PhD Thesis

Parallel Prefix Operations on Heterogeneous Platforms

Adrián Pérez Diéguez

2018

UNIVERSIDADE DA CORUÑA

Parallel Prefix Operations on Heterogeneous Platforms

Adrián Pérez Diéguez

PhD Thesis

October 2018

PhD Advisors:

Margarita Amor López

Ramón Doallo Biempica

PhD Program in Information Technology Research

UNIVERSIDADE DA CORUÑA

Dra. Margarita Amor López
Profesora Titular de Universidade
Dpto. de Enxeñaría de Computadores
Universidade da Coruña

Dr. Ramón Doallo Biempica
Catedrático de Universidade
Dpto. de Enxeñaría de Computadores
Universidade da Coruña

CERTIFICAN

Que a memoria titulada "*Parallel Prefix Operations on Heterogeneous Platforms*" foi realizada por D. Adrián Pérez Diéguez baixo a nosa dirección no Departamento de Enxeñaría de Computadores da Universidade da Coruña, e conclúe a Tese de Doutoramento que presenta para a obtención do título de Doutor en Enxeñaría Informática pola Universidade de Coruña coa Mención de Doutor Internacional.

En A Coruña, a de de 2018

Asdo.: Margarita Amor López
Directora da Tese de Doutoramento

Asdo.: Ramón Doallo Biempica
Director da Tese de Doutoramento

Asdo.: Adrián Pérez Diéguez
Autor da Tese de Doutoramento

Aos que me apoiastes nesta andaina,
mais sobre todo aos meus pais.

Agradecementos

En primeiro lugar, quixera agradecerlle aos meus directores, Marga e Ramón, a oportunidade que me deron ao confiar en mín, a súa dedicación e prezada axuda, pero especialmente a paciencia acadada conmigo. Foi unha honra ter traballado ao voso lado todos estes anos, unha etapa da miña vida que non esquecerei endexamais. Aínda que tamén me gustaría estender este recoñecemento a todos os profesores do Grupo de Arquitectura de Computadores.

Tamén quixera mostrar o meu agradecemento, dun xeito especial, aos meus compañeiros de laboratorio, aos que seguen nel e aos que xa marcharon. Unha relación que sen dúbida vai moito máis alá do académico, e que agardo que se manteña por moitos anos. Foi unha ledicia compartir todos eses momentos con vós. En particular, sería inxusto non destacar a labor de Jacobo nesta tese, o meu mentor e salvador asemade nos primeiros pasos desta aventura.

Pero se alguén debe ser nomeado, eses son os meus pais. Sacrificaron todo para darme a oportunidade de estar eu aquí. Non teño palabras para vós, moitas grazas de corazón. Tamén ao meu irmán e a toda a miña familia. Aos meus compañeiros de piso destes anos, aos meus amigos de Pontevedra, de Coruña e aos que están espallados por ahí adiante, aos que fixen no mundo do deporte, da cultura e das reivindicacións, non me esquezo de vós. A todos eles, envíolles hoxe unha aperta.

Of course, I do not want to forget the japanese people. I gratefully thank Dr. Satoshi Matsuoka and Dr. Akira Nukada for hosting and advising me during my visit to the Tokyo Institute of Technology, Japan, and also the Titech people, but especially Sergio, Shweta, Kevin, Pak, Xu and Artur. *Arigato gozaimasu.*

Finalmente, gustaríame estender o meu agradecemento a todas as entidades que financiaron este traballo. Dunha parte, ás redes estatais de investigación CAPAP-H4 e CAPAP-H5, ás redes europeas NESUS IC1305 e HiPEAC, á Xunta de Galicia (ref. GRC2013/055, ED431C 2017/04, R2014/041, ED431D R2016/05, ED431G/01 e a axuda predoutoral ED481A-2015/230) e ao Goberno de España (ref. TIN2013-42148-P, TIN2016-75845-P, axuda predoutoral FPU14/02801 e de estadía EST16/00579). Así mesmo, ao Grupo de Arquitectura de Computadores, ao Departamento de Enxeñaría de Computadores e á Universidade da Coruña. Tamén, á empresa Inditex polo financiamento dunha estadía.

Adrián Pérez Diéguez

Nothing has such power to broaden the mind as the ability to investigate systematically and truly all that comes under thy observation in life.

(Nada ten tanto poder para ampliar a mente coma a capacidade de investigar de xeito sistemático e real todo o que é susceptible de observación na vida)

Marco Aurelio

Resumo

As tarxetas gráficas, coñecidas como GPUs, aportan grandes vantaxes no rendemento computacional e na eficiencia enerxética, sendo un piar clave para a computación de altas prestacións (HPC). Sen embargo, esta tecnoloxía tamén é custosa de programar, e ten certos problemas asociados á portabilidade entre as diferentes tarxetas. Por outra banda, os algoritmos de prefixo paralelo son un conxunto de algoritmos paralelos regulares e moi empregados nas ciencias compuacionais, cuxa eficiencia é esencial en moitas aplicacións. Neste eido, aínda que as GPUs poden acelerar a computación destes algoritmos, tamén poden ser unha limitación cando non explotan axeitadamente o paralelismo da arquitectura GPU.

Esta Tese presenta dúas perspectivas. Dunha parte, deséñanse novos algoritmos de prefixo paralelo para calquera paradigma de programación paralela. Pola outra banda, tamén se propón unha metodoloxía xeral que implementa eficientemente algoritmos de prefixo paralelos, de xeito doado e portable, sobre arquitecturas GPU *CUDA*, mais que se centrar nun algoritmo particular ou nun modelo concreto de tarxeta. Para isto, a metodoloxía identifica os paramétros da GPU que inflúen no rendemento e, despois, seguindo unha serie de premisas teóricas, obtéñense os valores óptimos destes parámetros dependendo do algoritmo, do tamaño do problema e da arquitectura GPU empregada. Ademais, esta Tese tamén prové unha serie de funcións GPU compostas de bloques de código *CUDA* modulares e reutilizables, o que permite a implementación de calquera algoritmo de xeito sinxelo. Segundo o tamaño do problema, propóñense tres aproximacións. As dúas primeiras resolven problemas pequenos, medios e grandes nunha única GPU, mentras que a terceira trata con tamaños extremadamente grandes, usando varias GPUs.

As nosas propostas proporcionan uns resultados moi competitivos a nivel de rendemento, mellorando as propostas existentes na bibliografía para as operacións probadas: a primitiva *scan*, ordenación e a resolución de sistemas tridiagonais.

Resumen

Las tarjetas gráficas (GPUs) han demostrado grandes ventajas en el rendimiento computacional y en la eficiencia energética, siendo una tecnología clave para la computación de altas prestaciones (HPC). Sin embargo, esta tecnología también es costosa de programar, y tiene ciertos problemas asociados a la portabilidad de sus códigos entre diferentes generaciones de tarjetas. Por otra parte, los algoritmos de prefijo paralelo son un conjunto de algoritmos regulares y muy utilizados en las ciencias computacionales, cuya eficiencia es crucial en muchas aplicaciones. Aunque las GPUs puedan acelerar la computación de estos algoritmos, también pueden ser una limitación si no explotan correctamente el paralelismo de la arquitectura GPU.

Esta Tesis presenta dos perspectivas. De un lado, se han diseñado nuevos algoritmos de prefijo paralelo que pueden ser implementados en cualquier paradigma de programación paralela. Por otra parte, se propone una metodología general que implementa eficientemente algoritmos de prefijo paralelo, de forma sencilla y portable, sobre cualquier arquitectura GPU *CUDA*, sin centrarse en un algoritmo particular o en un modelo de tarjeta. Para ello, la metodología identifica los parámetros GPU que influyen en el rendimiento y, siguiendo un conjunto de premisas teóricas, obtiene los valores óptimos para cada algoritmo, tamaño de problema y arquitectura. Además, las funciones GPU proporcionadas están compuestas de bloques de código *CUDA* reutilizable y modular, lo que permite la implementación de cualquier algoritmo de prefijo paralelo sencillamente. Dependiendo del tamaño del problema, se proponen tres aproximaciones. Las dos primeras resuelven tamaños pequeños, medios y grandes, utilizando para ello una única GPU; mientras que la tercera aproximación trata con tamaños extremadamente grandes, usando varias GPUs.

Nuestras propuestas proporcionan resultados muy competitivos, mejorando el rendimiento de las propuestas existentes en la bibliografía para las operaciones probadas: la primitiva *scan*, ordenación y la resolución de sistemas tridiagonales.

Abstract

Graphics Processing Units (GPUs) have shown remarkable advantages in computing performance and energy efficiency, representing one of the most promising trends for the near-future of high performance computing. However, these devices also bring some programming complexities, and many efforts are required to provide portability between different generations. Additionally, parallel prefix algorithms are a set of regular and highly-used parallel algorithms, whose efficiency is crutial in many computer science applications. Although GPUs can accelerate the computation of such algorithms, they can also be a limitation when they do not match correctly to the GPU architecture or do not exploit the GPU parallelism properly.

