract the use of both models with respect to the target underlying hardware details. Programming using only these tools can be a tedious task.

2.1.1 Programming languages for GPUs

There are multiple global-purpose languages to take profit of GPU capabilities. A first classification criterion is to split them on *Specific* and *General* GPGPU language classes.

A Specific GPGPU language can only be used for a concrete architecture, such as CUDA [61] for NVIDIA platform, and MIC [70] developed by Intel for their architectures. These languages are specifically designed to squeeze the use of the underlying hardware resources. Besides, the programmer has the possibility of using specific code tuning strategies to take even more profit of the target GPU hardware design. On the other hand, a General GPGPU language provides support for multiple vendor devices, such as BrookGPU [19] developed at Stanford University, and the recent OpenCL [60], a standard API for programming both GPGPUs and multi-core CPUs. These general GPU languages do not have specific support to exploit, in the most efficient way, all the GPU hardware resources. Besides, they generally limit the programmer when she wants to optimize a parallel code.

2.1.2 GPU tuning strategies

In our study we are interested in how to transparently reach a high level of efficiency exploiting and taking advantage of the most used GPU architectures, such as those developed by NVIDIA (pre-Fermi, Fermi and Kepler). This goal leads us to choose CUDA as study case for GPGPU languages. There are multiple code tuning strategies in CUDA model [61]. These strategies are currently more complicated to understand and develop than those found on commercial CPUs. Then, to squeeze the computational power of a GPU technology requires much more programming effort.

For example, the work by Wynters [128] shows a naïve matrix multiplication implementation where several threadblock sizes are tested on pre-Fermi architecture. The author says that this configuration has a significant performance impact when a parallel problem is executed on a NVIDIA GPU.

One of most common tuning strategies is to choose a threadblock size that maximizes the SM (Streaming Multiprocessor) Occupancy in order to reduce the memory latencies when accessing the global device memory [61]. The authors focus on block shapes that simplify the programming task, such as square shapes, or dimensions that are power of two (an important part of the search space is not considered).

As [118] states, it is very important to adapt the values of configuration parameters (such as threadblock geometry, shared memory size, and L1 cache memory state) to the particular memory access pattern to squeeze, as far as possible, the computational capability of the heterogeneous systems. Despite the valuable work done in tools, such as FLAME [96] and MCUDA [108], the authors still need to manually determine by trial-

and-error the best values of configuration parameters. The first cite ([96]) focuses on programming dense linear algebra operations on complex platforms, including multi-core processors and hardware accelerators, such as GPUs and Cell. The authors abstract the target accelerator dividing the parallelism in two levels, the first one considering each accelerator device as a computation unit (coarse-grain parallelism), and the second one considering each hardware accelerator as a set of multiple cores (fine-grain parallelism). In the MCUDA paper [108] the authors present a framework to mix CPU and GPU programming. In this work it is mandatory to define separate kernels for all available devices. No data distribution policy is provided, and the toolkit can not make any assumption about the relative performance of the supported devices. Introducing any of these features would involve a redesign of the framework.

There are works, such as [11, 67, 126, 130], that use advanced compilation techniques to transform high-level primitives, or constructors, into optimized CUDA code. However, these frameworks take only into account architectures that are currently deprecated. A simple performance model is introduced in [101], in order to obtain a methodology for predicting execution times of GPU applications in single and multi-GPU environments. However, the citations do not explore the relationships between the threadblock size and shape, and their impact on the hardware resource utilization, which derives in performance impact.

Maximizing the Coalescing factor is another tuning strategy that tries to overlap the communication and computation in order to reduce the latencies in global memory accesses. In [132] the authors show several global memory access strategies in an attempt to reduce these latencies. This work is developed on the pre-Fermi architecture and does not consider global programming parameters, such as L1 cache configurations, threadblock (size or shape) choice.

Other contributions, such as [6, 98], study less common tuning strategies. These works present several problem implementations, testing significant parameters such as unroll factors, prefetching, and work-per-thread among others. The performance obtained by modifying these factors aims to reduce the search space for the optimal configuration. A performance model is also provided. Pre-Fermi is the only considered architecture and the impact of the hardware effects are omitted.

Focusing on more modern architectures (Fermi), in [129] the authors show how the cache memory hierarchy helps to take advantage of data locality, thus, significantly improving the global performance. However, taking into account the cache hierarchy leads to a too much complicated performance prediction model. The authors study a particular benchmark where the shared-memory vs. transparent cache configuration is adjusted automatically depending on the amount of data assigned to threads. Again, the effects of the block size and shape are not considered. Finally, in [36, 59], for the same GPU architecture, the authors show several interesting metrics related to hardware architecture.

The authors use these metrics in an attempt to predict the performance of CUDA kernel codes once the block shape is manually chosen.

2.2 Challenges in heterogeneous programming

2.2.1 Data partition and load balancing techniques

Load-balancing methods for heterogeneous systems try to distribute the workload among computing units according to their computational capabilities. There are several load-balancing methods suitable not only for traditional systems, but also for heterogeneous systems. In the rest of this section a brief discussion of these methods is shown.

The use of some heuristics or data partition policies can be very useful to improve the load balancing in any environment. Some works, such as [54, 131], exploit at the same time CPU and GPU devices, attempting to obtain a good load balance. However, data-structures partition and manipulation is not abstracted and authors do not support flexible mechanisms to add new partition and layout policies.

Chapel [27], proposes a transparent plug-in system for domain partitions in generic systems. It tries to hide the communication issues to the programmer. Most of the times efficient aggregated communications can not be automatically derived from generic codes. Moreover, the authors do not offer specific support for accelerator partition policies, or synchronization among different CPU and GPU devices. The authors in [66] create a model to estimate the execution time of each task (based on the number of instructions and input data size), thus deciding which hardware would be the best for each case. The size of each single task is fixed at compilation time. In [66, 100] the authors calculate the data transfer time between the different devices (GPUs and CPUs), and also create a model in order to reduce inter-GPU and CPU-GPU communication.

Other authors [56, 122] try to assign bigger data portions to the most powerful devices. However, the data portion size has to be initially fixed. On the other hand, in [32], all tasks have the same size, and the number of tasks assigned to each hardware depends on the computing capacity of each device.

Graph dependencies are commonly used to discriminate which data-portion or area are more profitable for each target architecture. An input data-portion represents the minimum data unit scheduled on the heterogeneous hardware devices. This technique is used to represent any kind of dependencies between the different input data-portions, improving the load-balance factor in heterogeneous environments [20, 26, 34]. Nevertheless, the recovery of the necessary information to fill the graph can be time consuming. Works like [34] and [26] describe how to create a complete dependence-graph in order to classify as dependent or independent the application tasks. Only independent tasks are launched to GPU devices in order to reduce the costly data transfers through inter-GPU commu-

nications. In [20], the authors make a study of DDAs (data dependencies algebras) and use this technique to improve the load-balance between a cluster of GPUs minimizing the memory bottlenecks.

Finally, there are training methods to estimate the most costly computation portions, and select an appropriate device to execute them. These methods select a small subset of input data and execute the related part of the computation in the target system. The execution is monitored and performance results registered. The information is then used to predict the global execution time, which are the most powerful devices, or even which devices are more appropriate for each input data-portion.

2.2.2 Memory size restrictions

Taking into account the memory size limitations of heterogeneous target devices is an additional challenge. Currently, many approaches do not focus in this problem, working with fixed sized middle-grain tasks [32], or assuming that the tasks fit, or are generated to fit into the devices [14, 20]. Other approaches simply advise to add more computation devices to allow finer partitions [10]. A simple way to tackle the problem is to generate more distributed processes than system nodes, mapping several of them to the same device [115]. In this way, each process is responsible for a smaller part of the computation. When enough processes are launched, the parts are small enough to fit in any target device. However, this leads to more costly inter-process communications and scalability problems. A more sophisticated approach is to consider the device memory limitations while creating the high-level partition [12]. This approach highly complicates the whole partitioning activity.

An associated problem for memory-restrictions-aware systems is to find a proper representation of the parallel computation that allows the system to locate, and measure the size, of the data portions required by a generic part of the computation. This information is needed for both generating a balanced partition, and mapping the parts adequately, even for libraries that make transparent the node to device communication (e.g. [3, 12]).

None of these works propose a solution to allow a hidden layer to split an arbitrarily large computation in parts that fit the memory limitations of an assigned target device, or to transparently launch the partial computations generated to the target devices.

2.2.3 Tiling support

We have already discussed tiling techniques in Sec. 1.1.5. There is a lack of tiling support in most programming languages, with the exception of some data-parallel languages such as HPF [69] and PGAS languages (e.g. UPC [22]). Both supply some constructors to align and distribute data among processors. HPF offers a limited set of patterns computed

at compile time. Moreover, HPF does not offer a truly composable distribution mechanism, since it is not possible to apply a second data distribution over the local part of a previous distribution. For example, block-cyclic distributions can not be programmed as a composition of cyclic over block distribution.

Regarding PGAS languages, it is responsibility of the programmer to define and distribute tiles, frequently in terms of the number of processors or specific architecture details. This leads to the development of code that is hard to read and maintain. Finally, HTA [13, 49] is an elegant implementation of hierarchically tiling arrays in object-oriented languages as an abstract data type. However, this implementation does not supports irregular or load-balancing data partitions with a common interface.

2.3 Benchmarking

2.3.1 Micro-Benchmarking for GPUs

The use of micro-benchmarks to evaluate hardware configurations has a long tradition. There is not too much related work in the scientific community about the use of these benchmarks to understand the GPU hardware behavior. The GPU devices present significant performance changes when both, the problem implementation, and the values of configuration parameters (such as threadblock size-shape or, L1 cache state) are even slightly modified. In the rest of this section several approaches to micro-benchmarking techniques are discussed.

A set of micro-benchmarks is described in [109] in order to obtain measures related to architectural features, and basic program characteristics. These features and characteristics include vectorization, burst write latency, texture fetch latency, global read and write latency, ALU/Fetch operation. However, the study only focuses on the AMD GPU architecture.

The authors in [127] introduce a suite of micro-benchmarks to measure the performance of GPUs as well as the performance impact when a specific optimization strategy is used. The authors measure execution times and obtain the different latencies for the same threadblock configuration. In [4], the authors try to find the lower and upper bounds of the partition camping problem [47]. They present performance results related to the global memory read and write operations, with and without partition camping. As a result, the authors provide a spreadsheet that calculates an estimation of the partition camping problem for a given kernel. The authors do not consider the full range of threadblock size and shape choices. All these studies are only focuses on the first NVIDIA released architectures.

Several authors like [55, 133] have tried to develop performance analytic models that help the user squeezing the GPU computational capabilities. In [133], the model is based

on the results of a set of micro-benchmarks in order to measure the time of each kind of instruction, and the global/shared memory accesses. The authors always use the same threadblock shapes (square geometry) and extrapolate the memory data transfer bandwidth. In [55], the authors identify two main parameters related to the efficiency of the hardware resources use: (1) The time that a warp is waiting for data, and (2) the number of operations that could be done during these delays. The authors estimate the values by a set of micro-benchmarks. Again, different threadblock shapes have not been considered.

In summary, all these works do not systematically explore all the threadblock configuration space. Moreover, several of these tests have been conducted using deprecated GPU architectures, and do not relate the threadblock configuration with the underlying hardware effects.

2.3.2 Choice of benchmarks used in this work

This section briefly discusses the most relevant benchmarks used throughout this Ph.D. Thesis. A more complete description of the different benchmarks is included in Appendix B at the end of this document.

The desirable properties of benchmarks to test the different tuning strategies studied on this work are the following: (1) benchmarks should have data reutilization and different memory access pattern for each input data structure; (2) specific benchmarks should be designed to exploit distributed memory systems where each one is composed by devices of different nature; and (3), real-world benchmarks characterized as embarrassing parallel applications to avoid the communication overhead, where each individual input datum can be processed independently, are desirable. This set of benchmarks covers the most common kernel features for GPU implementation codes. Therefore, we will use them as relevant benchmarks to test the GPU hardware effects under the influence of its configuration parameters.

First, we will consider simple linear algebra kernels that represent typical coalesced access patterns. A good example of these benchmarks, is the matrix-matrix multiplication where the simplicity of the algorithm helps to better understand the GPUs behavior. One matrix-matrix multiplication variant (Cannon's algorithm [21]) is also used, whose main feature is to exploit distributed-memory systems. This method distributes data portions among all available hardware accelerators.

Second, we will use real-world applications. These benchmarks are characterized by being more complicated than the previous ones since they are composed by more than one kernel with very different features. The first real-world benchmark that we will use is the Single-Source Shortest Path (SSSP) problem, a classical problem of optimization. Many problems that arise in real-world networks imply the computation of the shortest path and its distances from a source to one or more destination points. Some examples

include car navigation systems [99] or traffic simulations [9] among others. Algorithms to solve the shortest-path problem are computationally costly, and in many cases commercial products implement heuristic approaches to generate approximate solutions instead. Although heuristics are usually faster and do not need a large amount of data storage or pre-computation, they do not guarantee the optimal path.

The basic solution for the SSSP benchmark is Dijkstra's algorithm [33]. This algorithm constructs minimal paths from a source node s to the remaining nodes, exploring adjacent nodes following a proximity criterion. Two parallelization alternatives can be applied to Dijkstra's approach. The first one parallelizes the internal operations of the sequential Dijkstra algorithm, while the second one performs several Dijkstra algorithms through disjoint sub-graphs in parallel [103].

Finally, the second real-world benchmark selected is the All-Pair Shortest-Path (APSP) problem, that is a well-known problem in graph theory whose objective is to find the shortest paths between *any* pair of nodes. There are two ways to solve the APSP problem. The first solution is to execute n times, where n is the number of graph nodes, a NSSP algorithm selecting a new node as source in each iteration. The classical algorithm that solves the NSSP problem is also the Dijkstra's algorithm [33]. The second solution is to execute an algorithm that globally solves the APSP problem using dynamic programming, as the Floyd-Warshall algorithm [39, 123]. The former approach is used for sparse graphs whereas the latter is more efficient for dense graphs.

Chapter 3

The Hitmap Library for Homogeneous Systems

In the Trasgo research group, we have developed a runtime library for hierarchical tiling and mapping of arrays named Hitmap. Our research plan includes using the Hitmap abstractions for homogeneous systems as starting point to design new solutions focused on heterogeneous environments. The abstractions already proposed in Hitmap library are a very appropriate base to create a programming model and framework for heterogeneous systems, to encapsulate the selection of good values of GPU configuration parameters, the management of tile data structures for accelerators, mapping and load balancing functions, and synchronization/communication functionalities between CPU-GPU heterogeneous devices (this programming model and framework will be shown in Chapter 4).

This chapter explores the details of the Hitmap library for homogeneous systems. A complete description of Hitmap features and functionalities are shown, as well as details about its design and implementation. Finally, we will measure to what extend the abstractions introduced by the library simplify the complexity of codes and also if they entail any significant performance penalties.

Hitmap is a highly-efficient library for hierarchical tiling and mapping of arrays. It aims at simplifying parallel programming, providing functionalities to create, manipulate, distribute, and communicate tiles and hierarchies of tiles. Besides, Hitmap presents the following functionalities: (1) Generates a virtual topology structure; (2) maps the data to different processors with chosen load-balancing techniques, automatically determining inactive processors at any stage of the computation; (3) identifies the neighbor processors to use in communications; and (4) allows to build communication patterns to be reused across algorithm iterations.

Hitmap was designed to decouple the communication pattern from data partitioning, thanks to the use of abstract expressions of the communications that are automatically adapted at runtime depending on the partition finally used. This library presents an unique

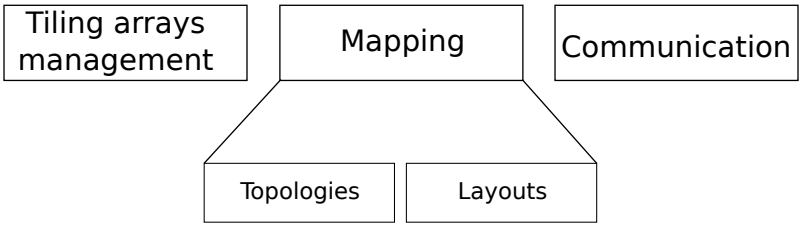


Figure 3.1: Hitmap library functionalities.

combination of features, including: (1) An extensible plug-in system, based on two different types of modules, to automatically compute data-partition and distributions of tiles as a function of the topology of the underlying architecture, hiding the details to the programmer; (2) a common framework to program new plug-ins with regular, irregular, static, or dynamic partitioning and load balancing techniques; (3) a flexible tool-set for data- and task-parallelism mapping with a common interface; (4) an API to create complex and scalable communication patterns in terms of an abstract partition and layout. Hitmap was designed with an object-oriented approach in mind. It internally exploits several efficient MPI techniques for communication, focusing on performance and on further native compiler optimizations. Hitmap can be used to support complex data structures, such as sparse matrices and graphs for irregular applications [41], using the same hierarchical tiling methodology. All these features allow to embed complex mapping decisions, some of them associated to compiler technology in a library. These features make Hitmap an excellent choice to develop higher-level programming models [44].

We will show how Hitmap achieves a good tradeoff between performance and memory usage. We will also compare Hitmap's programming effort with other alternatives.

3.1 Overview

3.1.1 Functionalities

The Hitmap library implements functions to efficiently create, manipulate, map, and communicate hierarchical tiling arrays for parallel computations. The library supports the following three sets of functionalities (see Fig. 3.1):

- Tiling functions: These operations allow to define and manipulate hierarchical array tiles. These functions can be used independently of the rest of the library functions to improve locality in sequential code, as well as to generate manually data distributions for parallel execution. Creation and tile selection functions typically receive as parameters the particular sizes and ranges of the tiles to be created from the input data structure.

- Mapping functions: This second set of functionalities implements the plug-in system for data distributions and layout functions, to automatically part an array into tiles, depending on a virtual topology selected. They define a common interface for static or dynamic partition techniques, or for fine or coarse grain techniques.

- Communication functions: Functions to create reusable communication patterns for distributed tiles. They are internally implemented using a message-passing distributed tool, such as MPI. These functions receive tiles and mapping information (including logical processes assignment and virtual topology information), and returns a handler that can be used to repeatedly send and receive tile data across physical processors.

3.1.2 Notations

In this section we introduce several key concepts and notations about arrays and tiles.

Signatures: We define a *Signature* S as a tuple of three integer elements representing a subspace of array indexes in a one-dimensional domain. It resembles the classical Fortran90 or MATLAB notation for array-index selections. The cardinality of the signature is the number of different indexes in the domain.

$$S \in \text{Signature} = (\text{begin} : \text{end} : \text{stride})$$
$$\text{Card}(s \in \text{Signature}) = \lfloor (\text{s.end} - \text{s.begin})/\text{s.stride} \rfloor$$

Shapes: We define a *Shape* h as a n-tuple of signatures. It represents a selection of a subspace of array indexes in a multidimensional domain (multidimensional parallelotope). The cardinality of the shape is the number of different index combinations in the domain.

$$h \in \text{Shape} = (S_0, S_1, S_2, ..., S_{n-1})$$
$$\text{Card}(h \in \text{Shape}) = \prod_{i=0}^{n-1} \text{Card}(S_i)$$

Our *shapes* unifies Chapel dense and stride domains [27] in a single type.

Tiles: We define a *Tile* as an $n-$dimensional array. Its domain is defined by a shape, and all the elements belong to a given type, provided by the chosen programming language.

$$\text{Tile}_{\,h \in \text{Shape}} : (S_0 \times S_1 \times S_2 \times \ldots \times S_{n-1}) \rightarrow < \text{type} >$$

A tile can represent a whole array, or a subset of array elements, determined by a subset of its index domain, expressed as a shape.

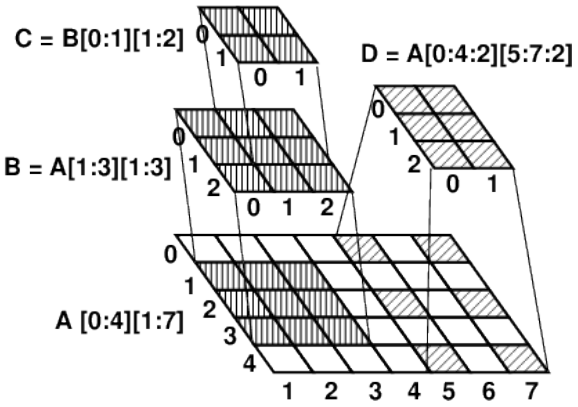


Figure 3.2: Tiling creation from an original array.

3.2 Tiling functionalities

In Hitmap, tiles are implemented with an abstract data type: *HitTile*. To use a new tile, it should be declared first as a *HitTile* variable, providing its dimensions and index ranges. This information constitute the *domain* of the array. See array A in Fig. 3.2, where we use a notation to specify dimensions and ranges that resembles Fortran90 and MATLAB conventions.

A new tile can be derived from another tile, specifying a *subdomain*, which is a subset of the index ranges of the parent tile. A *subtile* is indeed a tile with the same properties. The user can access the elements of the original array using two different coordinates systems, either the original coordinates of the array, or new tile coordinates starting at zero in all dimensions. See arrays B and C in Fig. 3.2, that are surrounded by their local coordinates indexes. We provide different functions to access elements and/or specify new subtiles using any of both coordinate systems. Tiles may also select subdomains with *stride*, transforming regular jumps in the original array indexes to a compact representation in tile coordinates. See array D in Fig. 3.2.

Tiling allocation. Tiles are not automatically allocated. Instead, we provide a function that allocates memory for a tile on demand. Accesses to a tile which has been already allocated are solved referencing its own memory, no matter if the coordinates system being used is the one belonging to the tile or to its ancestor. Accesses to tiles without their own memory are mapped to the *nearest ancestor* with allocated memory. The corresponding indexes are mapped transparently to the ancestor coordinates. This tile allocation system greatly simplifies data-partition and parallel algorithm implementation. For example, the programmer may define a global array without allocated memory, and create derived tilings directly or even recursively based on it. Only the subtiles which are needed locally should be allocated, while *all* the tiles generated have their associated tiles and array coordinates spaces. Subtiles with their own allocated memory never lose the reference to

the parent tile or array. Thus, the library provides functions to update data elements of an ancestor tiles with the values of the allocated memory containing elements also located on the ancestor, or vice-versa. This is useful to create double buffers or shadow copies for temporal use.

Tiling overlapping and range extension. Given an initial tile, it is possible to define two children tiles whose indexes *overlap*. Allocating overlapping tiles is a natural and easy way to generate local buffers for ghost zones in parallel stencil-based algorithms, where each cell should be updated taking into account its neighbors [64]. Hitmap also allows to define tiles that extend out of the range of the original array. Those elements outside of the original range can not be accessed unless memory is allocated for the new extended tile. Combining overlapping and extended tiles it is easy to implement boundary conditions and stencil operations in finite-element methods.

Multilevel tiling. The mechanisms shown above allow to create hierarchies of tiles with contiguous or regular-stride subselections. The access time to elements at any level of the tile hierarchy is uniform. For more generic, irregular tile hierarchies, or other more complex data structures, it is possible to define tiles with elements that are also of *HitTile* type. These *supertiles* store arrays of pointers to other tiles. We use them to store matrices by blocks, a useful solution for several linear algebra programs as discussed in [23]. The access time to elements of such a tile may be non-homogeneous, as adjacent indexed elements may be allocated at different referenced tiles, belonging to different levels on their respective hierarchies.

3.3 Mapping and communication functions

Other members of the research group contributed to Hitmap library with a layer for mapping functions. This layer will be used in combination with our proposal described in Chapter 4 to allow heterogeneous programming using distributed and shared memory environments with GPU accelerators.

Hitmap encapsulates all partition and mapping logic into separated modules avoiding the need to reason in terms of the number and identification of physical processors in the application code. One of the key characteristics of Hitmap is that clearly splits the mapping process in two independent parts, which have been observed to be related to different parallel algorithm features. *Topology* functions create virtual topologies using internal data, thus hiding the physical topology details. Data partition is done by a *Layout* function, that distributes domain indexes on a given virtual topology. The combination of a topology and a layout function automatically organizes the physical processors in a virtual topology, and automatically assigns a part of an index domain part to the local virtual processor.

Functions already available in Hitmap implement different topologies, such as grids of processors in several dimensions, or processor clustering depending on weights provided at run-time. All virtual topologies in Hitmap define a group of active processors, that may be hierarchical or even recursively splitted.

Layout functions implement partitions, and are also selected by name. The layout modules receive a virtual topology, the layout function name to be used, and a specification of the domain of the data structure to be distributed. The plug-in system allows to define specific parameters for a Layout module, that are provided by the application programmer when selecting the layout function name. These mapping modules return a single structure with: (*i*) the part of the domain mapped to the local processor (which may be defined and allocated using the Tiling functions), (*ii*) a mapping method to obtain information about other processors parts if needed, and (*iii*) information about virtual processors neighborhood for this specific mapping and virtual topology.

The layout building process also creates a more sophisticated neighborhood concept, taking into account that not all virtual processors may have been assigned data to compute. This solution allows neighbor communications to skip unassigned virtual processors transparently. This happens, for example, in V-cycle iterative PDE solvers, such as the MG program in the NAS Parallel Benchmarks [8, 40].

The Hitmap library supplies an abstraction to communicate selected pieces of hierarchical structures of tiles among virtual processors. It provides a full range of communication abstractions, including point-to-point communications, paired exchanges for neighbors, shifts along a virtual topology axis, collective communications, etc. Hitmap approach encourages the use of neighborhood and tile information automatically computed by Layouts to create communications which are automatically adapted to topology or data-partition changes.

The information needed to issue the real communication among physical processors is stored at run-time in a single data type named *Comm*. Abstract communication objects may be grouped in another data type named *Pattern*, generating reusable combinations of communication structures.

3.3.1 Combinations of topology and layout functions

Traditional message-passing interfaces, such as MPI, allow to create virtual processors topologies indicating their particular parameters. Hitmap provides a mechanism to create and select virtual topologies. The layout functions are used to map data-domains to virtual processors. Classical partition functions implementation are focused on splitting and assigning more data elements than processing resources. The Hitmap implementations deal also with the opposite case, to allow their use in hierarchical compositions of data and task parallelism.

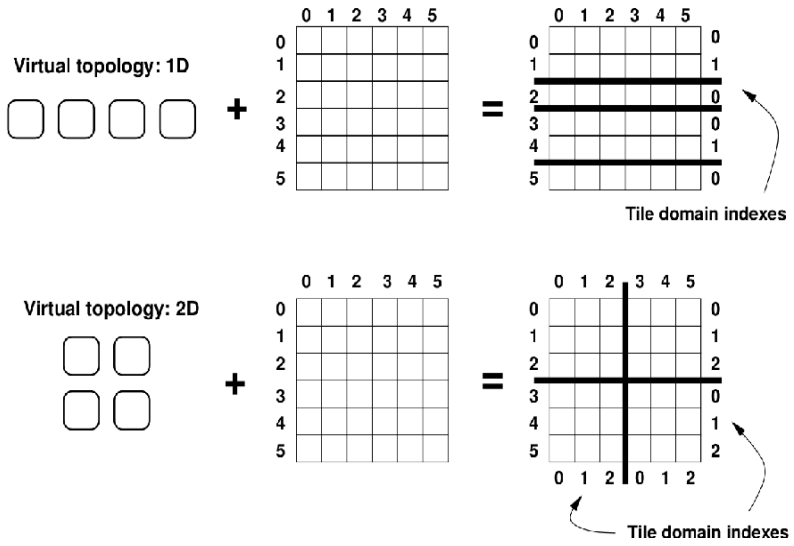


Figure 3.3: Different partitions automatically computed for different virtual processor topologies.

In the upper part of Fig. 3.3, a topology function has generated the requested one dimensional topology using the available processors. Given that topology and the domain of a tile, the partition of that tile and their assignment to different processors are computed automatically. If the programmer selects a *blocks* layout function, a one dimensional band partition is arranged dividing the odd numbers evenly among virtual processors. The lower part of Fig. 3.3 shows another example that uses a 2D topology, where *the same* layout function generates two dimensional blocks.

The correctness of some parallel algorithms depends on constraints on the topology or layout functions. For example, Cannon's algorithm for matrix multiplication is designed to work with a perfect- square mesh of tasks, since the topology must have $N \times N$ virtual processors. But the data-pieces produced by the partition of the matrices may have any kind of block shape. For this algorithm, the chosen layout function will surely have an impact on the performance, but not on correctness. On the other hand, the algorithm of the NAS MG benchmark [7] computes a 3-dimensional stencil convolution, leading to a 3D partition and communication pattern. However, this partition and pattern may be mapped on different 1D, 2D, or 3D arrangements of processors. In general, when an algorithm is tied either to a specific topology or to a specific layout function, the other one can be freely changed. Hitmap allows to test new topology/layout combinations with little effort, by changing only the name of a plug-in function in the whole code.

Regarding robustness issues, expressing communications in terms of layouts greatly helps the programmer to build deadlock-free communication patterns. Moreover, the automatic marshalling for tile communications helps to avoid programming errors.

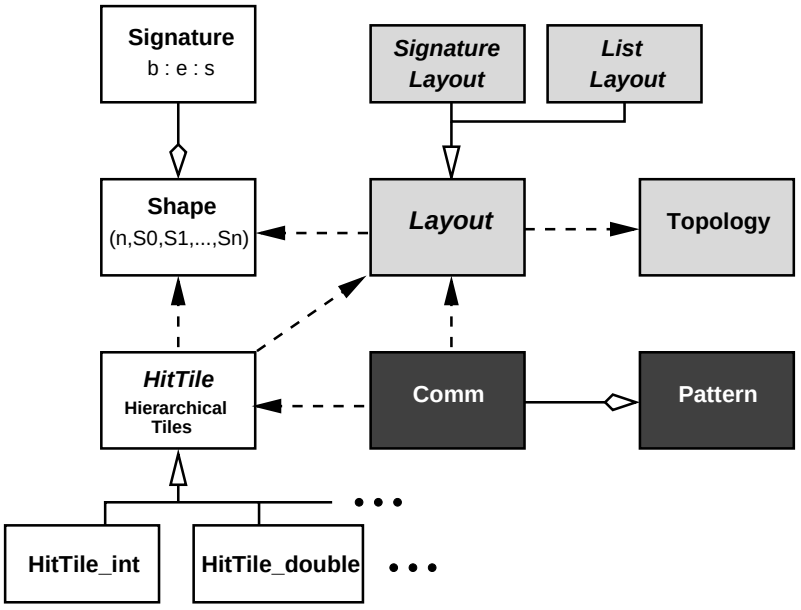


Figure 3.4: UML diagram of the architecture of the Hitmap library.

3.4 Design and implementation

In this section we will dive into the design and implementation details of the Hitmap library. Although Hitmap has been designed with an object-oriented approach, the current implementation has been written in C, to better exploit the Trasgo group members expertise in classical C compiler optimizations. The development of a C++ object-oriented interface for Hitmap is straightforward.

Figure 3.4 shows the UML classes diagram. Recall that the UML diagrams show dependencies as dotted arrows, where the arrow points to the class used, or referenced by, the class located at the tail of the arrow. Diamond headed arrows indicate that objects of the class located at the head of the arrow are composed by several objects of the tail class. The lines with a regular white arrow head indicate classes inheriting from an abstract class, whose name is depicted with italic font. The classes in white boxes implement tiling functionalities; the classes in light gray implement mapping functionalities; finally, the classes in dark gray implement communication functionalities.

3.4.1 Tiling classes

The classes *Signature, Shape*, and *HitTile*, define three data-types to support multidimensional and hierarchical tiling arrays. The library includes a macro function to specialize the *HitTile* data structure for any base type. The *HitTile* constructor only defines the tile domain:

```
hit_tileDomain( &tileName, type, #dimensions, cardinalities );
```

Available methods for *HitTile* data structure are described below:

- Data allocation: The *HitTile* data memory can be allocated at any time. The *HitTile* structure does not require any data allocation until programmer begins to make data accesses. We show an example of *HitTile* data allocation in the following code line:

```
hit_tileAlloc( &tileName );
```

- *HitTile* cloning: A tile data structure can be cloned independently of the *HitTile* data memory state through the following primitive:

```
hit_tileClone( &clonedTileName, &tileName );
```

- Access to elements: The data accesses may contain strides in any dimension of the data array. The elements location in memory have to be calculated using the coordinates, tile dimensional cardinalities, and taking into account the stride values.

  We show below an example of the use of a macro to access a 2-dimensional array elements. We also show the implement of the macros that allow the access using information in the tile variable.

```
hit_tileElemAt(tileName,2,rowIndex,columnIndex);

#define hit_tileElemAt(var,ndims,...)\
        hit_tileElemAt##ndims(var,__VA_ARGS__)
#define hit_tileElemAt2(var, pos1, pos2)\
    ((var).data[(pos1)*(var).qstride[0]* \
    (var).origAcumCard[1]+ (pos2)*(var).qstride[1]])
```

  The *qstride[dimension]* structure stores a stride value per each tile dimension and *origAcumCard[dimension]* contains the cardinality of each tile dimension multiplied by all its previous ones. The operations to calculate data coordinates with stride, when the programmer is using an array with no stride (that is, stride == 1), lead to performance degradation due to extra arithmetic operations. We provide a specific data access macro for tile class where stride is one. Thus, each data access is solved faster. We can see it in the following example:

```
#define hit_tileElemAtNoStride(var,ndims,...)\
            hit_tileElemAtNoStride##ndims(var,__VA_ARGS__)

#define hit_tileElemAtNoStride2(var, pos1, pos2)\
            ((var).data[(pos1)*(var).origAcumCard[1]+(pos2)])
```

For two dimensional accesses, it is used like in the following example:

```
hit_tileElemAtNoStride(tileName,2,rowIndex,columnIndex);
```

- Creation of sub-tiles (sub-selections): The programmer can define a new *HitTile* specifying with a shape an index subdomain of the original tile as follows:

```
hit_tileSelect( &tileChild, &tileParent, hit_layShape);
```

- To-from ancestor update operations: Each *HitTile* can update its data to the closer relative with allocated memory (we provide alternatives for ancestor or descendant). Internally, they use the *memcpy* C primitive, thus, the data contiguously stored in the last dimensions, can be copied in a single operation if the stride value is equal to the unit. This kind of update significantly reduces the data transfer between the relatives in a *HitTile* hierarchical structure. An example of update operation call is the following:

```
hit_tileUpdateToAncestor( &tileChild );
hit_tileUpdateFromAncestor( &tileParent );
```

3.4.2 Data partition and mapping subsystem

Mapping classes (*Topology* and *Layout*) are used for data distribution and mapping. *Topology* and *Layout* are designed as abstract classes. Each new topology function or data partition technique is implemented as an extension of one of these classes, providing the behavior of its abstract methods. A header file and a skeleton code in C are provided for both topologies and layout functions. Thus, new techniques may be easily programmed and compiled externally to the library, using them as plug-ins at compile time.

3.4.3 Topologies

Topologies are used from the application code invoking a constructor-like function *hit_topology (<name>)*. It receives only one parameter indicating the name of the chosen plug-in. The Topology class has only one abstract method. A new topology plug-in is implemented as a C function with a special name prefix. Topology functions receive an internal *HitPTopology* (physical topology) structure, containing information and details about the physical processors and the platform. This information is either automatically obtained by the library during initialization (e.g. querying MPI about the number of available processors and the local processor identifier), or provided statically in a platform

configuration file when an automatic query can not provide this information (e.g. the information about relative computing performance of each processor). The topology function fills up and returns a *HitTopology* structure. Currently, the internal representation supports mesh topologies for any number of dimensions.