This dissertation presents two different perspectives. On the one hand, new parallel prefix algorithms have been algorithmically designed for any parallel programming paradigm. On the other hand, a general tuning GPU methodology is proposed to provide an easy and portable mechanism to efficiently implement parallel prefix algorithms on any CUDA GPU architecture, rather than focusing on a particular algorithm or a GPU model. To accomplish this goal, the methodology identifies the GPU parameters which influence on the performance and, following a set of performance premises, obtains the convinient values of these parameters depending on the algorithm, the problem size and the GPU architecture. Additionally, the provided GPU functions are composed of modular and reusable CUDA blocks of code, which allow the easy implementation of any parallel prefix algorithm. Depending on the size of the dataset, three different approaches are proposed. The first two approaches solve small and medium-large datasets on a single GPU; whereas the third approach deals with extremely large datasets on a Multiple-GPU environment.

Our proposals provide very competitive performance, outperforming the state-of-the-art for many parallel prefix operations, such as the scan primitive, sorting and solving tridiagonal systems.

Preface

Introduction and Motivation

In recent years, GPUs (*Graphics Processing Units*) have experienced a noticeable increase in their relevance and usage in high performance computing, since they can perform much faster than regular CPUs (*Central Processing Units*). *Parallel computing* is a form of computation in which many calculations are performed simultaneously. Parallel computing involves different perspectives, being this work mainly focused on: *computer architecture* and *parallel programming*. *Computer architecture* (hardware aspect) refers to support parallelism at architectural level, whereas *parallel programming* (software aspect) focuses on fully using the computational power of the target architecture.

From a computer architecture perspective, modern GPUs can execute up to a thousand of physical threads per device, which are optimized for intensive arithmetic operations, performing especially well in regular algorithms with reduced flow control, and better hiding the execution latencies owing to overlap computation and communication. This overlapping is possible thanks to assign a certain number of logical threads to each core, reducing idle cycles through multi-threading.

From a programmability point of view, the CPU programmability has much more advantages. First, there are many *APIs* to facilitate the parallel adaptation from a serial code to a parallel approach, such as *OpenMP* [21], and parallel programming libraries like *MPI* [47]. Furthermore, there is a huge and experienced community behind the programming languages focused on a CPU, such as *C++*, *Python* and *Java*, providing easy and powerful tools for software development, profiling and debugging on these languages. In contrast, most of the high level *GPU*

languages are quite recent; thus, specialized developing tools, *APIs* and libraries are scarce. Additionally, this novel GPU capability is limited by the overall complexity of hardware and typical workloads. Programmers have to choose suitable parallel algorithms for these architectures that also require special languages such as *CUDA* [93] or *OpenCL* [68]; and also have to fully understand the hardware and the problem, considering optimization techniques to fully exploit the GPU resources and achieve the said performance.

There are several proposals in order to facilitate the programmability of these architectures: *Autotuning* [38], *directives* [131], *automatic compilers* [2] or *accelerated libraries* [44]. *Autotuning* [38] [73] is a very interesting option for applications whose execution time, memory usage or energy consumption can vary depending on a set of parameters and their execution environment. The autotuner determines the best parameter combination to maximise an user-defined metric. Nevertheless, this technique requires writing code in a parametrized way to accommodate various performance tuning parameters. Another approach is the *use of directives* such as *OpenACC* [131] or *hiCuda* [55]. Most of this kind of libraries require to have GPU expertise. Furthermore, the code is not easily readable and there are also some limits, for example, the programmer cannot use CUDA intrinsic functions within the accelerator region. *Automatic compilers* are another interesting option that automatically generate code for GPUs, such as *Par4all* [2] and *Bones* [88], saving time and effort to programmers. However, these approaches sometimes rely on the user knowledge for tuning applications. In addition, some systematic code translations, without a previous analysis of the problem, can lead to reduce performance. Finally, the use of *accelerated tuned libraries* for each architecture version, such as *SkePU* [44], *MAGMA* [64] or *SkelCL* [118], can enable applications to fully exploit the power of current heterogeneus parallel systems. Due to the fast GPU market evolution, each GPU architecture version highly vary its desing from one generation to another, and the parameters which influence on performance also change and must be re-adjusted.

This Thesis is primarily interested on the forth approach, tuning an *accelerated library*, as it provides an optimal implementation independently of the target architecture, providing generality and usability, and being transparent to the user.

On the one hand, a parallel computation is *work efficient* when it does not perform more work than its sequential version; in other words, both versions have the same complexity. An approach that is work efficient follows a *work and depth based model*. On the other hand, a *processor based model* takes into account computing costs, such as execution times, the number of processors, synchronization barriers or communications costs of the implementation, trying to minimize the execution time of the algorithm, but not the work efficiency. In this work, the complexity of the algorithms is not analyzed; only final execution times are considered, following a *processor based* model. On the other hand, the complexity of *GPU* hardware is reflected in the diversity of its performance models, and it is not easy predict how long an implementation takes or suggest a precise formula to find out its optimal GPU parameters. In [62], a very complex model is presented, using more than 20 equations; whereas a model formulation based on graphs was presented in [5]. Additionally, other proposals can be found in [1] and [22].

Objectives and Work Methodology

The aim of this Thesis is to propose a *GPU* performance parameter tuning methodology to predict the best GPU parameter configuration that influences on the performance for each *GPU* architecture generation, especially focused on a set of regular and highly-used parallel algorithms, called *pararell prefix algorithms* [75] [76], as well as designing and developing new pararell prefix algorithms that match well to the parallel paradigm. These algorithms are regular algorithms whose communication pattern does not depend on execution values, as it is given by a linear function which is well suited to GPU architectures. Furthermore, each resulting element is a combination of previous results from other elements with common calculations that can be reused. Therefore, using the proposed methodology is posible to efficiently parallelize and solve several frequently used operations: the Fast Fourier Transform (FFT), the scan primitive, tridiagonal system solvers (TS) or sorting. Regarding this methodology, these problems can be classified depending on their size:

- *Small datasets.* The problem data fit in the GPU scratchpad memory (also known as *shared-memory* in CUDA architectures).

- *Medium and Large datasets.* The problem size is bigger than the scrathpad memory but still fits into the device memory of a single GPU.
- *Extremely Large datasets.* The problem size is bigger than the device memory of a single GPU, and the dataset is distributed among several GPUs.

To accomplish our goal, this research work has evolved across these three stages, progressively developing an incremental methodology for each kind of dataset and adapted to different CUDA architectures. Firstly, a tuning methodology was proposed for small datasets, providing an efficient implementation for both tridiagonal systems, the scan primitive and a sorting algorithm. After this, we have increased the methodology to support medium and large datasets, implementing different scan and tridiagonal system solver approaches under this methodology. Finally, the methodology was extended to extremely large datasets, introducing several GPUs and computing nodes in the design, providing an efficient proposal for the scan primitive. It should be observed that the scope of this work is limited to CUDA GPUs, as it is the leading programming model and pioneer for general-purpose computing on GPUs, but the proposals of this Thesis could be applied to other frameworks, as OpenCL, as long as similar hardware architecture is used. In addition to the tuning methodology, this Thesis also provides three new parallel prefix algorithms, two for solving tridiagonal systems and one for sorting. These algorithms were designed from a algorithmical perspective to match well to any parallel paradigm, demonstrating their efficiency on GPUs.

In [8], a dissertation about tuning GPU performance parameters for *Index-Digit* algorithms and small datasets is presented. In contrast to that text, this work extends the methodology for *parallel prefix* algorithms, a superset that also includes *Index-Digit* algorithms, as will be explained later, as well as supporting medium, large and extremely large datasets.

Main Contributions of the Thesis

The main contributions of this Thesis are the following:

- A literature review about the most employed parallel prefix algorithms and recent Graphic Processing Units (GPUs).
- Design, development and algorithmic formulation of new parallel prefix algorithms.
- Development of a general tuning methodology for parallel prefix algorithms and Index-Digit algorithms on different GPU systems.
- Experimental analysis of the proposed methodology for several parallel prefix operations.
- Provide an accelerated GPU library that outperforms the state-of-the-art for the corresponding operations.
- Thorough peformance evaluation of the library using real-world applications.

Structure of the Thesis

This Thesis is organized as follows:

- Chapter 1 introduces the Graphcis Processing Units (GPUs) and describes the basics of CUDA programming and its execution model. Additionally, it also summarizes the GPU architectures employed in this Thesis.
- Chapter 2 defines both the parallel prefix algorithms and a subset of them called Index-Digit algorithms. Specifically, the following algorithms are analyzed in this chapter: Tridiagonal system solvers, scan primitive and sorting algortihms.
- Chapter 3 presents the new parallel prefix algorithms designed and developed in this Thesis. These new algorithms are algorithmically formulated in this

chapter, and also a hand-tuned GPU implementation is provided for some of them. Concretely, two new algorithms are created to solve tridiagonal systems, and a new algorithm for sorting is also proposed.

- Chapter 4 addresses the development of a general GPU tuning methodology for both parallel prefix algorithms and Index-Digit algorithms, considering datasets that fit in the shared memory a of CUDA GPU, and providing an accelerated library with the corresponding implementations. The proposed methodology is analyzed against the state-of-the-art, and the experimental results are presented.

- Chapter 5 conducts the extension of the previous methodology to larger datasets which do not fit in the shared memory of a GPU but still can be stored in the device memory of a single GPU. The methodology is also tested for well-known operations, surpassing the state-of-the-art on different GPU architectures.

- Chapter 6 extends the proposed methodology for extremely large datasets which cannot be stored in a single GPU, needing a Multiple-GPU system. The resulting library based on the methodology is tested for tridiagonal systems and the scan primitive in different Multiple-GPU environments, analyzing the experimental results of their execution.

- Chapter 7 analyzes the efficiency of the final library built on this Thesis in real-world applications. Specifically, the multiplication of high-precision integers, which is used in many computer science fields, such as cryptography, is tested using our proposal.

- Chapter 8 extracts the conclusions of the Thesis and presents the future work.

Funding and Technical Means

The following means and funding have been used to carry out this Thesis:

- Working material, as well as human and financial support provided by the Computer Architecture Group (GAC) of the University of A Coruña.

- Fellowships funded by the Ministry of Education, Culture and Sport of Spain (FPU program, ref. FPU14/02801) and by the Galician Government (Xunta de Galicia, ref. ED481A-2015/230).
- Access to bibliographical material through the library of University of A Coruña.
- Additional funding through the following research projects:
  - European funding: "Network for Sustainable Ultrascale Computing" (NESUS COST Action ref. IC1305), "High-Performance Embedded Architecture and Compilation Network of Excellence" (HiPEAC3 NoE).
  - State funding by the Ministry of Economy and Competitiveness of Spain through the projects "New Challenges in High Performance Computing: from Architectures to Applications" (refs. TIN2013-42148-P and TIN2016-75845-P); and the "Red de Computación de Altas Prestaciones en Arquitecturas Heterogéneas" (CAPAP-H4 and CAPAP-H5).
  - Regional funding by the Galician Government (Xunta de Galicia) under the Consolidation Program of Competitive Research Groups (Computer Architecture Group, refs. GRC2013/055 and ED431C 2017/04); Network of Cloud and Big Data Technologies for HPC (refs R2014/041 and ED431D R2016/045) and Funding for the Accreditation, Structuring and Improvement of the Remarkable Research Centre on Information and Communication Technology (CITIC ref. ED431G/01).
- Access to clusters, supercomputers and other computing platforms:
  - *Pluton* cluster (Computer Architecture Group, University of A Coruña, Spain).
  - *Roma* and *Warsaw* cluster (Global Scientific Information and Computing Center, Tokyo Institute of Technology, Japan).
  - *TSUBAME-KFC* supercomputers (Global Scientific Information and Computing Center, Tokyo Institute of Technology, Japan).
  - *DGX-1* workstation by the Real World Big-Data Computation Open Innovation Laboratory (RWBC-OIL) (National Institute of Advanced Industrial Science and Technology (AIST), Japan and NVIDIA Company)

  - *NVIDIA Tesla Kepler K40* card donated by NVIDIA Company.

- Two three-month research visits to the Global Scientific Information and Computing Center at Tokyo Institute of Technology in Japan, funded by the Ministry of Education, Culture and Sport of Spain (ref. EST16/00579), by the INDITEX-UDC 2016 collaboration grant and by the High-Performance Embedded Architecture and Compilation Netfowrk of Excellence Collaboration 2017 Grant.

Registered Software

The following software product has been registered in the IP registry as outcome of this Thesis:

- Adrián Pérez Diéguez, Jacobo Lobeiras Blanco, Margarita Amor López, Ramón Doallo Biempica. *BPLG: A Tuned Butterfly Processing Library for GPUs architectures*. December 2016. Record entry number: C-241-2016. Owning entity: Universidade da Coruña. Priority country: Spain.

Publications from the Thesis

Journal Papers (5)

- Adrián P. Diéguez, Margarita Amor, Ramón Doallo. Tree Partitioning Reduction: A New Parallel Partition Method for Solving Tridiagonal Systems. In *ACM Transactions on Mathematical Software*. (Accepted under second revision).
  JCR Impact Factor (2017): 2.905, Q1 in Computer Science, Hardware & Architecture.

- Adrián P. Diéguez, Margarita Amor, Ramón Doallo, Akira Nukada, Satoshi Matsuoka. Efficient High-Precision Integer Multiplication on the GPU. In *International Journal of High Performance Computing Applications.* (Submitted)
  JCR Impact Factor (2017): 2.015, Q2 in Computer Science, Hardware & Architecture.

- Adrián P. Diéguez, Margarita Amor, Ramón Doallo. Parallel Prefix Operations on GPU: Tridiagonal System Solvers and Scan Operators. In *The Journal of Supercomputing* (Accepted under second revision).
  JCR Impact Factor (2017): 1.532, Q2 in Computer Science, Hardware & Architecture.

- Adrián P. Diéguez, Margarita Amor, Jacobo Lobeiras, Ramón Doallo. Solving Large Problem Sizes of Index-Digit Algorithms on GPU: FFT and Tridiagonal System Solvers. In *IEEE Transactions on Computers*, volume 67, issue 1, pages 86-101. January 2018.
  JCR Impact Factor: 3.052, D1/Q1 in Computer Science, Hardware & Architecture.
  DOI 10.1109/TC.2017.2723879

- Adrián P. Diéguez, Margarita Amor, Ramón Doallo. BPLG-BMCS: GPU-Sorting Algorithm using a Tuning Skeleton Library. In *The Journal of Supercomputing*, volume 73, issue 1, pages 4-16. January 2017.
  JCR Impact Factor: 1.532, Q2 in Computer Science, Hardware & Architecture.
  DOI 10.1007/s11227-015-1591-9

International Conferences (7)

- Adrián P. Diéguez, Margarita Amor, Ramón Doallo. A Tuning Strategy for Tridiagonal System Solvers on GPU. In *18th International Conference on Computational and Mathematical Methods in Science and Engieering (CMMSE'18).* July 2018.
  ISBN 978-84-697-7861-6

- Adrián P. Diéguez, Margarita Amor, Ramón Doallo, Akira Nukada, Satoshi Matsuoka. Efficient Solving of Scan Primitive on Multi-GPU Systems. In *32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'18)*, pages 794-803. May 2018.
  ISSN 1530-2075

- Adrián P. Diéguez, Margarita Amor, Ramón Doallo. Solving Multiple Tridiagonal Systems on a Multi-GPU Platform. In *26th Euromicro International Conference on Parallel, Distruted and Network-based Processing (PDP'18)*, pages 759-763. March 2018.
  ISSN 2377-5750

- Adrián P. Diéguez, Margarita Amor, Ramón Doallo. New Tridiagonal System Solver on GPU architectures. In *22th International Conference on High Performance Computing (HiPC'15)*, pages 85-93. December 2015.
  ISBN 978-1-46738487-2

- Adrián P. Diéguez, Margarita Amor, Ramón Doallo. Solving Tridiagonal Systems with BPLG Library. In *11th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES'15)*, pages 267-270. July 2015.
  ISBN 978-88-905806-3-5

- Adrián P. Diéguez, Margarita Amor, Ramón Doallo. BS-Comb: An Efficient Sorting Algorithm for GPUs. In *15th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE'15)*, pages 461-473. July 2015.
  ISBN 978-84-617-2230-3

- Adrián P. Diéguez, Margarita Amor, Ramón Doallo. Efficient Scan Operator Methods on a GPU. In *26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'14)*, pages 190-197. October 2014.
  ISSN: 1550-6533

Book Chapters (1)

- Adrián P. Diéguez, Margarita Amor, Ramón Doallo. Chapter: *Techniques for Autotuning Algorithms on Heterogeneous Platforms*. In First PhD Symposium on Sustainable Ultrascale Computing Systems, pages 25-28. Computer Architecture, Communications and Systems Group (ARCOS) University Carlos III Madrid. Spain (2016).
  ISBN 978-84-608-6309-0

National Conferences (1)

- Adrián P. Diéguez, Margarita Amor, Ramón Doallo. CUDA Optimization Techniques for Scan Operator. In *XXV Jornadas de Paraalelismo*, pages 287-292. September 2014.
  ISBN: 84-697-0329-3

Contents

List of Tables

List of Figures

Chapter 1

An Introduction to GPU Computing

Graphics Processing Units (GPUs) are parallel processors designed to accelerate portions of a program, but not to replace CPU computing. The main program is executed on the CPU, and code fragments, called *kernels*, are executed on the GPU. CPU is suitable for control-intensive jobs, whereas GPU is suitable for data-paralell computation-intensive jobs. CUDA [93] is a general-purpose parallel computing platform and programming model that leverages the parallel compute engine in *NVIDIA* GPUs to solve data-parallel computation-intensive problems in a more efficient way. The CUDA programming language allows programming the CUDA GPUs using an extension of C language.

In the following sections, we describe the basics of CUDA programming and its execution model, as well as the GPU architectures used in this thesis. A more detailed description can be found in [17], [70] and [92].

1.1. The CUDA Programming and Execution Model

A kernel is expressed as a sequential program, and then, from the *host* (CPU), the user specifies how this code is mapped to the logical thread hierarchy of the *device* (GPU). Internally, CUDA handles the execution of the program by scheduling and

processing the logical threads over CUDA physical cores. A typical processing flow in a CUDA program is as follows:

- Allocate space in the device memory.
- Copy data from host memory to device memory.
- Invoke kernels from the host to perform the computation in the GPU
- Copy results back from device memory to host memory.
- Release memory space in the device.

As can be observed, the CPU and the GPU have separated memories. One of the most important features of CUDA is its memory hierarchy, where the device has different memory types, depending on the purpose.