3.4.4 Layouts overview

Data-partition and mapping is done by classes inherited from the *Layout* class. Hitmap introduces a generic layout constructor tile function, *hit_layout()*. This function receives at least three parameters: (a) The name of a plug-in that implements the particular layout to be used; (b) a virtual topology created with *hit_topology()*; and (c) a shape, either created on the fly or extracted at runtime from any tile, representing the domain to be mapped data size. Each layout function may define further compulsory parameters if needed. The *hit_layout()* function returns a *HitLayout* structure which may be queried for the generated mapping information. If there are less domain elements than processors on any dimension in the virtual topology, the layout function transparently determine the active virtual processors and assigns domain elements to virtual neighbors, without manual intervention.

The Layout class defines a common interface for both regular and irregular data-partition techniques. We define two different inherited abstract subclasses for layout functions implementations. *Signature Layout*s are more appropriate for regular partitions, because they describe the relationship between processors and data indexes using signatures. *List Layout*s are more appropriate for irregular partitions. Instead of using signatures, they implement generic mapping algorithms that associate lists of indexes to processors. The latter are more generic, but not as efficient as shapes to represent big quantities of indexes organized in regular (signature) form.

3.4.5 Layout plug-ins implementation

To define a new signature layout, the programmer should provide a function to compute the partition in one dimension. The function fills up an output signature with the local part for the local processor, and returns true/false to indicate whether a part has been assigned, or the virtual process should be marked as non-active for communications using this mapping. For example, a generic block partition function in a signature layout plug-in receives four parameters: The local virtual processor index (or rank), p; the number of virtual processors on this dimension, P; the signature of the shape in this dimension to be divided, $S = (b, e, s)$; and a pointer to the resulting signature object, S'. The local part of the signature assigned to this processor, $S' = (b', e', s')$, is calculated according to:

$$b' = \frac{p * \text{Card}(S)}{\min(P, \text{Card}(S))}$$

$$e' = \left(\frac{(p+1) * \text{Card}(S)}{\min(P, \text{Card}(S))} - 1 \right) \cdot s + b$$

$$s' = s$$

For this example, if there are less index elements in the signature than the number of processors, the last processors remain inactive. It is necessary to add a condition to handle this situation.

After defining the layout function, the programmer should define a plug-in that uses it. This plug-in simply declares the function to be used and their properties, and calls a method, provided by the library, that applies the function either to each dimension of the input shape present in the topology, or to a selected dimension (chosen by the application programmer with an optional parameter when calling the plug-in). Therefore, the library internally handles all the interactions between the topology and the layout function.

The shape calculations implemented in this plug-in provide valid outputs for any combination of input parameters. Many parallel algorithms, both in literature and in real implementations, assume cardinalities that are powers of 2, or input shapes that are multiples of the number of processors, thus generating simpler codes for partition and communication. With our approach, this complexity is encapsulated in the plug-ins, making the general-case implementation of algorithms as simple as the restricted ones.

We also implement an optional wrapping flag, to generate toroidal neighbor relationships. Thus, all the complexity of detecting neighbors in the general case is again encapsulated in the plug-in, not in the application code.

The *HitLayout* structure returned by the library contains information automatically generated only for the local part. Pointers to the signature and neighbor functions are also stored in the structure, to generate neighbor or non-local parts information on demand. Thus, the amount of local information is fixed, instead of growing with the number of processors, allowing a better scalability.

Writing a new signature plug-in is straightforward. It is enough to implement the corresponding formula into a layout function. List plug-ins, on the other hand, are more difficult to develop, because they are implemented using an algorithm instead of a signature formula. The *HitLayout* structure contains a C union with different internal data for Signature and List layouts. Hitmap provides a common API to query most of their internal information. The most relevant change is that signatures are internally substituted by lists that map each processor to a collection of associated indexes that do not need to follow a regular pattern.

In summary, the development of a plug-in consists in encapsulating in our abstract API the functions or algorithms that the programmer would otherwise hardwire into the application code, an improvement in terms of reusability with a negligible performance penalty.

3.4.6 Groups and hierarchical partitions

Layout functions may create *Groups*. A group associates a collection of virtual processors together, which are considered a single virtual processor. Layout functions can assign a part of the original shape not only to a processor, but to a group. The part assigned can be a signature-based shape or an index list. This allows the processors in the same group to use further levels of partition (new layouts) on their assigned sub-domain of indexes. This is useful for recursive data-domain decompositions, or for mapping small quantities of highly-loaded tasks with more inner levels of parallelism.

Each group has a leader processor. It is the current active processor of the group, the one doing the serial computations before further data partitions are used, and the one issuing communications to neighbor group leaders if needed. Group management is almost transparent to the programmer. Hitmap provides a macro function including a conditional structure that executes a program block only if the local processor is the group leader in a given layout.

3.4.7 Topology and layout techniques currently implemented

Typical available topology plug-ins are currently included in Hitmap. The first one is *plain*, representing a one-dimensional arrangement. This virtual topology simply enumerates the available processors. The second is mesh, *mesh*, that arranges all available processors P into an n-dimensional mesh, for a given n supplied as parameter. It is based on a prime-factors decomposition. It tries to balance the number of processors on each dimension. If P is prime, it falls back to a $P \times 1 \times 1 \times ...$ arrangement. Finally, we have *square*, that is similar to *mesh* in two dimensions, but arranges as many processors as possible into a perfect-squared mesh, leaving the rest of processors available for other parallel routines.

Signature-based layout functions include several *blocks* functions, with different policies to allocate the group leaders (active processors), and a *cyclic* function. We do not need to introduce a explicit *block-cyclic* function, as it may be generated in two-levels using first a blocking function, and then a cyclic layout to distribute the blocks. In [31] this composability layout property is described, explaining how to use it to efficiently implement for example an LU factorization.

Lists-based layouts include two techniques for load-balancing. The first one is based

on the partition needed by bucket sort algorithm implemented in the IS NAS benchmark, where it is used to redistribute data buckets in terms of the buckets sizes. The second is a similar technique, also using extra information about arbitrary loads that can be for example cardinality domain indexes. It may associate non-neighbor virtual processors to the same group to create a smoother load balance on non-symmetric systems. Both techniques for load balancing are also useful in recursive decomposition algorithms, such as Quicksort.

3.4.8 Communications implementation

The implementation uses several features of the MPI communication library. Communications across virtual processors are encapsulated in the *Comm* objects. A Comm object stores information about either a single operation, a pair of send/receive operations, or a collective one. We provide different constructors for different communication operations. The constructors have a very similar interface with the following parameters, some of them optional for certain communication types: (a) Sending and/or receiving tile buffers (tile subselections); (b) sending and/or receiving virtual processes (expressed as indexes or neighbor relations); and (c) a Layout object with the information about neighborhoods generated by the data distribution over the virtual topology. Internally, the MPI communicator is retrieved from the layout. The Layout constructor generates and stores in the layout object a particular MPI communicator that contains only the processors that have associated data domains after the domain distribution.

The pointers to the sending and/or receiving tile data buffers are stored in the object. The structure of the sending or receiving tiles is examined to generate MPI-derived data types that represent the tile data displacements. Any hierarchical HitTile subselection can be represented by a combination of *contiguous* and *vector* MPI-derived data types. Tiles with base elements that are also tiles need also to combine the previous types with *struct* MPI-derived data types. The result is a single, combined type that is committed and stored in the Comm object. Thus, buffering, marshalling, and unmarshalling of data is automatically managed by the MPI layer in the best possible way.

The programmer may directly provide values for the sending/receiving virtual process parameters. Nevertheless, Hitmap methodology encourages the use of the Layout methods for calculating neighbor processes indexes. In this way, a change in the real topology, in the policies selected for the virtual topology or layout building, or in data sizes, is automatically captured in the communication objects during their construction, storing different real processor indexes, or different MPI-derived data types for the data location.

Once built, the Comm object contains all the information to issue the real data transfer as many times as needed. The Comm class provides a method to activate the communication in synchronous (normal) mode, and two methods to start and end the communication

at different points of the code, allowing to implement an asynchronous mode.

Communication Patterns are implemented as a queue of Comm objects. The same activation modes are provided for Patterns. Associated methods simply traverse the internal queue calling the corresponding method on each Comm object.

Although the current backend implementation relies on the rich API of MPI, these functions are abstract enough to be ported to other backends.

3.5 Experimental evaluation of Hitmap

Experimental work has been conducted to show that the abstractions introduced by the library do not only simplify the complexity of codes, but also they do not entail significant performance penalties.

3.5.1 Design of experiments

We have designed experiments with the following guidelines: (1) Choose parallel applications which are well-known and representative of important applications classes and programming paradigms. They should present different challenges, and imply the use of different library resources. (2) Obtain or generate a manually programmed and optimized version of each application in C language to be used as reference, since some of the selected benchmarks may be originally in Fortran. The Fortran codes should be manually ported to C. (3) Write a new code version based on Hitmap. (4) Execute both versions with the same inputs and conditions on selected machines, and (5) compare the codes and the execution times obtained.

The codes have been run on two different machines which represent two different types of common architectures: A multicore, and a distributed memory cluster of commodity computers.

The first one, Geopar, is an Intel S7000FC4URE server, equipped with four quad-core Intel Xeon MPE7310 processors at 1.6GHz and 32GB of RAM. Geopar runs OpenSolaris 2008.05, with the Sun Studio 12 compiler suite. The second architecture is a homogeneous Beowulf cluster of up to 36 Intel Pentium 4 nodes, interconnected by a 100Mbit Ethernet network. The MPI implementation used in both architectures is MPICH-2, compiled with a backend that exploits shared memory if available.

The codes were instrumented to measure the execution times, distinguishing between the execution of the main computation part, the communication times, and the creation of data-partition, mapping, and communication information and structures. Thus, we may fairly compare the performance of the library and its applicability in real cases.

The benchmarks chosen include: Two programs from the NAS Parallel Benchmarks [8] (the MG multigrid program, and the IS integer sort program); an LU factorization and

back-substitution solver based on the ScaLAPACK package [15]; and a matrix multiplication kernel based on the generalized Cannon's algorithm.

3.5.2 Performance comparison

The execution times obtained for different versions of the benchmarks evaluated are shown in Figs. 3.5, 3.6, and 3.7. The times include the stage of computing tiling hierarchies, mapping, and communication information. According to the ScaLAPACK documentation, the ScaLAPACK's LU factorization implementation does not scale well if the interconnection network can not deliver several messages simultaneously, such as Ethernet. Thus, this benchmark is not suitable for the Beowulf cluster, and we did not carry on experiments for this program and platform. In the Geopar machine, when all 16 processors are used, the operative system, the MPI daemon, and the computations interfere with each other, producing additional context changes and cache misses on at least one core, delaying the overall computation. Thus, some applications exhibit a scalability limitation for 16 processors in Geopar. None of the experiments have led to incorrect results or runtime errors of any kind.

Tiling efficiency

Results show that the use of Hitmap library does not imply a significant performance penalty, being less than 8.5 % in the worst case (LU Scalapack). Sequential time measures for the Cannon's matrix multiplication and the LU factorization programs show that the accesses to tile elements in Hitmap are almost as efficient as direct memory accesses. Thus, the tiling abstraction layer does not introduce appreciable performance degradation. Moreover, the efficient management of MPI derived data types and reutilization of communication patterns produces positive effects.

In general, the cost of initializing Hitmap data-structures, layouts and communication patterns is similar to the cost of the manually programmed calculations in the reference versions. This cost is amortized by their reutilization across many iterations of the computation. It is remarkable that the IS program needs the computation of several different patterns on each repetition of the code. However, the performance delivered by the Hitmap and reference versions is almost the same.

General efficiency

Figure 3.7 shows a performance comparison between different MG benchmark implementations using state-of-the-art parallel programming models, with input set of size C and D as defined by the NAS benchmark. The comparison include shared-memory models (OpenMP), PGAS models (UPC), and distributed-memory-based libraries (C+MPI,

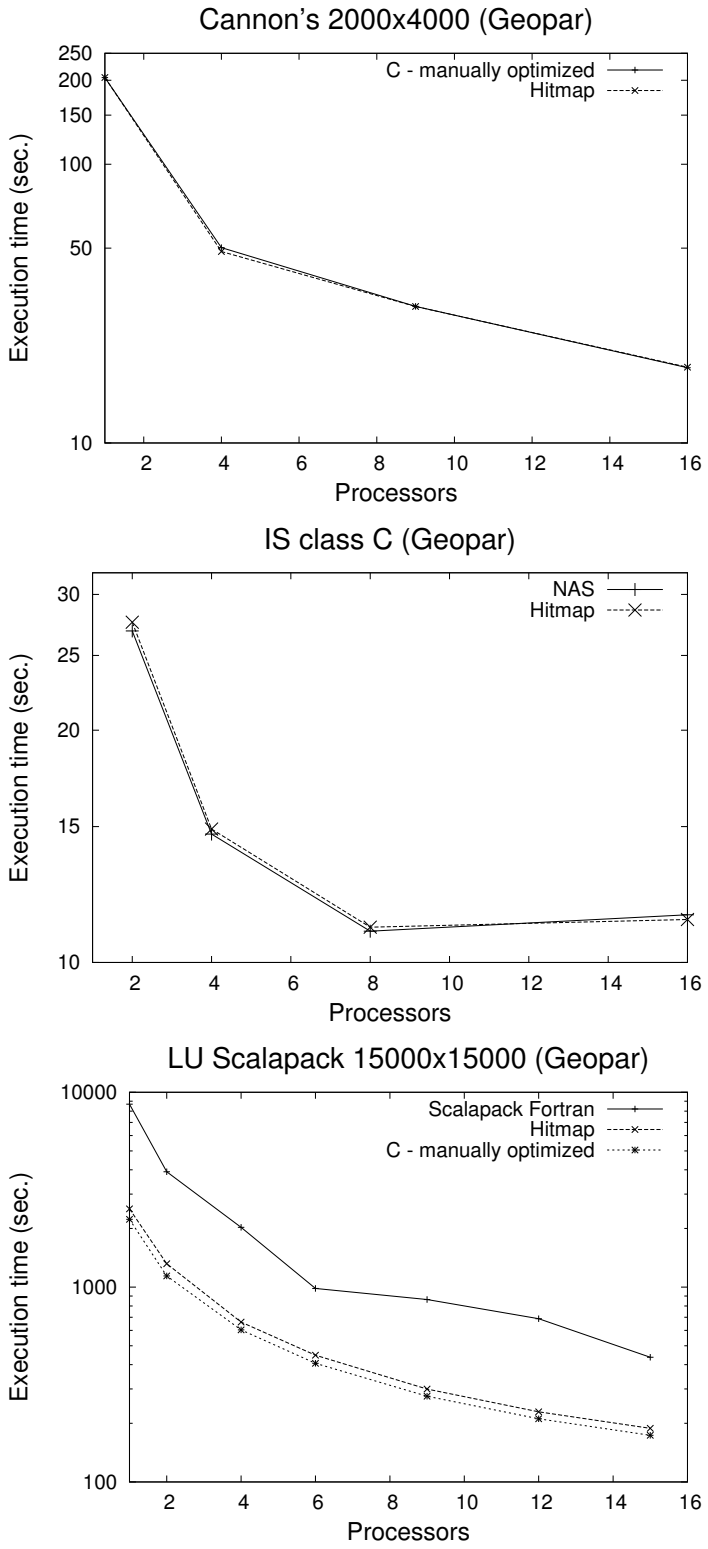


Figure 3.5: Performance results for some representative parallel kernels and benchmarks in Geopar, a shared memory system. Results for MG are shown in Fig. 3.7.

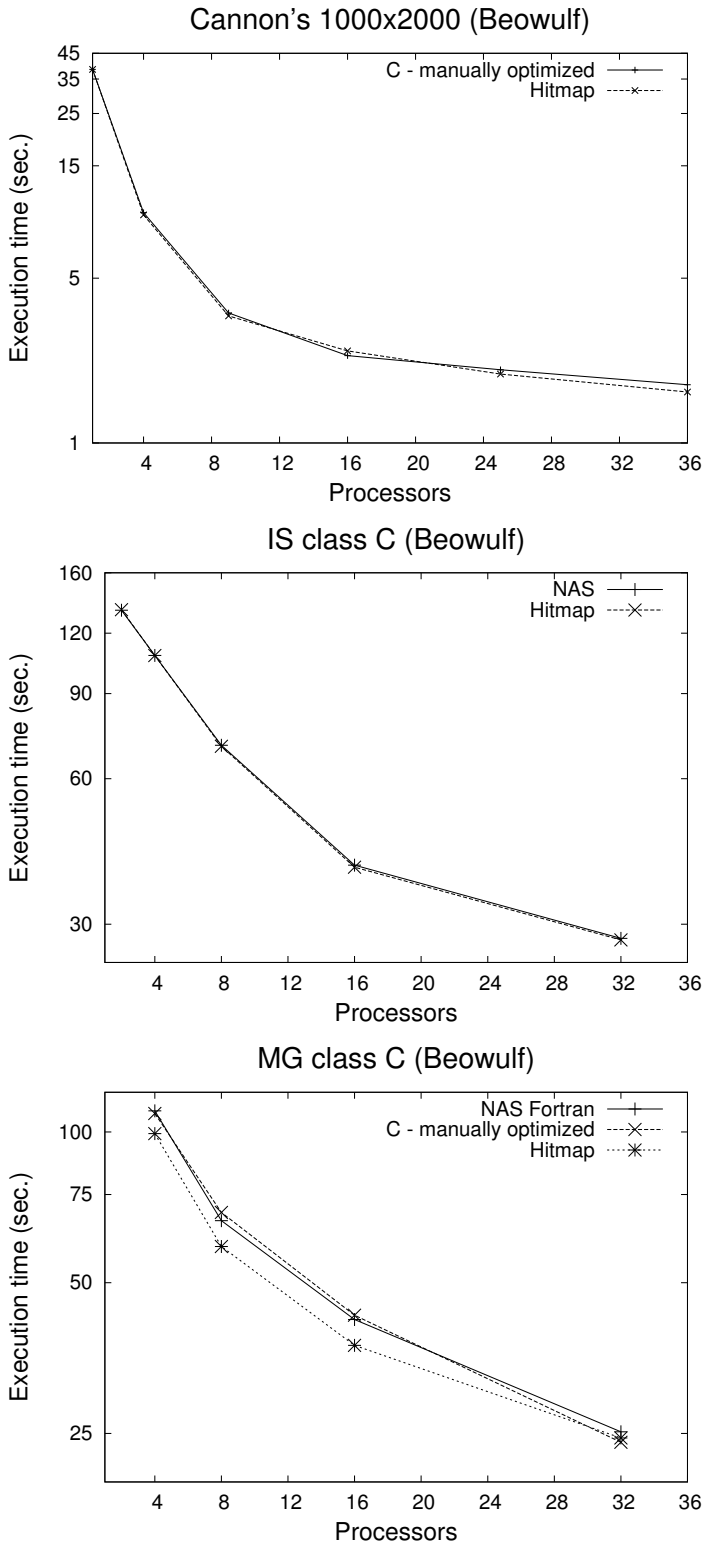


Figure 3.6: Performance results for some representative parallel kernels and benchmarks in a Beowulf cluster.

HTA, Hitmap). Results for the original NAS implementation, using Fortran+MPI, are also shown. To allow the comparison of shared-memory and distributed-memory models in terms of performance, all experiments were run in the shared-memory machine: Geopar. All implementations were compiled with GCC and equivalent optimization flags, except HTA, that requires the use of the Intel C compiler. MG implementations that support the D input size (the biggest one that fits in the machine memory) need to be compiled using the *medium* memory model in Geopar architecture. Each bar includes the overall time as measured by the NAS benchmark, and the *additional time* spent in initializations.

From the results obtained we can draw the following observations. First, UPC delivers good performance for the C input size. However, UPC's memory footprint is three to four times bigger than in other implementations. For this reason, the UPC implementation for the D input size does not fit in Geopar's memory. Second, the use of HTA to execute MG with the D input set leads to much higher execution times and *unsuccessful* results, so performance values for class D are not shown. Third, although C+MPI version is generally faster than the Fortran+MPI version, it does not beat the performance obtained with the OpenMP version. Nevertheless, OpenMP can not be directly used in distributed memory environments.

Finally, the results show that Hitmap performance is comparable with the performance with C+MPI and generally faster than HTA, thanks to a more efficient communication management. Besides this, Hitmap presents a more flexible interface and a lower development effort, as we will show in the following section.

Development effort

To compare Hitmap development effort with respect to other implementations, we have used several complexity and development effort metrics, including-number of lines of code, Halstead development effort (Halstead's D.E.) [51], KDSI (COCOMO) [16] development effort, and McCabe's cyclomatic complexity (McCabe's C.C.) [73].

Figure 3.8 shows a comparison of the Hitmap version of the benchmarks considered with the other implementations in terms of lines of code. The comparison separates the lines devoted to parallelism (data layouts and communications), sequential computation, declarations, and other non-essential lines (input-output, etc).

With respect to MG, our results show that the use of Hitmap library leads to a significant reduction in the number of lines, specially those devoted to parallelism (partitioning and communication), even including the 14 lines of the new layout plug-in developed and introduced in the library for this example (see [40] for the details). Although MG uses a multilevel data partition, Hitmap automates the generation of communication patterns and hides the particular cases that occur in smaller grids.

Significant reductions are also obtained for Cannon's algorithm, because it only uses

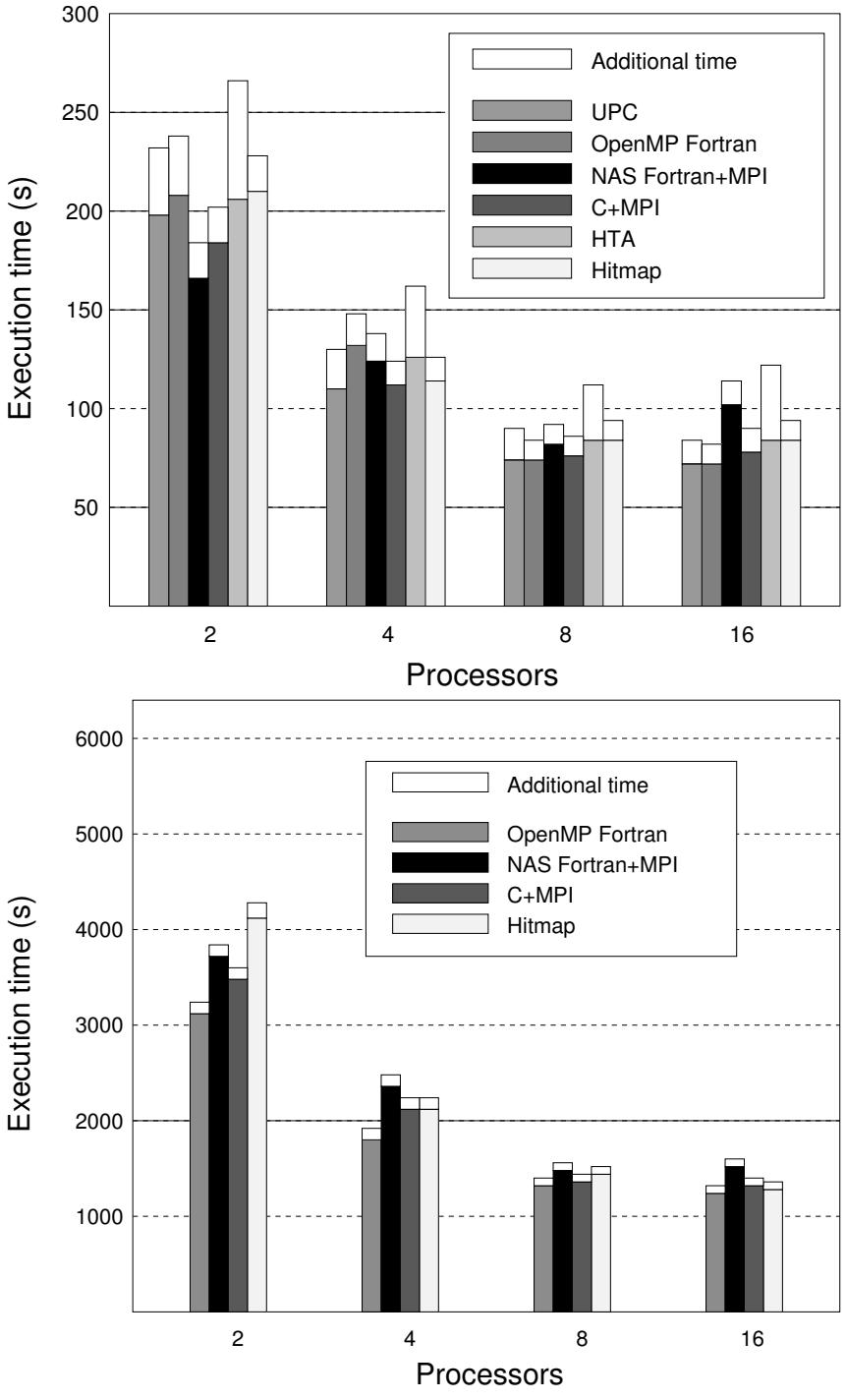


Figure 3.7: NAS MG benchmark performance comparison. Class C problem size (up) and Class D problem size (down).

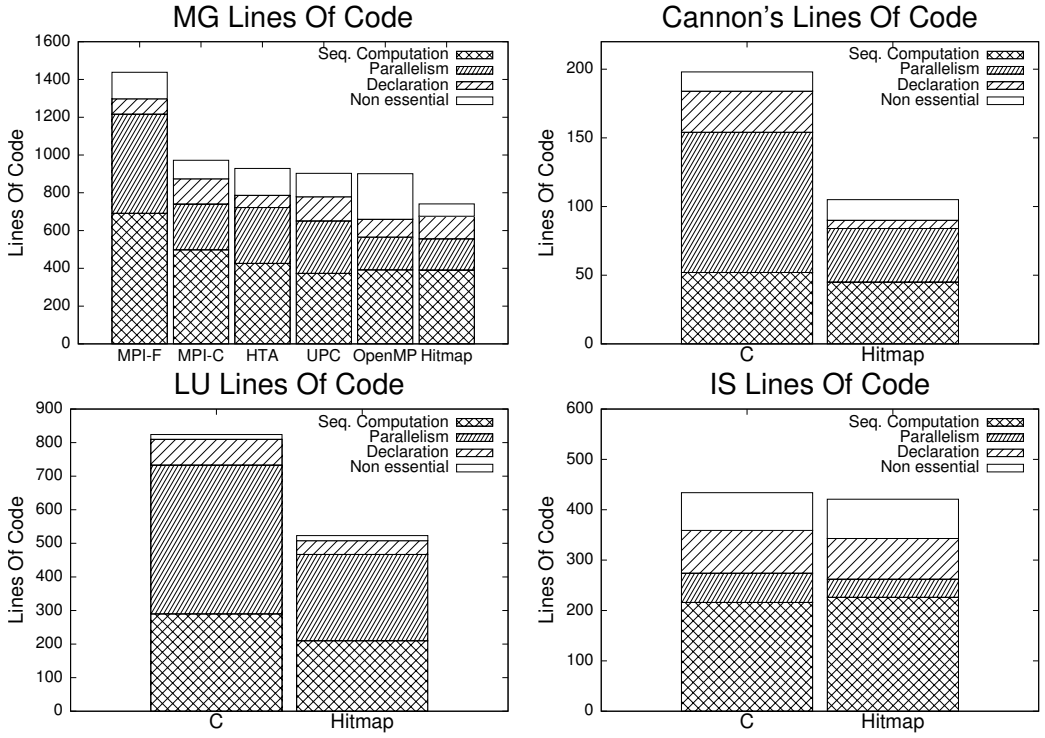


Figure 3.8: Comparison of code lines.

a single communication pattern that is derived directly from the data partition. Regarding LU, the use of Hitmap leads to a compact representation of blocks of tiles, thus reducing many computations needed to handle size, paddings, and block-cyclic distribution management. Moreover, the use of layouts hides to a great extent the details on how to build LU's complex communication patterns. Finally, IS presents smaller reduction ratios, because most of the code is sequential. Even so, lines devoted to parallelism are reduced by 38 %.

In Table 3.1 the McCabe's cyclomatic complexity indicates the total number of execution paths in a piece of code. As can be seen in the table, cyclomatic complexity is greatly reduced for Cannon's (58.82 %), MG (49.04 %), LU (31.49 %) and IS (34.52 %) code. The reason is that Hitmap hides many decisions to the programmer, thus avoiding unnecessary conditional branches in the resulting code.

Table 3.1 also shows the Halstead's development effort and the KDSI metric used in the COCOMO model for the benchmarks considered. As can be seen in the table, the use of Hitmap leads to a great reduction with all benchmarks.

We can conclude that Hitmap greatly simplifies the programmer effort for data distribution and communication, compared with the equivalent code needed to manually calculate the information used by MPI routines, or to handle the synchronization details needed in other models. Moreover, Hitmap encapsulates generic calculations into plug-

Metric	Cannon's		MG				
	C + MPI	Hitmap	C + MPI	HTA	UPC	OpenMP	Hitmap
McCabe's C.C.	34	14	210	148	218	168	107
Halstead D.E.	1 892K	359K	29 568K	54 366K	35 084K	-	19 265K
KDSI (COCOMO)	201	104	1 389	1 277	1 413	1 343	945

Metric	LU		IS	
	C + MPI	Hitmap	C + MPI	Hitmap
McCabe's C.C.	216	148	84	55
Halstead D.E.	27 822K	7 576K	2 683K	2 193K
KDSI (COCOMO)	919	606	608	496

Table 3.1: Complexity metrics and development effort for the benchmarks considered.

ins, allowing the programmer to skip the use of tailored formula to compute local tile sizes in the application code, and neighborhood relationship at the different grain levels.

3.6 Conclusions

This chapter introduced Hitmap, a runtime library for hierarchical tiling and mapping of arrays in homogeneous systems. Hitmap was designed to simplify parallel programming, providing functionalities to create, manipulate, distribute, and communicate tiles and hierarchies of tiles.

After analyzing the experimental results, we can conclude that the use of Hitmap does not imply a significant performance degradation. The tiling abstraction layer do not lead to a noticeable efficiency loss, while the use of this library greatly simplifies the programmer effort compared to the equivalent code needed in other implementations analyzed.

In the following chapter, we will use Hitmap library as a starting point to analyze the possibility of creating a programming model to take profit of all available hardware resources in heterogeneous environments.

Chapter 4

New Abstraction Layers for an Heterogeneous Hitmap

In Chapter 3 we described the foundations of the Hitmap library, where dense data structures (tiling arrays) can be handled independently of the target platform. Nevertheless, that work is only focused on CPU based, distributed systems environments.

In this chapter we analyze the possibility of creating a programming model and framework that support heterogeneous platforms encapsulating the selection of (a) good configuration parameters values for heterogeneous devices, (b) the management of arbitrary tile data structures, (c) mapping and load balancing functions, and (d) synchronization/-communication functionalities between CPU-GPU heterogeneous devices. To achieve it, we present a programming framework that extends the Hitmap library. We introduce this framework as a prototype tool that integrates a parallel computation model which takes profit of all available hardware resources (CPU-GPU) in heterogeneous environments. This framework allows to generate abstract codes which are transparently adapted to heterogeneous systems with mixed types of accelerator devices. Moreover, we present a local data partition policy for polyhedral computations with affine expressions. This policy transparently creates a tuned partition of computations to be launched in a single accelerator device when it is not possible allocate the whole data structures of a computation in the target device.

4.1 Mapping synchronization issues

4.1.1 Conceptual approach

Heterogeneous systems can be built with very different hardware devices (CPU-cores, accelerators) in several interconnected nodes, in a distributed environment. Portable codes for such systems should implement parallel algorithms, abstracting them from the map-

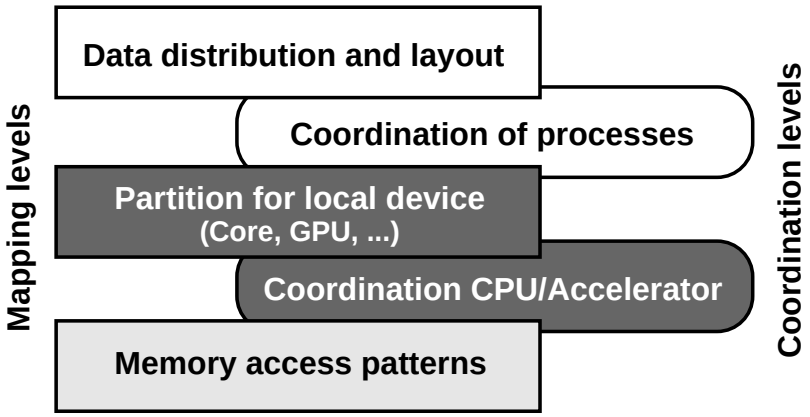


Figure 4.1: Mapping/Coordination levels. White boxes show the original Hitmap approach.

ping activities that adapt the computation to the platform. Thus, the programming model should encapsulate the mapping techniques and the CPU/accelerator synchronization with appropriate abstractions.

We propose a programming model and framework for heterogeneous target platforms based on the same Hitmap concepts: (1) Several layers of plug-in modules that encapsulate the mapping functions; and (2) functionalities to build the coordination (synchronization and communication) structures of the algorithms, in a way that they are transparently adapted at run-time in terms of the results of the mapping functions.

Our work extends the Hitmap approach. Figure 4.1 shows the different mapping levels of the original Hitmap, among with our proposed extension for this transparent synchronization solution. As we stated in the previous chapter, Hitmap has a single level of data-partition and layout. It is designed to encapsulate coarse-grain mapping techniques, appropriate for distributed-memory nodes.

We propose to add a second, middle-grain partitioning level, that allows to exploit a further level of parallelism, adapting the local part of data to the specific characteristics and architecture of the actual device associated to the logical process by the virtual topology. We name *Partitions* to new local partition policy modules.

The programmer naturally introduces a third level of mapping inside the kernel code by implementing specific, thread-level memory access patterns.

Our second-level mapping plug-ins use information about the device and the global memory access pattern of the kernel, to generate domain partitions that exploit locality, maximum occupancy, coalescence, or other device properties that affects performance. The result is an object encapsulating information about a partition of the local computation in a grid of blocks. The same abstraction can be used for techniques of very different nature: tiling techniques for CPU-cores, threadblock size-shape choice policies for GPUs, or other sophisticated tuning techniques.

Finally, the coordination of the data movement between the CPU and accelerators,

together with the kernel launch, is automatized by a run-time system, using the second-level partition results. Padding can be automatically added to tiles when needed, properly aligning data to the memory banks of the particular device, alleviating memory bottlenecks, and improving cache usage.

This approach can be used together with techniques to automatically generate kernels for different architectures from common specifications (see e.g. [37, 104]), avoiding the need to supply optimized kernels for all the architectures that compose the target heterogeneous system. By encapsulating the CPU/accelerator coordination in a transparent system, we also allow to integrate calls to library kernels specifically optimized for a given architecture, such as CUBLAS [88] for GPUs.

We also promote the abstraction of hierarchical tiles to specific programming languages for accelerators. In this work we use CUDA as a proof of concept, doing this exercise for more generic languages, such as OpenCL, is straightforward. Thus, we introduce a common array abstraction, simplifying the porting of code between CPU cores and accelerators.

This conceptual framework adds new functionalities to the Hitmap library without modifying the original structure. This imposes a minimal impact on the original Hitmap codes, which takes care of the coordination of processes in the upper level. The new extension takes care of adapting the local parts to the device automatically assigned to the logical process. Plug-ins with new mapping techniques may be included and tested without modifying the framework implementation.

4.1.2 Design and implementation

We have developed a prototype implementation of this framework by extending Hitmap. In this section we describe some design and implementation considerations, and problems that ought to be solved.