When a kernel is invoked from the host, the execution is performed in the device, where a large number of logical threads are created and organized following the user's indications. These threads follow a two-level thread hierarchy abstraction: threadblocks and grids of blocks. A grid is composed of many threadblocks; and a threadblock is a group of threads that cooperate with each other. However, threads from different threadblocks cannot communicate with each other in the same kernel. The user specifies how both the grid and the threadblock are scheduled to perform the execution on the GPU, organizing grids and threadblocks into three dimensions. This configuration is very important, since it determines how the GPU resources are distributed and how the GPU memory system is accessed, which has a bearing on performance.

The GPU architecture is built of an array of *Streaming Multiprocessors* (SM), as shown in Figure 1.1. The parallelism is achieved by the replication of these SMs. Each SM is mainly composed of many CUDA cores for single and double precision, also known as *Streaming Processors* (SP), a shared memory, an L1 cache, a register file, load/store units, special function units (SFU), warp schedulers and memory controllers.

Each SM is designed to execute hundreds of threads concurrently. When a kernel is invoked, the threadblocks of the grid are distributed among the available SMs for

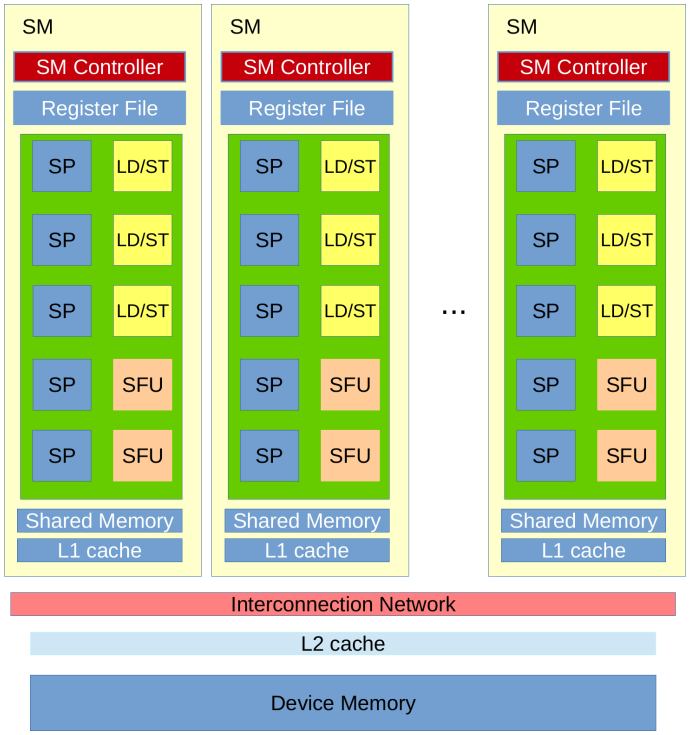


Figure 1.1: A GPU composed of an array of Streaming Multiprocessors (SM)

execution, and an SM can hold several threadblocks concurrently, which are called *resident* or *active* threadblocks. Once dispatched, their threads execute concurrently on that assigned SM only. Also, once the execution of a threadblock is started, the threadblock remains resident in that SM until it finishes. Each threadblock groups its threads into *warps*, a set of 32 threads that executes instructions in lockstep; i.e, all threads in a warp execute the same instruction at the same time. Thus, each SM partitions its assigned threadblocks into warps, and these warps are scheduled for execution on available SM resources. All warps which are scheduled to be executed concurrently in an SM are called *active* warps, and their threads are *active* threads. There are two types of instruction latency: arithmetic instruction latency (around 10-20 cycles) and memory instruction latency (between 400-800 cycles for global memory accesses). Switching active warps makes it possible to hide the latency.

It should be observed that the term *thread* can be confusing: while all threads in a threadblock run logically in parallel, not all threads can execute physically at the same time. The GPU programming model executes *Single Instruction Multiple*

Thread (SIMT) operations by matching each physical thread as a number of logical threads. The SIMT architecture is similar to the *Single Instruction, Multiple Data* (SIMD) architecture: both broadcast the same instruction to multiple execution units. However, SIMD requires that all elements in a vector execute together in a synchronous group; while SIMT allows multiple threads in a warp to execute independently. This allows different threads in the same warp to take different instruction paths (*branch divergence).* In case of divergence, CUDA disables some of the threads (using a *mask*) and executes instructions on one path; then it disables the other threads and executes instructions on the other path.

Memory management and accesses are an important part of CUDA, having a particularly large impact on performance. CUDA presents a low-latency but lower-capacity memory subsystem to optimize performance, which is depicted in Figure 1.2. This subsystem is composed of multiple levels of memory with different latency, bandwidth and capacities. *Global memory* is the largest, but highest-latency, memory on a GPU. It can be accessed by any thread, even after kernel execution, as Figure 1.3 shows. Global memory resides in the device memory, an off-chip on-board DRAM memory. *Registers* are the fastest memory space. Each thread has its own set of private registers, and any variable declared in the kernel is generally stored in a register. Once a kernel completes the execution, a register value cannot be accessed. Variables in a kernel that are eligible for registers but which cannot be stored into the register space by the compiler are saved into *local memory*, a portion of global memory which is accessed by the corresponding thread only, so local memory accesses have higher latency and lower bandwidth than registers. This behavior is called *local memory spilling.* Furthermore, *shared memory* is a programmable on-chip memory, and it has much higher bandwidth and much lower latency than global memory. Shared memory shares the lifetime of the kernel, and serves as a inter-thread communication inside a threadblock; thus, only threads within a threadblock can access this memory space. When a threadblock is finished, its allocation of shared memory is released. *Constant memory* is other memory which resides in device memory. The constant memory must be initialized by the host and kernels can only read from it. This memory performs well when all threads in a warp read from the same memory address. Another memory that resides in the device is the *texture memory.* Each SM caches this memory with a *read-only data cache*, and is only accessed through this dedicated read-only cache. Texture memory is optimized

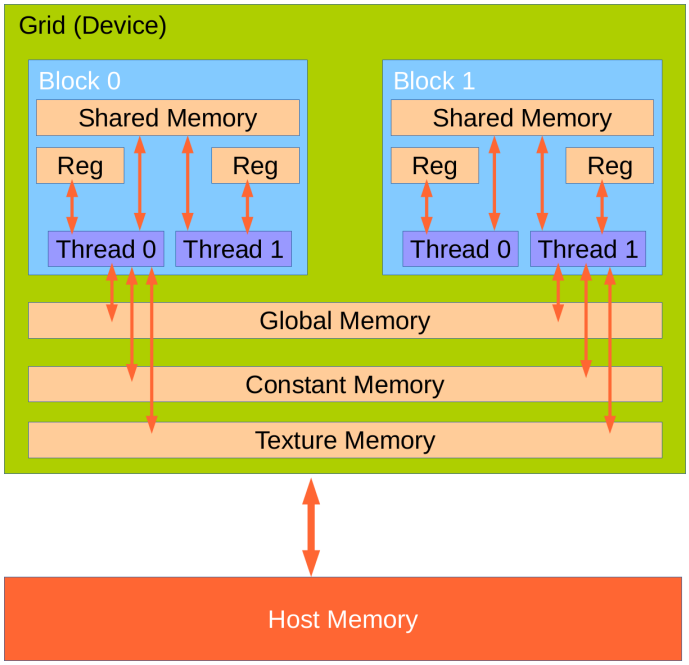


Figure 1.2: CUDA memory subsystem

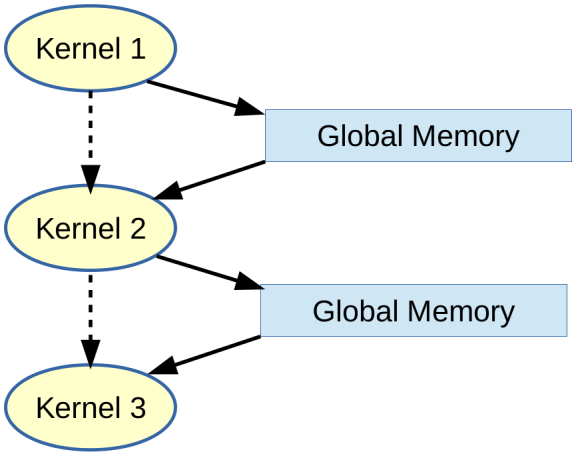


Figure 1.3: Communication among kernels across global memory

for 2D spatial locality.

When sharing data across the memory hierarchy, it is necessary to pay attention how to avoid *race conditions* or hazards; i.e., unordered accesses by multiple threads to the same memory location. It is not defined how warp schedulers issue warps and the order they follow. In order to synchronize threads in a system, there are four

types of synchronization barriers:

- *System-level.* Wait for all threads on both the host and the device to complete. This is possible with a *cudaDeviceSynchronize()* instruction in the host side.
- *Device-level.* Wait for a GPU task to complete. This is done by assigning one GPU to a CUDA *Stream*, a sequence of operations that execute in issue-order on the GPU, and using a *cudaStreamSynchronize()* instruction in the host side.
- *Grid-level.* There is not an explicit instruction to wait for all the threadblocks of a grid to complete. There are several strategies to cover this point, which are explained below.
- *Block-level.* Wait for all threads within a threadblock to complete. This is done by using the *_syncthreads()* instruction in the kernel.