As described in Chapter 3, Hitmap is designed to manipulate hierarchical tiling arrays. The *HitShape* class implements tile domains. A shape object represents a subspace of array indexes defined as an n-dimensional rectangular parallelotope. Its limits are determined by n *Signature* objects. Each *Signature* is a tuple of three integer numbers (begin, end, and stride), representing the indexes in a domain axis.

Hitmap defines an API for data-partition modules, named *Layout* plug-ins. It defines a wrapper function that links the main code with the chosen plug-in. The Layout plug-ins receive as parameters: (1) A virtual topology object (*HitTopology*), (2) a domain to be mapped (a HitShape object), and (3) optional parameters for the specific technique. They return a *HitLayout* object containing a local domain (another HitShape), information about neighbor relations, and other mapping details. These objects are used as parameters in the constructors of *HitComm* objects that express tile communications across logical

processes.

- **Partitions**: We follow the same approach for the new second-level of local partition plug-ins. The wrapper function is similar, but also selects different implementations of the same plug-in name depending on the architecture of the target device at run-time assigned to the logical processes. Our current wrapper differences between CPU-cores, and several NVIDIA's CUDA supported architectures.

  The Hitmap initialization function gathers information about the particular system devices and builds an internal physical topology object. The virtual topology constructors attach each logical process to one device or devices subset (e.g set of CPU-cores). The Partition plug-ins receive as parameters: (1) The attached devices data; (2) a HitLayout object with information of the local domain to be mapped. Optional parameters indicating the memory-access patterns of the low-level threads can be supplied. The result is a new *HitPartition* object containing information about block shapes, grid sizes, and information to generate tile paddings if needed.

  As example, we have implemented a trivial partition plug-in called *Default*. The CPU-cores implementation simply creates a grid with one element containing the full local shape. The GPU implementations split the local domain in rectangular blocks with 1×512 threads. This solution is based on an NVIDIA's recommendation [118]. More sophisticated policies can be integrated as new plug-ins, as will be described in Sects 5.1, 5.2, and 5.3.

- **Assigning several logical processes to the same device**: We have introduced a new technique in the virtual topology modules of Hitmap. It allows to assign more than one logical process to the same device. This has two purposes: As a potential load-balancing technique, and to transparently use accelerators to perform large computations whose data do not directly fit in the accelerator global memory. Thus, the full computation is carried out in smaller parts, coordinated by the Hitmap upper-level communication structures. (Another solution for this problem is studied in Sect. 4.2.)

- **Kernel definition and launch**: We provide a macro function to declare the function headers of different kernel versions for different architectures, using a common interface. The following example shows the headers of two implementations (one for CPU-cores, another for pre-Fermi GPUs) of the same kernel:

```
hit_kernelDefinition( CORE, mmult, HitTile_float *A,
                          HitTile_float *B, HitTile_float *C ) {

hit_kernelDefinition( GPU_R1, mmult, HitTile_float *A,
                            HitTile_float *B, HitTile_float *C ) {
```

We develop a kernel launching function that transparently do the coordination with the assigned device. It receives the kernel name, a partition object, and the kernel parameters, indicating which ones are inputs and which ones are outputs (see *hit_kernelLaunch* function call in the full example of Figs. 6.1).

This function deals with linking issues of kernels written in specific languages. For example, the launch of a kernel for an NVIDIA GPU needs a special syntax, and the launching code has to be compiled with the CUDA compiler. We use internal wrapper functions with different implementations for different architectures. Each implementation is compiled with the proper tools before linking. A selection mechanism checks at run-time the nature of the assigned device architecture and calls the appropriate implementation for the local device.

For CPU cores, the wrapper simply calls the proper C function passing the indicated arguments. For accelerators the process is more complex, and involves communication between the main system memory and the device memory. We have implemented the synchronization for NVIDIA's GPUs with the following stages: (1) Move to the GPU memory the input tiles (the data and the tile handler structure). Padding restrictions expressed in the HitPartition object are applied to the memory allocation in this step. (2) Launch the kernel, using the grid parameters from the Partition object, passing the pointers to the new tile handlers in GPU memory. (3) Copy data from output GPU-memory tiles to the CPU, eliminating padding if needed. Finally, a mutual exclusion mechanisms has been added in the kernel launch function to allow several processes assigned to the same device to coordinate themselves for the use of the device.

These abstractions completely encapsulate the synchronization and coordination between CPU and different devices, such as cores and accelerators. The same primitive call automatically invokes CPU-core functions written in plain C language, or launches CUDA kernels.

- **Running the programs**: Hitmap programs are started like any MPI program, using the *mpiexec* command. The MPI hosts file is used to select the machines where the processes are started. Processes in the same machine are automatically attached to CPU-cores or GPU devices. If data do not fit into the memory of an accelerator device, more MPI processes are required to obtain a finer partition. In Sect. 4.2 we propose a better solution for this problem for some application classes.

4.1.3 Mapping and synchronization issues: Summary

The framework presented in this section encapsulates the mapping techniques into plug-ins at two different layers of abstraction: One related to logical processes coordination,

and another related to adapting the computations to the inherent parallelism and architecture details of the actual device associated to each logical process. Thereby, the proposed high-level API transparently deals with all the details of communication and synchronization between logical processes and accelerator devices, such as GPUs. Finally, this framework allows to generate codes which are transparently adapted to heterogeneous systems with mixed types of accelerator devices.

4.2 Memory size restrictions

In Sect. 4.1 we proposed an heterogeneous programming approach and framework based on: (1) Several layers of plug-in modules that encapsulate the mapping functions; and (2) functionalities to build the coordination (synchronization and communication) structures of the algorithms, which are transparently adapted at run-time in terms of the results of the mapping functions. The approach has been incorporated into Hitmap [31, 40], a parallel programming library where partition policies are implemented through a set of plug-ins with a common interface.

In the mapping approach described in Sect. 4.1, the computation partitioning is done top-down. The whole computation is first split and coordinated among logical processes in a distributed memory environment (Load balancing techniques can be used at that level to adapt the amount of computation of each part to the computation power and characteristics of each device assigned to a process).

Device memory restrictions can be considered at that level in the partition policies creating a finer grain, partition to generate smaller tasks that fit into devices memories. However, these policies would become much more complicated, and for huge computations, they will lead to the creation of a higher number of logical processes, with the associated penalties for coordination and communication.

We propose to keep using simple partition policies at the highest level, that do not take into account the memory restrictions of heterogeneous accelerator devices attached to the system nodes. Then, we propose to introduce a hidden abstraction layer that splits the computation in several parts whose memory requirements fit the device limits. This layer is applied after determining the appropriate tile or block geometry (see the dark shaded box in Fig. 4.2). To keep the obtained optimizations by the tiling techniques of the upper layer, this new internal partition would use the tiles or blocks as basic mapping elements. Sections of the grid of tiles/blocks are then sequentially launched to the device as separate kernels.

In general, due to communication costs between the main node and the device memories, the partition of a computation should maximize the part sizes. Besides this, when launching a subpart of a computation, the exact pieces of data structures accessed by

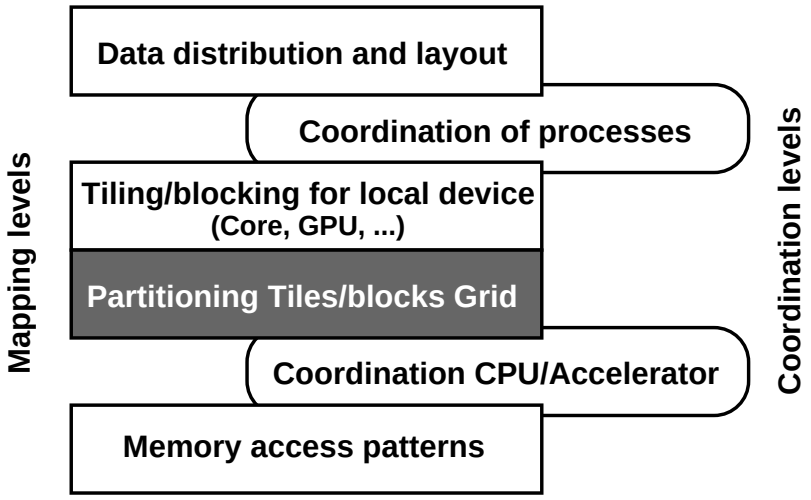


Figure 4.2: Mapping/Coordination levels. The new level of automatic partitioning is highlighted with a dark-grey shadowed box.

the different blocks (mapping units) of that subpart are determined by the program algorithm and the design details of the parallel solution. In our approach, we introduce a simple abstraction to help the programmer to express the threads access patterns to any data structure involved. Thus, the system will be able to automatically derive expressions to compute at run-time the exact memory requirements, and the exact locations of data pieces needed for a given computation subpart.

4.2.1 Model for parallel computations

Polyhedral domain spaces

We define a *domain* D as a collection of n-tuples of integer numbers that define a space of n-dimensional indexes. For dense arrays, the index domain is a subspace of Z^n, defined by a rectangular parallelotope. In this approach we also allow *strided* domains, where the parallelotopes are defined by its dimensional limits, and a stride value for each dimension. A *Signature* is a 3-tuple of integer numbers $S = (b, e, s) : b, e, s \in \mathbb{Z}$ representing a subset of integer numbers where the begin or lower limit is b, the end or upper limit is e, and the elements are selected using the stride value s. We denote this subset of integer numbers as the *range* of the signature $\breve{S}$.

$$S = (b, e, s);\ \breve{S} = \{x \in \mathbb{Z} : x \geq b,\ x \leq e,\ (x - b) \mod s = 0\}$$

$$D < S_0, \ldots, S_n >= \{(p_0, \ldots, p_n) : p_i \in \breve{S}_i\}$$

Domains are used in this work to represent the index space of a data structure, a set of indexed threads, the geometry of a tile/block of threads, a grid of tiles/blocks, or a

superblock geometry (a subdomain of a grid of tiles/blocks).

Parallel computations

A data structure or tile T is a map between elements of a domain and data elements of a given type: $T : D \rightarrow dataType$. We denote with $d(T)$ the Domain of a tile.

We define a *Parallel Computation* $P < D, f, T_0, \ldots, T_m >$ as a collection of threads manipulating data in one or more data structures or tiles $T_0, \ldots, T_m$. The domain of the computation D defines the number and indexes of the threads to be executed. The computation is the application of the function f (or collection of statements) by each thread on data elements. A *Polyhedral Computation* is a parallel computation where its domain D can be expressed as a parallelotope, and where the function f uses affine expressions on the thread indexes to locate and access data elements in any data structure T_i.

Access patterns

An *Access Pattern* AP is a set of access expressions. An *Access Expression* represents a domain transformation $A : D, \mathbb{Z}^n \rightarrow D$. It is a tuple of n *Signature Functions* $A = (A_0, \ldots, A_n)$. Each signature function maps a signature, and one domain element, to another signature: $A_i : S, \mathbb{Z}^n \rightarrow S$.

Affine Access Expressions are those whose signature functions determine the resulting signatures using affine expressions in terms of the input domain element $\vec{x} \in D$. Let $S = (b, e, s)$ be the signature of the dimensional domain of data structure T_i:

$$\begin{aligned} A_i < \vec{a}_b, b_b, \vec{a}_e, b_e, c > (S, \vec{x}) = (b', e', s') : \\ b' = \vec{a}_b \cdot \vec{x} + b_b, \\ e' = \vec{a}_e \cdot \vec{x} + b_e, \\ s' = c \times s \end{aligned}$$

In some real parallel computations one dimension of a data structure is fully traversed by any thread. We model this special behavior using infinity values in the signature function to refer to the limits of the input signature. If $b_b = -\infty$, then $b' = b$. If $b_e = \infty$, then $e' = e$.

Union of domains

The union of generic domains expressed by signatures, cannot always be expressed themselves by signatures. As an example, consider the situation where there is a gap between

their extremes, such in $S = (2, 100, 2), S' = (250, 300, 2)$, or when the strides are not compatible, such in $S = (2, 100, 2), S' = (2, 100, 3)$.

We define the *Signature coarse union* operator $\sqcup$ as: $S \sqcup S' = (b'', e'', s'') : b'' = \min(b, b'), e'' = \max(e, e'), s'' =$ m.c.d.(s, s'). We can also extend the operator definition to n-dimensional domains. The *Domain coarse union* of two domains is calculated applying the signature coarse operator to each pair of signatures with the same index: $D \sqcup D' = (S_0 \sqcup S'_0, \ldots, S_n \sqcup S'_n)$. The application of this operator to merge two strided parallelotope domains generates another strided parallelotope that can be expressed with signatures, with minimal number of extra added elements.

Domain transformations

We define a *Domain transformation* $\Gamma : D, AP, D \rightarrow D$ as the coarse union of the domains obtained applying each access pattern to each element of the second domain, using as reference or data-structure domain the first parameter domain.

$$\Gamma(D, AP, D') = \sqcup\{A(D, \vec{x})\}\ \forall \vec{x} \in D' \wedge \forall A \in AP$$

We call *Regular access expressions* to those that for two given input domain elements $\vec{x}, \vec{y}$, the signatures $A_i(D, \vec{x}) = (b, e, s)$ are a coordinates translation of the signatures, $A_i(D, \vec{y}) = (b', e', s')$ such that $\forall i$: (1) $b' = b, e' = e, s' = s$, or (2) $b' = b + (y_i - x_i), e' = e + (y_i - x_i), s' = s$. A *Regular access pattern* is a pattern with only regular access expressions. Memory requirements of regular access patterns grow linearly when the threads space grows in only one dimension.

4.2.2 Partition of regular computations

In this section we present a general algorithm that, given a polyhedral parallel computation with regular access patterns, determines how to split the grid of tiles/blocks of threads in regular parts, in such a way that the number of parts is minimal, and the memory requirements of each part do not exceed a given memory limit. To introduce the basic concept we first present the special case for 1-dimensional domains. Then, we present the solution for 2-dimensional domains. Algorithms for higher dimensions can be deduced from these ones.

To simplify the presentation, in the following algorithms we assume that the thread index space starts at 1 and, has stride 1 for all dimensions. It is straightforward to extend the algorithm to use generic thread index domains with any stride or starting positions.

Inputs/Outputs

The algorithms have the following parameters:

Input: The device memory limit $devLim \in \mathbb{N}$.

Input: The dimensional sizes of the grid of tiles/blocks $\vec{g} \in \mathbb{N}^n$.

Input: The dimensional sizes of any tile/block $\vec{b} \in \mathbb{N}^n$.

Input: A collection of data structures or tiles $T_0, \ldots, T_m$.

Input: A collection of access patterns, one for each tile $AP_0, \ldots AP_m$.

Output: The number of blocks in each dimension that will form a subpart $\vec{r} \in \mathbb{N}^n$.

Algorithm for 1-dimensional spaces

The algorithm is based on determining the linear increasing rate of memory requirements when more blocks are grouped together, representing it with a linear equation. Substituting the device memory limit into the equation, we can obtain the higher number of blocks which memory requirements fits in the available space.

---

1. $B_1 = ((1, b, 1)), B_2 = ((1, 2 \times b, 1))$
2. $s_1 = \bigsqcup_i |\Gamma(d(T_i), AP_i, B_1)|,\ s_2 = \bigsqcup_i |\Gamma(d(T_i), AP_i, B_2)|$
3. Compute $\alpha, \beta, \gamma : 0 = \alpha x + \beta y + \gamma$, the linear equation that contains both $(1, s_1)$ and $(2, s_2)$.
4. Return $r = \lfloor -(\beta \cdot devLim + \gamma)/\alpha \rfloor$

Algorithm for 2-dimensional spaces and beyond

For two dimensional spaces we obtain a plane equation for the memory requirements of three samples of block groups. Substituting the device memory limit into the equation, we obtain a linear equation. The points of this equation determine the best candidates for the solution. These candidates are checked to determine which one leads to less number of parts due to better alignment of multiples of the new superblock sizes with the grid dimensions.

---

1. $B_1 = ((1, b_0, 1), (1, b_1, 1)), B_2 = ((1, b_0, 1), (1, 2 \times b_1, 1)), B_3 = ((1, 2 \times b_0, 1), (1, b_1, 1))$
2. $s_1 = \bigsqcup_i |\Gamma(d(T_i), AP_i, B_1)|$, $s_2 = \bigsqcup_i |\Gamma(d(T_i), AP_i, B_2)|$, $s_3 = \bigsqcup_i |\Gamma(d(T_i), AP_i, B_3)|$
3. Compute $\alpha, \beta, \gamma, \delta : 0 = \alpha x + \beta y + \gamma z + \delta$, that is, the plane equation that contains $(1, 1, s_1)$, $(1, 2, s_2)$, and $(2, 1, s_3)$.
4. Substitute $z = devLim$ to obtain a linear equation $0 = \alpha x + \beta y + \delta'$.
5. $\forall\, \vec{r} = (r_0, r_1) : r_0 = \lfloor q_0 \rfloor, r_1 = \lfloor q_1 \rfloor : 0 = \alpha q_0 + \beta q_1 + \delta'$
5.1. Compute $k(\vec{r}) = \lceil g_0/r_0 \rceil \times \lceil g_1/r_1 \rceil$
6. Return $\vec{r}$ with the minimum value of $k(\vec{r})$.

Vector addition

1. $\forall i \in d(\vec{z})$
1.1. $z_i = x_i + y_i$
2. Return $\vec{z}$

Cellular automata

1. for i=1...t
1.1. $A' = A$
1.2. $\forall (i,j) \in d(A)$
1.2.1. $A(i,j) = (A'(i-1,j) + A'(i+1,j) + A'(i,j-1) + A'(i,j+1))/4$
2. Return A

Matrix-matrix multiplication

1. $C = 0$
1. $\forall (i,j) \in d(C)$
1.1. $\forall\ k \in [0, m-1]$
1.1.1. $C(i,j) = C(i,j) + A(i,k) \times B(k,i)$
2. Return C

Figure 4.3: Algorithms for the three cases studied.

Study cases

We are going to present some examples of regular kernels and applications to show how our model can be used to express different access patterns. The base algorithms for the study cases are presented in Fig. 4.3.

- **Vector addition**: This simple kernel computes $\vec{z} = \vec{y} + \vec{x}$ using one thread to compute the result of each z_i element. It uses a 1-dimensional thread space of as many threads as elements in the arrays. The access pattern for this kernel have a single access expression:

  $$A_0 < 1, 0, 1, 0, 1 >$$

  Thus, the resulting signature

  $$S' = A_0(S, \vec{x}) = (1 \times x_0 + 0, 1 \times x_0 + 0, 1) = (x_0, x_0, 1)$$

  contains only one point in its range $\breve{S}' = \vec{x}$.

- **Stencil program: Cellular automata**: This is an example of an stencil application in a two dimensional array space. It implements a PDE solver to compute the heat distribution is a 2-dimensional discretized space using the Jacobi method. The application has a step loop that applies a stencil computation, computing the new value of a matrix position using the old values of its four neighbors. There is only one input/output parameter, a matrix A.

  The thread domain is the same as the matrix index domain. Each thread compute one matrix position. All threads synchronize on each i loop step.

The access pattern for this kernel can be expressed with one access expression for each matrix access, or in a compact form with only one expression:

$$A = (A_0 < 1, -1, 1, 1, 1 >, A_1 < 1, -1, 1, 1, 1 >)$$

Thus, the resulting signatures are:

$$S'_0 = A_0(S_0, \vec{x}) = (x_0 - 1, x_0 + 1, 1)$$
$$S'_1 = A_1(S_1, \vec{x}) = (x_1 - 1, x_1 + 1, 1)$$

This compact form directly includes in the access pattern result the four *corner* elements that are not really accessed. However, the resulting domain is a parallelotope. When the pattern is applied to a subset of the thread index space, the amount of added data is negligible, and the parallelotope shape conveniently simplifies the movement of data between node and device memories.

Note that, for threads in the limits of the thread domain, the resulting accessed pattern exceeds the limits of the original matrix. To avoid the use of costly conditional evaluations in the fine-grain threads, the A matrix should be extended with *ghost borders*, or the thread index space should be reduced by one element on each border.

- **Matrix multiplication**: In all the previous examples the resulting domains do not need to take into account the domain description of the data structures. Thus, the input signatures on the access expressions are simply ignored.

  This study case is a direct implementation of the classical matrix-matrix multiplication $C_{n,n} = A_{n,m} \times B_{m.n}$, with three loops. It implements a fine-grain parallelization of the first two loops. Each thread executes the third loop to compute one position of the resulting matrix.

  There are three different access patterns for this application, one for each matrix. Each pattern has a single access expression:

  For matrix A: $(A_0 < 0, -\infty, 0, +\infty, 1 >, A_1 < 1, 0, 1, 0, 1 >)$

  For matrix B: $(A_0 < 1, 0, 1, 0, 1 >, A_1 < 0, -\infty, 0, +\infty, 1 >)$

  For matrix C: $(A_0 < 1, 0, 1, 0, 1 >, A_1 < 1, 0, 1, 0, 1 >)$

This access patterns indicate that each thread accesses to a full row of the A matrix, a full column of the B matrix, and one element of the C matrix, with the same indexes as the thread.

4.2.3 Memory size-restrictions: Summary

In this section we have introduced a partition technique that abstracts memory restrictions of heterogeneous accelerator devices. It can be used to create a transparent mapping layer for Hitmap library to split the computation in several parts whose memory requirements fit the device limits. The objective of this policy is to compute the pieces of data structures required by a generic Hitmap partition, and determine the best sub-partition that ensures that each subpart fits in the target device memory.

We will discuss in Chapter 6 the feasibility of the local partition technique proposed to be included in Hitmap, with the help of some experimental results.

4.3 Conclusions

In this chapter we have proposed an extensible framework model to encapsulate runtime decisions related to data partition, granularity, load balance, synchronization, and communication for systems including assorted GPUs.

Our goal is to create a set of policies to select GPU configuration-parameter values for Hitmap, exploiting efficiently the GPU capabilities. However, to squeeze the performance of parallel applications in heterogeneous environments it is necessary to study how the values of GPU configurations parameters affect to the parallel application performance. Thus, in the following chapter we will present an in-depth study of how the GPU configuration parameters impact on the GPU performance. That study, that helps to determine good values for threadblock geometries, can be used to create threadblock geometry selection policies, and will allow us to incorporate these policies in the framework already discussed.

Chapter 5

Study of GPU Configuration Parameters

The implementation of the abstraction layers described in the previous chapter requires new level of partitioning methods. For the particular case defined by GPU architectures, this in turn requires to study the optimal choice of some GPU configuration parameters. This topic will be covered in this chapter. As we will see, these values have a heavy impact on the GPU parallel programs performance. The knowledge obtained in this study will be used in the next chapter to integrate it into our framework, allowing it to transparently choose good values of GPU configuration parameters at the lower partition level.

5.1 Threadblock geometry

In this section we explain our hypotheses about the effects of changing the threadblock geometry and other parameters on performance. The subsequent sections are devoted to the design and execution of appropriate benchmarks to evaluate these hypotheses.

The threads launched to any target GPU architecture are grouped in blocks of threads (named threadblock in CUDA). This first section presents a discussion on how CUDA architecture details (Fermi architecture as example in our study) affect the programmer decisions about threadblock size and shape. We will also discuss the implications derived from the changes in other configurable parameters, such as deactivating the L1 cache, or modifying its size. A description of CUDA GPU architectures is presented in Appendix A.

5.1.1 Threadblock size and occupancy tradeoff

Maximize occupancy

A warp in CUDA is a group of 32 threads that is the logical scheduling unit processed in SIMD by a CUDA multiprocessor (SM). One SM can execute two half-warps at the same time, each one in a different computational unit. The warps of several threadblocks can be queued on the same SM, and the warps of the same block are always scheduled to the

same SM. When the threads of a warp issue a global memory request, these threads are blocked until the data arrives from memory. During this high latency time period, other warps in the SM's queue can be scheduled and executed. Thus, it is important to have enough warps queued in the SM to hide the global memory latencies by overlapping them with computation, or with other memory accesses.

"the number of active warps over the number of max warps supported on one Stream Multiprocessor" [61] is named Occupancy. The first consideration to maximize Occupancy is to select a proper block size. For example, to maximize the occupancy in Fermi, the number of threads per block should be an integer divisor of the maximum number of threads per SM, and higher or equal to 192, to allow to fill up the maximum number of threads per SM with no more than 8 blocks that is the maximum number of blocks supported by the SM at a given instant of time ($1\,536/8 = 192$). The only values that fulfill this requirements are: 192, 256, 384, 512, and 768 (see the Occupancy calculator spreadsheet developed by NVIDIA [79]).

However, we can also observe that in computations that use a grid with a small number of total threads, it may be beneficial to use very small blocks to distribute the computational load across the available SMs in the GPU. For example, to execute 512 threads, using only one block for all of them forces to execute all the load in one SM [118].

Coalescing and high ratio of global memory accesses

Memory Coalescing is a technique used in kernel coding to force consecutive threads in the same warp to concurrently request consecutive logical addresses from global memory. This allows to minimize the number of transaction segments requested to the global memory. Typically, it is done by properly associating consecutive data-structure indexes to thread indexes when traversing dense data-structures, such as multidimensional arrays. Classical examples include many dense matrix and linear algebra operations. Coalescing is particularly important on codes with a high ratio of global memory accesses.

For this study, we consider that a kernel has a high-ratio of global memory accesses when each thread contains more than 100 arithmetic instructions per global memory access. (see Formula 5.1). We omit the arithmetic instructions usually added to compute the global thread indexes at the start of the kernel.

$$\frac{\#instructions_{per\_thread}}{\#memory\_accesses_{per\_thread}} > 100 \tag{5.1}$$

A common technique to ease the programming of coalescing is to ensure that the dense data-structures are aligned with the global memory banks and transfer segments. Thus, arrays with a cardinality of their last dimension that is multiple of 32, simplify the programming of Coalescing. Moreover, to avoid partition camping problems [80], it is

better to use data structures that are aligned to the number of global memory controllers or banks on the target architecture.

We suggest that for kernels with a high-ratio of coalesced global memory accesses, the best performance would be obtained with a block size that maximizes occupancy, and at the same time maximizes the number of blocks in any SM (e.g. 192 in Fermi). The rationale behind this hypothesis is that programs with continuous accesses to global memory need at all times the maximum number of supported threads in an SM to properly hide latencies (e.g. in Fermi 48 warps, or 1 536 threads). This effect should be especially noticeable on kernels with a short number of total arithmetic instructions per thread, because when blocks finish, they need to be replaced as fast as possible.

It is highly recommended to activate the L1 cache for kernels with data reutilization to reduce the global memory accesses [61]. When the L1 memory is not active the size of transaction segment is reduced and a higher number of them are necessary to supply the same number of data (see Appendix A). For coalesced kernels, the number of transactions segments required is inversely related to the size of them. Thus, we expect that for coalesced and no data reutilization kernels deactivating the L1 cache in Fermi does not lead to a performance degradation. Similarly, for coalesced and data reutilization kernels to active the L1 cache memory could relax the global memory latencies.

On the other hand, when the code in the kernel is not requesting data fast enough to need these number of warps to hide latencies, we expect that it is not needed to force the maximum Occupancy to obtain the best performance. Examples include codes with high computational workload between global memory requests, not issued at the same time. In this situation, the computation of one warp may hide the latencies of the memory requests of other warps, and the best performance results could also be obtained with block sizes that does not maximize Occupancy.

Non-coalesced accesses

Codes with non-coalesced global memory accesses request many different memory transfer segments from the same warp at the same time, up to one different segment for each thread. These memory requests are serialized: Therefore, it is much more difficult to have a code with enough computational workload vs. memory accesses overlapping, to hide such latencies. In these cases, reducing the number of requests could also have beneficial impact on the partition camping problems that arise in the global memory banks. Reducing the blocks up to 32 threads, may preserve at the same time all possible parallelism (e.g. in Fermi eight blocks with their maximum of 32 threads per block), minimum global memory bandwidth requested, and minimum number of bank conflicts.

The global memory bandwidth bottleneck, and the partition camping problems, can become so expensive that it may even compensate to reduce the number of active SPs per

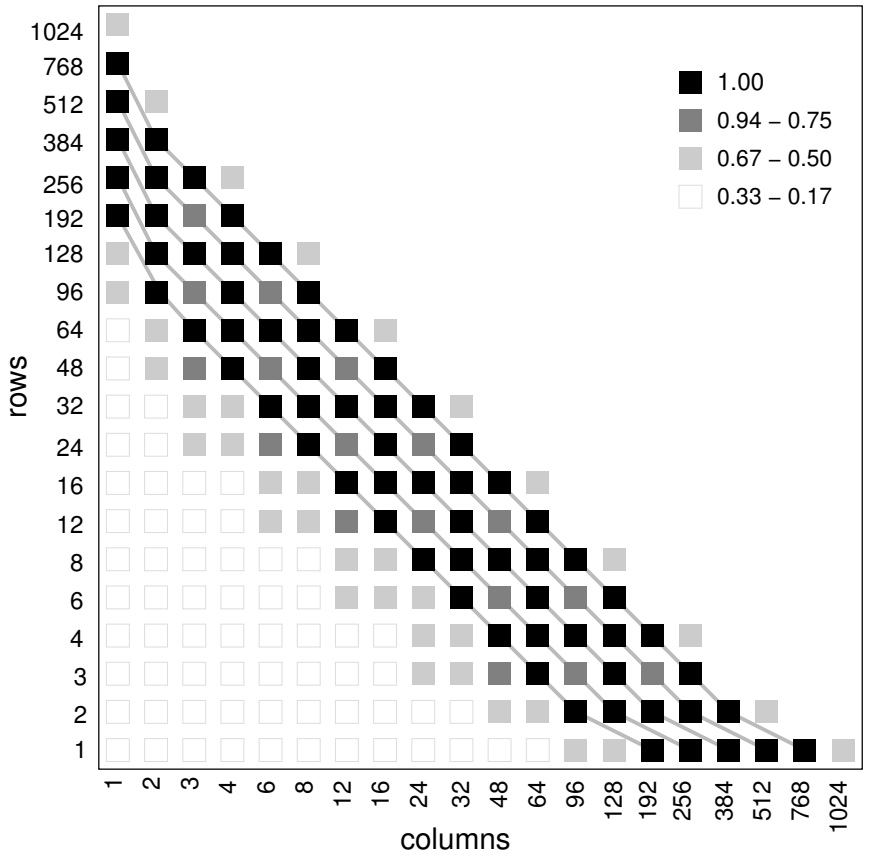


Figure 5.1: Maximum Occupancy for different threadblock shapes. It is not possible to cover the ranges not considered in this figure.

warp using blocks with less than 32 threads. However, reducing it too much may lead to waste parallelism capabilities and to lose performance. Without using more information about the architecture details it is difficult to predict the optimum block size. Codes with scatter accesses could benefit from deactivating the L1 cache memory, since the transaction segment size is reduced, thus alleviating bottlenecks.

5.1.2 Shape in several dimensions

Besides the effect of size on Occupancy, the chosen shape has also a significant impact on coalescing, partition camping, and memory bottlenecks.

We show in Figure 5.1 the Occupancy obtained for different combinations of threadblock shapes for 2-dimensions in a Fermi architecture, when the code does not exhaust SM resources (registers and shared memory). The block shapes with the same block size, that also maximize Occupancy, are linked by a gray line.

We expect that in programs with good Coalescing, the best performance results will be obtained with shapes with no less than 32 columns. One warp should request 32 integer

or float contiguous elements to request full transaction segment and consequently reduce the total memory bandwidth for the some number of threads. With perfect Coalescing, in Fermi, the first half-warp request 16 elements, obtaining a full cache line with 32 elements: The 16 elements requested by the first half-warp, and 16 more elements which are needed by the second half-warp. Thus, the second half-warp finds all needed elements in the cache, skipping the global memory access and its latency.

In Fermi, we find 320 or 380 (depending on the exact number of memory banks in the device) consecutive elements spread across the full span of the five or six memory banks. Thus, perfectly coalesced codes using a threadblock shape with that amount of columns, minimize the global memory bank conflicts. We expect that reducing the number of columns to increment the number of rows immediately derives in an increment of these conflicts, and in performance degradation.

5.1.3 Tuning techniques and threadblock size and shape

Some of the common code tuning strategies [61] heavily interact with, or are dependent on, the chosen block size and shape. For example, the CUBLAS optimized matrix operations [88] uses specific block shapes to improve performance while keeping correctness. It is an open question when, and where, it is possible to isolate the threadblock configuration from the application of other tuning techniques.

5.1.4 ThreadBlock size and shape in OpenCL

OpenCL [107] (Open Computing Language) is an open, royalty-free standard for general purpose parallel programming across CPUs, GPUs, and other processors. It provides developers with a platform to build portable and efficient software to manipulate these heterogeneous processing platforms.

OpenCL provides a mechanism to select manually the threadblock shape. The behavior of programs when using the CUDA driver with OpenCL are expected to be similar to direct CUDA programming. The effects discussed on previous sections are dependent on the hardware architecture, and thus, they will affect performance in the same way.

OpenCL includes a convenient mechanism to automatically select the threadblock shape, letting the programmer to skip this decision. However, our study on the use of this OpenCL automatic mechanism shows that it is focused on using the maximum number of threads per block (1024 on Fermi). Based on the previous discussion, the reader can expect that other smaller block sizes will lead to better performance.

5.2 Experimental study

In this section we introduce the design of experiments to verify the previously presented hypotheses and deductions derived from the architecture observation. We will run different benchmarks on a Fermi architecture platform to isolate and test different application features. Kernels used are intentionally simple, to minimize the interactions among different hardware effects, such as coalesced vs. scattered accesses, different ratios of workload vs. number of memory accesses, cache reutilization, etc.

5.2.1 Setup

The algorithms and coding ideas of some of the benchmarks are obtained from examples included in the CUDA and OpenCL SDK, or well-known linear algebra libraries, such as CUBLAS [88] and MAGMA [77]. The original codes cannot be directly used in our study because their optimizations and tuning strategies are dependent on specific threadblock sizes and shapes. For example, the threadblock sizes for the basic matrix multiplication on CUBLAS and MAGMA libraries is fixed to 512 and 256 respectively. We have adapted and simplified the codes to make them completely independent of the threadblock shape. We avoid the use of sophisticated tuning techniques to isolate the different effects of the block shape on each benchmark.

Although we focus on 1- and 2-dimensional problems, results can be extrapolated to 3-dimensional cases. The programs have been tested for different combinations of square- and rectangular-shaped threadblocks. We use shapes with a number of rows or columns which are powers of 2, or powers of 2 multiplied by 3, to include all the combinations that maximize Occupancy for Fermi. With this two shape configurations all blocks that maximize the SM Occupancy are obtained for Fermi [79]. Thereby, the power of 2 multiplied by 3 leads to blocks that does not perfectly exploit the coalescence. That is because multiple of 32 (the warp size) threads cannot be reached as cardinality in any of the two block dimensions.

The experiments have been conducted using integer and float elements. We present results for the integer arrays experiments. As the storage size of both types is the same, the effects on the memory hierarchy are similar. Float arrays experiments simply present slightly higher execution times due to the extra computation cost associated to the floating point operations.

We present one 1-dimensional benchmark. This benchmark uses one input vector with 1024k-elements. Therefore, we can use blocks with eight or more threads, to generate grids with no more than the maximum number of blocks allowed in CUDA for any dimension (e.g. in Fermi 65 535). For 2-dimensional benchmarks we use input matrices of 6 144 rows and columns. The size chosen has several advantages. First, this size is

small enough to allocate up to three matrices in the global memory of the GPU device used for the experimentation (see bellow). Second, this size ensures that not all blocks can be executed at the same time. This size mimics the behavior of bigger matrices, as long as data is aligned in the same way. Moreover, the dimensions of the matrices are multiples of: (1) all the block-shape dimensions tested; and (2) the number of global memory banks in our test platform (described below). Thus, matrix accesses on any threadblock are always aligned with the matrix storage, generating the same access pattern.

The experiments have been run on an NVIDIA GeForce GTX 480 device. The host machine is an Intel(R) Core(TM) i7 CPU 960 3.20GHz, 64 bits compatible, with a global memory of 6 GB DDR3. It runs an UBUNTU desktop 10.10 (64 bits) operative system. The programs have been developed using CUDA and OpenCL. The CUDA driver used was the version included in the 4.0 toolkit. All benchmarks were also executed with both OpenCL 1.0 and 1.1, with the same performance for both versions.

We measure performance considering only the total execution times of the kernel functions in the GPU. We skip initialization and CPU-GPU communication times because they are not related to our study. Our results are the arithmetic mean of the measures of at least three more executions.

5.2.2 Benchmarks with coalesced accesses

Vector reduction

We use a reimplementation of one of the CUDA SDK examples modified to allow the modification of the threadblock shape. The kernel is launched in several synchronized stages. On each stage each thread reduces two contiguous data elements, storing the result in a properly compacted ancillary data structure, used as input for the next stage. Thus, each thread issues two contiguous read requests to global memory, in two consecutive load operations. After the single arithmetic operation, each full warp performs a write request. No matter the number of active threads, the coalescing makes the warp to write the results in a single transaction segment (when the L1 cache is active, otherwise, results are stored in four transaction segment). The number of blocks is divided by two on each stage. The main code executes this kernel 16 times to generate enough load to obtain stable results.

Adaptive-block vector reduction

We suggested in previous sections that applications that executing a small amount of threads in the whole GPU can benefit from using many small blocks instead of a single bigger one, in order to spread the workload and exploit more parallelism across the SMs. Taking into account this observation, we have introduced an improvement on the

vector reduction code. The first stages are computed with a fixed block size. When we need less than 15 blocks (the number of SMs in our GPU testing device) to process the data, we divide the threadblock size to increase the number of blocks and the potential parallelism. This improvement is done on stages with low workload. We expect a slight performance improvement, more noticeable for small input data sets.

Matrix addition

This benchmark consists on a direct matrix addition ($C = A + B$) algorithm. Each thread is associated with a particular matrix position. This implies three global memory accesses per thread (two reads and one write). It presents a full coalesced memory access pattern, with no reutilization of the same matrix elements by other threads

Overloaded kernel

We have generated a synthetic program based on the matrix addition code. It simply adds an arbitrary number of dummy arithmetic operations (10 000) to the original single addition after loading the two elements. This code keeps the matrix addition access pattern, but introduces an overload between the load and the store global memory accesses obtaining a high-ratio of arithmetic operations per thread.

Overlapped memory accesses kernel

We propose another modification to the matrix addition code to force different warps in the same block to issue load global memory operations at different times. In Fermi there is a maximum of 48 warps in a SM. We use the warp number to select the exact amount of dummy arithmetic operations carried out before the loads (warpId $\times$ 1 000), and between the loads and stores ((48 - warpId) $\times$ 1 000). We have tested that 1000 dummy arithmetic operations take more time than the global memory latency. Thus, the warps of the same block that are scheduled to the Sm at the same time completely overlap the communication latencies of the load operations with computation across the different warps.

Naïve matrix multiplication

This benchmark is very simple and straightforward even for a non-experienced programmer. Each element of the C matrix, C_{ij}, is calculated through the scalar product of two arrays. The first one corresponds to the i-th row of the A matrix and the second one to the j-th column of the B matrix. There is reutilization of data between threads in the same block at different stages of the dot product. This naïve version is interesting due to the relationship of the reutilization of caches with the coalesced memory access pattern

Matrix multiplication by block products

We have coded two versions of matrix multiplication ($C = A \times B$) which are independent of the block size and shape. A *naïve matrix multiplication* The previously described, and an *iterative block-product matrix multiplication*.

In this new benchmark the values of result matrix are updated iteratively on each iteration. Each block of threads calculates a specific area of the result matrix, bringing to cache on each iteration as many elements of the two input matrices as threadblock size.

5.2.3 Benchmarks with non-coalesced accesses

Regularly-scattered accesses

This synthetic benchmark is designed to create a simple scattered access pattern in which each thread requests a different memory transfer segment. Each thread accesses a different matrix element. The position is computed multiplying the column index of the thread by the maximum size of a transfer segment (32 elements when L1 cache is active). The obtained global memory datum is modified with a single arithmetic operation and copied into the same element to reuse the same transfer segment.

Random accesses

This benchmark is a modified version of the previous one. Each thread copies one value from a random position of a matrix, in the same position of another matrix. The workload associated with computing the random indexes is higher than in the previous benchmark, and comprises around 20 arithmetic operations. Two memory transfer segments are requested per thread, one per each matrix access. The random indexes force most threads to request elements on different transfer segments, with little or no reuse of transaction segments contents. This benchmark simulates scattered accesses that typically appears in graph traversing algorithms, or codes for other sparse data structures.

5.2.4 Experimental results

In this section we present the results obtained by our experiments for both, CUDA and OpenCL implementations, in order to verify the hypotheses proposed in the previous sections. We discuss the results in terms of the effects related to the Fermi architecture features commented at the beginning of this chapter, and in Appendix A. We first describe the results obtained with CUDA, and then, we discuss the differences with OpenCL.

The results tables shown in this chapter contain the execution time of each scenario. The tables axes indicate the cardinality of threadblock dimensions.

Coalesced global memory accesses

Small kernels with no data reutilization: Table 5.1(A) shows the execution times of the matrix addition benchmark for different shapes. The results for the block sizes that maximize Occupancy are presented with a dark-grey background. The table does not show the first columns where the warps have at least three quarters of their threads idle, deriving in a quick grow of the execution time.

The table confirms the expected results, as previously discussed in Sect. 5.1.1: (1) The best performance is obtained with block sizes that maximize Occupancy; (2) considering the maximum Occupancy block sizes, diagonals of smaller blocks present better performance, with the optimum in blocks of 192 threads; (3) blocks with less than 32 columns perform really bad due to the loss of coalescing and idle threads in the warp; and (4) blocks with more columns and less rows, imply less conflicts when accessing the global memory banks (with this kind of shapes the same warp accesses to a lower number of global memory banks).

Table 5.4(A) shows the execution times of the vector reduction benchmark. For the vector sizes chosen, we cannot choose blocks sizes below 32, due to the maximum number of blocks per dimension supported in CUDA (reducing the size of the blocks increments the number of them for the same input set). Having similar properties than the matrix addition, the effects observed are similar, and the best performance is found in 192 threads per block. Our results indicate that the adaptive-block vector reduction improves the performance only for small input data sets. For example, 3 % to 4 % for 1023 k-elements input vector. This technique has more impact on computations with a higher workload per thread.

As described in Sect. 5.1.1, for coalesced codes without data reutilization, deactivating the L1 cache does not affect performance. There are four more transaction segment requests, but the segments are four times smaller. The final requested global memory bandwidth does not significantly change.

Higher loaded kernels: Table 5.1(B) shows the execution times of the overloaded kernel. As expected for high loaded coalesced codes, with no data reutilization across threads, and low number of memory accesses comparing with arithmetic operations, the results indicate that any shape that maximizes occupancy produces a similar performance. The big number of computation compared to the number of global memory accesses per thread reduces the effect produced by the block shape on the memory accesses. In these cases, the computations govern the performance behavior.

The effect of the faster replacement of ending warps when smaller blocks finish, is negligible comparing with the overall computation. In this code, all warps begin executing the load accesses at almost the same time. Thus, latencies are not really well hidden across

(A) Matrix Addition: Execution Times

Rows \ Columns	8	12	16	24	32	48	64	96	128	192	256	384	512	768	1 024
128	9,01														
96	6,35														
64	5,49	5,48	5,78												
48	5,24	5,60	4,00												
32	4,86	4,80	3,36	3,88	4,23										
24	4,70	4,68	3,28	4,17	3,07										
16	5,05	4,43	3,23	3,45	3,04	3,38	4,14								
12	5,42	4,58	3,28	3,55	3,02	3,43	3,05								
8	6,05	4,97	3,42	3,41	2,96	3,05	2,95	3,17	4,45						
6	7,18	5,52	3,87	3,53	2,95	3,06	2,94	3,46	3,19						
4	9,70	6,73	5,06	3,92	3,11	3,10	2,90	3,00	3,05	3,19	4,44				
3	11,94	8,48	6,16	4,64	3,53	3,12	2,89	2,96	2,95	3,35	3,18				
2	16,43	11,43	8,55	6,12	4,54	3,68	3,08	2,93	2,93	2,95	2,96	2,99	3,95		
1	29,97	20,24	15,16	10,51	7,74	5,73	4,40	3,49	3,08	2,89	2,89	2,91	2,94	3,01	3,94

(B) Overloaded Kernel: Execution Times

Rows \ Columns	8	12	16	24	32	48	64	96	128	192	256	384	512	768	1 024
128	1353														
96	1351														
64	1349	1350	1351												
48	1349	1351	1350												
32	1349	1349	1349	1349	1350										
24	1349	1349	1349	1349	1348										
16	1349	1349	1349	1349	1349	1349	350								
12	1349	1499	1349	1349	1348	1349	1348								
8	1352	1349	1349	1349	1349	1349	1349	1348	1350						
6	1802	1799	1349	1499	1349	1349	1348	1349	1348						
4	2322	1803	1352	1349	1349	1349	1349	1349	1349	1348	1350				
3	3096	2403	1802	1799	1349	1499	1349	1349	1349	1349	1348				
2	4648	3099	2325	1803	1352	1349	1349	1349	1349	1349	1349	1348	1350		
1	9286	6195	4645	3099	2325	1803	1352	1349	1349	1349	1349	1348	1349	1348	1350

Table 5.1: Execution times for the benchmarks considered (A,B,part 1). Time in milliseconds. Maximum Occupancy with dark-gray background in Fermi architecture.

(C) Overlapped Memory Access Kernel: Execution Times

Rows \ Columns	8	12	16	24	32	48	64	96	128	192	256	384	512	768	1 024
128	901														
96	873														
64	851	872	895												
48	841	851	873												
32	831	840	850	873	896										
24	827	835	841	849	873										
16	822	828	831	842	852	873	896								
12	819	915	828	834	842	847	873								
8	828	819	824	828	832	842	851	873	896						
6	1103	1092	819	914	828	834	842	848	873						
4	1438	1104	828	819	823	828	832	842	851	873	896				
3	1916	1469	1103	1092	818	914	828	834	842	848	873				
2	2871	1915	1436	1103	828	819	823	828	831	842	852	873	896		
1	5741	3828	2871	1915	1437	1103	828	818	823	828	832	842	852	873	896

(D) Regularly-scattered: Execution Times

Rows \ Columns	8	12	16	24	32	48	64	96	128	192	256	384	512	768	1 024
128	12,82														
96	12,68														
64	12,58	12,60	12,63												
48	12,44	12,47	12,51												
32	12,29	12,34	12,33	12,39	12,33										
24	12,13	12,18	12,16	12,17	12,19										
16	11,89	11,99	11,98	12,01	12,00	12,05	11,93								
12	11,63	11,78	11,83	11,87	11,86	11,81	11,88								
8	11,25	11,41	11,53	11,71	11,74	11,73	11,71	11,75	11,65						
6	10,89	11,15	11,28	11,50	11,63	11,65	11,63	11,54	11,67						
4	11,53	11,52	11,52	11,16	11,31	11,52	11,55	11,53	11,52	11,52	11,28				
3	10,28	10,42	10,56	10,95	11,08	11,36	11,45	11,48	11,46	11,31	11,44				
2	10,23	9,99	10,06	10,53	10,67	11,01	11,14	11,39	11,41	11,38	11,35	11,35	10,92		
1	12,80	11,42	10,02	9,79	9,91	10,43	10,54	10,88	11,09	11,34	11,35	11,31	11,24	11,24	11,15

Table 5.2: Execution times for the benchmarks considered (B,C,part 1). Time in milliseconds. Maximum Occupancy with dark-gray background in Fermi architecture.

(E) Regularly-scattered without L1 cache: Execution Times																		
Rows	Columns																	
	8	12	16	24	32	48	64	96	128	192	256	384	512	768	1 024			
128	10,49																	
96	10,38																	
64	10,27	10,04	10,22															
48	10,09	9,78	10,07															
32	9,90	9,73	9,86	9,89	9,68													
24	9,61	9,48	9,58	9,59	9,52													
16	9,13	9,20	9,33	9,38	9,27	9,36	9,06											
12	8,57	8,88	9,06	9,12	9,05	8,96	9,02											
8	7,79	8,02	8,42	8,89	8,85	8,90	8,80	8,85	8,63									
6	7,10	7,50	7,79	8,41	8,65	8,70	8,63	8,44	8,69									
4	8,29	8,30	8,31	7,60	7,85	8,29	8,37	8,34	8,34	8,30	7,83							
3	6,97	6,58	6,63	7,21	7,25	8,02	8,14	8,14	8,16	7,84	8,05							
2	7,94	6,79	6,44	6,61	6,59	7,17	7,22	7,84	7,86	7,82	7,80	7,86	7,30					
1	10,91	8,78	7,64	6,64	6,29	6,42	6,42	6,73	7,13	7,74	7,69	7,67	7,48	7,62	7,34			

Table 5.3: Execution times for the benchmarks considered (E,part 1). Time in milliseconds. Maximum Occupancy with dark-gray background in Fermi architecture.

(A) Vector reduction											
	Columns										
Rows	32	48	64	96	128	192	256	384	512	768	1 024
1	1,1087	0,9048	0,7893	0,6852	0,6582	0,6268	0,6358	0,6312	0,6330	0,6635	0,7885

(B) Naïve Matrix Multiplication											
	Columns										
Rows	32	48	64	96	128	192	256	384	512	768	1 024
32	6441										
24	5842										
16	5218	6094	6579								
12	5121	6478	5979								
8	4982	5862	5265	5479	6470						
6	4860	5775	5293	5940	5457						
4	6177	4746	4855	4898	4915	4743	6066				
3	7960	5918	4653	4928	4649	5421	4520				
2	11890	8121	6103	4415	4339	4325	4450	4288	6172		
1	23730	16086	12073	8399	6967	5855	5866	5909	6120	5951	7223

(C) Random access											
	Columns										
Rows	32	48	64	96	128	192	256	384	512	768	1 024
32	*										
24	347,27										
16	388,41	345,63	*								
12	334,81	367,85	347,79								
8	330,47	332,90	388,28	347,56	*						
6	324,86	325,46	334,77	369,78	347,23						
4	325,50	347,18	330,47	334,99	388,41	347,18	*				
3	328,07	357,41	324,82	327,35	334,75	369,59	347,30				
2	370,40	326,11	325,48	324,88	330,43	334,72	388,42	347,17	*		
1	634,43	490,32	370,41	328,04	325,47	324,82	330,43	334,80	388,49	347,45	*

Table 5.4: Execution times for the benchmarks considered (part 2). Time in milliseconds. Maximum Occupancy with dark-gray background in Fermi architecture.

the warps of a block.

Hiding global memory latencies: Table 5.2(C) shows the execution times of the overlapped memory accesses kernel. Recall that this benchmark code ensures that the latencies of global-memory loads on any warp are completely overlapped with the computation of other warps in the block. We observe that the best performance is obtained for blocks with less than 192 threads. As expected, in this type of code maximum Occupancy is not needed to hide latencies, because they are hidden by the computation overlap.

Intensive data reutilization: Table 5.4(B) shows the execution times of the naïve matrix multiplication code. The tables for this benchmark skip all the columns where the warps have idle threads and the execution time explodes. Due to the high reutilization of data, bigger block sizes have more opportunities to reuse the cache lines. Thus, the best performance results are for the biggest block size that maximizes Occupancy (768). For a given block size, we also observe a trend that leads to obtain better performance results when using a shape with more columns and less rows. Blocks with more columns,

	Columns										
Rows	32	48	64	96	128	192	256	384	512	768	1 024
32	30										
24	344										
16	352	568	38								
12	362	257	342								
8	357	625	366	331	64						
6	357	580	372	196	306						
4	201	610	334	354	300	256	124				
3	177	379	291	313	264	166	190				
2	200	343	253	277	255	250	249	246	245		
1	366	603	437	490	513	529	520	509	504	496	488

Table 5.5: Naïve matrix multiplication. L1 Cache misses.

up to 384 (due to the 6 global memory banks on our device) reduce the number of bank conflicts. It also impacts on the reutilization and trashing of the L1 cache due to the algorithm properties. Table 5.5 shows the number of L1 cache misses as reported by the visual profiler included in the CUDA toolkit.

For maximum Occupancy we observe a clear correlation between performance and the number of L1 and L2 cache misses. Blocks with one row have no opportunity to exploit any reutilization of data on the second matrix. Thus, they produce many more cache misses and their performance is degraded. The best performance is found for the shape 2×384. With this block shape a proper cache misses balance between A-B matrices is reached and then, the global memory accesses are reduced.

Increasing the L1 cache size to 48 KB, reduces cache misses and produces an improvement of performance between 0.3 % to 7.5 % for block sizes with maximum Occupancy.

Matrix multiplication by block products performs worse than the naïve implementation for the same shapes. The naïve version achieves better reutilization of data, as all warps work on the same parts of the first matrix during the dot product evolution. Naïve multiplication algorithm is more suitable for multi- or many-cores architectures with cache hierarchies, while block is more appropriate for distributed-memory with higher communication latencies.

Non-coalesced accesses

Regularly-scattered accesses: Table 5.2(D) shows the execution times of the regularly scattered accesses benchmark. As discussed in Sect. 5.1.1, we expect that this type of code does not need to maximize Occupancy, due to the big amount of simultaneous memory requests whose latencies cannot be hidden. Moreover, reducing the number of threads per block also alleviates the global memory bandwidth bottleneck. We observe that the best performance is obtained for blocks with only 24 threads (shapes with 1×24, or 2×12 threads). As reasoned in Sect. 5.1.1, having even some idle SPs per warp is compensated

by the reduction of the memory bandwidth bottleneck, and the bank conflicts.

Table 5.3(E) shows the execution of this benchmark with L1 cache deactivated. As expected, reducing the transfer segments alleviates the global memory bandwidth problem up to the point that it is not needed to have idle SPs, and the best block size moves from 24 to 32 threads per block (warps with all threads active). The performance improvements are in the range of 20 % to 40 % for block sizes of 16 or more threads.

Random accesses: Table 5.4(C) shows the execution times of the random accesses code. This program uses many registers for the computation of the random indexes. Thus, the Occupancy level is reduced for any shape. The cells with a star indicate block sizes that cannot be executed due to the exhaustion of resources, leading to zero Occupancy.

This program has a medium ratio of global memory accesses per arithmetic operations (2 global memory accesses vs. 20 arithmetic operations). This ratio is not small enough to eliminate the effect of improving performance when reducing the block size, up to 192 for maximum Occupancy. For this type of codes, the block size is the key decision. The shape is not relevant due to the random access pattern used on each thread, that distributes global memory accesses across global memory banks, and no coalescing can be exploited.

The medium workload on the threads helps to hide the global memory access latencies when the L1 is active. Although this code has a non-coalescent pattern, the deactivation of L1 cache only improves the performance slightly (less than 1 % for any shape).

OpenCL results

All the results obtained with OpenCL 1.0 and 1.1 consistently show the same effects discussed for CUDA. The versions of OpenCL tested introduces a performance penalty on the Fermi architecture used for the experimentation, due to the OpenCL abstraction layer.

We have tested the mechanism provided by OpenCL to automatically select the block size and shape. For all our benchmarks, the results indicate that this mechanism systematically chooses threadblock sizes of 1024 threads, maximizing the number of threads per block. However, due to the Fermi architecture particular features, this block size does not maximize Occupancy. Thus, the performance obtained with this mechanism is always far from the optimum, with performance degradations between 28 % and 65 % in our benchmarks.

This technique can easily be improved using a simple and conservative strategy, selecting blocks of 768 threads for big kernels, and 192 for small ones. More sophisticated techniques can be devised using code analysis to detect other code features, such as coalescing vs. scattered accesses, ratio of global memory accesses, total workload, etc. as we will present in Sect. 5.3.

5.2.5 Limitations of this experimental study

One of the most important decisions when programming a GPU is to choose specific values for global programming parameters, such as the threadblock size and shape, in order to achieve the highest performance. However, these parameters are usually chosen by a trial-and-error process.

The choice of global parameters are closely related to the particular parallel problem implementation. We have shown that a combined analysis of the knowledge of a specific GPU card architecture, code features such as the type of global memory access pattern (coalesced vs. scatter), the total workload per thread, and the ratio of global memory read-write operations, can significantly affect the choice of important programming parameters, such as threadblock shape, and the L1 cache memory configuration. Nevertheless, there are some significant gaps in this study that will be considered in the following section.

The most important gaps that have not been considered so far are:

- More types of global-memory access patterns.
- Ratio of L1 cache memory lines evictions compared to the size of this memory.
- Ratio of global memory data reutilization across threads in the same block, and across blocks, compared to the number of global memory accesses per thread.
- The use of more GPU architectures.

5.3 Micro-benchmarks (uBench)

This section describes a suit of micro-benchmarks, that we named uBench [116, 117, 120], in order to overcome the limitations of our previous study, described above. To do so, we will explore the impact on performance of (1) the thread-block size and shape choice criteria, and (2) the GPU hardware resources and configurations, with the help of a tailored micro-benchmark suite designed to explore the performance impact of the GPU configuration parameters. The knowledge gained thanks to the uBench suite will be used as a guideline to help the programmer to choose good values for the GPU configuration parameters.

5.3.1 The uBench suite

uBench design principles

We first discuss the design principles of the uBench micro-benchmark suite. To better understand the effects produced in the L1 cache, and to isolate effects derived from the

global memory access patterns, all benchmarks use as input/output parameter a single array structure. Some benchmarks logically access it as a two dimensional matrix, while others access it as a vector.

- **Data sizes and storage order**: We have decided to simplify communications and data transfers between the main programs written in C and the CUDA GPU kernel. Matrices are stored in row-major order. All benchmarks work with integer elements. Float elements use the same memory space as integers, and each SP has one unit for integer and one unit for single precision arithmetic operations. This kernels working with single-precision float elements have a similar behavior as integer computations. On the other hand, double precision numbers require double memory space, and there are not as many double precision arithmetic units as SPs. Instead, two SPs are coordinated to issue one single double precision operation. Thus, considering one double precision number as two floats, most performance effects can be extrapolated.

  The number of threads launched by the uBench kernels is equal to the number of matrix elements. We have designed micro-benchmarks so that each one fulfills the following guidelines: (a) Each thread access only one global memory location, a different one for each thread, and (b) each thread access several global memory locations with a given pattern, exploring effects that appear in scenarios where there are more input data elements than the number of kernel threads.

  Coalescing is one of the most important issues that affect the code performance, as it is mentioned in Chapter 2 and Appendix A. Different micro-benchmarks explore different classes of coalescing patterns. Nevertheless, negative performance effects can also appear due to conflicts in global memory banks (this effect is known as Partition Camping [4, 47]). Due to the introduction of L2 cache memory in Fermi and Kepler architectures, the global-memory bank conflicts are significantly reduced on applications with high transaction-segment reutilization. However, with low data reutilization, the L2 cache has a limited beneficial effect, and the global-memory bank conflicts can appear. We design micro-benchmarks with different data reutilization degrees to test this effect.

- **Data sizes and alignment**: Threads accessing out of the bounds of data structures should be avoided in kernels. In codes that correlate thread indexes with data-structure indexes, CUDA programmers tackle this problem in two different ways. Either the kernels include divergent branches to skip processing for out-of-bounds threads, or data padding is added to the data structures to align them with the chosen threadblock shape. In both cases the performance impact is very small. More irregular applications may need more sophisticated codes to deal with the align-

ment of threads and data structures. Their behavior can be extrapolated from results of micro-kernels that test access patterns not correlated to thread indexes in array structures.

Alignment of data structures to threadblock sizes. To keep the micro-kernel and the launcher codes simple, we select matrix sizes with dimensional cardinalities which are multiples of any of the threadblock dimension cardinalities to be considered in the study. In this way, we can avoid data padding, or trivial divergent branches, without losing generality.

Memory bank alignment. In order to selectively introduce memory access patterns that reproduce or skip the effects of the global-memory bank conflicts described above, the cardinalities should also include multiples of (a) the number of memory banks, and (b) the width of the memory-bank, for any CUDA architecture.

Choosing the order of magnitude of array sizes. Coalescing patterns can effectively hide global memory latencies when there are enough warps scheduled in the SM during the computation. If the total amount of threads is not enough to fill all the SMs in the GPU device, there are not enough active warps to hide global memory latencies. In our study, the *Total Amount of Threads* in the whole computation (TT) has been forced to be the same as the total size of the data set. The maximum number of active threads in the whole device (MT) is the product of the maximum of active threads per SM by the number of SMs (recall that in Fermi $MT = 1\,536 \times [14, 16]$, while in current Kepler release $MT = 2\,048 \times 8$). We choose different matrix sizes with the following criteria: (1) TT less than MT; (2) TT slightly higher than MT (latency hiding may start to happen); and (3) TT much higher than MT (latency hiding can be fully exploited). To keep execution times bounded, for this last category we select two different matrix sizes: The first one, for the kernels with high computational load, is 1 024 times bigger than the size in category (2); the second one, for those kernels with low computational load, is 9×1024 times bigger than the size in category (2) (to generate an array that is near to fill up the global memory of the devices).

The matrix sizes chosen to achieve all the previous criteria in the different CUDA architectures considered are:

- Category-1 N=96×96
- Category-2 N=192×192
- Category-3 N=6 144×6 144 or N=18 432×18 432

- **Computational load and independence of L1 cache configuration**: The micro-kernels are designed to execute at most one basic arithmetic operation for each

data element accessed. Some of them include an extra loop with dummy computations with constant values, or with a value already read before the dummy loop. In this way, we generate a configurable load by changing the number of iterations. Additionally, overload can be added to ensure that the computation times between memory accesses are higher than the global-memory latencies. This allows to test the impact of hiding global-memory latencies by coalescing, and/or by overlapping computation with communication.

All uBenchs are designed to correctly work regardless of the L1 cache configuration. In this way the performance behaviors can be tested using the different cache configurations (enabled with 16K, enabled with 48K, or disabled).

- **Access patterns**: We have designed a first subset of uBench micro-kernels with only one global memory access access per thread, in order to isolate the effects produced by single global-memory access patterns. These kernels do not read data. Instead, they use constant values or computed data to write in global memory. To study the interaction of read-write patterns, we have developed a second set of micro-kernels combining more sophisticated read patterns with a simple writing pattern.

  These read/write patterns do not intend to represent the whole space of access patterns that can appear in an application. Instead, we focus on pattern classes that can produce different performance trends due to hardware effects. The selected patterns cover: The main categories of coalesced patterns; patterns that spread concurrent accesses across the global memory banks vs. patterns that stress the bank-conflicts; and patterns that interleave accesses across threadblocks vs. patterns that make threads in different blocks traverse data structures simultaneously.

  Regarding writing patterns, we have chosen first two basic types of coalesced patterns. Both represent patterns where each thread accesses only one data element. Then, we also add three more types, to study non-coalesced patterns and other special situations.

  - *Pattern I:* Each thread accesses one matrix element using the thread global coordinates (y index indicates the row, and x index indicates the column):

$$\begin{aligned} \underline{row} &\leftarrow blockIdx.y \times blockDim.y + threadIdx.y \\ \underline{column} &\leftarrow blockIdx.x \times blockDim.x + threadIdx.x \end{aligned}$$

– *Pattern II:* Each thread is assigned to a single uni-dimensional coordinate used to access the array as a vector. For a given grid, independently of the thread-blocks shape and size, each thread always accesses to the same data position.

$$\underline{index} \leftarrow (blockIdx.y \times gridDim.x + blockIdx.x) \times blockSize + (threadIdx.y \times blockDim.x + threadIdx.x)$$

– *Pattern III:* The threads use their local block indexes to compute a flattened index: $threadIdx.y \times blockDim.x + threadIdx.x$. Then, each thread writes in the matrix element corresponding to element of the main diagonal with the thread index on both dimensions. This completely non-coalesced pattern ensures that all threads in the block access to a different transaction segment. But all blocks access to the same small set of transaction segments. Thus, there are not cache trashing effects, and there is high reutilization across blocks.

– *Pattern IV:* Only one thread per block (the one with $threadIdx.x = threadIdx.y = 0$) writes in a matrix position selected as in Pattern I. The remaining threads do not perform any access. This pattern recreates the effect of sparse patterns but, due to minimum number of accesses per block, there are no cache thrashing effects involved.

– *Pattern V:* All threads access to the first position of the array ($vector[0] = value$). This pattern has been designed to produce a high degree of memory bottleneck.

Finally, reading patterns include different types of coalescing and memory alignment techniques, including examples in which threads read many data elements. They produce different degrees of data reutilization.

– *Pattern A:* Each thread reads the full column of the matrix with its global x index. It is a perfectly coalesced pattern with reutilization of the data by other threads of the same column in different iterations. Thus, reutilization inside the block is dependent on the exact shape. In different iterations the same memory banks are accessed.

– *Pattern B:* Each thread reads the full row of the matrix with its global y index. Data that are in consecutive positions in global memory are read in different loop iterations. This pattern can be considered coalesced in the sense that on each read operation, threads in the same warp are accessing to data in the same transaction segment. There is reutilization of the transaction segments on the cache across iterations, and also by threads in the same row. Reutilization

inside the block is dependent on the exact shape. The accesses of all the threads from the same block are concentrated in the same memory bank on each iteration, cyclically changing the bank as the loop advances.

- *Pattern C:* The threads compute a starting position in the array using the block global indexes and the block size. All the threads in the same block obtain the same position. Threads from different blocks obtain positions which differ in a multiple of the block size. The threads execute the same loop to read all the array positions corresponding to the block. All threads in the same block reuse the same data independently of the shape.
- *Pattern D(s):* All threads traverse once the whole array structure as a vector. However, threads from different blocks start at a different position, traversing the vector cyclically. The position is computed using the global block identification: $blockIdx.y \times gridDim.x + blockIdx.x$, multiplied by a stride parameter (s). When $s = 1$, there is a high overlapping of blocks accessing to the same transaction segments, producing bank conflicts, but there is also a very high reutilization of L1 and L2 caches. When $s = 32$, we ensure that each block is accessing to different transaction segments, and the accesses are balanced across memory banks. Bank conflicts are reduced but reutilization of caches, specially L2, also decreases. Note that L1 caching in Kepler architectures is reserved only for local memory accesses, such as register spills and stack data. Thus, in Kepler, global loads are cached in L2 only (see Sect. A.5 of Appendix A.1).

uBench suite description

This section include an enumeration and short description of the benchmarks included in the uBench suite.

uBench-0 This kernel is designed to test the performance impact of the device schedulers when facing different threadblock shapes, without memory access interferences. It does not do computation, and neither do the threads access any data.

uBench-1 No reads. Each thread copies the same constant value in its position using Pattern I.

uBench-2 No reads. Each thread copies the same constant value in its position using Pattern II.

uBench-3 No reads. The same as uBench-1 but with an overload loop with one thousand iterations.

uBench-4 No reads. The same as uBench-2 but with an overload loop with one thousand iterations.

uBench-5 No reads. Each thread copies the same constant value in a position using Pattern III.

uBench-6 No reads. Only the thread with $threadId.x = 0$ and $threadId.y = 0$ of each block stores in its global matrix position the same constant value.

uBench-7 No reads. Each thread copies a calculated value to the first vector position. The values are calculated using a loop of one thousand iterations.

uBench-8 Each thread copies in its position the sum of the matrix values in the column with its global x index.

uBench-9 Each thread copies in its position the sum of the matrix values in the row with its global y index.

uBench-10 Similar to uBench-8 but with an overload loop with one thousand iterations.

uBench-11 Similar to uBench-9 but with an overload loop with one thousand iterations.

uBench-12 Each thread sums the values of a matrix block selected with Pattern C, and stores the result in a position selected using Pattern II.

uBench-13 Each thread stores in its position the sum of all the elements of the whole data structure. Each thread starts to cyclically traverse the array at a different position with stride 1.

uBench-14 The same as uBench-13 but using stride 32 to compute the starting position.

uBench classification criteria

The uBench benchmarks have been classified according to the following criteria (summarized in Table 5.6):

1. Types of global-memory access patterns.
2. Ratio of arithmetic instruction per thread compared to the number of global memory access (high, low, or none).
3. Ratio of L1 cache memory lines evictions compared to the size of this memory (low, medium, or high).

uBench	(1) Access patterns	(2) Load ratio	(3) L1 eviction ratio	(4) Data reutilization in/across blocks
uBench-0	-.-	None	None	None
uBench-1	-.I	Low	Low	Low
uBench-2	-.II	Low	Low	Low
uBench-3	-.I	High	Low	Low
uBench-4	-.II	High	Low	Low
uBench-5	-.III	Low	Low	Low/High
uBench-6	-.IV	Low	Low	Low/High
uBench-7	-.V	High	Low	High/High
uBench-8	A.I	Low	High	Shape/Medium
uBench-9	B.I	Low	Medium	Shape/Medium
uBench-10	A.I	High	High	Shape/Medium
uBench-11	B.I	High	Medium	Shape/Medium
uBench-12	C.II	Low	Medium	High/Low
uBench-13	D(1).I	Low	Medium	High/High
uBench-14	D(32).I	Low	Medium	High/Medium

Table 5.6: uBench classification, according to the criteria proposed.

4. Ratio of global memory data reutilization across threads in the same block, and across blocks, compared to the number of global memory accesses per thread (low, medium, or high). We also consider a special class (*Shape*) for those benchmarks in which in-block reutilization is dependent on the exact shape of the block.

5.3.2 uBench evaluation

Experiments have been conducted for all the benchmarks described using both Fermi and Kepler architectures. Due to the maximum number of threads per threadblock supported by Fermi and Kepler architectures (1 024 threads) the threadblock shapes should fulfill the following criteria: (1) (#rows and #columns) $\in [1, 1\,024]$; (2) (#rows $\times$ #columns) $\leq 1\,024$. To reduce the search space, we only use threadblock geometries where (3) #rows and #columns are multiple of two and/or three. This ensures that we include in the tests all possible combinations that can derive in maximum Occupancy in any current CUDA architecture. Recall that the maximum number of concurrent threads per SM in Fermi is 1 536, which is a multiple of three. We have explored, in terms of execution time, all the combinations of sizes for each threadblock dimension that complies with the previous shape restrictions. (1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 128, 192, 256, 384, 512, 768, 1 024), of L1 configurations (enabled, disabled, increased), and of input data sizes (input_data_size=$ids_i/i \in [1, 2, 3A, 3B]$; ids_1= N$\times$M= 96$\times$96= 9 216,

ids_2= N×M= 192×192= 36 864, and ids_{3A}= N×M= 6 144×6 144= 37 748 736 or ids_{3B}= N×M= 18 432×18 432= 339 738 624).

To show an example of how the results obtained with uBench can be extrapolated to real life applications, we have tested two different CUDA implementations, of one real-world application: Cannon's algorithm for matrix-matrix multiplication (see Appendix B). The first implementation is a direct translation of the original algorithm, and the second one modifies the order in which matrix blocks are accessed to force each threadblock to start reading from a different global memory bank. The codes are included in the uBench release.

The experiments have been run on a GeForce GTX 480 (Fermi) and a GForce GTX 680 (Kepler) NVIDIA GPU devices. The uBenchs have been developed using CUDA 4.2 toolkit and the 295.41 64-bit driver.