1.1.1. SM Resource Partition

When a warp idles due to any dependence or latency, the SM can schedule another available warp from any threadblock resident on the SM. Switching between concurrent warps has no penalty. While warps within an SM can be scheduled in any order, the number of active warps is limited by SM resources. Registers and shared memory are scarce resources in the SM, and CUDA has to partition these resources among the threads resident on an SM. Thus, these resources limit the number of active warps in an SM. Each SM has a set of 32-bit registers that is partitioned among active threads, and a fixed amount of shared memory that is partitioned among active threadblocks. The number of threadblocks and warps that can simultaneously reside on an SM depends on the number of registers and shared memory available on the SM. Reducing the number of registers a kernel means that more warps will be executed concurrently. Reducing the amount of shared memory used by each threadblock will result in it being possible to execute more threadblocks concurrently. If there are insufficient registers or shared memory on each SM to execute at least one threadblock, the kernel invocation fails; whereas, if the number of threads per threadblock is not a multiple of 32, some threads will be executed as inactive, but they consume SM resources.

Specifically, a threadblock becomes an active threadblock when the resources it needs are assigned. The warp scheduler of the SM chooses active warps on each cycle to be dispatched to execution. To be eligible, two requirements must be met: 32 CUDA cores are available and all arguments of the current instruction are ready. If on every cycle, all the warps schedulers have an eligible warp, then a complete resource utilization is achieved, ensuring that the latency of each instruction can be hidden by issuing other instructions. Switching the warp context between active warps has no penalty, since the required state (program counter, registers and shared memory) is already on-chip, as is maintained during the entire life-time of the threadblock.

Therefore, instructions are executed in a sequential way within each CUDA core. If a warp stalls, the warp scheduler finds other eligible warps to keep the cores occupied, and hide latency. *Warp occupancy* is the ratio of active warps to maximum number of warps per SM. In a similar way, the *block occupancy* defines the ratio between active threadblocks to maximum number of threadblocks supported per SM owing to the fixed amount of shared memory and registers.

During the execution, the grid is divided into *waves of threadblocks*. The wave size depends on the number of active threadblocks and the number of SMs. For example, if 128 threadblocks have to be executed on a Tesla Kepler K20 that has 13 SMs, with a hypothetical number of 4 active threadblocks per SM, then each full wave is composed of $13 \times 4 = 52$ threadblocks. Thus, the kernel is executed in 2 full waves and a much smaller wave with only 24 threadblocks. The last wave consumes a significant fraction of the runtime, although is under-utilizing the GPU. This event is called *tail effect*.

1.2. Efficient Memory Accesses in CUDA

When comparing the measured program values to theoretical peak values, it is easy to determine if the execution is limited by arithmetic (*arithmetic bound* problem) or by memory bandwidth (*memory bound* problem).

1.2.1. Global Memory Accesses

Currently, most HPC workloads are bound by memory bandwidth. Especially in GPUs, most applications tend to be limited by the global memory bandwidth. Certain conditions need to be met to achieve the maximum performance when reading and writing data in this memory.

The allocated host memory is *pageable*; i.e, the operating system can move the data allocated in this memory to different physical locations (*virtual memory system*). This enables us to use more memory than that physically available. If the GPU has to transfer data from/to this pageable host memory, a *page-locked* or *pinned* host buffer will need to be created to move data safely. Thus, data are first moved from host memory to the pinned buffer, and then to the device memory. *Pinned* host memory can be allocated directly, to avoid the initial data transfer (from pageable host memory to the pinned buffer), achieving a high speed-up.

Zero-copy memory is a pinned host memory space that is mapped into the device address space, being possible to access this memory from both host and device, performing data transfers across PCI-e by demand. This is useful for leveraging host memory when insufficient device memory, avoiding explicit data transfers between host-device, and improving PCI-e transfer rates. However, frequent accesses to this memory will slow performance down, due to the latency of the PCI-e communications.

To improve the *zero-copy* behavior, CUDA 6.0 introduces the *Unified Memory* to simplify the memory management. The zero-copy memory is allocated in the host, and the kernel suffers from the latency of PCI-e transfers. However, the unified memory decouples the memory spaces to the host and the device, thus data are transparently migrated on demand, improving locality and performance. This is possible thanks to the *Unified Virtual Addressing* (UVA) support, which provides a single virtual memory address space for all CPU and GPU memories, although it does not automatically migrate data, which is only done by the unified memory.

All accesses to global memory go through the L2 cache, and depending on the architecture version, some accesses also pass through the L1 cache. In order to take advantage of the global memory bandwidth, two requirements must be met, otherwise the global memory performance slows down significantly: *aligned memory*

accesses and *coalesced memory acceses*. To perform *aligned memory accesses*, the first memory address of the transaction must be a multiple of the cache granularity (either 32 bytes for L2 or 128 bytes for L1). *Coalesced memory accesses* occur when all the threads in a warp access to contiguous memory addresses: this reduces the number of transactions to service the maximum number of memory requests.

Regarding the memory accesses and the compiler, *static indexing* represents the fact that constant indices are derived by the compiler in all accesses to an array, placing elements directly into registers, and it is the most efficient way to reference an array. However, when the compiler cannot resolve indices to constants, it places them into local memory, with the consequent performance loss (*dynamic indexing*). Indices must be determined by the compiler and must not depend on a value determined at runtime. In this case, if all threads within a warp access the same index (*uniform access*), performance is fairly high owing to the GPU cache system. Otherwise, if the threads of a warp access elements using different indices, it is called *non-uniform indexing*, this being the worst scenario.

1.2.2. Shared Memory Accesses

Shared memory is faster than global memory as it is a low-latency on-chip memory. Shared memory is smaller and it is only reachable by the threads within the same threadblock, but offering much higher bandwidth.

The shared memory is divided into 32 memory modules called *memory banks*. These modules are accessible simultaneously and have a *bank width* that depends on the architecture. For example, if the 32 threads of a warp access to different banks simultaneously, the operations are serviced by one memory transaction. Otherwise, the bank width defines how many threads can access simultaneously to the same bank; but if this amount is surpassed, it needs more memory transactions to serve data, decreasing performance. If more threads than those supported by the bank width capacity try to write into the same memory bank, a *bank conflict* occurs, and the operation is replayed. However, different threads can read from the same memory bank, a *broadcast access*, with no penalty.

There are two different bank widths depending on the architecture: 4-byte or

8-byte widths. In the first case, successive 4-byte words are mapped to consecutive banks, and each bank has a bandwidth of 4 bytes per two clock cycles. In the second case, there are also two address modes: 8-byte or 4-byte modes. In the 8-byte mode, successive 8-byte words are assigned to consecutive banks, and each bank has a bandwidth of 8 bytes per cycle. Hence, with this address mode, two threads can access any sub-word within the same 8-byte word with no penalty. In the 4-byte mode, succesive 4-byte words are mapped to consecutive banks, and it is possible to access two 4-byte words in the same bank at once, with no conflict thanks to the 8-byte bandwidth.

1.2.3. Shuffle Instructions

Shuffle instructions enable threads within the same warp to exchange data through registers directly, rather than through shared or global memory. This instruction has higher bandwidth and it is highly interesting for rapidly interchanging data among threads. To do so, each thread has a unique identifier inside the warp, called the *lane*, and there are two datatypes supported by shuffle instructions: integers and float variables.

On the one hand, shuffle instructions can be used to free up shared memory to be used for other data or to increase the occupancy. On the other hand, they are faster than shared memory since they only require one instruction versus three for shared memory (write, synchronize, read).

There are several communication patterns supported by the shuffle instructions. The general shuffle instruction, *_shfl(var, src, width)*, returns the value *var* stored in a register from any other thread. The *src* thread is identified by its lane, and if this value is constant, the *var* value from *src* is broadcast to all threads. It is also possible to create thread-groups inside the warp specifying the *width* of the group, which is 32 by default. The *_shfl_up(var, delta, width)* and *_shfl_down(var, delta, width)* instructions return the *var* value from a source thread with a lower or higher lane defined by the delta argument. As of CUDA 9.0, these functions have been deprecated and changed to *_shfl_sync*, *_shfl_up_sync*, *_shfl_down_sync*, keeping the same arguments and behavior.

1.2.4. Atomic Operations

Another common access pattern in computing applications is to access and modify a single memory location by several threads, but ensuring no interference from others in each access until completing the operation, in order to avoid race conditions. This memory pattern is called *atomic access*, since it is necessary to guarantee the atomicity of read-modify-write operations.

CUDA provides 32-bit and 64-bit atomic operations for global and shared memory. The most common are *atomicAdd*, *atomicSub*, *atomicMin* and *atomicMax*. Depending on the version of the architecture, the hardware provides native support for these instructions. Otherwise, the desired function can be software implemented, based on a *Compare-And-Swap* (CAS) implementation. Instead of writing directly in memory ensuring no inference (native support), a CAS implementation compares the contents of a memory location with the given value and, only if they are the same, does it modify the contents of that memory location to a new given value. If the value had been updated by another thread in the meantime, the write would fail.

1.3. Multiple-GPU Programming

So far, different features and techniques of CUDA have been introduced, focusing on a single GPU. However, when several GPUs participate in the system, a *multiple-GPU* programming model should be considered.

The most two common cases for performing a multiple-GPU execution are:

- *Memory space limits.* The datasets are too large to be executed in the memory of a single GPU.
- *Scalability.* Although the datasets can fit into a single GPU, better performance can be obtained by partitioning and executing the problem among several GPUs concurrently.

The efficiency of the execution depends on how the inter-GPU communication is designed. It is possible to distinguish two types of environments: a *Multi-GPU*

environment represents a single computing node composed of several GPUs; whereas if the system consists of several of these nodes connected through a low-latency bus, it is called a *Multi-Node* environment. When GPUs are arranged in several nodes a Multi-Node communication is required.

When designing a multiple-GPU execution, the workload is divided among devices. There are two common communication cases, depending on the program. On the one hand, no data exchange is needed between the partitions of the problem, thus there is no communication among devices. On the other hand, each partition of the problem needs to communicate partial data to other partitions, requiring redundant data storage and communication among GPUs. The first case is trivial, as each partition runs independently in each GPU. The second case is more challenging, since it is necessary to consider how data can optimally be moved among GPUs.

CUDA presents a number of features to facilitate Multi-GPU programming. Kernels executed under 64-bit applications on modern devices can directly access the global memory of any GPU connected to the same PCIe network using the CUDA peer-to-peer (P2P) API, avoiding communication via the host. This is possible thanks to their sharing a common memory address space (UVA). Hence, data are copied between these devices asynchronously along the shortest PCI-e path, enabling communication-computation overlapping. Specifically, *peer-to-peer accesses* enable direct load and store operations within a kernel across GPUs. If the GPUs are not connected to the same PCI-e bus, it is possible to transfer data from host, *peer-to-peer transactions*, through host memory rather than directly across the PCI-e bus. Synchronization between devices can be performed by assigning a CUDA stream to each GPU and using the *cudaStreamSynchronize()* instruction from the host for all GPUs.

In the case of Multi-Node programming, the communication is performed across a cluster composed of several computing nodes. In this case, *Message Passing Interface* (MPI), a well-known standard and portable API, is employed. Using MPI, the contents of host memory can be transmitted directly by MPI functions. Instead of copying data from the device memory to the host buffers, and then calling the MPI API, MPI and CUDA can be combined, sending data directly to the GPU buffers. This CUDA support is called *CUDA-Aware MPI*, enabling direct MPI communi-

cation between GPU global memories. Moreover, *RDMA - GPU Direct* technology enables low-latency transfers over an *Infiniband* connection between GPUs in different nodes without host processor involvement, reducing CPU overhead and communication latency.

1.4. CUDA Architectures

Several generations of CUDA-capable GPUs have been released so far. In the following subsections, a global overview of the CUDA architectures used in this thesis is given from Fermi to Volta.

1.4.1. Fermi Architecture

Each Fermi SM [91] is composed of 32 CUDA cores, 16 load/store units (LD/ST units) to address memory operations for sixteen threads per clock, four special function units (SFU) to execute transcendental mathematical instructions, a memory hierarchy and warp schedulers, as Figure 1.4 represents.

The board has six 64-bit memory partitions with a 384-bit memory interface which supports up to 6 GB of GDDR5 DRAM memory. The CPU is connected to the GPU via a PCI-e bus. Each CUDA core has a fully pipelined arithmetic logic unit (ALU) as well as a floating point unit (FPU). In order to execute double precision, the 32 CUDA cores can perform as 16 FP64 units. Each SM has two warp schedulers which enable issue and execute 2 warps concurrently.

A key block of this architecture is the memory hierarchy, as Figure 1.5 shows. It introduces 64 KB of configurable shared memory and an L1 cache per SM, which can be configured as 16 KB of L1 cache with 48 KB of shared memory; or 16 KB of shared memory with 48 KB of L1 cache. Whereas the CPU L1 cache is designed for spatial and temporal locality, the GPU L1 is only optimized for spatial locality. Frequent accesses to a cached L1-memory location does not increase the probability of hitting the datum, but it is attractive when several threads are accessing to adjacent memory spaces. The 768 KB L2 cache is unified and shared among all SMs that services all operations (load, store and texture). Both caches are used

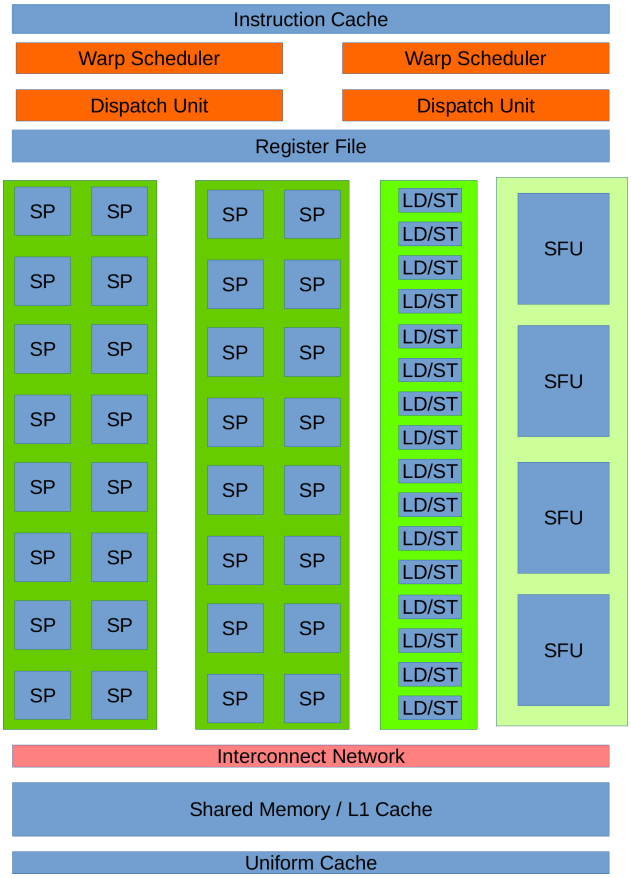


Figure 1.4: Description of an SM in the Fermi architecture

to store data in local and global memory, including register spilling. However, it is necessary to configure whether reads are cached in both L1 and L2, or only L2. This architecture is represented as *compute capability 2.x*, a special term to describe the hardware version of the GPU which comprises a major revision number (left digit) and a minor revision number (right digit). Devices with the same major revision number belong to the same core architecture, whereas the minor revision number corresponds to an incremental improvement to the core architecture.

1.4.2. Kepler Architecture

Kepler [96] includes up to 15 SMs and six 64-bit memory controllers. Each SM has 192 single-precision CUDA cores, 64 double-precision units, 32 SFUs, 32 LD/ST

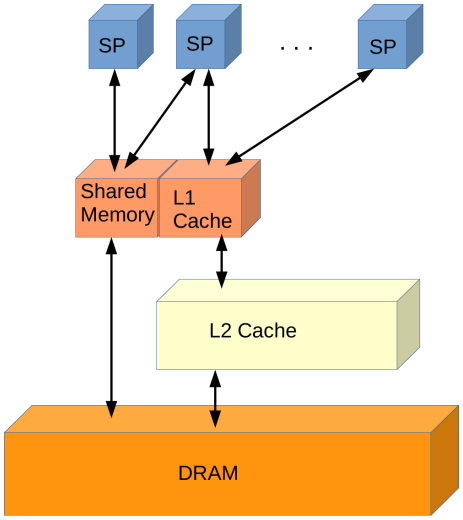


Figure 1.5: Description of the Fermi memory subsystem

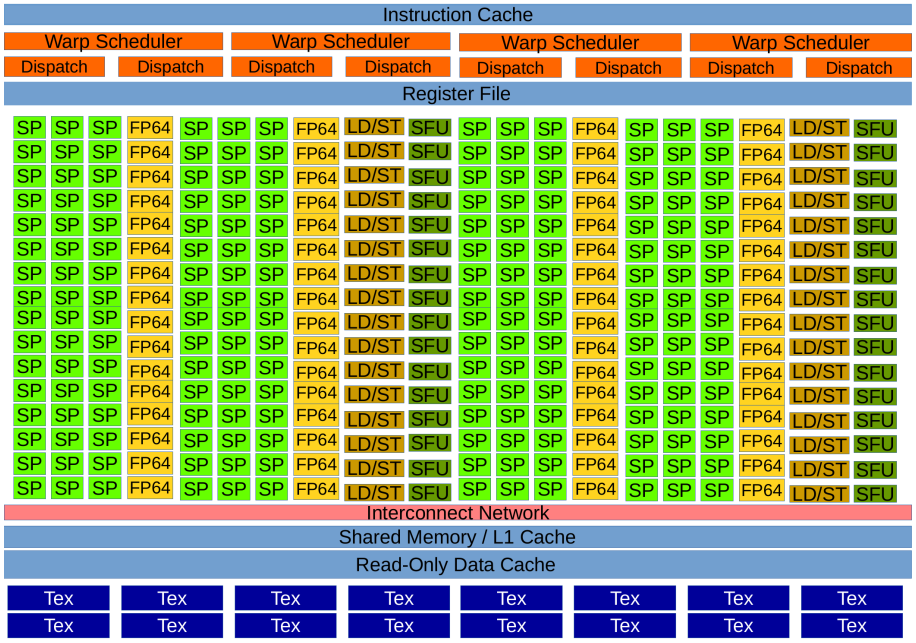


Figure 1.6: Description of an SM in the Kepler architecture

units and 16 texture units, as Figure 1.6 shows.