All the numerical results are compiled into 270 tables for the uBench kernels, and 36 tables for the Cannon's algorithm implementations. The tables are presented in a technical report publicly available [117]. The tables are accompanied by graphical diagrams. We include two example diagrams in Fig. 5.2 to help the reader to understand how we used them in the following discussion of results. During the following discussion, we consider *good choices* for threadblock geometry those that lead to execution times with less than 5 % difference with the best execution time obtained with any geometry.

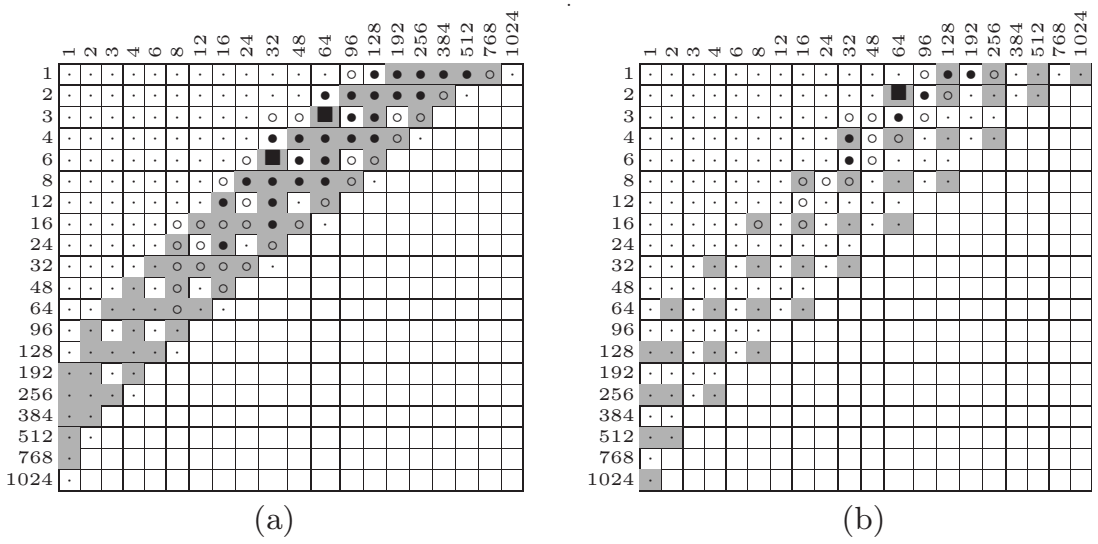


Figure 5.2: Example of diagrams for results tables for uBench-1, with default L1 configuration, matrix size 6144 × 6144 ($TT >> MT$), in (a) Fermi, and (b) Kepler architectures. Grey shaded cells indicate block geometries that lead to maximum occupancy (recommended block geometries). Symbols used: ■ for best performance results; • for execution times up to 5 % more than the optimum; ○ for execution times up to 25 % more than the optimum.

ThreadBlock geometry and the scheduling system

The scheduling system of both Fermi and Kepler architectures works faster with blocks of medium size. The results of uBench-0, with no memory accesses and no computation on any thread, show this effect. As the total number of threads and blocks increases the threadblock sizes that are close to the optimal value, that we name *good blocks*, concentrate more and more in the geometries with a medium size (384 in Fermi, 256 in Kepler), independently of the shape. Compared with Fermi, Kepler presents a performance degradation of up to 20 % for the good threadblock geometries. The reason is that Kepler's block/warp scheduling system has more complex units and it is slower than Fermis's. The impact of this effect can also be partially seen in kernels with scarce data accesses and low computation, like uBench-6.

Coalesced patterns for simple writing operations

The threadblock geometry choice has a great impact in the performance of typical coalesced patterns. Results for uBench-1 to uBench-4 show the key trends. uBench-1 explores a classical coalesced pattern, with low ratio of arithmetic operations per thread, and reutilization of the same transaction line only due to coalescence. Not only the size, but the shape of the threadblock, highly influences the performance (see Fig 5.2). Good results are obtained for the smallest block sizes that lead to near maximum occupancy (more than 90 % of Occupancy), and shapes with a number of columns equal to, or greater than, the size of the SM's scheduling unit: Half a warp in Fermi (16 threads); a whole warp in Kepler (32 threads). Smaller blocks derive in faster replacement of the blocks that finish by new blocks in the SM queues. While in Fermi similar good results are obtained for sizes from 192 to 512, the trend to obtain better results for smaller blocks is more clear in Kepler, where good candidates are those with 128 and 192 threads. In Fermi architecture, even blocks with 128 threads (leading to occupancy of 67 %) also derive in good performance. These results confirm and generalize the observations discussed in Sect. 5.1.

uBench-2 results also confirm that the performance obtained for good candidates in uBench-1, can be obtained for any shape of the same size if the proper pattern can be devised for the specific application.

For a very small number of total threads there are not enough warps to hide latencies due to coalescing. Thus, the restrictions on the threadblock shape, and the policy of trying to achieve maximum occupancy of the SMs become less relevant. This is confirmed by the results of all the benchmarks with small input sets ($TT < MT$).

When there is a high ratio of arithmetic operations per data access, the global-memory latencies are overlapped with computation, obtaining good performance for all sizes and shapes that lead to near maximum occupancy. uBench-3 and 4 are versions of 1 and 2 with extra computational load. Their results for enough number of threads $TT >> MT$,

show this effect. The same effect is also noticeable in the results of uBench-7, that also includes an overloaded loop that allows to hide memory latencies.

For these benchmarks, when $TT >> MT$, performance for the best threadblock geometry choices is not affected by changing the L1 cache configuration.

Non-Coalesced and scarce patterns

Non-coalesced patterns can not benefit from the previous global-memory latency hiding effects. We do not find shape restrictions, and for certain applications the best performance results are found with small threadblock sizes that lead to medium occupancy factors. This effect is more noticeable in Fermi due to the faster scheduling system. For example, uBench-5 results for Fermi show that the best choices are blocks with 64 threads, while in Kepler the best choices are blocks with 128 to 192 threads. uBench-6 uses a pattern where only one thread per block writes, but each block writes in a different position. The amount of writing operations is very small and there is no memory bottleneck. Due to the small amount or writing operations, the good block sizes are bigger than for uBench-5: 256 to 768 threads in Fermi, and 256 to 512 in Kepler.

Changing the configuration or the L1 cache does not significantly change the performance results for the best threadblock geometry choices.

Reading multiple data elements with coalesced patterns

Each thread accessing to all elements of the data structure: When each thread traverses whole parts of a data structure, the amount of data reutilization across the threads of the same block depends on the form of the shape. For example, uBench-8 presents different types of coalesced patterns, while traversing a complete row or column of a matrix. This uBench has the classical coalesced pattern with consecutive threads in a warp reading consecutive data elements from the same transaction segment. Good results are obtained for the smallest block sizes that lead to near maximum occupancy, and shapes with a number of columns equal to or greater than the size of the SM's scheduling unit. The same conclusions as presented for uBench-1, which have a similar pattern for writing.

Warp threads accessing to the same data: uBench-9 presents a complete different type of coalescence, where consecutive threads in the warp access to the same data element in the same loop iteration, and to consecutive data elements across loop steps. There is only one transaction segment required per block row simultaneously. Results indicate that, in this case, the columns limitation due to the scheduling unit also appears, but any block size that achieves a near to maximum occupancy produces good performance. uBench-10 and 11 are versions of uBench-8 and 9 with a loop that introduces extra computational load between consecutive accesses. As expected, the exact shape becomes irrelevant and

any block with a size that produces near maximum occupancy obtains good performance results.

Data reutilization in the same block: A different scenario appears when threads in the same block highly reuse the same data, and there is no reutilization across blocks. In this case, the good choices for block size are related to the transaction segments size.For example, the results of uBench-12 show that the good block sizes are between 24 and 32 for both architectures.

L1 cache for Kepler and Fermi: The impact of L1 cache configurations (enabled or disabled) is much more noticeable in Kepler than in Fermi. Kepler supports twice the same blocks and threads in an SM than Fermi, while the L1 size is the same.

Memory bottlenecks: There are several techniques to alleviate memory bottlenecks that lead to performance improvements. In general, these techniques improve performance without changing significantly the conclusions about the threadblock geometry choice. For example, uBench-13 overlaps the accesses of consecutive blocks of the grid. Results show similar behavior as the reading coalescing pattern in uBench-8, with L2 cache alleviating the bank conflicts to obtain better performance. uBench-14 tries to alleviate memory bottlenecks distributing the accesses of consecutive blocks across banks, like it is explained in [47]. For uBench-14 the good threadblock geometries are those with 384 threads, independently of the shape.

Extrapolation to non-synthetic applications

Observation of the main loop in the threads of the CUDA implementation of Cannon's algorithm easily reveals the access patterns for the two input matrices (Pattern A, and Pattern B, respectively), and for the output matrix (Pattern I). The results show that the choice of the threadblock geometry is influenced mainly by Pattern A and Pattern I (smallest block sizes that lead to near maximum occupancy, and shapes with columns equal to or greater than the size of the SM's scheduling unit). But the influence of the writing Pattern B also relaxes the first condition, leading to similar performance in blocks with slightly bigger or smaller sizes than the previously proposed good candidates.

For the modified version, we apply the technique discussed in [47] to alleviate bank conflicts spreading accesses from consecutive blocks to different banks. As expected, the results show performance improvements for the same, good threadblock geometries. For the best threadblock choices we observe a reduction in execution times of 23 % in Fermi, and 18 % in Kepler. It is more noticeable in Fermi due to its higher number of banks. See the complete results in [117].

5.3.3 Summary

Our results show that uBench can be used to gain insight of the performance impact of the threadblock geometry and L1 cache configuration choices for different architectures and applications. This understanding improves the ability of a programmer to develop better policies for threadblock selection, and also to apply code tuning techniques. Finally, uBench could be used as a test bed for auto-tuning techniques that automatically select the threadblock geometry.

5.4 Conclusions

This chapter discusses how configuration parameters and their relationship with the underlying GPU architecture affects GPU performance. We started with a number of hypotheses that describe the expected effects of different set of configuration choices. Later, we conducted a study to validate the main effects described. This study revealed that several combinations of kernel characteristics were not yet covered. To solve it, we designed uBench, a more exhaustive benchmark suite used to study in more depth the relationship between configuration parameters and performance. Finally, we have applied the knowledge obtained to the execution of two real-world problems to verify the proposed hypotheses.

This study has produced the knowledge needed to design the mapping and configuration policies needed to develop a version of Hitmap suitable for heterogeneous environments. The following chapter presents an experimental study on how the new Heterogeneous Hitmap fulfills the requirements imposed by our research question.

Chapter 6

Experimental Evaluation of an Heterogeneous Hitmap

After designing the solution needed to adapt Hitmap for its use on heterogeneous environments, in this chapter we show how the new Heterogeneous Hitmap performs when executing different benchmarks.

6.1 Mapping and synchronization issues

In this section we show with an example how Hitmap abstractions lead to codes which are independent of the encapsulated mapping techniques.

6.1.1 Case study

We have chosen as study-case the Cannon's algorithm for matrix multiplication (see Sect. B.1 in Appendix B). It is a task-parallel algorithm focused on reducing local memory usage for distributed systems.

Figure 6.1 shows the Cannon's matrix multiplication algorithm implemented with the Hitmap library for heterogeneous systems. We use float base elements to allow better exploitation of parallel resources of involved GPU devices. The code is the same used in previous versions of Hitmap for distributed-memory systems except lines 40–41 (that encapsulate the low-level partition for the assigned device), and lines 47 and 50, that encapsulates the coordination between the CPU and the accelerators.

Lines 3–6 declare the full domain of the three matrices with a global-view approach. Memory is not yet allocated. Line 9 builds a virtual topology enforcing a perfect square of processes, as required by the algorithm. Lines 12–14 create layout objects that distribute the matrices domains across the virtual topology. The layout plug-in modules used are different for the three matrices. Figure 6.2 shows a diagram of the resulting layouts for 4

```
void cannonsMM( int n, int m, int p ) {
    /* 1. DECLARE FULL MATRICES WITHOUT MEMORY */
    HitTile_double A, B, C;
    hit_tileDomain( &A, float, 2, n, m );
    hit_tileDomain( &B, float, 2, m, p );
    hit_tileDomain( &C, float, 2, n, p );

    /* 2. CREATE VIRTUAL TOPOLOGY */
    HitTopology topo = hit_topology( plug_topSquare );

    /* 3. COMPUTE PARTITIONS */
    HitLayout layC = hit_layout( plug_layBlocksLB, topo, C, 0 );
    HitLayout layA = hit_layoutWrap( plug_layBlocksLB, topo, A, 0 );
    HitLayout layB = hit_layoutWrap( plug_layBlocks, topo, B );

    /* 4. CREATE AND ALLOCATE TILES */
    HitTile_double tileA, tileB, tileC;
    hit_tileSelectNoBoundary( &tileA, &A, hit_layMaxShape(layA,1) );
    hit_tileSelectNoBoundary( &tileB, &B, hit_layMaxShape(layB,0) );
    hit_tileSelect( &tileC, &C, hit_layShape(layC) );
    hit_tileAlloc( &tileA ); hit_tileAlloc( &tileB ); hit_tileAlloc( &tileC );

    /* 5. INITIALIZE MATRICES */
    hit_tileFileRead( &tileA, "matrixA.dat" );
    hit_tileFileRead( &tileB, "matrixB.dat" );
    float aux=0; hit_tileFill( &tileC, &aux );

    /* 6. INITIAL ALIGNMENT PHASE */
    HitComm commRow = hit_comShiftDim( layA, 1, -hit_layRank(layA,0), &tileA );
    HitComm commCol = hit_comShiftDim( layB, 0, -hit_layRank(layB,1), &tileB );
    hit_comDo( commRow ); hit_comDo( commCol );
    hit_comFree( commRow ); hit_comFree( commCol );

    /* 7. REUSABLE COMM PATTERN */
    HitPattern shift = hit_pattern( HIT_PAT_UNORDERED );
    hit_patternAdd( &shift, hit_comShiftDim( layA, 1, 1, &tileA ) );
    hit_patternAdd( &shift, hit_comShiftDim( layB, 0, 1, &tileB ) );

    /* 8. COMPUTE DEVICE PARTITION USING ACCESS PATTERN INFO */
    HitPartition parts = hit_partition( plug_partBlocks, hit_layShape(layC),
                        2, hit_shape( 2, ALL, THIS ), hit_shape( 2, THIS, ALL ) );

    /* 9. DO COMPUTATION */
    int loopIndex;
    int loopLimit = max( hit_layNumActives(layA,0), hit_layNumActives(layB,1) );
    for (loopIndex = 0; loopIndex < loopLimit-1; loopIndex++) {
        hit_kernelLaunch( mmult, parts, 3, IN, tileA, IN, tileB, INOUT, tileC );
        hit_patternDo( shift );
    }
    hit_kernelLaunch( mmult, parts, 3, IN, tileA, IN, tileB, INOUT, tileC );

    /* 11. WRITE RESULT */
    hit_tileFileWrite( &tileC, "matrixC.dat" );

    /* 12. FREE RESOURCES */
    hit_partitionFree( parts );
    hit_layFree( layA ); hit_layFree( layB ); hit_layFree( layC );
    hit_patternFree( &shift );
    hit_topFree( topo );
}
```

Figure 6.1: Heterogeneous Hitmap implementation of Cannon's matrix multiplication.

processes. Matrix B uses a classical block data partition, with evenly sized parts. Matrices C and A use a load-balancing technique inserted in the library as a plug-in. The rows dimension is split unevenly according to a *Balance factor*, decided in terms of the relative computing power of the device types as recorded in the low-level topology description, a value experimentally determined previously. In lines 17–21 each logical process creates and allocates the local part of the matrices.

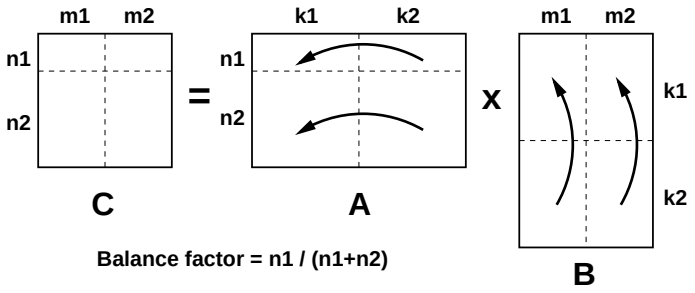


Figure 6.2: Load balancing layout scheme in the Cannon's matrix multiplication example for 4 processes.

Thanks to the *maxShape* padding function, n and m do not need to be exact multiples of the number of processes in a given axis. Lines 24–26 read in parallel the tiles of the input matrices. The C matrix is initialized with zeros.

Lines 29–32 perform the initial relocating stage prescribed by the Cannon's algorithm, shifting A and B tiles. Lines 35–37 build the shifting communication pattern that will be used between the computation stages. The layout objects and the tiles provide all the information needed to internally find neighbors and build MPI derived data types to optimize the communications. Thus, communications are adapted to the partition transparently. For this example we choose synchronous communication to avoid the need of double buffers, exploiting our full system memory to do larger computations.

Lines 40–41 generate a partition object tailored to the device assigned to the logical process. Line 41 is a shape expression that represents the global memory access pattern; indicating, in relative coordinates, which elements are accessed by a thread. Lines 44–50 implement the main loop of the algorithm. The computation stage of the last iteration has been unrolled to avoid the last unneeded communication stage. The computation is launched by the *hit_kernelLaunch* primitive, independently of the actual device. The shifting communication pattern is activated by the *hit_patterDo* primitive. Line 53 writes the output matrix tiles to a file in parallel. Lines 56–59 free all the Hitmap resources before finishing.

6.1.2 Experimental work

We have designed experimental work to show that: (1) Our new abstractions do not impose a significant overhead on the computation; and (2) this framework allows to easily exploit different devices to obtain performance benefits.

In order to show the efficiency of the Hitmap codes, we have manually developed and optimized reference codes for matrix multiplication: (a) A direct MPI implementation of the Cannon's algorithm (see Appendix B); and (b) a direct CUDA implementation that may also split and multiply the matrices block by block if they are too big to fit into the GPU device memory.

For our experiments we have used two different platforms. The first one is a Beowulf cluster with up to 18 dual-core PC computers. The second one is an Intel(R) Core(TM) i7 CPU 960, 3.20GHz with active hyper-threading. This system has two GPUs: a GeForce 8500 GT, and a GerForce 9600 GT, both managed by the CUDA driver included in the 4.0 toolkit. From now on, we identify the different available devices in this machine with the following letters: (A) GeForce 9600 GT; (B) GeForce 8500 GT; (C) Cores of the CPU.

We use three square matrices of 2 048, 8 192, and 12 288 rows/columns. The first size is small enough to allocate the three matrices in any of the devices of both systems. The second size cannot be fully allocated on the second GPU (device B) of the mixed CPU-GPU machine. The last size cannot be allocated in any of the GPUs. We also test that our automatic padding mechanisms do not impose a significant performance effect on the results using modified sizes (e.g. 2 039 or 2 057 rows/columns).

Figures 6.3 and 6.4 present execution times obtained in different scenarios. Note that all y-axis are in logarithmic scale. The experiments in the Beowulf cluster show that, even for the small matrix size, Hitmap implementations have the same scalability and overall performance than the manually optimized MPI code. A minimal Hitmap performance overhead is observed in all our experimental work.

In the heterogeneous machine the best performance results are obtained for a small number of processes. Remind that Cannon's algorithm forces more synchronization stages when the number of processes increase. Thus, for Cannon's algorithm, more MPI processes lead to bigger communication overhead, while reducing the computation load of each task. In this machine, our experiments show the best results for four MPI processes. We show results for the following scenarios.

- *Reference code* (manually developed):

    1. (A) CUDA code executing the whole computation with only one kernel launch in device A, the fastest GPU;
    2. (A4) For matrices that do not fit in the GPU device memory. The reference code parts the matrices in four even parts and executes the computation in

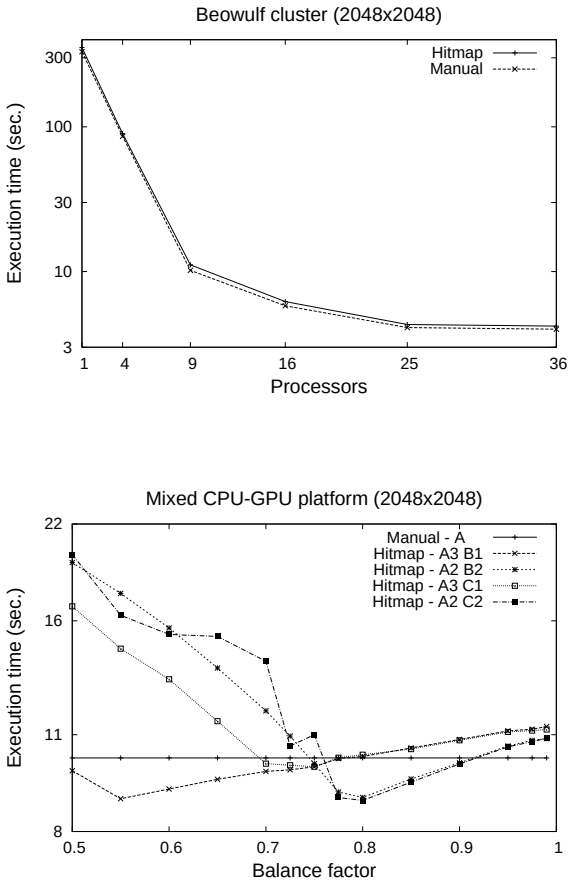


Figure 6.3: Hitmap abstraction results (1st part).

several kernel launches.

- *Hitmap code:*

  Changing the topology module we can easily experiment with different assignments of devices to logical processes.

  1. (A3-B1) Mixed GPUs: 3 processes mapped to device A, and 1 process to device B;
  2. (A2-B2) Mixed GPUs: 2 processes mapped to device A, and 2 process to device B;
  3. (A3-C1) GPU and core: 3 processes mapped to device A, and 1 process to one CPU-core;
  4. (A2-C2) GPU and cores: 2 processes mapped to device A, and 2 more processes, each one mapped to a different CPU-core.

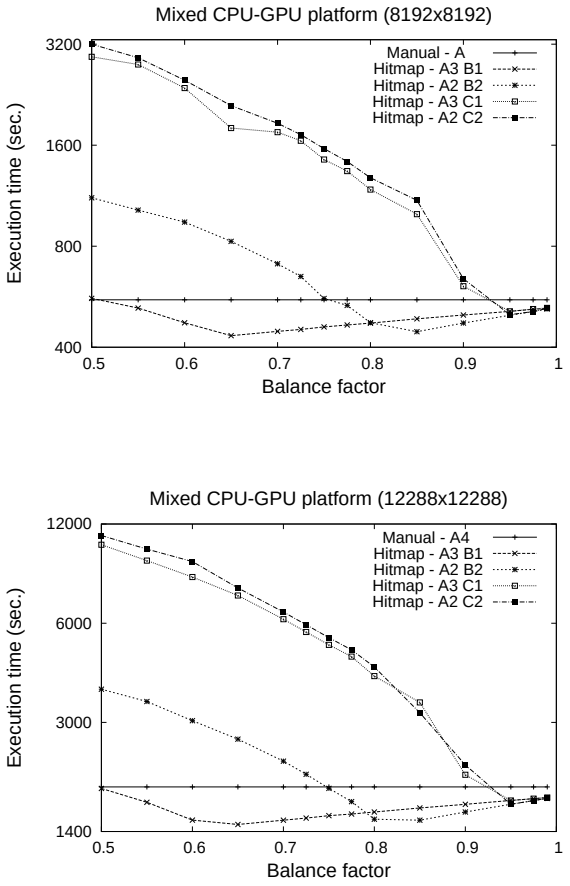


Figure 6.4: Hitmap abstraction results (2nd part).

For all the experiments with GPUs and Hitmap we have manipulated the partition plug-in to experiment with different load-balance factors, between 0.5 and 0.975.

Let us consider the execution time of the reference CUDA code (A and A4). The results show that it was always possible to improve these performance results with the Hitmap code, exploiting heterogeneity with more than one device. The results for the small matrix size are more unstable, and affected by the kernel initialization times, including the communication between CPU and GPU. However, as the matrix size increases, the results are more stable, and show exactly the same trends. We obtain performance improvements of up to 10 % for the small matrices, and a consistent best improvement of 20.5 % for medium and big input data sizes. Traces of the executions show that the MPI communication times are always less than 10 % of the total execution time for the small matrix size, and their impact quickly decreases as the data input size grows.

On the left part of the plots (load-balance factor of approximately 0.5), the load is evenly distributed, not taking into account the different computing powers of the devices. The critical path is dominated by the slower devices. As the load-balance factor grows, the

balance is improved proportionally, reducing the total execution time. After the optimum balance point is found, an increase of the factor leads to too few computation on the slower devices. Thus, the critical path is dominated by the fastest device, proportionally reducing performance again.

An important question is: Is it possible to predict the best load-balance factor for a given set of devices? Profiling tests with simple benchmarks show that the relative computing power between devices A and B is approximately $r = 3.826$; and between device A and a core (device C) it is $r = 14.153$. In order to assign to each device a computation proportional to its relative computing power, the load-balance factor may be calculated as $LB = r/(r+1)$ for the A2 scenarios, and $LB = (r-1)/(r+1)$ for the A3 scenarios.

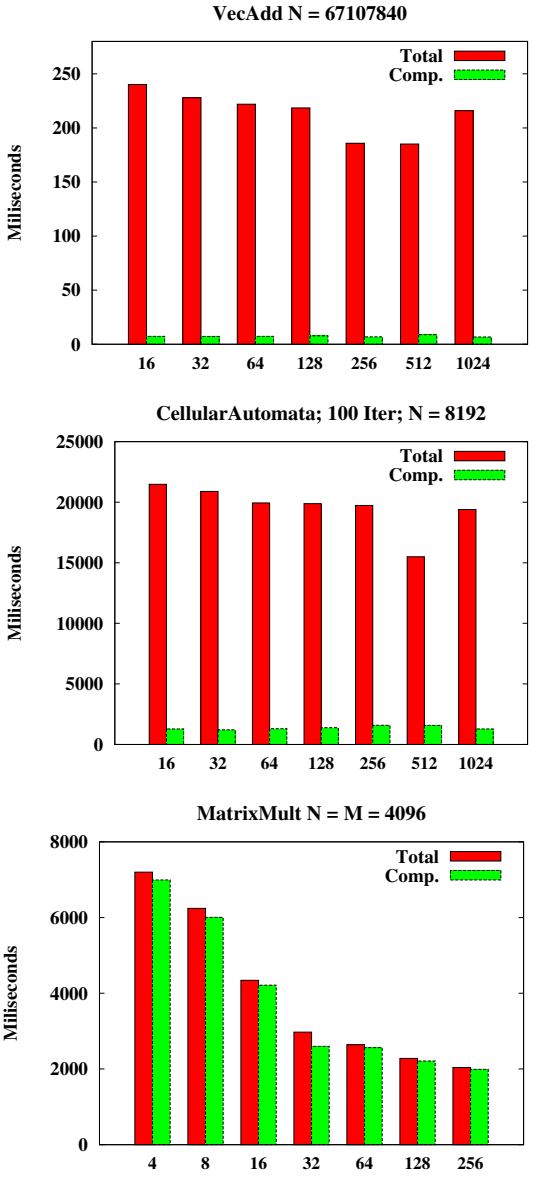
The experimental results show that, for big enough matrices, this estimation is always too conservative with respect to the value that leads to the best performance: 10 % less than the best performance in both A3 scenarios, 2 % and 6 % on A2 scenarios. A more sophisticated model, taking into account the synchronization stages, is needed to automatically predict the best factor in the Layout plug-ins.

6.1.3 Synchronization issues: Conclusions

The framework presented in Sect. 4 encapsulates the mapping techniques into plug-ins at two different layers of abstraction: One related to logical processes coordination, and another related to adapting the computations to the inherent parallelism and architecture details of the actual device associated to each logical process. Thereby, the proposed high-level API transparently deals with all the details of communication and synchronization between logical processes and accelerator devices, such as GPUs. Finally, this framework allows to generate codes which are transparently adapted to heterogeneous systems with mixed types of accelerator devices, taking into account different computational powers, and with a minimal performance degradation due to abstractions or synchronization overheads.

6.2 Memory size restrictions

Our prototype layer implements the technique described in Sect. 4.2 for automatic mapping of computations to accelerators with memory size restrictions. It computes the best partition, issues the transparent movement of the required portions of the data structures to/from the target device memory, and forces the sequential execution of each part as a different kernel. This hidden layer is integrated in a new kernel launching function, that receives one access pattern specification along with each tile parameter.



Vector Addition

Memory limit MBs	1	2	4	8	16	32	64	128	256	512	1024
#Kernels	.	.	.	.	49	25	13	7	3	2	1
Kernel size	.	.	.	.	16	32	64	128	256	512	767

Cellular Automata

Memory limit MBs	1	2	4	8	16	32	64	128	256	512	1024
#Kernels	.	.	.	.	48	24	13	7	4	2	1
Kernel size	.	.	.	.	11	21	43	85	170	241	512

MM Multiplication

Memory limit MBs	1	2	4	8	16	32	64	128	256	512	1024
#Kernels	.	.	1103	433	128	43	19	9	1	.	.
Kernel size	.	.	0.4	1.2	4	12	28	60	192	.	.

Figure 6.5: Execution times for: (a) Vector addition; (c) Stencil computation; (d) Matrix-matrix multiplication. The tics in the x-axis indicate the value of the memory-size-limit parameter. The table shows for each program and each memory-size-limit value, the number of sub-kernels generated by our system for this case, and the memory size actually used.

We have implemented the three study cases presented in Sect. 4.2.2 using the new tools. The codes are similar to the original ones, with expressions of the access patterns for each data structure involved in the computation. We have tested this prototype implementation with a GPU target device, manually changing the memory-size-limit parameter to simulate different scenarios.

Our experimental platform is a GForce GTX 680 (Kepler, 2048MB GDDR5) NVIDIA GPU device. The host machine is a 64-bits Intel(R) Core(TM) i7 CPU 960 3.20GHz, with a global memory of 6 GB DDR3. It runs an UBUNTU desktop 10.10 (64 bits) operating system. The applications have been developed using CUDA 4.2 toolkit and the 295.41 64-bit driver.

The use of integer or float data element lead to practically the same execution time in CUDA. Thus, we select integer as data type. The data structures size chosen for each benchmark are different, in order to obtain stable execution time values. The number of items are the following: (1) Vector addition, $n = 67\,107\,840$; (2) cellular automata, $n = m =$8 192, and (3) matrix multiplication, $n = m = 4\,096$. These sizes are multiple of the selected threadblock size to avoid any padding operation.

To simulate results for different kinds of devices, we decided to manually change the memory-size-limit parameter. We have selected values that are powers of two in the range of 1 to 1024 Mbytes. For each kernel there is a different range of this parameter that leads to a feasible number of sub-kernels with a reasonable kernel size. Figure 6.5 shows the execution times (in milliseconds) obtained for some memory-size-limit parameter values. The first bar indicates the total execution time, while the second bar indicates the time devoted to real computation. The rest of the time is spent in host-device communications.

The results show that, as expected, for the kernels with low computational load per thread (vector addition and cellular automata), the ratio of communication vs. computation is very high, being very small in the remaining cases. When communication times dominate the total execution time, we observe a trend to reduce the communication times for particular memory restrictions. This effect can be explained by the fact that the PCI Express bus works faster for memory transactions of particular sizes. Thus, when the sub-kernels generated require memory sizes that fit well in the PCI bus, the communication times are reduced. This information can be exploited by a library to split the communication in proper block sizes [106].

For the unidimensional example, vector addition, we can see that the algorithm generates kernels that fit the memory limit almost perfectly. However, this is not the case for 2-dimensional problems. In the current implementation, the stage 5 of the 2-dimensional partition algorithm shown in Sect. 4.2.2 has not been yet implemented, leading to sub-optimal partition results. However, the performance results show the same trends when manually selecting the best candidate.

The intensive reutilization of caches by the concurrent dot products in matrix multi-

plication application, leads to reduced total execution times when the kernels have bigger sizes.

The results show that the hidden layer does not impose a substantial overhead on the execution of the whole computation, and it can take away the burden of considering memory-size restrictions at upper mapping layers. Moreover, a deeper research on the information provided by the access patterns may also lead to detect situations where the system can take profit of the artificial automatic partition of the kernels to improve performance results.

6.2.1 Memory size-restrictions: Conclusions

The proposed layer requires the programmer to specify the access patterns of the computation threads in a simple abstract form. This information is used at run-time to compute the pieces of data-structures required by a generic partition to determine the best partition that ensures that each subpart fits in the device memory.

We have discussed an implementation of this concept into an automatic mapping tool that allows to apply high-level distributions in heterogeneous devices without the need to take into account the memory limitations of the target devices. Our experimental results show the feasibility of the solution proposed.

6.3 A real-world benchmark: The SSSP problem

We have selected the Dijkstra's algorithm (see Sect. B.2 of Appendix B) to solve the SSSP problem. In this section, we will adapt this algorithm to be exploited on GPU devices efficiently. In the following section 6.4, the implemented code will be used to test the hypotheses enumerated in Sect. 5.1 on applications with similar characteristics. This problem meets the specifications listed in Sect. 2.3.2, since it is a benchmark with data reutilization, and a different memory access pattern for each input data structure. It is an embarrassingly-parallel application where the computation shares data in a common sparse data structure.

6.3.1 Parallel Dijkstra for GPUs

This section describes how our implementation parallelizes the Dijkstra algorithm focusing on the *outer loop*, and following the ideas of Crauser *et al.* [28]. As we have explained in Appendix B, the main problem of this kind of parallelization is to identify as many nodes as possible that can be inserted in the following frontier set.

Defining the frontier set

Dijkstra's algorithm calculates in each iteration i the minimum tentative distance of the nodes belonging to the unsettled set, U_i. The node whose tentative distance is equal to this minimum value can be settled and becomes the frontier node. Its outgoing edges are traversed to relax the distances of the adjacent nodes.

In order to parallelize the Dijkstra algorithm, it is needed to identify which nodes can be settled and used as frontier nodes at the same time. Martín *et al.* [72] inserts into the frontier set, F_{i+1}, all nodes with this minimum tentative distance with the aim to process them simultaneously. Crauser *et al.* [28] introduces a more aggressive enhancement, augmenting the frontier set with nodes with bigger tentative distance. The algorithm computes in each iteration i, for each node of the unsettled set, $u \in U_i$, the sum of: (1) Its tentative distance, and (2) the cost of its outgoing edges. Afterwards, it calculates the minimum of these computed values. Finally, those nodes whose tentative distance are lower or equal than this minimum value can be settled becoming the frontier set.

We define the concept of Δ_i as the limit value computed in each iteration i that holds that any unsettled node u with $\delta(u) \leq \Delta_i$ can be safely settled. The bigger the Δ_i value, the more parallelism is exploited. However, depending on the particular graph being processed, the use of a very ambitious Δ_i may induce overheads that destroys any performance gain with respect to sequential execution.

Our implementation of Dijkstra's algorithm follows the idea proposed by Crauser *et al.* [28] for incrementing each Δ_i. For every node $v \in V$, the minimum weight of its outgoing edges, that is, $\Delta_{\text{node } v} = \min\{w(v,z) : (v,z) \in E\}$, is calculated in a precomputation phase. For each iteration i of the external loop, having all tentative distances of the nodes in the unsettled set, we define

$$\Delta_i = \min\{(\delta(u) + \Delta_{\text{node } u}) : u \in U_i\} \tag{6.1}$$

Thus, it is possible to put into the frontier set F_{i+1} every node v whose $\delta(v) \leq \Delta_i$.

Our GPU implementation: The general variant

The four Dijkstra's algorithm steps described in Apendix B can be easily transformed into a GPU general algorithm (see Alg. 1). It is composed of three kernels that executes the internal operations of the Dijkstra vertex outer loop.

The *relax kernel* (Alg. 2, invoked in line 3 of Alg. 1) decreases the tentative distances for the remaining unsettled nodes of the current iteration i through the outgoing edges of the frontier nodes $f \in F_i$. A GPU thread is associated to each node in the graph. Those threads assigned to frontier nodes, $f \in F_i$, traverse their outgoing edges, relaxing the distances of their unsettled adjacent nodes.

Algorithm 1 GPU implementation of Dijkstra's algorithm. CUDA kernels are delimited by <<< ... >>> ..

```
1: <<<initialize>>> (U, F, δ);          //Initialization
2: while (Δ ≠ ∞) do
3:    <<<relax>>> (U, F, δ);            //Edge relaxation
4:    Δ =<<<minimum>>> (U, δ);          //Settlement step_1
5:    <<<update>>> (U, F, δ, Δ);        //Settlement step_2
6: end while
```

Algorithm 2 Pseudo-code of a CUDA thread in *relax kernel*.

```
1: tid = thread.Id;
2: if (F[tid] == TRUE) then
3:    for all j successor of tid do
4:       if (U[j] == TRUE) then
5:          BEGIN ATOMIC REGION
6:             δ[j] = min{δ[j], δ[tid] + w(tid, j)};
7:          END ATOMIC REGION
8:       end if
9:    end for
10: end if
```

The *minimum kernel* (invoked in line 4 of Alg. 1) computes the minimum tentative distance of the nodes that belongs to the U_i set. To do so, the advanced *reduce3* method of the CUDA SDK [52] has been modified to accomplish this task. Our *minimum kernel* is adapted in order to: (1) Add the corresponding $\Delta_{\text{node } v}$ value to $\delta(v)$, and (2) compare its new assigned values to obtain the minimum one. The resulting value of the *minimum kernel* is the Δ_i.

The *update kernel* (Alg. 3, invoked in line 5 of Alg. 1) settles those nodes from U_i whose tentative distances are lower or equal to Δ_i. This task is carried out extracting them from the following-iteration unsettled set, U_{i+1}, and putting them to the following-iteration frontier set F_{i+1}. Each single GPU thread checks, for its corresponding node v, whether $U(v) \wedge \delta(v) \leq \Delta_i$. If so, the thread assigns v to F_{i+1} and deletes v from U_{i+1}.

Our implementation supports two types of graph representations, both adjacency lists and matrices. The nodes are numbered from $0 \ldots n-1$. Besides the basic structures to hold nodes, vertices, and their weights, three vectors are defined:

- Vector U, that stores in $U[v]$ whether node v is an unsettled node.
- Vector F, that stores in $F[v]$ whether node v is a frontier node.
- Vector δ, that stores in $\delta[v]$ the tentative distance from source to node v.

Algorithm 3 Pseudo-code of a CUDA thread in *update kernel*.

```
1: tid = thread.Id;
2: F[tid]= FALSE;
3: if (U[tid]==TRUE and δ[tid] <= Δ) then
4:     U[tid]= FALSE;
5:     F[tid]= TRUE;
6: end if
```

An economic variant

We have developed an additional version (we named *economic variant*) that needs less memory space at the cost of being less powerful than the previous variant already described.

Our *general variant* uses a vector of size n to store the $\Delta_{\text{node } v}$ value of each node v. Instead, the *economic variant* uses a single value, Δ_{base}, that is a lower bound for every $\Delta_{\text{node } v}$. This value is the minimum weight associated to any edge e of the graph, $\Delta_{\text{base}} = \min\{w(e) : e \in E\}$. Then, for each iteration i of the external loop having the tentative distance of the following node to be settled, we define

$$\Delta_i = \min\{\delta(u) : u \in U_i\} + \Delta_{\text{base}} \tag{6.2}$$

For the development of this approach we need to calculate Δ_{base} in a precomputation phase. Note that the computation of the minimum value in the *minimum kernel* is simplified because every thread does not need to add $\Delta_{\text{node } v}$ to the tentative distance. Now, the Δ_{base} is added to the value returned by the *minimum kernel*.

Regarding the exploited parallelism degree, this *economic variant* cannot include as many nodes into frontier set as the *general variant*, leading to more iterations of the external loop. Thus, in spite of consuming less space, the exploited parallelism degree is lower than for the *general variant*.

Martín et al. successor variant

In this subsection we will describe the first GPU approach of Dijkstra's algorithm developed by Martín *et al.* [72]. They have presented some parallel implementations of Dijkstra's algorithm executed on the first CUDA architecture (now called pre-Fermi).

In order to parallelize the Dijkstra algorithm, they have introduced a conservative enhancement to increase the frontier set, inserting all nodes with the same minimum tentative distance. According to our notation presented above, their frontier set at any iteration i, F_{i+1}, is composed by every node $x \in U_i$ with equal tentative distance $\delta(x)$ than Δ_i

being

$$\Delta_i = \min\{\delta(u) : u \in U_i\} \tag{6.3}$$

Their *update kernel* also differs from ours in the frontier-set check condition, $U(v) \wedge \delta(v) = \Delta_i$.

Martín et al. predecessor variant

The same authors have presented a different variant of Dijkstra's algorithm, called the *predecessor variant*. They have implemented both a sequential version for CPU, and a parallel one for GPU devices.

This variant differs from the previous one, in the way that it relaxes the tentative distances of the unsettled nodes. That is, for every unsettled node, the algorithm checks if any of its predecessor nodes belongs to the current frontier set. In that case, the tentative distance is relaxed if the new distance through this frontier node is lower than the previous one.

The GPU predecessor implementation assigns a single thread for each node in the graph. The *relax kernel* only computes those threads assigned to unsettled nodes $u \in U_i$. Every thread traverses back the incoming edges of its associated node looking for frontier nodes.

6.3.2 Experimental setup

We will first describe the methodology used for our experiments, in order to test the Dijkstra's algorithm implementation for GPU devices and check the hypotheses of GPU configuration parameters.

Methodology

We have compared our GPU algorithm implementation, the *general approach*, with: (a) The *CPU successor variant*, and (b) the *GPU successor variant* of Martín *et al.*. We have adapted our algorithm to also support the *predecessor variant* in order to compare with (c) the *CPU predecessor variant*, and (d) the *GPU predecessor variant* of Martín *et al.*. To fairly compare the performance improvement of our algorithms, we also use the same run-time configuration of the grid (threadblock size and geometry) as Martín *et al.*.

Moreover, in order to check the performance degradation due to reduction the memory space, both *general* and *economic variant* are also compared. This experiment is also carried out with the same kernel configuration described before.

After checking that results were consistent, we repeat the experiments using a GeForce GTX 680 (Kepler) NVIDIA GPU device. The following discuss compares results obtained with Fermi platform.

In order to compare the results of our approach with the implementation of Martín *et al.*, we have replicated the experimental examples using their graph creation tools.

Divergent branch and dummy computation

The threads of the *relax kernel* for both, *predecessor* and *successor* variants, have a divergent branch. Two different kinds of threads are identified due to this divergent branch: (1) *dummy threads*, that do not make any computation for its assigned node, and (2) *working threads*, that carry out the relax operation from the assigned node.

Usually, the effect of the divergent branch is negative for the performance application due to the serialization of the work-flow. Thus, in order to discuss if its presence causes a significant performance degradation, we have carried out an experiment to measure the efficiency ratio of divergent branch. (This efficiency tries to show how many branches diverged. 100 % means no branches diverged. When a branch diverges the warp thread active mask is reduce to be less than 32 so the execution is not as efficient. In addition the branch may have to be executed multiple times based upon the number of ways the branch diverged). See cite [89]). With the aim of knowing if it is possible to compute this kernel more efficiently, we have measured both the total number of executed threads and the number of *working threads*.

Target architecture

The performance results described by Martín *et al.* were obtained using a pre-Fermi architecture release. We started by replicating these results, in our case using a GeForce GT 9600 with a Fermi.

Regarding the host machine, we used an Intel(R) Core(TM) i7 CPU 960 3.20GHz, 64-bits compatible, with a global memory of 6 GB DDR3. It runs an UBUNTU Desktop 10.10 (64 bits) operative system. The experiments have been launched using CUDA 4.2 toolkit and the 295.41 64-bit driver.

Input set characteristics

Due to the different efficiency of the two storage methods (adjacency matrices and adjacency lists), the input set for each implementation should be different. Adjacency list allows to use much bigger graphs for the same memory footprint. For both storage methods we have generated 25 different graphs instances for each size with the same random graphs generator used by Martín *et al.*. We have maintained seven as graph degree.

- Adjacency Matrices: The example graphs we have chosen to be stored as adjacency matrices have sizes that range from $1 \cdot 2^{10}$ to $15 \cdot 2^{10}$ vertices. The edge weights are integers that randomly range from 1 to 10.

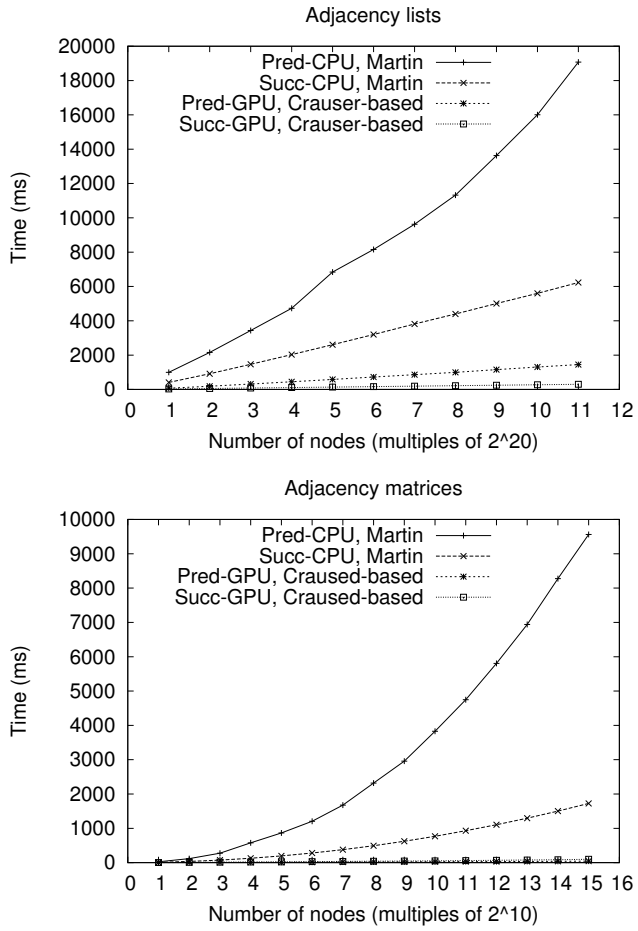


Figure 6.6: CPU-Martín vs. our Crauser-based GPU implementation execution times for both input sets considered (1st part).

- Adjacency Lists: The example graphs we have chosen to be stored in adjacency list have sizes that range from $1 \cdot 2^{20}$ to $11 \cdot 2^{20}$ vertices. Martín *et al.* had also inverted the generated graphs in order to study approaches based on the successor version. Note that the degree seven cannot be kept for these inverted graphs. The edge weights are integers that randomly range from 1 to 10.

Code Versions Evaluated

From the suite of different implementations described in [72], we have taken the fastest ones that use a CPU and a GPU computation. That means we have left out the hybrid approaches that mix the execution of some phases in the CPU and others in the GPU. We denominated implementations as: (1) "Pred-CPU and Pred-GPU" for the *predecessor variants* for CPU and GPU, and (2) "Succ-CPU and Succ-GPU" for the *successor variants* for CPU and GPU.

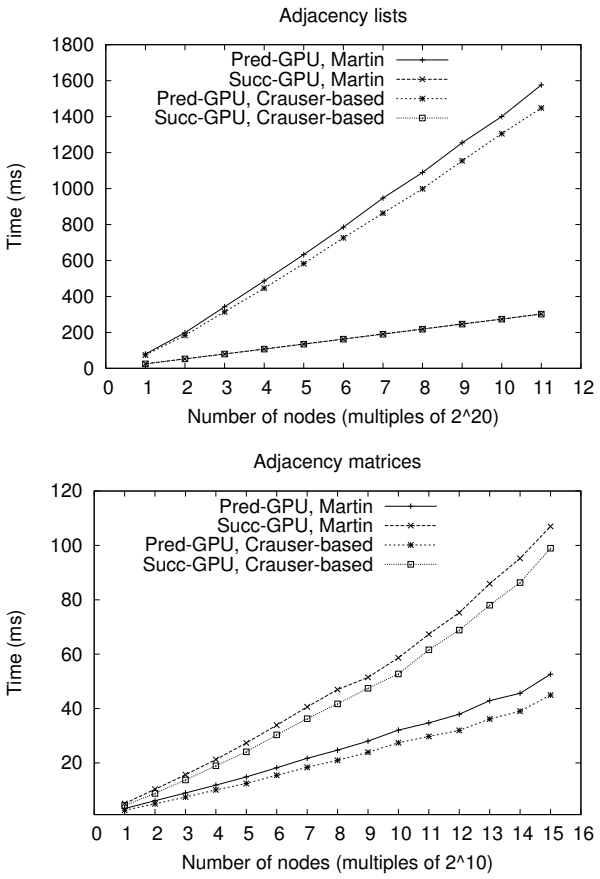


Figure 6.7: GPU-Martín vs. our Crauser-based GPU implementation execution times for both input sets considered (2st part).

6.3.3 Experimental results

This section discusses the results of the experimental work.

Data structure	successors	predecessors
Adjacency Lists CPU vs. GPU	20.65×	13.17×
Adjacency Matrices CPU vs. GPU	17.43×	219.79×

Table 6.1: Speedup obtained by our GPU implementation considering Martín *et al.* CPU version as reference.

Performance improvement

Figure 6.6 shows the execution time of *predecessor* and *successor* variants for CPU Martín *et al.* and our GPU implementation in Kepler's architecture for adjacency lists (top) and for adjacency matrices (bottom). We can observe in Table 6.1 a performance speed-up from 13× to 220× with respect to the CPU times (in the predecessor variant for

Data structure	successors	predecessors
Adjacency Lists GPU vs. GPU	1.07%	8.14%
Adjacency Matrices GPU vs. GPU	7.51%	16.97%

Table 6.2: Performance improvement of our GPU Implementations vs. Martín *et al.* GPU versions.

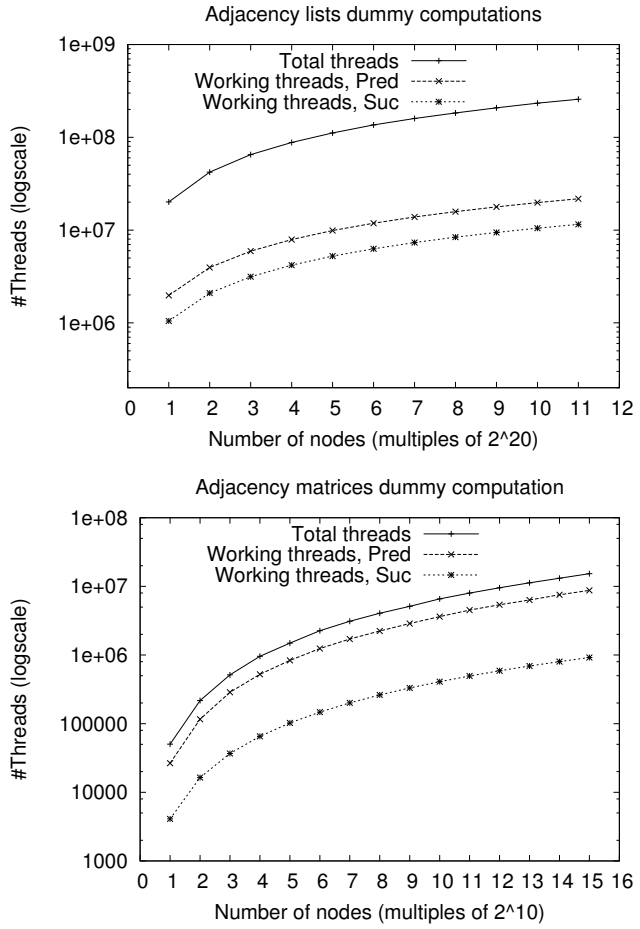


Figure 6.8: Total vs. Number of Working Threads in the *relax kernel* for Adjacency List (top) and Matrices (down).

1×2^{10} and 15×2^{10} graph sizes respectively). We have always better performance than the obtained by Martín *et al.*

Figure 6.7 shows the execution time of *predecessor* and *successor* variants in Kepler's architecture for both GPU implementations, Martín *et al.* and ours, for adjacency lists (top) and for adjacency matrices (bottom). The Table 6.2 shows a performance gain up to 17 % with respect to the GPU-Martín algorithm.

Economic vs. general

For the input sets used in the experiments, the *economic variant* has a similar performance compared with the *general variant*. The used graphs for this experimental work have seven outgoing adjacent nodes on average, with integer weights that range from $1..10$. Therefore, there is a high probability that for every node v the values $\Delta_{\text{node } v} = \Delta_{\text{base}}$, leading to an analogous behavior. With graphs with less outgoing adjacent nodes and a bigger range of weights, this probability is decreased. In such cases, the *general variant* will take more advantage of its precomputed data with respect to the *economic variant*.

Divergent branch and dummy Computations

Figure 6.8 shows the total number of executed threads and the number of *working threads* in *relax kernel*. In this case, the number of *working threads* is significantly lower than the total number of launched threads. The percentage of *dummy threads* vs. total threads goes from 42 %, for the *predecessor variant* with adjacency matrices, to 96 %, for the *successor variant* with adjacency lists.

The results of the CUDA VisualProfiler have shown that the efficiency ratio of the divergent branch in the *relax kernel* is good, from 94.3 % to 99.5 %. Thus, the serialized work-flows, due to the divergent branch, hardly have affected the performance of this kernel.

The execution of *dummy warps*, that are warps of 32 *dummy threads*, do not lead to serialize different work-flows, because all threads of these warps processes the same dummy instruction. Therefore, the fact of having much more *dummy warps* than *mixed warps*, warps filled with both *dummy* and *working threads*, is the reason because the performance is hardly affected by the divergent branch.

6.3.4 The SSSP problem: Conclusions

Our implementations obtain up to 220× speed-up with respect to the CPU version and a performance improvement up to 17 % with respect to other GPU versions, as Martín *et al.* selecting a good threadblock geometry.

We have also shown that, although the *relax kernel* has divergent branches, they do not affect significantly the performance. In spite of the good performance improvements obtained, we have detected that there is a high amount of dummy instructions executed in *relax kernel*, up to 96 %. Thus, we expect that modifying the kernel to reduce the the dummy computational load for the same inputs could lead to better performance times.

6.4 APSP problem

In this section we face the APSP problem (see Sect. B.2.5 in Appendix B) for sparse graphs using the parallelized version of Dijkstra's algorithm for GPUs discussed in Sect. 6.3 combined with optimization methods. These methods include the selection of a good criteria for the threadblock size (as we introduced in sections 5.1 and 5.3), and the deployment of concurrent kernels in the same application context on a single GPU device to better exploit the underlying hardware resources (see Appendix A).

The experimental results obtained aim to verify, through another real-world benchmark, the conclusions presented in Sect. 5.1 and 5.3 regarding the best values of CUDA configuration parameters.

6.4.1 Experimental setup

Methodology

We have compared the performance of the GPU solution proposed in Sect. 6.3 with different configurations of the threadblock size and the number of concurrent kernels for the *relax kernel*. Both techniques are closely related to the use of the hardware resources, and therefore, we have considered to test them together to efficiently squeeze the GPU capabilities.

Additionally, for our experiments we have used sparse graphs whose number of nodes is 1 049 088. This size has been chosen because it is a multiple of all recommended values for the threadblock size. With the aim to reduce the total experimental workload, we have randomly execute reduced set of tasks for each graph, with each task starting at a different source node, randomly chosen following an uniform distribution.

Target architectures

The GPU device used for our experiments is the GeForce GTX 480 that has a FermiCUDA architecture (GF 110 series). The experiments have been launched using CUDA 4.2 toolkit and the 195.36 64-bit driver. Regarding the host machine, we used an Intel(R) Core(TM) i7 CPU 960 3.20GHz, 64-bits compatible, with a global memory of 6 GB DDR3. It runs an UBUNTU Desktop 10.10 (64 bits) operating system.

Input set characteristics

The input set is composed by a collection of graphs randomly generated by a graph-creation tool used the experiments presented in [72]. The graphs have been created adding seven adjacent predecessors to each node of the graph. These graphs are represented

through adjacency lists, with the nodes numbered from $0 \ldots |V| - 1$, and integer weigh for the edges randomly chosen with a uniform distribution between 1 an 10.

Tuning the threadblock and concurrent kernels

We have chosen the recommended threadblocks sizes that maximize the SM occupancy in the Fermiarchitecture (192, 256, 384, 512 and 768), that are explained in [61]. Moreover, following the recommendations described in Sect. 5.1 and 5.3 of this Thesis, we have tested lower values for this threadblock size, that are 64, 96 and 128 threads per block, with a medium ratio of SM occupancy (for no full-coalesced/scatter access patterns). Regarding the grid and the block shape, both have a single horizontal dimension to create a direct relation between the thread indexes and the array storage.

We also wanted to check if the deployment of concurrent kernels will lead to better performance times due to a good resource exploitation. In order to test this behavior, we have evaluated the GPU solution for the APSP with the following number of concurrent kernels: 1, 2, 4, 8, 16, 32 and 64. As long as Fermi architecture cannot support real extension of more than 16 concurrent kernels, we suppose that the performance gain from this point will not be significantly improved. The kernels launched after this limit are stored in a queue to later be dispatched.

Since the threadblock size and the number of concurrent kernels are closely related to the use of hardware resources, they should be evaluated testing all combinations of these parameters. Due to the large amount of computational time involved to solve the APSP problem we have only launched this first experiment by computing only the distance from 1 024, 2 048 and 8 192 source nodes to all nodes. These values are selected because they are multiples of the chosen numbers for the concurrent kernels launched.

Second, we want to show the performance difference between the worst and the best recommended configurations and to evaluate its scalability with bigger input sets. So, we have also evaluated the performance for the 16 384 and 32 768 source-node to all.

6.4.2 Experimental results

In this section we present the experimental results obtained for the different parameter configurations evaluated and the behavior of the worst and the best configurations for the *relax kernel*.

Tuning the threadblock and concurrent kernels

Figure 6.9 shows the execution times for the different configurations for the 1 024-source-node to all (left) and 8 192-source-node to all (right). The 4 096-source-node to all plot is not shown because it presents a similar performance. In all cases, the best configuration

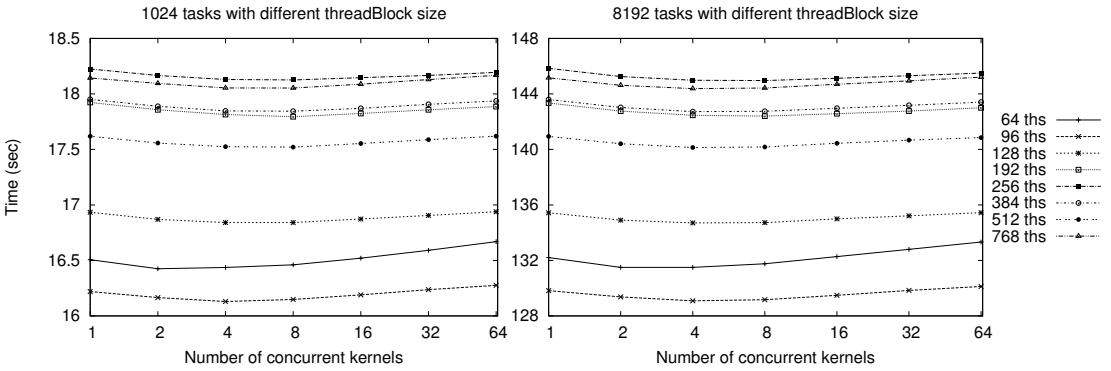


Figure 6.9: Relax-kernel execution times for different input sets with different configurations between the number of threads per block (ths) and number of multi-kernels.

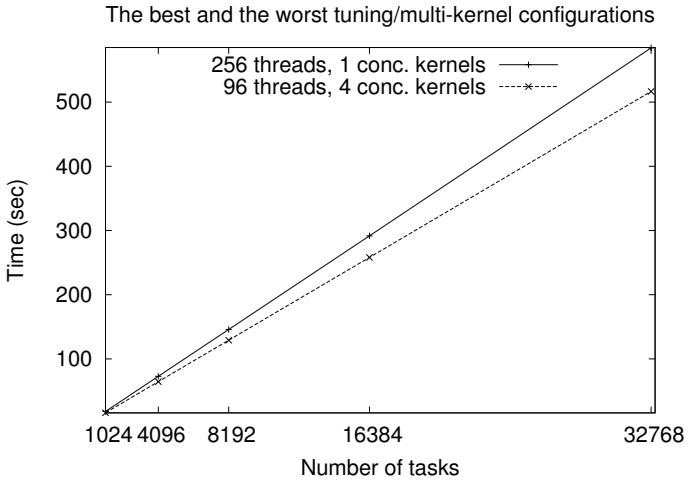


Figure 6.10: Execution times of the *relax kernel* for the best and the worst tuning/multi-kernel configurations.

from the recommended values is reached with a block size of 96 threads and the execution of 4 concurrent kernels.

The results show that there are performance improvements using from one kernel to four (or eight, for bigger threadblock size) concurrent kernels due to the exploitation of the data-caches and block/warp dispatchers. Through the CUDA VisualProfiler we have observed that the use of concurrent kernels slightly reduces the number of L1 data-caches misses for this kind of kernels. However, although the performance times from this number of concurrent kernel are also good, the performance cannot be improved any more by introducing new kernels, because the global memory bottlenecks and data-cache trashing effects increase too much.

Total performance gain

From the previous experiments, the worst configuration is obtained with 256 threads per block and 1 single concurrent kernel. On the other hand, the best results are reached with 96 threads per block and 4 concurrent kernels. Figure 6.10 shows the execution times for the best and the worst recommended configurations for bigger problems such as 16 384 and 32 768-source-node to all. The global performance gain reached between the worst configuration (256 threads + 1 kernel) and the best (96 threads + 4 concurrent kernels) is an 11.5 %.

6.4.3 APSP-problem: Conclusions

The results of our experiments corroborate the conclusions of the work described in Sect. 5.1 and 5.3. For kernels with the same features as our *relax kernel*, an optimal performance is achieved if smaller threadblock sizes that do not reach the maximum occupancy are used.

6.5 Load balancing techniques for the APSP problem

We first describe briefly the equitable scheduling and numbered ticket load-balancing techniques evaluated, and after, the methodology used for our experiments.

6.5.1 Load-balancing techniques evaluated

A simple way to apply load-balancing to a heterogeneous system is to equitably distribute the work without taking into account the computational capabilities of the devices. This kind of techniques usually lead to easy implementations, but at the expense of having a temporal cost equal to the time that the worst device needs to compute its work. Equitable Scheduling can be classified as a static load-balancing technique at compile time.

Our *Equitable Scheduling* (ES) approach statically divides the workspace between the computing threads giving to each one the same quantity of tasks. If nc represents the number of computing threads, id the thread identifier, and $nt = |V|/nc$ the number of tasks per thread, this approach makes each thread responsible for computing the tasks from $id \cdot nt$ to $id \cdot nt + nt - 1$. If this task division is not exact, each of the first threads takes one of the remaining tasks until there is no more work to do.

Numbered Ticked Scheduling is commonly employed to accomplish a dynamic work scheduling between any kind of hardware device. All hardware devices of the heterogeneous system can steal a task from the global task queue. Note that the access to the global task queue must be implemented with some kind of synchronization in order to avoid that

two or more devices steal the same task. Usually, this synchronization involves a bottleneck in the execution times. Work-Stealing scheduling can be classified as a dynamic load-balancing technique at runtime.

Our *Numbered Ticked Scheduling* (NT or WS) approach lets to an idle thread that has finished its previous work to steal the following task t_i. This task is immediately eliminated from the queue at the moment it is taken. Then, the thread computes the corresponding NSSP problem with node i as source. Finally, when the thread ends its work, it comes back to the global task queue in order to take another one, repeating the process till there is no more pending work. The synchronization of the task stealing has been implemented using an atomic region. That means that only one thread can be taking the following work at any moment.

6.5.2 Methodology

We have compared our heterogeneous implementations against a single GPU implementation described in Sect. 6.4, that we have denominated baseline, in order to evaluate the performance gain of using heterogeneous systems for the Crauser *et al.* APSP problem. The algorithm implemented for GPU devices is an adaptation of [28] ideas for the CUDA architecture presented in [92]. Moreover, the sequential version of this algorithm is used for the CPU devices.

Several instances with different number of OpenMP threads, for both load-balancing methods presented, have been executed in order to determine the best configuration. These instances have been tested with graphs of $1 \cdot 2^{20}$ nodes solving the complete APSP problem. Additionally, we have used for our experiments bigger graphs whose number of nodes is ranging from $1 \cdot 2^{20}$ to $11 \cdot 2^{20}$. However, due to the large amount of computational load needed to solve the APSP in these big graphs, we have bounded the problem to a *512-source-nodes-to-all* in order to reduce the global execution time. For the selection of these source nodes we have used the family of random functions *srand48()* from standard C library.

6.5.3 Target architectures

For this experiments, the evaluated heterogeneous system is composed by different computational units that are grouped in two categories:

- The shared-memory CPU system cores of the host machine. It is an Intel(R) Core(TM) i7 CPU 960 3.20 GHz, 64-bits compatible, with a global memory of 6 GB DDR3.
- Two GPU devices of different architectures attached to the host machine:
  - A GeForce GTX 680 (Kepler) NVIDIA GPU device, and

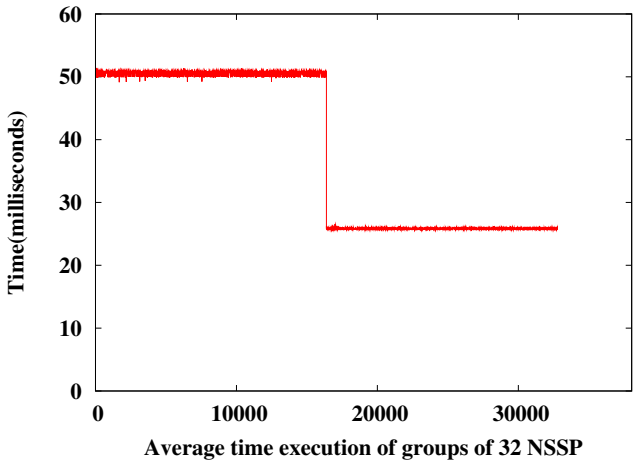


Figure 6.11: Temporal cost of the different source nodes in the graph for the Kepler GPU.

– A GeForce GTX 480 (Fermi) NVIDIA GPU device.

The baseline implementation is executed in the same shared memory host machine of the previously described heterogeneous system, but it only uses the most powerful GPU device as computational unit: The GeForce GTX 680 (Kepler) GPU device.

Regarding the software used, the host machine runs an UBUNTU Desktop 10.10 (64 bits) operating system, and the experiments have been launched using CUDA 4.2 toolkit and the 295.41 64-bit driver.

6.5.4 Input set characteristics

The input set is composed by a collection of graphs randomly generated by a graph-creation tool used by [72] in their experiments. They have been created adding seven adjacent predecessors to each node of the graph. Afterwards, they have inverted the graphs in order to store the node successors sequentially. These graphs are represented through adjacency lists, the nodes are numbered from $0 \ldots |V| - 1$, and the edge weights are integers that randomly range from $1 \ldots 10$.

The node distribution of this kind of graphs shows an irregular behavior for the computational time of the APSP problem in terms of each NSSP subproblem. The iterations of the first nodes of the graph need more computational time to solve its NSSP problem than the final ones. Figure 6.11 shows, using intervals of 32 nodes, how the time needed is considerably reduced when the baseline implementation executes with an starting node of the second half of the nodes set. Due to the nature of the problem, there are no inter-NSSP dependencies and communication in the complete APSP computation.

Legend	Description
G_1	Single GPU thread (Kepler)
E_2 / W_2	2 GPU threads (Fermi & Kepler)
E_3 / W_3	2 GPU threads + 1 CPU threads
E_4 / W_4	2 GPU threads + 2 CPU threads
E_6 / W_6	2 GPU threads + 4 CPU threads
E_8 / W_8	2 GPU threads + 6 CPU threads
E_{14} / W_{14}	2 GPU threads + 12 CPU threads
E_{16} / W_{16}	2 GPU threads + 14 CPU threads

Table 6.3: Experimental instances.

6.5.5 Load-balancing techniques evaluated

Both load-balancing techniques described, Equitable Scheduling and Numbered Ticket, have been implemented with support to different number of OpenMP threads. Several instances with different number of threads have been evaluated against the baseline implementation.

On our results tables and plots we have tagged each instance, depending on which load-balancing technique implements, with the label "E" for Equitable Scheduling, and "W" for Numbered Ticket scheduling instances, followed by a number that represents the number of OpenMP threads used (see Table 6.3). Thus, the evaluated instances "E_3" and "W_8" are a implementation of equitable scheduling with 3 threads, and a implementation of numbered ticket scheduling with 8 threads respectively.

The first two threads are always assigned to the two GPU devices, one for each graphic accelerator. The rest of the threads are executed in the CPU-cores. Therefore, the instances "E_2" and "W_2" only use the GPUs resources.

6.5.6 Experimental results

In this section we present the experimental results obtained for the execution of the complete APSP with $|V| = 1 \cdot 2^{20}$, and the 512-source-to-all for graphs which number of nodes ranges from $1 \cdot 2^{20}$ to $11 \cdot 2^{20}$.

Complete APSP

- **Equitable Scheduling**: Figure 6.12(a) presents the execution times of equitable scheduling technique for instances with different number of OpenMP threads. The performance of the baseline approach (G_1) is significantly improved when a second GPU device is used (E_2). However, a $2\times$ speed-up is not reached because the architectures of the used GPUs are different. This means that the total execution

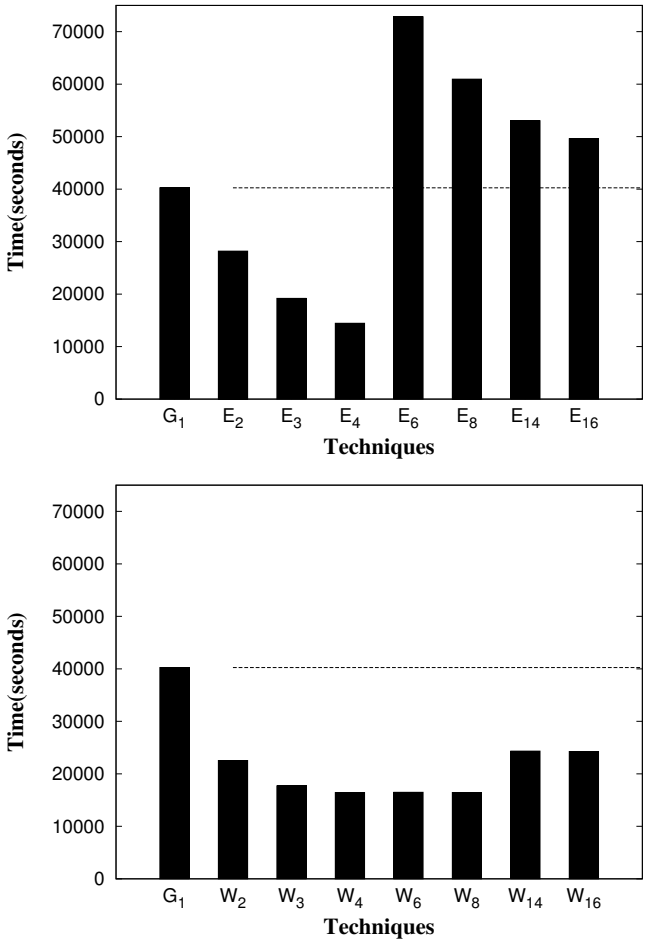


Figure 6.12: Execution times of (a) Equitable and (b) Numbered Ticket Scheduling policies.

time corresponds with the total execution time of the less powerful GPU device. Nonetheless, E_2 presents a 30 % of performance improvement against the baseline.