Also, four warp schedulers, each with 2 dispatch units, which allow four warps to be issued and executed concurrently. It also increases the number of registers accessed by each thread, from 63 in Fermi, to 255; it introduces the shuffle instruc-

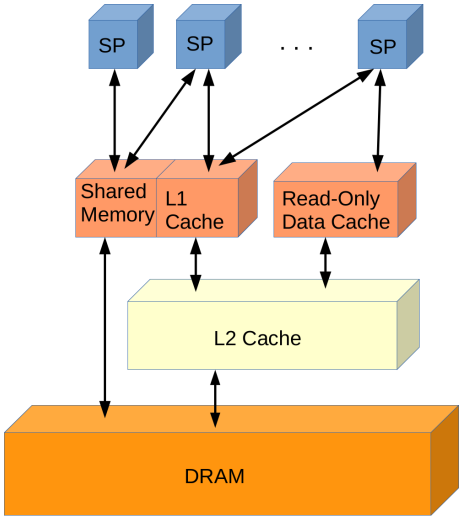


Figure 1.7: Description of the Kepler memory subsystem

tions and improves the atomic operations by introducing native support for FP64 atomics in global memory. It also introduces the CUDA Dynamic Parallelism, the capacity of launching kernels from a kernel. Additionally, the memory hierarchy is organized similarly to Fermi, as Figure 1.7 depicts.

The 64 KB shared memory / L1 cache is improved by permitting a 32 KB / 32 KB split between the L1 cache and shared memory. It also increases the shared memory bank width from 32 bits in Fermi to 64 bits, and introduces a 48 KB Read-Only Data cache to cache constant data. The L2 cache is also increased to 1536 KB, doubling the Fermi L2 cache capacity. Additionally, Kepler compute capabilities are represented with the 3.x code.

1.4.3. Maxwell Architecture

Maxwell [99] consists of up to 16 SMs and four memory controllers. Each SM has been reconfigured to improve performance per watt. It contains four warp schedullers, each capable of dispatching two instructions per warp every clock cycle. The SM is partitioned into four 32-CUDA core processing blocks, each with eight texture units, 8 SFUs and 8 LD/ST units. Figure 1.8 shows this new partition. Regarding the memory hierarchy (see Figure 1.9), it features a 96 KB dedicated shared memory (although each threadblock can only use up to 48 KB), while the L1

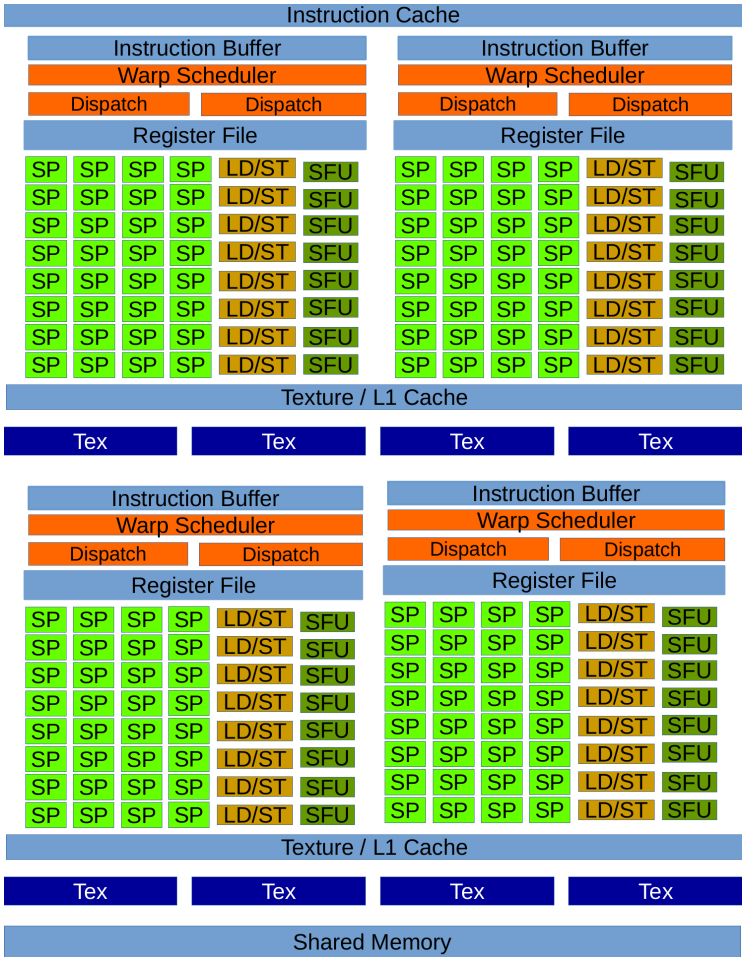


Figure 1.8: Description of an SM in the Maxwell architecture

cache is shared with the texture caching function. The L2 cache provides 2048 KB of capacity. The memory bandwidth is also increased, from 192 GB/sec in Kepler, to 224 GB/sec, and native support is introduced for FP32 atomics in shared memory. Maxwell is represented as compute capabilities 5.x.

1.4.4. Pascal Architecture

A Pascal board [102] is composed of up to 60 SMs and eight 512-bit memory controllers. Each SM has 64 CUDA cores and four texture units, as Figure 1.10 shows. It has the same number of registers as Kepler and Maxwell, but provides much more SMs, thus many more registers overall. It has been designed to sup-

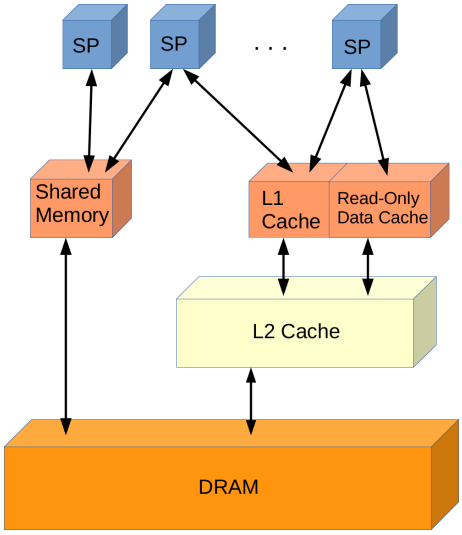


Figure 1.9: Description of the Maxwell memory subsystem

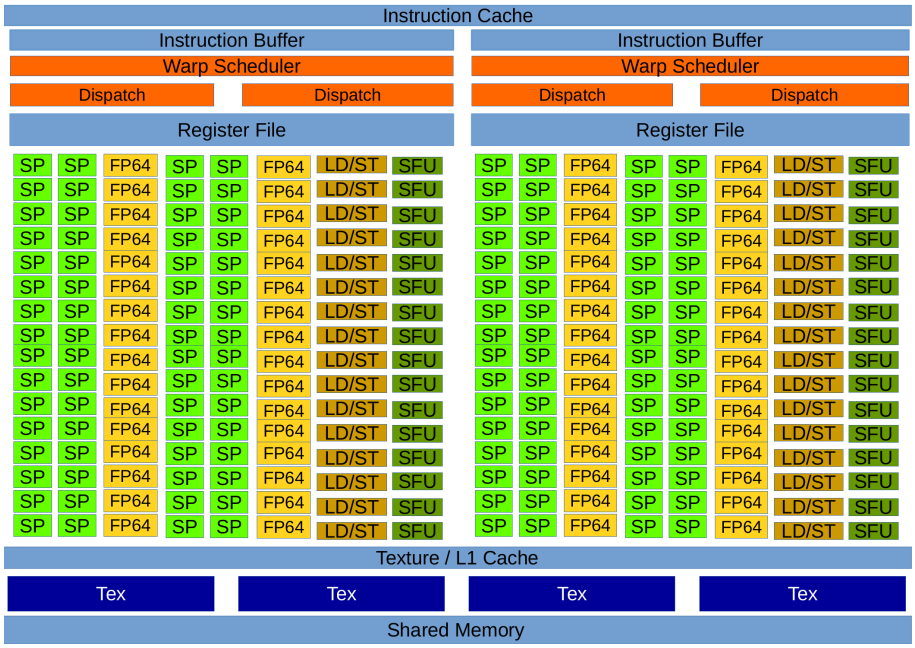


Figure 1.10: Description of an SM in the Pascal architecture

port many more active warps and threadblocks than previous architectures. The shared memory bandwidth is doubled to execute code more efficiently. It allows the overlapping of load/store instructions to increase floating point utilization, also improving the warp scheduling, where each warp scheduler is capable of dispatching

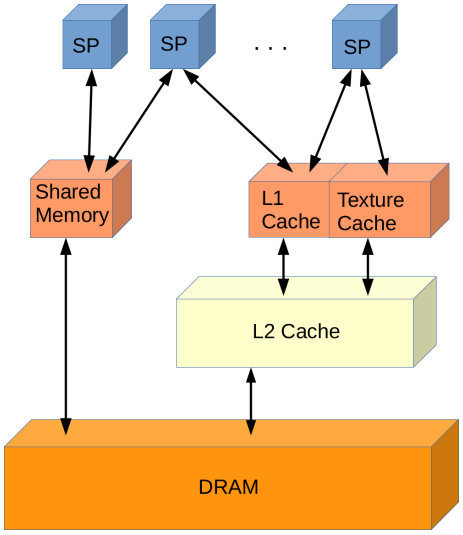


Figure 1.11: Description of the Pascal memory subsystem

two warp instructions per clock. CUDA cores are able to process both 16-bit and 32-bit instructions and data, facilitating the use of deep learning programs, but also providing 32 FP64 CUDA cores for numerical programs. Global memory native support is also extended to include FP64 atomics.