The use of one and two CPU-cores (E_3 and E_4) helps to decrease this critical execution time because the number of subproblems (SSSP problems) that the critical GPU has to resolve is reduced. The instance E_4 shows a 65 % of performance improvement. The higher the number of launched threads, the lesser the computation load given to each GPU device. Nevertheless, due to the irregular nature of the graph (see the distribution time in Fig. 6.11), there is a threshold where the equitable partition overloads so much the work that the CPU-cores have. This occurs when the most costly tasks, that they were assigned to GPUs before, are assigned to CPU-cores. For this reason, the total execution time of the approach E_6 is significantly increased even surpassing the baseline time. Furthermore, as more threads are launched from this point, the total time execution is reduced.

- **Numbered-Ticket Scheduling**: Figure 6.12(b) shows the execution time results

of the Numbered ticket technique for instances with different number of OpenMP threads. The performance of the baseline approach (G_1) is significantly improved by any experimental instance that uses the numbered ticket method (W_i). The instance that uses only two GPUs has a 44 % of performance improvement against the baseline. As we increase the number of OpenMP threads, more hardware devices are used, reducing the execution times. Although the most costly tasks are also taken by the CPU-cores, while they are computing their subproblem, the GPUs are continuously stealing tasks. The instance with the fastest execution times is the W_4 instance, leading to a 60 % of performance improvement. However, when the number of launched threads exceeds the number of heterogeneous computational units (W_{14} and W_{16}), the execution of threads that belong to the same CPU-core is concurrent. This behavior leads to slightly penalty times, reaching a performance improvement of 40 % against the baseline.

512-Source-node-to-all shortest path

- **Equitable Scheduling**: Figure 6.13(a) presents the execution times for scenarios using the equitable scheduling implementation, for different number of OpenMP launched threads. The best performance is obtained with the E_2 configuration, leading to a 45 % of performance improvement against the baseline.

  The heterogeneous approaches with CPU-cores ($E_{\{3...16\}}$) have worse execution times than the baseline due to memory access bottlenecks. That is because the CPUs are taking costly tasks due to the random nature of the 512 nodes selection. However, as it happened in the complete APSP scenario, this time is reduced when more threads are launched.

- **Numbered-Ticket Scheduling**: Figure 6.13(b) shows the numbered ticket implementation for different OpenMP launched threads. As it happens in the APSP scenario, the execution time of any numbered ticket instance ($W_{\{2...16\}}$) is better than the baseline (G_1). The instance of two threads that only uses GPU devices, W_2, has a very good performance against the baseline (46 % of performance improvement). Inserting an additionally CPU-core to the heterogeneous system, W_3, leads to an even better performance improvement of 47 %. However, adding more than one CPU-cores to the heterogeneous system, $W_{\{4...16\}}$, leads to slightly worse execution times compared with the best.

Experimental conclusions

The best execution time for the complete APSP scenario is achieved with an equitable scheduling implementation, E_4, leading to an 65 % of performance improvement com-

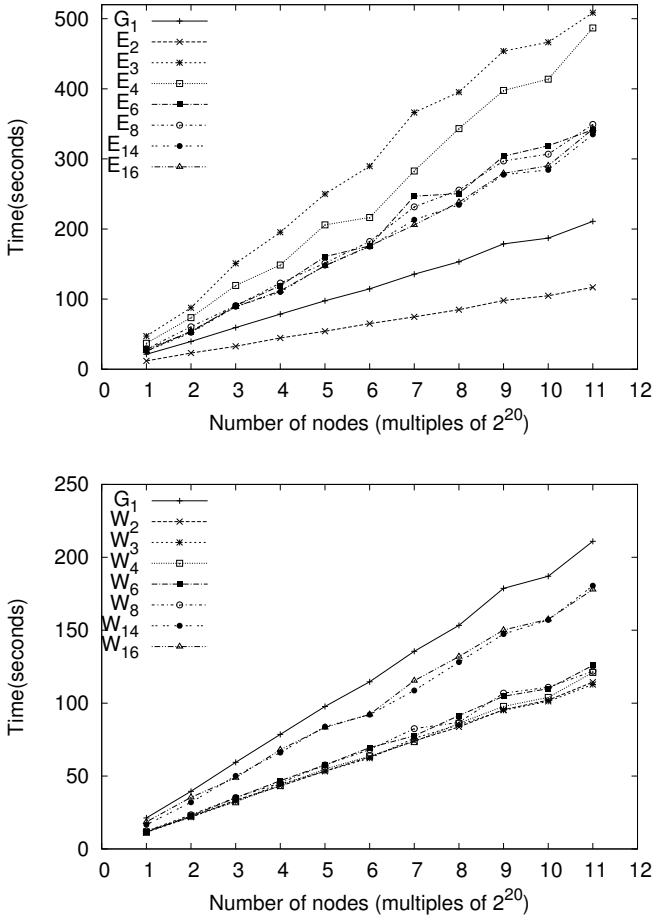


Figure 6.13: 512 nodes execution times of (a) Equitable and (b) Numbered Ticket Scheduling.

pared with the baseline G_1. However, the next approaches that closely follows this improvement are those that use a work-stealing implementation, ($W_{\{3...8\}}$), instead of other equitable scheduling instances with similar thread configurations.

For the 512-source-to-all scenario, the best results are reached with a numbered ticket implementation, W_3, with a 47 % of improvement compared to G_1. The equitable scheduling approach looses performance compared to the baseline for any thread configurations excepting the version that only uses GPUs, E_2.

These results show that (1) the equitable scheduling can be tuned up to achieve the best performance times avoiding critical code regions but it is very sensible to changes of the input graph, and (2) the numbered ticket implementations have a more robust performance than the equitable scheduling because it is more independent of the graph nature.

6.5.7 Load balancing techniques: Conclusions

The solutions described have achieved a performance improvement up to 65 % compared with the baseline single-GPU solution. Moreover, the results of our experiments have shown that the numbered ticket implementation with the same number of OpenMP threads have given a good performance for all tested scenarios. However, the equitable scheduling implementation, that involves CPU-cores, has not shown a significantly performance improvement if the nature of the graph is not taken into account.

Our first conclusion is that the jointly use of very different computational power devices is useful to improve the total execution time compared with the fastest GPU implementation. Second, the previous study of the nature of the input problem allows us to better mapping the most costly tasks to the most powerful devices. For our case, the equitable scheduling that maps all costly tasks to the GPUs, and leaves light ones to the CPU-cores, leads to the best performance. Finally, the application of the numbered ticket technique results in a more robust implementation compared to the equitable scheduling because it is less sensible to the nature of the input problem.

Chapter 7

Conclusions

In this dissertation we have presented a study of several problems related to the programming of heterogeneous systems. We discuss several policies to select good thread-block geometry and other execution parameters to exploit the GPU device architecture efficiently. We have also developed a software tool, synthetic benchmarks, and some real-world applications to study the feasibility of automating these high-risk decisions as well as automatic data partitioning techniques and communication tools for heterogeneous systems. This chapter summarizes the main contributions provided by this Thesis, presents its main conclusions with the answer to the research question proposed in Sect. 1.2.1, and finally discusses the future directions of this work.

7.1 Summary of contributions

First

We contributed to the development of Hitmap, a library designed to decouple the communication pattern from data partitioning, thanks to the use of abstract expressions of the communications, that are automatically adapted at runtime depending on the partition policy finally chosen. We then used the Hitmap abstractions for homogeneous systems to design new ones focused on heterogeneous environments.

1. Arturo Gonzalez-Escribano, Yuri Torres, Javier Fresno, and Diego R. Llanos. An Extensible System for Multilevel Automatic Data Partition and Mapping. Parallel and Distributed Systems, IEEE Trans. on Parallel and Distributed Systems, PP(99):1–1, 2013. [45].

Second

We provided new insights into the relationship between performance and several key programmer decisions when using GPU architectures. This includes occupancy,

threadblock size and shape, Fermi cache hierarchy configuration, and thread access pattern on the global memory.

2. Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Understanding the impact of CUDA tuning techniques for Fermi. In High Performance Computing and Simulation (HPCS), 2011 International Conference on, pages 631–639, 2011. [113].
3. Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. CUDA Tuning and Con- figuration Parameters on Fermi Architectures. In Advanced Computer Architecture and Compilation for High Performance and Embedded Systems (ACACES 2011), 2011. [112].
4. Yuri Torres, Arturo Gonzalez -Escribano, and Diego R. Llanos. Uso del conocimiento de la arquitectura Fermi para mejorar el rendimiento en aplicaciones CUDA. In Actas XXII Jornadas de Paralelismo, 2011. [114].
5. Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Using Fermi architecture knowledge to speed up CUDA and OpenCL programs. In Proc. ISPA'12, Leganes, Madrid, Spain, 2012. [118].

Third

We introduced a complete suite of micro-benchmarks (called uBench) to further explore the impact on performance of: (a) The thread-block size and shape choice criteria, and (b) the GPU hardware resources and configuration. This benchmark suite covers the hardware details of Fermi and Kepler architectures.

6. Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Measuring the Impact of Configuration Parameters in CUDA Through Benchmarking. In The 12th International Conference Computational and Mathematical Methods in Science and Engineering, CMMSE 2012, 2012. [116].
7. Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. uBench: Performance Impact of CUDA Block Geometry. Technical Report IT-DI-2012-0001, Depto. Informática, Universidad de Valladolid, Dec 2012. [117].
8. Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. uBench: exposing the impact of CUDA block geometry in terms of performance. The Journal of Supercomputing, 65(3):1150–1163, 2013. [120].

Fourth

We developed and used two real-world applications, such as, the SSSP and APSP problem to test and verify the conclusion obtained in [112, 113, 114, 116, 117, 118, 120] in these representative benchmarks.

9. Hector Ortega-Arranz, Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. A New GPU-based Approach to the Shortest Path Problem. In The 2013 International Conference on High Performance Computing & Simulation, (HPCS 2013), pages 505–511, 2013. [92].
10. Hector Ortega-Arranz, Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. A Tuned, Concurrent Multi-Kenel Approach to the APSP problem. In The 13th International Conference Computational and Mathematical Methods in Science and Engineering, CMMSE 2013, 2013. [93].

Fifth

We presented a programming framework extending Hitmap library in order to analyze the possibility of creating a programming model and framework that supports heterogeneous platforms encapsulating: (a) Policies for the selection of good values of GPU configuration parameters, (b) the management of the abstract tile data structures, (c) mapping and load balancing functions, and (d) synchronization/communication functionalities between CPU-GPU heterogeneous devices.

11. Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Automatic Data Layout at Multiple Levels for CUDA. In The 10th International Conference Computational and Mathematical Methods in Science and Engineering, CMMSE 2010, 2010. [111].
12. Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Data partition and synchronization in heterogeneous systems. In HPC-EUROPA2 project (project number: 228398) with the support of the European Commission - Capacities Area - Research Infrastructures, 2013. [119].
13. Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Encapsulated Synchronization and Load-Balance in Heterogeneous Programming. In Euro-Par 2012 Parallel Processing, volume 7484 of LNCS, pages 502–513. Springer Berlin Heidelberg, 2012. [115].
14. Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Automatic run-time mapping of polyhedral computations to heterogeneous devices with memory-size restrictions. In The 2013 International COnference on Parallel and Distributed Processing Techniques and Applications, 2013. [53].

7.2 Conclusions

Based on the information, discussions, and results shown throughout the previous chapters of this Ph.D. Thesis, the main conclusions are:

- Both homogeneous and heterogeneous Hitmap lead to more abstract programs, easier to code and maintain, still obtaining good performance results. These libraries allow to achieve similar performance as carefully-implemented manual versions for several, well-known parallel kernels and benchmarks in distributed and multi-core systems, and substantially reduces programming effort.

- The choice of global GPU configuration parameters are closely related to the particular parallel problem implementation. A combined analysis of the knowledge of a specific GPU card architecture, and code features, such as the type of global memory access pattern (coalesced vs. scatter), the total workload per thread, and the ratio of global memory read-write operations, can significantly help to choose programming parameters, such as threadblock geometry and L1 cache memory configuration, that have a significant impact on performance.

- It is possible to develop a portable and transparent programming system that incorporates hierarchical tiling and scheduling policies, in order to take advantage of heterogeneous computing capabilities.

7.3 Future directions

There are still some issues we would like to address. These issues and paths define the future directions of this work:

- We are currently working on designing more sophisticated mapping policies that better exploit CPU-cores and GPU architecture information. We want to test the applicability of these techniques to more types of programs, including well-know benchmarks and real applications.

- We plan to study the influence of hardware effects and threadblock configuration parameters in other GPU architectures vendors, such as AMD and Intel.

- We are curious to study the FPGAs (Field Programmable Gate Array) devices because, its hardware functionality can be reconfigured several times. We would like to automate decisions that any programmer should make before launching any FPGA function. We would like to give support for FPGAs devices in our programming framework.

Appendix A

CUDA Programming Model

A.1 CUDA model

CUDA (an acronym for Compute Unified Device Architecture) [78, 82, 85, 90] is a parallel computing architecture developed by NVIDIA Company. CUDA model is the computing engine in NVIDIA graphics processing units (GPUs) that is accessible to software developers through variants of industry standard programming languages.

This model allows the programmer to focus on algorithm efficiency and also to develop scalable parallel programs. The University of Virginia carried out a study that uses CUDA during several weeks to check how easy is to learn this model. The students of Virginia were able to write correct parallel programs in only three weeks, and to see the advantages of multi-core GPUs execution. The projects that have been addressed by this university belong to different areas, such as medical imaging (Leukocyte, Heart Wall) [57, 74], bioinformatics (MUMmerGPU) [75, 105], fluid dynamics (CFD Solver) [38, 121], linear algebra (LU Decomposition) [24, 25], physics simulation (HotSpot) [35, 48], pattern recognition (back propagation) [50, 63], data mining (Kmeans) [1, 68], and graph algorithms (Breadth-First Search) [46, 95], among others.

There are many applications [42, 91, 134] that use this parallel programming model to get more execution performance. For example, FHD-spiral MRI reconstructions, molecular dynamics, N-body astrophysic simulations, and images treatment. These applications need many computation resources to minimize the global execution time. For this reason, the GPUs architectures and GPGPU languages are well suited to these problems.

CUDA parallel programming model is a C or C++ language extension that works with shared-memory architecture and allows to exploit the GPUs technologies. This parallel programming extension was implemented by the NVIDIA company, that also produces several GPU models like GeForce TESLA and QUADRO. CUDA forces the programmer to explicitly use thread-groups which are related to the memory hierarchy. The thread-groups contain threads that are executed concurrently, sharing a cache memory. The size

of these local memories should be considered by the programmer to efficiently exploit them, having an important impact of performance. Indeed, this architecture provides a hierarchy of memory levels. For example, in the low level, we find the main or system memory. In a second higher level the GPU main memory. Finally, we find the memory shared by the threads of a threadblock. Different synchronization techniques and issues are related with the different memory levels.

The programmer writes his program in a serial form with several kernel calls (functions that will be executed on the GPU devices), which are simple sequential functions. These functions will be replicated across multiple threads that will execute the tasks concurrently. Threads executing the same kernel function are organized in threadblocks. In CUDA the threadblock is expressed as an array of threads with one, two, or three dimensions. The actual size (number of threads) is the product of this array domain cardinalities. Each thread is identified by as many indexes as dimensions on the threadblock declaration.

The threads in different threadblocks may communicate among themselves via shared memory and synchronize with certain primitives. A grid is the array of threadblocks that may be executed independently on a single GPU device. One grid is started or launched by a single GPU call with the corresponding parameters for the kernel function to be executed by the threads.

Each thread of a threadblock has an ID number called *threadIdx*, and also, each threadblock has a unique block ID number within its grid, called *blockIdx*. Thus, we can identify each thread and threadblock within a grid, and then, assign work to each thread. A grid may have two or three dimensions depending on CUDA version.

A.1.1 Brief examples

Matrix multiplication

A very easy example could be the multiplication between a vector and a scalar number. This vector would have N elements, and a efficient form to execute this operation on a GPU with CUDA model would be to assign each vector element multiplication to a different thread. In a sequential implementation, it takes N steps to execute all multiplications in sequence. Using the CUDA model, each multiplication would still take one step but all of them may be executed in parallel in only one concurrent execution step (without considering other costs in the synchronization). In this way, we have reduced significantly the global execution time.

In the CUDA implementation we have several declaration qualifiers, for example __global__. This declaration specifies a kernel entry point. The programs launch parallel kernels with the following programming language extension:

```
kernel<<<dimGrid, dimBlock>>>(...functions parameters...);
```

Where *dimGrid* and *dimBlock* are declarations that specify the number of blocks and the number of threads per dimension respectively.

Computing y ← ax + y with a Serial Loop

```
void saxpy_serial(int n, float alpha, float *x, float *y)
{
   for(int i = 0; i<n; ++i)
      y[i] = alpha*x[i] + y[i];
}

// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Computing y ← ax + y in parallel using CUDA

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;

   if( i<n )  y[i] = alpha*x[i] + y[i];
}

// Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Figure A.1: Computing $Y \leftarrow Ax + y$ in CUDA parallel model (this figure is obtained from [78]).

In Example A.1, the grid would have one threadblock with N threads, that will be spawned in a single GPU device call. By default, CUDA assumes that there are no dependencies among the threadblocks in a grid, executing them in parallel. The threads in each threadblock are executed concurrently depending on potential synchronization primitives that can be added. We can put several grids in our program but CUDA launches them in sequential form. In other words, the grid can not be executed in parallel with other grids. This limitation will be further explained in later sections. However, in the new GPU device architecture (Fermi or Kepler [81, 84, 86]), it is possible to launch several grids to a single GPU,and they are executed in parallel.

Parallel reductions

Other easy-to-understand problem is the *parallel reductions*. This algorithm initially seems to be sequential, as we are used to sum up one element at a time, remembering only one partial result along the whole operation. However, as it is shown in Fig. A.2, several threads may concurrently compute partial results which are also reduced in further steps, until we obtain one single value. Thus, a parallel algorithm may reduce the global execution time.

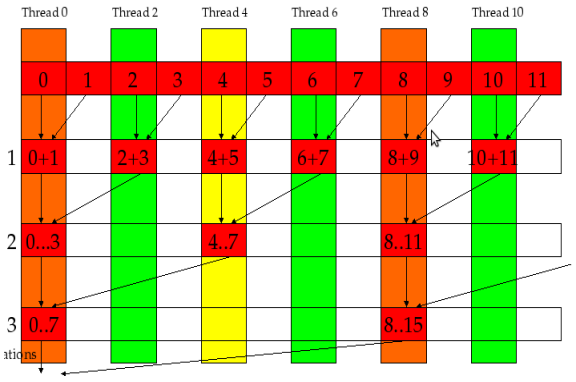


Figure A.2: Parallel sum reduction tree (this figure is obtained from [83]).

A.1.2 Thread organization

Figure A.3 illustrates the organization of threads within a threadblock. In this figure, each threadblock is organized as an array of 2×4 threads. All blocks within a grid have the same dimensions for the threads array. The specifications provided by each CUDA architecture technology define a maximum number of threads per block. This number is 512 or 1024 threads, depending on the technology. In this example, we have 6 threadblocks with 2×3 threads (a grand total of $12 \times 6 = 72$ threads in the grid). Typical CUDA grids contain thousands millions of threads.

A.1.3 Synchronization barriers

In CUDA model, different issues appear when synchronizing the threads. The threads of a threadblock can be synchronized by __syncthreads() calls. These calls allow to stop the threads that are involved in the barrier until all threads reaches it. This guarantees that a thread may see all concurrently computed data without any race conditions.

There is no direct mechanism to synchronize thread execution of different threadblocks. However, there exist atomic operations to avoid race conditions when threads of different blocks access to global memory positions. All the blocks in a grid are synchronized at the end of its execution with the equivalent of a barrier.

We may declare and use several grids in our program. These grids can be dependent or independent. Dependent grid will be executed in sequential mode with old GPU technologies, while the independent grids may be executed concurrently, given sufficient hardware resources, with current Fermi technology. By default, concurrent grids are not synchronized at their end. However, GPU to CPU memory blocking operations may be used to create global synchronization points for grid sets.

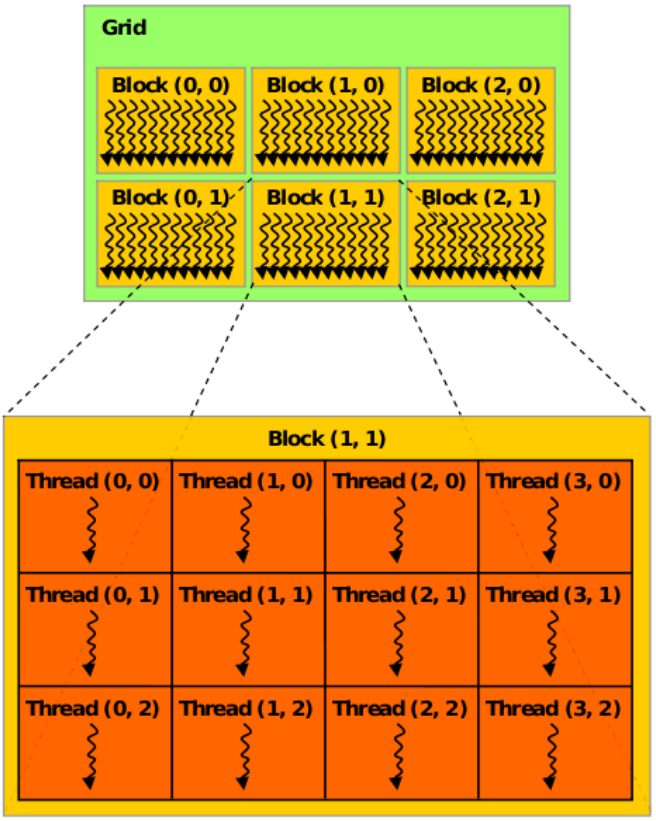


Figure A.3: Grid of threadblocks (this figure is obtained from [83]).

A.1.4 Memory accesses

All threads of a grid can access elements on several memories. For example, accesses to the global memory (memory hosted on a Host) pass across the GPU to CPU communication system (typically the system bus). When a thread executes a global memory call, it spends a significant amount of time, slowing the program execution. There also exists the GPU main memory, used to save the global variables for the kernel computation, together with big sets of data. Finally, it exists the shared memory between the threads of a threadblock. This memory is named GPU cache memory. As its name suggests, it is a faster memory. However, its main disadvantage is its small size. There is a memory hierarchy, where the high level is the faster and smallest-sized memory (cache), and in the low level we find the slowest and the most abundant memories. To book shared memory we have to use the *__shared__* qualifiers within the GPU code (code that will be executed by GPU). There are two other specific-purpose memories that are in the same level than the shared cache memory: (1) The constant cache that store the different constant values used in a code, and (2) the texture cache that was initially thought to improve the image-processing operators related to texture application.

One of the most important things to consider when programming in CUDA is the

cache memory size. We have said that this memory is very fast, and we have to use it as much as possible. When we choose a little threadblock (few threads in the block), the amount of threads may also limit the amount of shared data that may be located and exploited on the cache memory to reduce the global memory latencies. This situation could be improved by increasing the number of threads per threadblock. However, if the threadblock has too many threads, the cache size may be too small to locate all the shared data they need, forcing to use global memory with higher latencies.

Thus, when we try to choose a threadblock size it is very important to be acquainted with all levels of the memory architecture, as well as to be aware that threadblocks with different sizes can give us very different behaviors (in terms of computation efficiency). The sequential program executed in the CPU may manage the global memory space visible to kernels or grids by calls to CUDA run-time functions. For example, *cudaMalloc()* and *cudaMemcpy()* primitives allows to copy portions of data from system global memory towards GPU device memory.

A.1.5 CUDA architecture

CUDA programming model allows the execution of general purpose applications. For this reason, thousands of data-parallel threads can be used on NVIDIA's CUDA architecture [82, 85]. But is very important to know how the multiples kernels in a NVIDIA GPU are organized and executed. On an NVIDIA GPU there are several streaming multiprocessor called SMs (or MP Multi-processors). They are shown in the Fig. A.4. All these SMs share the same device memory, that is the global memory of a GPU card. Each SM has a set of scalar processing elements called SPs. Each SP executes a different thread, but they work with a SIMD model (Simple Instructions, Multiple Data). In other words, these sets of SPs within the same SM execute the same instruction at the same time. Each SM has a cache memory and a set of registers shared by the threads executed by the SPs.

Each SM has 8 or 32 SPs depending on the technology version. However, CUDA groups the threads in sets of 32, to be executed in the same SM with the SIMD model. This sets are called *Warps*. Each warp of 32 concurrent threads is scheduled on the 8, 32 or 192 SPs. When threads in a warp access memory they stall, waiting for the data. It is convenient to have other warps (from the same or from a different threadblock) to keep the SM busy while data arrives from memory. A sufficient number of warps help to hide global memory latencies. However, all the warps of a threadblock must be scheduled on the same SM, and the architecture and technology version impose a maximum number of warps supported by the SM.

When the user launches a kernel in a CUDA program, she can declare several threadblocks executing the same kernel function. Each threadblock consists of at most 1024 threads (with the last technology) and each one of them is assigned to a single SM, being

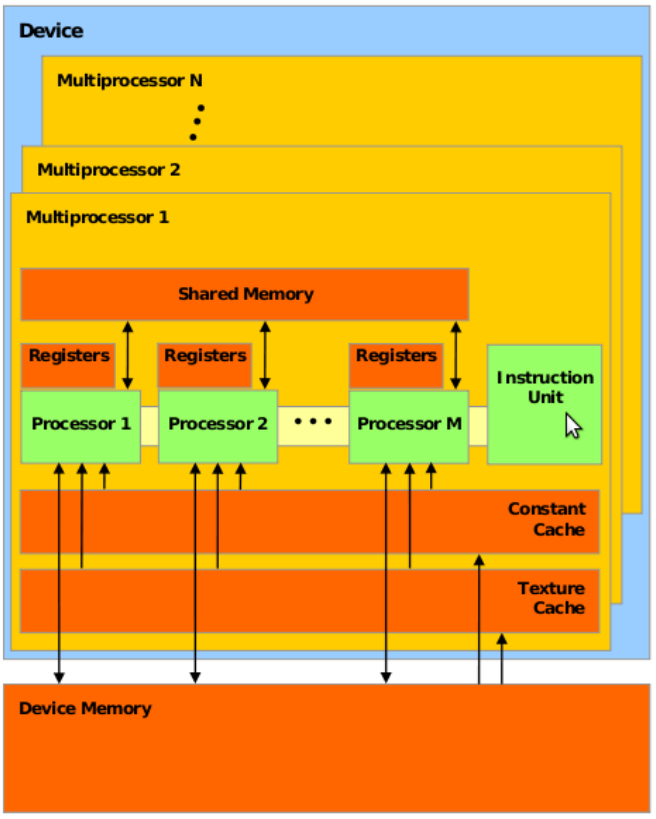


Figure A.4: A set of SIMT multiprocessors with on-chip shared memory (this figure is obtained from [81]).

executed without preemption. This set of threads are executed in SIMD model. There is also a limited number of warps scheduled to a single SM at a given moment. This also sets a maximum number of threads per SM, independently of the threadblock sizes.

The CUDA model specifies that the order of the execution of threadblocks within a single kernel is undefined. With this situation, communications between the different threadblocks are not allowed. These communications are substituted by access to the global memories.

A.2 Concurrent kernels

Since the advent of the second generation of NVIDIA architectures, CUDA supports concurrent kernel execution. With this feature, different kernels can be executed concurrently, allowing better utilization of GPU resources. The maximum limit of concurrent kernels are 16 for Fermi (GF110 serie) and Kepler (GK104 serie).

Moreover, the advantages of concurrent kernel execution are automatically exploited by the CUDA kernel dispatcher, providing a high efficiency without an extensive user

intervention. The authors in [85] show that the best performance using concurrent kernels is achieved with small kernels. The maximum number of concurrent kernels that can be allocated efficiently in the GPU depends on the use of the hardware resources made by each one. The less resources used by each kernel, the more kernels that can be efficiently computed concurrently.

A.3 CUDA heterogeneous programming

Heterogeneity is a term that is related to execute the same code across several devices of a different nature. The heterogeneous computing [17, 76] is a technique that uses a collection of different devices, for example CPUs and GPUs, to solve a particular parallel problem. To exploit efficiently the parallel program with an heterogeneous model it is highly recommended to know the GPUs device architectures.

CUDA programming model is oriented to support heterogeneity [82] on the program executions. In other words, the parallel code will be executed by CPU and GPU devices working jointly. We will have to specify the parts that we want to execute onto the CPU and onto the GPU device. Thus, it is possible to exploit all devices in parallel.

In Fig. A.5, we can see a generic diagram of the sequential model of execution for CUDA programs. First, we have the serial code executed by the CPU device (in this example called Host) and then, the parallel execution launched by CPU that will be executed on the GPU device. The Fermi architecture does not block the Host execution flow when a single kernel is launched.

A.4 CUDA strengths and weaknesses

So far, we have only spoken about CUDA programming model features, but it is very important to summarize and highlight the CUDA low-level software and hardware, that imposes both advantages and limitations.

A.4.1 Advantages

CUDA provides an easy-to-understand model to implement parallel programs on NVIDIA GPU cards. Some of the most important features of the CUDA programming model are, among others, asynchronous memory copy between GPU and CPU, and support for integer and bitwise operations, including integer texture lookups. The cache of each SM is divided in 16 individual modules that access to multiple cache modules at the same time. For this reason, the cache bandwidth is considerably increased. This cache memory is a shared memory between all threads of a threadblock, then, the threads may read and

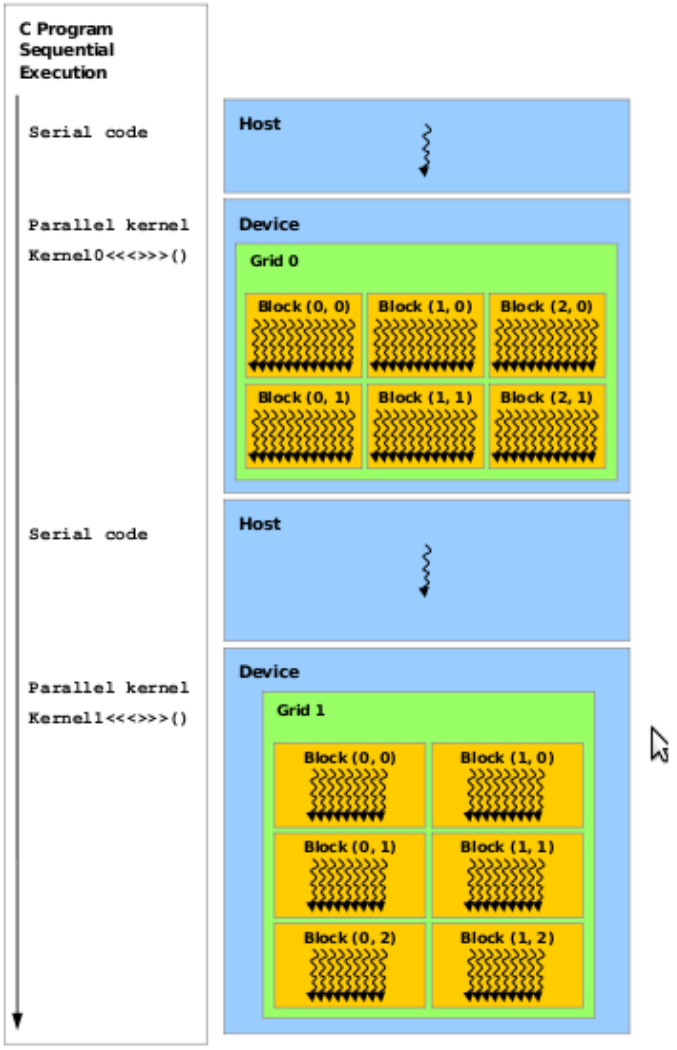


Figure A.5: Serial code executes on the host while parallel code executes on the device (this figure is obtained from [83]).

write data (taken into account the synchronization primitives) that are immediately visible for the rest of the threads. The global memories are more slower than the cache modules. Therefore, when a GPU device wants to read a CPU global memory, it has to go through the CPU device, and this situation produces a noticeable delay. CUDA provides high bandwidth to mitigate this situation. There is also a primitive that allows the copy of data to a global GPU device memory at same time that a kernel is called to be executed. With this primitive, it is possible to start the kernel execution without waiting for the data copy.

A.4.2 Constraints

The studied CUDA versions (up to CUDA 4.2) do not allow recursion techniques. Besides this, a single process should run across multiple disjoint memory spaces, unlike other C

language runtime environments, and the texture rendering is not supported.

CUDA programming synchronization may be insufficient [78]. In the CUDA programming model, the threadblock are independent. In other words, there is not communication and synchronization mechanisms among them. Each threadblock is launched to a single SP (with eight cores), then, each threadblock will have its own cache without access to another cache.

For double precision (only supported in newer GPUs like GTX 260) there are some deviations from the IEEE 745 standard: round-to-nearest-even is the only supported rounding mode for reciprocal, division, and square root. In single precision, abnormals and signaling NaNs are not supported; only two IEEE rounding modes are supported (chop and round-to-nearest even), and those are specified on a per-instruction basis rather than in a control word; and the precision of division/square root is slightly lower than single precision. The bus bandwidth and latency between the CPU and the GPU may be a bottleneck.

The atomic operations are slower than any other memory access operation affecting the system performance. The CUDA atomic operations are operations that are performed without interference from any other threads. In CUDA, atomic operations are often used to prevent race situations which are common problems in multithreaded applications. In CUDA model, an atomic operation can read, modify, and write a single value without interference of any other threads, which guarantees that a race condition will not occur. The time requested to finish the atomic operations is closely related with the number of threads in a single grid.

A.4.3 Summary

In order to program with CUDA model it is very important to know the different restrictions imposed by the model. The threads and threadblocks can only be created invoking a parallel kernel (functions executed in the GPU devices). A kernel function may not invoke new kernels. The levels of parallelism are restricted. Parallel tasks also are expressed at the threadblock level. Synchronization mechanisms provided are limited to synchronization barriers inside the threadblocks. Different threadblocks within the same grid, may read/write on the global memory to communicate. But there is no mechanism to synchronize such memory operations between threads of a different threadblock. Thus, threadblocks are designed to be executed independently.

When we define the threadblock size, we have to know these limitations. This size will be closely related to the global performance. It is related to the use of synchronization barriers, memory hierarchy, data transfers and latencies, communications, and shared memory sizes, among others. As we stated above, CUDA model does not allow recursive calls. The reason is that using ten of thousands of threads with recursive calls will need

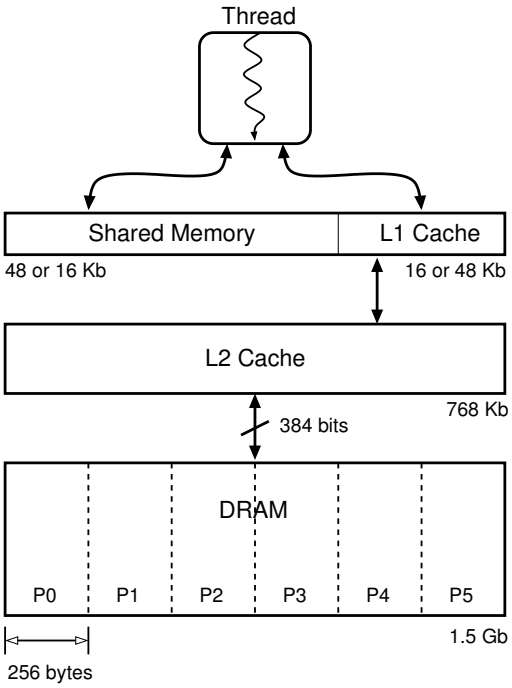


Figure A.6: Fermi memory hierarchy (NVIDIA GTX-480).

a big stack management, not supported by the studied architecture (up to Kepler GK 104 serie). It is possible to manually develop programs which exploit heterogeneity, but using explicit asynchronous memory transfers between the CPU and GPU devices. These memory copy operations across devices may take long times when compared with the GPU-CPU performance, generating a bottleneck for the computation.

A.5 Review of NVIDIA GPUs architectures

Pre-Fermi is the first NVIDIA CUDA supported architecture, launched in early 2007 [83]. Fermi is the second generation of CUDA architectures [83], launched on early 2010. The latest generation of CUDA architecture at the time of writing is Kepler [85, 86], released on early 2012. Table A.1 summarizes their main characteristics.

Each new architecture generation has increased the number of SPs (Streaming Processors), and the maximum number of threads, per SM (Streaming Multiprocessor). This leads to different relations between the threadblock configuration and the occupancy on the SMs. Thus, the policies to select a good threadblock configuration are potentially subject to changes.

The main change introduced by Fermi is a transparent L1/L2 cache hierarchy that has been maintained in Kepler (see Fig. A.6). L1 caching in Kepler GPUs is reserved only for local memory accesses, such as register spills and stack data. Global loads are cached in L2 only or in the Read-Only data cache [86, 87]. The sizes and configurations possibilities

Parameter	Pre-Fermi	Fermi	Kepler
SPs (per-SM)	8	32	192
Max. number of blocks (per-SM)	8	8	16
Max. number of threads (per-SM)	1 024	1 536	2 048
Max. number of threads (per-block)	512	1 024	1 024
L2 cache	-	768 KB	$\geq$ 512 KB
L1 cache (per-SM)	-	0/16/48 KB	0/16/32/48 KB
Size of global memory transaction	32/64/128 B	32/128 B	32/128 B
Global memory banks	8-9	5-6	4
Number of concurrent kernels	-	16	16/32

Table A.1: Summary of CUDA architecture parameters (pre-Fermi, Fermi and Kepler).

of this memory are different. The programmer can select to enable/disable the L1 cache. When the L1 cache is active, the size of the cache and local SM memory has two possible configurations in Fermi, and three in Kepler.

The size of the memory transaction segment can be adjusted. In Pre-Fermi is automatically chosen by the compiler, while in Fermi and Kepler is associated with the L1 configuration chosen. Nevertheless, this maximum size is always 128 bytes. This size is relevant for the alignment of data in memory.

The global memory is organized is several banks. The number of banks has been decreased on Fermi and Kepler. Concurrent data accesses in the same bank produce access conflicts that can affect the performance. Thus, the number of memory banks becomes important for decisions related to code optimizations, data alignment, and threadblock shape. Finally, the concurrent kernel feature is only supported by Fermi and Kepler architectures.

Appendix B

Benchmarks

Throughout this Ph.D. Thesis we have implemented and experimented with different common benchmarks. These benchmarks have been modified and adequately adapted to take profit the hardware architectures mentioned in this document.

This appendix describes the different benchmarks that we have worked, such as matrix-matrix multiplication, Cannon's algorithm, and the Single-Source Shortest Path Problem (SSSP).

B.1 Matrix-matrix multiplication

The matrix-matrix multiplication is a common lineal-algebra algorithm used in the scientific area in which this Thesis is focused. There are three different matrices (C= A $\times$ B) and each element of the C matrix, C_{ij}, is calculated through the scalar product of two arrays. The first one corresponds to the i-th row of the A matrix and the second one to the j-th column of the B matrix. This algorithm imposes a single restriction: $\#columns(Amatrix) = \#rows(Bmatrix)$.

B.1.1 Cannon's algorithm

In Cannon's algorithm [21] the available processes are organized in a perfect square topology to generate neighbor relations. Each matrix A, B and C is divided into rectangular blocks, distributing them across processes.

It starts with an initial communication stage. Matrices A and B data are realigned or reassigned so that, if there is a two-dimensional array of $P \times P$ processors, the element or submatrix A in row i and column $(j+i)modP$, $a_{i,(j+i)modP}$, and also the element or submatrix of B in row $(i+j)modP$ and column j, $b_{(i+j)modP}, i$, are assigned to processor P_{ij}. In other words, each data of row $i/(0 \leq i \leq P-1)$ of the elements or submatrices of A are transferred or shifted i times towards the left processors, and each data or column

$j/(0 \leq j \leq P-1)$ of the elements or submatrices of B are transferred or shifted j times towards upper processors.

Figure B.1 shows the initial assignment (a), and initial relocation (b), imposed by Cannon's algorithm for matrices of 3×3 elements in a 3×3 processors torus.

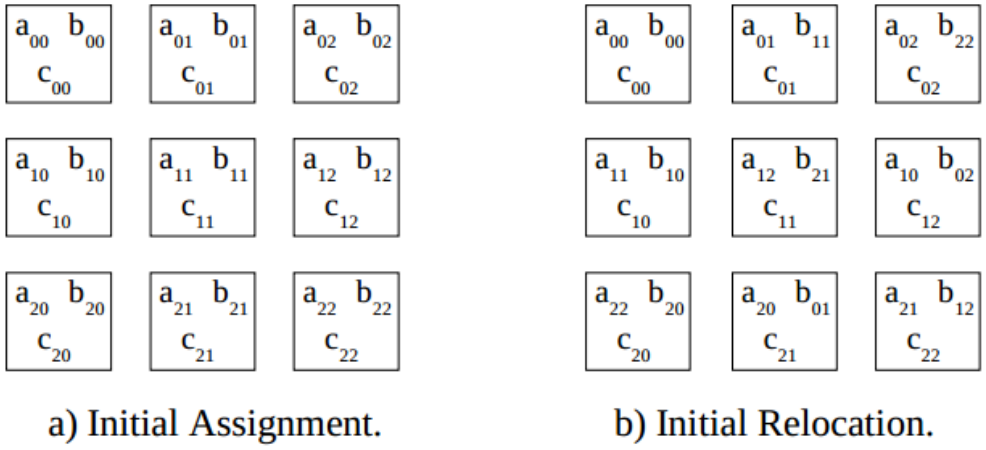


Figure B.1: 3×3 Data Location for Cannon's Algorithm.

From the initial relocation, the following steps are carried out iteratively:

- Local multiplication of data assigned in each processor for a partial result computation
- Left rotation of the elements or submatrices of A
- Upwards rotation of the elements or submatrices of B and after P of these steps, thoroughly computed values of matrix C are finally obtained

B.2 Shortest Path Problem

B.2.1 Graph Theory Notation

We will first present some graph theory concepts and notations related to the shortest-path problem. A graph $G = (V, E)$ is composed by a set of vertices V, also called nodes, and a set of edges E, also called arcs. Every vertex v is usually depicted as a point in the graph. Every edge e is usually depicted as a line that connects two and only two vertices. An edge is a tuple (u, v) that represents a link between vertices u and v. The number of edges connected to a vertex v is called the *degree* of v. In an *undirected graph* all edges can be traversed in both directions, whereas an edge (u, v) of a *directed graph* only can be traversed from u to v. There is a weight function $w(u, v)$ associated to each edge, that represents the cost of traversing the edge.

A *path* $P = \langle s, ..., u, ..., v, ..., t \rangle$ is a sequence of vertices connected by edges, from a source vertex s to a target one t. The *weight* of a path, $w(P)$, is the sum of all the weights associated to the edges involved in the path. The *shortest path* between two vertices s and t is the path with the minimum weight among all possible paths between s and t. Finally, the minimum distance between s and t, $d(s,t)$ or simply $d(t)$, is the weight of the shortest path between them. We denote $\delta(s,t)$, or simply $\delta(t)$, to a temporal tentative distance between s and t during the computation of $d(t)$.

B.2.2 Dijkstra's Algorithm

The basic solution for the Non-negative, Single-source, Shortest-Path problem (NSSP) is Dijkstra's algorithm [33]. This algorithm constructs minimal paths from a source node s to the remaining nodes, exploring adjacent nodes following a proximity criterion.

The exploring process is known as *edge relaxation*. When an edge (u, v) is relaxed from a node u, it is said that node v has been *reached*. Therefore, there is a path from source through u to reach v with a tentative shortest distance. Node v will be considered *settled* when the algorithm has found the shortest path from source node s to v. The algorithm finishes when all nodes are settled.

The algorithm uses an array, D, that stores all tentative distances found from source node s to the rest of nodes. At the beginning of the algorithm, every node is unreached and no distances are known, so $D[i] = \infty$ for all nodes i, except current source node $D[s] = 0$. Note that both reached and unreached nodes are considered unsettled nodes.

The algorithm proceeds as follows:

1. (Initialization) The algorithm starts on the source node s, initializing distance array $D[i] = \infty$ for all nodes i and $D[s] = 0$. Node s is considered as the *frontier node* f ($f \leftarrow s$) and it is settled.

2. (Edge relaxation) For every node v adjacent to f that has not been settled, a new distance from source is found using the path through f, with value $D[f] + w(f, v)$. If this distance is lower than previous value $D[v]$, then $D[v] \leftarrow D[f] + w(f, v)$.

3. (Settlement) The node u with the lowest value in D is taken as the new frontier node ($f \leftarrow u$). After this, current frontier node f is now considered as settled.

4. (Termination criteria) If all nodes have been settled the algorithm finishes. Otherwise, it proceeds to step 2.

In order to recover the path, every reached node stores its predecessor, so at the end of the query phase the algorithm just runs back from target through stored predecessors till the source node is reached. The *shortest path tree* of a graph from source node s is the composition of every shortest path from s to the remaining nodes.

B.2.3 Parallel Dijkstra

The key of the parallelization of a single sequential Dijkstra algorithm resides in the inherent parallelism of its loops. For each iteration of Dijkstra's algorithm, the *outer loop* selects a node to compute new distance labels. Inside this loop, the algorithm relaxes its outgoing edges in order to update the old distance labels, that is the *inner loop*.

Parallelizing the *outer loop* implies to compute in each iteration i a frontier set F_i of nodes that can be settled in parallel without affecting the algorithm correctness. The main problem here is to identify this set of nodes v which tentative distances $\delta(v)$ from source s must be the minimum shortest-distance $d(v)$. Some algorithms that are based on this idea are [28, 29]. Parallelizing the *inner loop* implies to traverse simultaneously the outgoing edges of the frontier node. One of the algorithm presented in [94] is an example of this kind of parallelization.

B.2.4 SSSP problem

Given a graph $G = (V, E)$, a function $w(e) : e \in E$ that associates a weight to the edges of the graph, and a source node s, it consists on computing the shortest paths from s to every node $v \in V$. If the weights of the graph range only in positive values, $w(e) \geq 0 : e \in E$, we are facing the so-called Non-negative Single-source Shortest-Path (NSSP) problem.

B.2.5 APSP problem

Given a graph $G = (V, E)$ and a function $w(e) : e \in E$ that associates a weight to the edges of the graph, it consists in computing the shortest paths for all pair of nodes $(u, v) : u, v \in V$. The APSP problem is a generalization of the classical problem of optimization, the Single-Source Shortest-Path (SSSP), that consists in computing the shortest paths from just one source node s to every node $v \in V$. If the weights of the graph range only in positive values, $w(e) \geq 0 : e \in E$, we are facing the so-called Non-negative Single-source Shortest-Path (NSSP) problem.

Bibliography

[1] Lokman A. Abbas-Turki, Stephane Vialle, Bernard Lapeyre, and Patrick Mercier. High dimensional pricing of exotic European contracts on a GPU Cluster, and comparison to a CPU cluster. In *Proceedings of the 2009 IEEE International Symposium on Parallel& Distributed Processing*, IPDPS '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[2] W. Richards Adrion. Research methodology in software engineering: Summary of the Dagstuhl workshop on future directions on software engineering. *SIGSoft Software Engineering Notes*, 18:36–37, January 1993.

[3] A.M. Aji, J. Dinan, D. Buntinas, P. Balaji, Wu chun Feng, K.R. Bisset, and R. Thakur. MPI-ACC: An Integrated and Extensible Approach to Data Movement in Accelerator-based Systems. In *Proc. HPCC-ICESS'2012*, pages 647 –654, june 2012.

[4] Ashwin M. Aji, Mayank Daga, and Wu-chun Feng. Bounding the effect of partition camping in GPU kernels. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 27:1–27:10, New York, NY, USA, 2011. ACM.

[5] S.B. Baden and S.J. Fink. The Data Mover: A Machine-Independent Abstraction for Managing Customized Data Motion. In *LCPC'99*, volume 1863 of *LNCS*, pages 333–349. Springer, 2000.

[6] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wenmei W. Hwu. An adaptive performance modeling tool for GPU architectures. *SIGPLAN Not.*, 45:105–114, January 2010.

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[8] E. Bailey, E. Barszcz, J. Barton, D. Browning, and R. Carter. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[9] J. Barceló, E. Codina, J. Casas, J. L. Ferrer, and D. García. Microscopic traffic simulation: A tool for the design, analysis and evaluation of intelligent transport systems. *J. Intell. Robot. Syst.*, 41:173–203, 2005.

[10] J. Barnat, P. Bauch, L. Brim, and M. Ceska. Employing Multiple CUDA Devices to Accelerate LTL Model Checking. In *Proc. ICPADS'2010*, pages 259 –266, dec. 2010.

[11] Muthu Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In Rajiv Gupta, editor, *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, chapter 14, pages 244–263. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010.

[12] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *Proc. ACM PPoPP'08*, pages 1–10, New York, NY, USA, 2008. ACM.

[13] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, María J. Garzarán, David Padua, and Christoph von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Proc. of the ACM SIGPLAN PPoPP*, pages 48–57, New York, New York, USA, 2006. ACM.

[14] A.P.D. Binotto, C.E. Pereira, and D.W. Fellner. Towards dynamic reconfigurable load-balancing for hybrid desktop platforms. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium*, pages 1–4, april 2010.

[15] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial Mathematics, 1987.

[16] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1 edition, 1981.

[17] TracyD. Braun, HowardJay Siegel, and AnthonyA. Maciejewski. Heterogeneous Computing: Goals, Methods, and Open Problems. In Burkhard Monien, ViktorK. Prasanna, and Sriram Vajapeyam, editors, *High Performance Computing — HiPC 2001*, volume 2228 of *Lecture Notes in Computer Science*, pages 307–318. Springer Berlin Heidelberg, 2001.

[18] J.C. Brodman, B.B. Fraguela, M.M. Garzarn, and D. Padua. New Abstractions for Data Parallel Programming. In *First USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, March 2009.

[19] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[20] Eva Burrows and Magne Haveraaen. A Hardware Independent Parallel Programming Model. *Journal of Logic and Algebraic Programming*, 78:519–538, 2009.

[21] Lynn Elliot Cannon. *A cellular computer to implement the kalman filter algorithm*. PhD thesis, Montana State University, 1969.

[22] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[23] M. Castillo, E. Chan, F.D. Igual, R. Mayo, E.S. Quintana-Ortí, G. Quintana-Ortí, R. van de Geijn, and F.G. Van Zee. Making Programming Synonymous with Programming for Linear Algebra Libraries. Tech.Rep. TR-08-20, The University of Texas at Austin, Department of Computer Sciences, Apr 2008.

[24] Joseph M. Cavanagh, Thomas E. Potok, and Xiaohui Cui. Parallel latent semantic analysis using a graphics processing unit. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, GECCO '09, pages 2505–2510, New York, NY, USA, 2009. ACM.

[25] José María Cecilia, José Manuel García, and Manuel Ujaldón. CUDA 2D stencil computations for the Jacobi method. In *Proceedings of the 10th international conference on Applied Parallel and Scientific Computing - Volume Part I*, PARA'10, pages 173–183, Berlin, Heidelberg, 2012. Springer-Verlag.

[26] Daniel Cederman and Philippas Tsigas. On Sorting and Load Balancing on GPUs. *SIGARCH Comput. Archit. News*, 36(5):11–18, June 2009.

[27] B.L. Chamberlain, S.J. Deitz, D. Iten, and S-E. Choi. User-Defined Distributions and Layouts in Chapel: Philosophy and Framework. In *2nd USENIX Workshop on Hot Topics in Parallelism*, June 2010.

[28] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of Dijkstra's shortest path algorithm. In Luboš Brim, Jozef Gruska, and Jirí Zlatuška, editors, *Mathematical Foundations of Computer Science 1998*, volume 1450 of *LNCS*, pages 722–731. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0055823.

[29] J. R. Crobak, J. W. Berry, K. Madduri, and D. A. Bader. Advanced Shortest Paths Algorithms on a Massively-Multithreaded Architecture. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, march 2007.

[30] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997.

[31] Carlos de Blas Cartón, Arturo Gonzalez-Escribano, and Diego R. Llanos. Effortless and Efficient Distributed Data-Partitioning in Linear Algebra. In *HPCC'2011*, pages 89–97. IEEE, September 2010.

[32] C.S. de la Lama, P. Toharia, J.L. Bosque, and O.D. Robles. Static Multi-device Load Balancing for OpenCL. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 675–682, july 2012.

[33] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[34] Everton Hermann, Bruno Raffin, François Faure, Thierry Gautier, and Jérémie Allard. Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. In *Europar 2010*, Ischia-Naples Italy, 09 2010.

[35] R. Farivar, A. Verma, E.M. Chan, and R.H. Campbell. MITHRA: Multiple data independent tasks on a heterogeneous resource architecture. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1 –10, 31 2009-sept. 4 2009.

[36] Naila Farooqui, Andrew Kerr, Gregory Diamos, S. Yalamanchili, and K. Schwan. A framework for dynamically instrumenting GPU compute applications within GPU Ocelot. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 9:1–9:9, New York, NY, USA, 2011. ACM.

[37] Naila Farooqui, Andrew Kerr, Gregory Frederick Diamos, Sudhakar Yalamanchili, and Karsten Schwan. A framework for dynamically instrumenting GPU compute applications within GPU Ocelot. In *GPGPU*, page 9, 2011.

[38] Vincent Favre-Nicolin, Johann Coraux, Marie-Ingrid Richard, and Hubert Renevier. Fast computation of scattering maps of nanostructures using graphical processing units. *Journal of Applied Crystallography*, 44(3):635–640, Jun 2011.

[39] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345–345, June 1962.

[40] Javier Fresno, Arturo Gonzalez-Escribano, and Diego R. Llanos. Automatic Data Partitioning Applied to Multigrid PDE Solvers. In *PDP'2011*, pages 239–246. IEEE, February 2011.

[41] Javier Fresno, Arturo Gonzalez-Escribano, and Diego R. Llanos. Extending a hierarchical tiling arrays library to support sparse data partitioning. *The Journal of Supercomputing*, 2012. Online-first version available.

[42] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel Computing Experiences with CUDA. *IEEE Micro*, 28(4):13–27, 2008.

[43] Arturo Gonzalez-Escribano and Diego R. Llanos. Trasgo: A Nested-parallel Programming System. *J. Supercomput.*, 58(2):226–234, November 2011.

[44] Arturo Gonzalez-Escribano and Diego R. Llanos. Trasgo: A Nested-Parallel Programming System. *The Journal of Supercomputing*, 58(2):226–234, 2011.

[45] Arturo Gonzalez-Escribano, Yuri Torres, Javier Fresno, and Diego R. Llanos. An Extensible System for Multilevel Automatic Data Partition and Mapping. *Parallel and Distributed Systems, IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–1, 2013.

[46] Oded Green, Robert McColl, and David A. Bader. GPU merge path: a GPU merging algorithm. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 331–340, New York, NY, USA, 2012. ACM.

[47] Paulius Micikevicius Greg Ruetsch. NVIDIA Optimizing Matrix Transpose in CUDA. http://developer.download.nvidia.com/compute/cuda/3_0/sdk/website/CUDA/website/C/src/transposeNew/doc/MatrixTranspose.pdf, June 2010. Last visit: Dec 2, 2013.

[48] Max Grossman, Alina Simion Sbîrlea, Zoran Budimlić, and Vivek Sarkar. CnC-CUDA: Declarative programming for GPUs. In *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, LCPC'10, pages 230–245, Berlin, Heidelberg, 2011. Springer-Verlag.

[49] Jia Guo, Ganesh Bikshandi, Basilio B. Fraguela, Maria J. Garzaran, and David Padua. Programming with tiles. In *Proceedings of the ACM SIGPLAN PPoPP*, pages 111–122, Salt Lake City, UT, USA, 2008. ACM.

[50] Ziyu Guo, Bo Wu, and Xipeng Shen. One stone two birds: Synchronization relaxation and redundancy removal in GPU-CPU translation. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 25–36, New York, NY, USA, 2012. ACM.

[51] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[52] Mark Harris. Optimizing Parallel Reduction in CUDA. Technical report, nVidia, 2008.

[53] Hector Ortega-Arranz, Yuri Torres, Arturo Gonzalez-Escribano, Diego R. Llanos. *High-Performance Computing on Complex Environments, ComplexHPC 2013*, chapter The All-Pair Shortest-Path Problem in Shared-Memory Heterogeneous Systems. Series on Parallel and Distributed Computing. Addison-Wesley, 2013.

[54] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. MapCG: Writing parallel program portable between CPU and GPU. In *PACT'2010*, PACT '10, pages 217–226, New York, NY, USA, 2010. ACM.

[55] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th International Sym-*

posium on Computer Architecture, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[56] Dominique Houzet, Sylvain Huet, and Anis Rahman. SysCellC: A data-flow programming model on multi-GPU. *Procedia Computer Science*, 1(1):1035–1044, 2010. ICCS 2010.

[57] Edward S. Jimenez, Laurel J. Orr, and Kyle R. Thompson. An Irregular Approach to Large-Scale Computed Tomography on Multiple Graphics Processors Improves Voxel Processing Throughput. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 254–260, Washington, DC, USA, 2012. IEEE Computer Society.

[58] N.P. Karunadasa and D.N. Ranasinghe. Accelerating high performance applications with CUDA and MPI. In *ICIIS'2009*, pages 331 –336, dec. 2009.

[59] Andrew Kerr, Gregory Diamos, and Sudakhar Yalamanchili. Modeling GPU-CPU Workloads and Systems. In *Third Workshop on General-Purpose Computation on Graphics Processing Units*, Pittsburg, Pennsylvania, USA, 4 2010.

[60] Khronos. Open Computing Language (OpenCL), 2010. http://www.khronos.org/opencl/, Last visit: December 2, 2013.

[61] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[62] Stephen M. Kofsky, Daniel R. Johnson, John A. Stratton, Wen-Mei W. Hwu, Sanjay J. Patel, and Steven S. Lumetta. Implementing a GPU Programming Model on a non-GPU Accelerator Architecture. In *A4MMC 2010 - 1st Workshop on Applications for Multi and Many Core Processors*, Saint Malo France, 2010.

[63] Akintola Kolawole and Alireza Tavakkoli. Robust foreground detection in videos using adaptive color histogram thresholding and shadow removal. In *Proceedings of the 7th international conference on Advances in visual computing - Volume Part II*, ISVC'11, pages 496–505, Berlin, Heidelberg, 2011. Springer-Verlag.

[64] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *Proceedings of the ACM SIGPLAN PLDI*, pages 235–244, San Diego, California, USA, 2007. ACM.

[65] Qing kui Chen and Jia kang Zhang. A Stream Processor Cluster Architecture Model with the Hybrid Technology of MPI and CUDA. In *ICISE'2009*, pages 86 –89, dec. 2009.

[66] Alan Leung, Ondřej Lhoták, and Ghulam Lashari. Automatic parallelization for graphics processing units. *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java PPPJ 09*, page 91, 2009.

[67] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Manikandan Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2010*, volume 425 of *ACM International Conference Proceeding Series*, pages 51–61, Pittsburgh, Pennsylvania, March 2010.

[68] Yuping Lin and Gérard Medioni. Mutual information computation and maximization using GPU. *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 0:1–6, 2008.

[69] D.B. Loveman. High Performance Fortran. *Parallel & Distributed Technology: Systems & Applications*, 1(1):25–42, Feb 1993.

[70] Rama Malladi, Richard Dodson, and Vyacheslav Kitaeff. Intel many integrated core (MIC) architecture: Portability and performance efficiency study of radio astronomy algorithms. In *Proceedings of the 2012 workshop on High-Performance Computing for Astronomy Date*, pages 5–6, New York, NY, USA, 2012. ACM.

[71] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. In *ACM SIGGRAPH 2003 Papers*, pages 896–907, San Diego, California, 2003. ACM.

[72] P. Martín, R. Torres, and A. Gavilanes. CUDA Solutions for the SSSP Problem. In Gabrielle Allen, Jaroslaw Nabrzyski, Edward Seidel, Geert van Albada, Jack Dongarra, and Peter Sloot, editors, *Computational Science – ICCS 2009*, volume 5544 of *LNCS*, pages 904–913. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-01970-8_91.

[73] T.J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[74] Richard Membarth, Frank Hannig, Jurgen Teich, and Harald Kostler. Towards Domain-Specific Computing for Stencil Codes in HPC. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 1133–1138, Washington, DC, USA, 2012. IEEE Computer Society.

[75] Julian Francis Miller and Simon Harding. Cartesian genetic programming. In *Proceedings of the 2008 GECCO conference companion on Genetic and evolutionary computation*, GECCO '08, pages 2701–2726, New York, NY, USA, 2008. ACM.

[76] V. K. Murthy and E. V. Krishnamurthy. Heterogeneous programming with concurrent objects. In *Proceedings of the 1997 ACM symposium on Applied computing*, pages 454–463, San Jose, California, United States, 1997. ACM.

[77] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An Improved Magma Gemm For Fermi Graphics Processing Units. *Int. J. High Perform. Comput. Appl.*, 24:511–515, November 2010.

[78] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, March 2008.

[79] NVIDIA. Occupancy calculator spreadsheet. http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls, Last visit: Jun 2013.

[80] NVIDIA. Optimizing Matrix Transpose in CUDA. http://docs.nvidia.com/cuda/samples/6_Advanced/transpose/doc/MatrixTranspose.pdf, Last visit: Jun 2013.

[81] NVIDIA. Fermi home page, 2010. http://www.nvidia.es/object/fermi_architecture_es.html, Last visit: Nov, 2013.

[82] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture Programming Guide Version 2.0, 2010. http://docs.nvidia.com/cuda/, Last visit: Nov, 2013.

[83] NVIDIA. NVIDIA CUDA Programming Guide 3.0 (Fermi), 2010. https://developer.nvidia.com/cuda-toolkit-30-downloads, Last visit: Nov, 2013.

[84] NVIDIA. Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2010. http://www.nvidia.com, Last visit: Nov, 2013.

[85] NVIDIA. NVIDIA CUDA Programming Guide 4.2: Kepler, 2012. https://developer.nvidia.com/cuda-toolkit-42-archive, Last visit: Nov, 2013.

[86] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110 architecture, 2012. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, Last visit: Jun 2013.

[87] NVIDIA. CUDA C Best Practices Guide: Kepler, 2013. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/, Last visit: Nov, 2013.

[88] NVIDIA. *CUDA CUBLAS Library*, 2013. http://www.nvidia.com/object/fermi_architecture.html, Last visit: Mar, 2013.

[89] NVIDIA. Cuda: Visual Profiel documentation, 2013. http://docs.nvidia.com/cuda/profiler-users-guide/, Last visit: Nov, 2013.

[90] NVIDIA. The CUDA Compiler Driver NVCC, 2013. http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/, Last visit: December 2, 2013.

[91] Lars Nyland, Mark Harris, and Jan Prins. Fast N-Body Simulation with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 31. Addison Wesley Professional, August 2007.

[92] Hector Ortega-Arranz, Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. A New GPU-based Approach to the Shortest Path Problem. In *The 2013 International Conference on High Performance Computing & Simulation, (HPCS 2013)*, pages 505–511, 2013.

[93] Hector Ortega-Arranz, Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. A Tuned, Concurrent Multi-Kenel Approach to the APSP problem. In *The 13th International Conference Computational and Mathematical Methods in Science and Engineering, CMMSE 2013*, 2013.

[94] Marios Papaefthymiou and Joseph Rodrigue. Implementing Parallel Shortest-Paths Algorithms. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 59–68, 1994.

[95] Meng Qi, Thanh-Tung Cao, and Tiow-Seng Tan. Computing 2D constrained Delaunay triangulation using the GPU. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 39–46, New York, NY, USA, 2012. ACM.

[96] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. In *PPoPP'2009*, PPoPP '09, pages 121–130, New York, NY, USA, 2009. ACM.

[97] J. Rose and G. Steele. C*: An Extended C Language for Data Parallel Programming. In *ACM Proceedings of the International Conference on Supercomputing*, pages 2–16. ACM, 1987.

[98] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, John A. Stratton, Sain-Zee Ueng, Sara S. Baghsorkhi, and Wen-Mei W. Hwu. Program optimization carving for GPU computing. *Journal of Parallel and Distributed Computing*, 68(10):1389–1401, October 2008.

[99] Peter Sanders, Dominik Schultes, and Christian Vetter. Mobile Route Planning. In *ESA'08*, pages 732–743, Berlin, 2008. Springer.

[100] Nadathur Rajagopalan Satish. *Compile Time Task and Resource Allocation of Concurrent Applications to Multiprocessor Systems*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2009.

[101] Dana Schaa. Modeling execution and predicting performance in multi-GPU environments. In *Electrical and Computer Engineering Master's Theses*, Boston, Mass, 2009. Department of Electrical and Computer Engineering, Northeastern University.

[102] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[103] Dhirendra Pratap Singh and Nilay Khare. A Study of Different Parallel Implementations of Single Source Shortest Path Algorithms. *International Journal of Computer Applications*, 54(10):26–30, September 2012. Published by Foundation of Computer Science, New York, USA.

[104] Satnam Singh. Computing without processors. *Commun. ACM*, 54:46–54, August 2011.

[105] James Smaldon, Natalio Krasnogor, Cameron Alexander, and Marian Gheorghe. Liposome logic. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, pages 161–168, New York, NY, USA, 2009. ACM.

[106] Fengguang Song and Jack Dongarra. A scalable framework for heterogeneous GPU-based clusters. In *Proc. ACM SPAA'2012*, pages 91–100, New York, NY, USA, 2012. ACM.

[107] J.E. Stone, D. Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering*, 12(3):66 –73, may 2010.

[108] John A. Stratton, Sam S. Stone, and Wen-Mei W. Hwu. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In José Nelson Amaral, editor, *LCPC'2008*, pages 16–30, Berlin, Heidelberg, 2008. Springer-Verlag.

[109] R. Taylor and Xiaoming Li. A Micro-benchmark Suite for AMD GPUs. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 387 –396, sept. 2010.

[110] Top500. Top500 home page, 2013. http://www.top500.org/, Last visit: October 2, 2013.

[111] Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Automatic Data Layout at Multiple Levels for CUDA. In *The 10th International Conference Computational and Mathematical Methods in Science and Engineering, CMMSE 2010*, 2010.

[112] Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. CUDA Tuning and Configuration Parameters on Fermi Architectures. In *Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES 2011)*, 2011.

[113] Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Understanding the impact of CUDA tuning techniques for Fermi. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 631–639, 2011.

[114] Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Uso del conocimiento de la arquitectura Fermi para mejorar el rendimiento en aplicaciones CUDA. In *Actas XXII Jornadas de Paralelismo*, 2011.

[115] Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Encapsulated Synchronization and Load-Balance in Heterogeneous Programming. In *Euro-Par 2012 Parallel Processing*, volume 7484 of *LNCS*, pages 502–513. Springer Berlin Heidelberg, 2012.

[116] Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Measuring the Impact of Configuration Parameters in CUDA Through Benchmarking. In *The 12th International Conference Computational and Mathematical Methods in Science and Engineering, CMMSE 2012*, 2012.

[117] Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. uBench: Performance Impact of CUDA Block Geometry. Technical Report IT-DI-2012-0001, Depto. Informatica, Universidad de Valladolid, Dec 2012. http://www.infor.uva.es/investigacion/publicaciones.html.

[118] Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Using Fermi architecture knowledge to speed up CUDA and OpenCL programs. In *Proc. ISPA'12*, Leganes, Madrid, Spain, 2012.

[119] Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Data partition and synchronisation in heterogeneous systems. In *HPC-EUROPA2 project (project number: 228398) with the support of the European Commission - Capacities Area - Research Infrastructures*, 2013. http://www.hpc-europa.org/files/2012/ICT_0688_TORRES%20DE%20LA%20SIERRA%20Yuri.pdf.

[120] Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. uBench: exposing the impact of CUDA block geometry in terms of performance. *The Journal of Supercomputing*, 65(3):1150–1163, 2013.

[121] Christian Trott and Lars Winterfeld. General purpose Molecular Dynamics Simulations on GPUs: Issues of Pair Forces and Scaling to large Clusters. *CoRR*, abs/1009.4330, 2010.

[122] Stanley Tzeng, Anjul Patney, and John D. Owens. Task Management for Irregular-Parallel Workloads on the GPU. In Michael Doggett, Samuli Laine, and Warren Hunt, editors, *High Performance Graphics*, pages 29–37. Eurographics Association, 2010.

[123] S. Warshall. A Theorem on Boolean Matrices. *Journal of the ACM*, 9(1):11–12, January 1962.

[124] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proc. of the ACM SIGPLAN PLDI*, pages 30–44, Toronto, Ontario, Canada, 1991. ACM.

[125] M. Wolfe. More Iteration Space Tiling. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, Supercomputing '89, pages 655–664, New York, NY, USA, 1989. ACM.

[126] Michael Wolfe. Implementing the PGI Accelerator Model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 43–50, New York, NY, USA, 2010. ACM.

[127] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 235 –246, march 2010.

[128] Erik Wynters. Parallel processing on NVIDIA graphics processing units using CUDA. *J. Comput. Small Coll.*, 26:58–66, January 2011.

[129] Changyou Zhang Xiang Cui, Yifeng Chen and Hong Mei. Auto-tuning Dense Matrix Multiplication for GPGPU with Cache. In *Proceedings of 16th international conference on parallel and distibuted systems, 2010*, ICPADS international conference on parallel and distibuted systems, Shanghai, China, December 2010.

[130] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. An optimizing compiler for GPGPU programs with input-data sharing. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '10*, page 343, Bangalore, India, 2010.

[131] Ping Yao, Hong An, Mu Xu, Gu Liu, Xiaoqiang Li, Yaobin Wang, and Wenting Han. CuHMMer: A load-balanced CPU-GPU cooperative bioinformatics application. In *HPCS'2010*, pages 24–30, July 2010.

[132] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 369–380, New York, NY, USA, 2011. ACM.

[133] Yao Zhang and J.D. Owens. A quantitative performance analysis model for GPU architectures. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 382 –393, feb. 2011.

[134] Simon F. Portegies Zwart, Robert G. Belleman, and Peter M. Geldof. High-performance direct gravitational N-body simulations on graphics processing units. *New Astronomy*, 12(8):641–650, November 2007.