The memory hierarchy configuration is also changed, as Figure 1.11 shows. Each memory controller is attached to 512 KB of L2 cache, providing 4096 KB of L2 cache, and introduces HBM2 memory, providing a bandwidth of 732 GB/sec. It presents 64 KB of shared memory per SM, and an L1 cache that can also serve as texture cache, which acts as a coalescing buffer to increase warp data locality. Its compute capability is represented with the 6.x code.

1.4.5. Volta Architecture

The most recent CUDA architecture is called Volta [103] and delivers the highest GPU performance so far. A Volta board has up to 84 SMs and eight 512-bit memory controllers. Each SM has 64 FP32 CUDA cores, 64 INT32 CUDA cores, 32 FP64 CUDA Cores, 8 tensor cores for deep learning matrix arithmetic, 32 LD/ST units, 16 SFUs, a new L0 instruction cache to provide higher efficiency than previous intructions buffers and a warp scheduler with a dispatch unit, as Figure 1.12 shows.

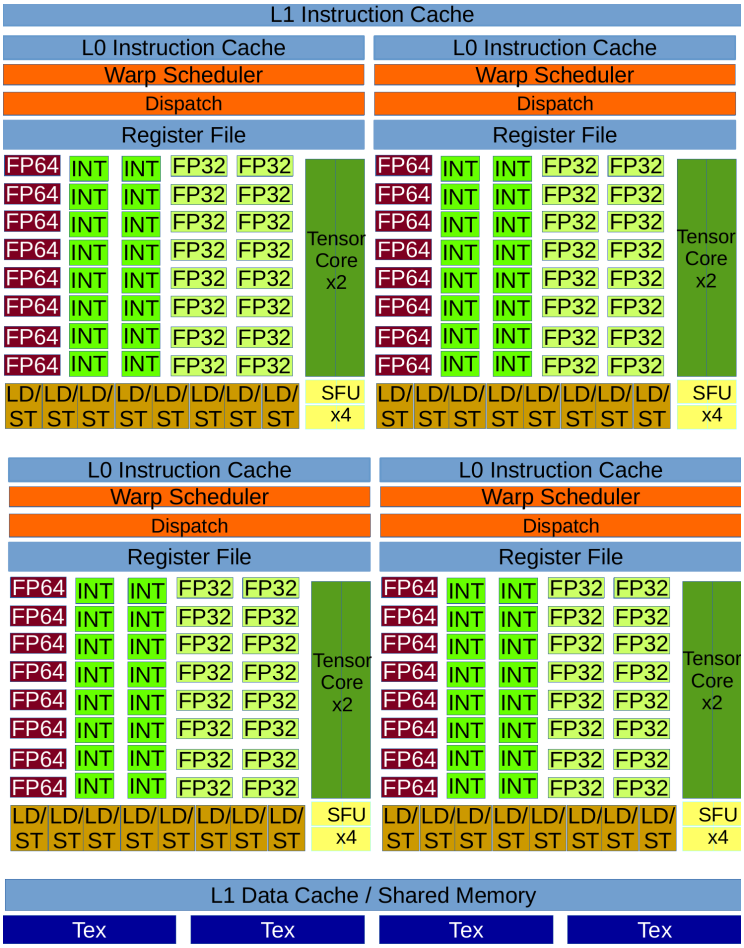


Figure 1.12: Description of an SM in the Volta architecture

A merged 128 KB L1 Data Cache / shared memory is introduced, providing 96 KB of shared memory, see Figure 1.13. The HBM2 bandwidth is also improved, obtaining 900 GB/sec. Additionally, the full GPU includes a total of 6144 KB of L2 cache and its compute capability is represented with the 7.0 code.

However, the biggest change comes from its independent thread scheduling. Previous architectures execute warps in SIMT fashion, where a single program counter is shared among the 32 threads. In the case of divergence, an active mask indicates which threads are active at any given time, leaving some threads inactive and serializing the execution for the different branch options. Volta includes a program counter and call stack per thread. It also introduces a schedule optimizer that determines what threads from the same warp must execute together into SIMT units,

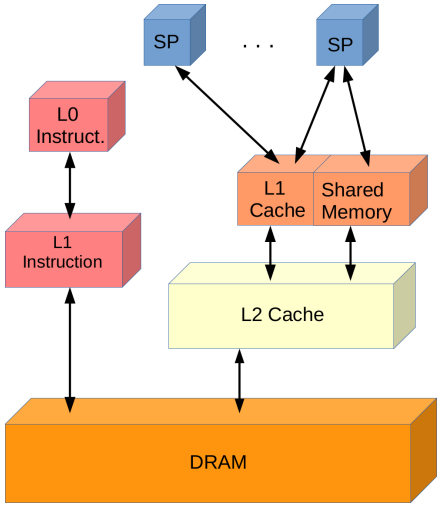


Figure 1.13: Description of the Volta memory subsystem

giving more flexibility, as threads can now diverge at sub-warp granularity.

The newest breakout feature of Volta is called a *Tensor Core*, which makes up to 12x faster for deep learning applications compared to previous Pascal P100 accelerator. They are essentially arrays of mixed-precision FP16/FP32 cores. Each of the 640 tensor cores operates on a 4×4 matrix, and their associated datapaths are custom-designed to increase floating-point compute throughput of the operations over this kind of matrix. Each tensor core performs 64 floating-point fused-multiply-add (FMA) operations per clock, delivering up to 125 TFLOPS for training and inference applications.

Chapter 2

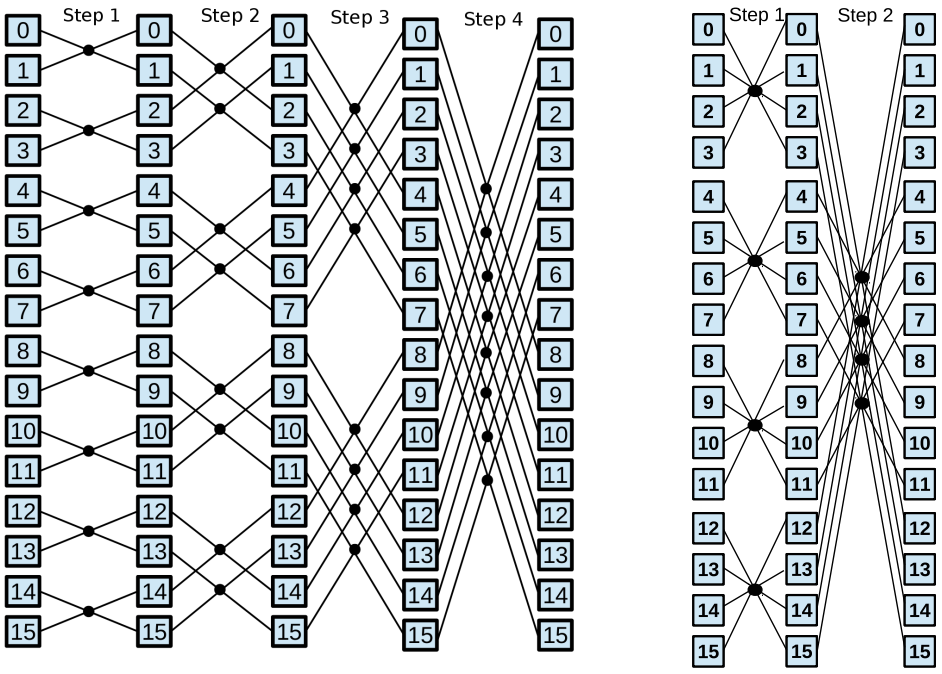
Parallel Prefix Algorithms

As introduced in the Preface, this work focuses on the efficient execution of parallel prefix algorithms. This chapter covers the definition of these algorithms, and a subset of them called *Index-Digit algorithms*. Different parallel prefix algorithms are analyzed in detail: the Fast Fourier Transform (FFT), tridiagonal system solvers, the scan primitive and sorting algorithms. An efficient implementation of these algorithms will be developed in the remaining chapters of this document.

2.1. Parallel Prefix Definitions

A parallel prefix algorithm [75] [76] solves a problem of size $N = r^n$, where r is called *radix*, in K steps, which may be depicted by a directed acyclic oriented graph called prefix circuit [133]. The computations are performed by the *Node operator*, which executes the core operation over the corresponding elements. This operator is represented by small circles in the prefix circuit, as can be observed in Figure 2.1, which shows the prefix circuit of a parallel prefix algorithm.

Parallel prefix algorithms match well to the GPU architecture. Their communication patterns are regular and known at compile-time. The pattern is static, does not depend on the runtime and can be expressed as a linear function with the element index as a variable. Furthermore, each resulting element is a combination of the results of other elements.



(a) Radix-2 pattern (b) Radix-4 pattern

Figure 2.1: A parallel prefix algorithm, Cooley-Tukey, with $N = 16$.

The Node operator is responsible for performing the computation in parallel prefix algorithms. Specifically, the Node operator is defined by four aspects: *fan in*, the number of input data; *fan out*, the number of output data; *size*, which represents the size in bytes of each data; and the specific core operation, which depends on the algorithm. The radix r, which is given by the algorithm pattern, has a direct bearing on the number of steps taken, K. Thus, r and n usually appears in the expression that calculates K. In general, most of the parallel prefix algorithms use binary Node operators and employ $r = 2$; hence, in most cases $N = 2^n$.

2.1.1. Index-Digit Algorithms

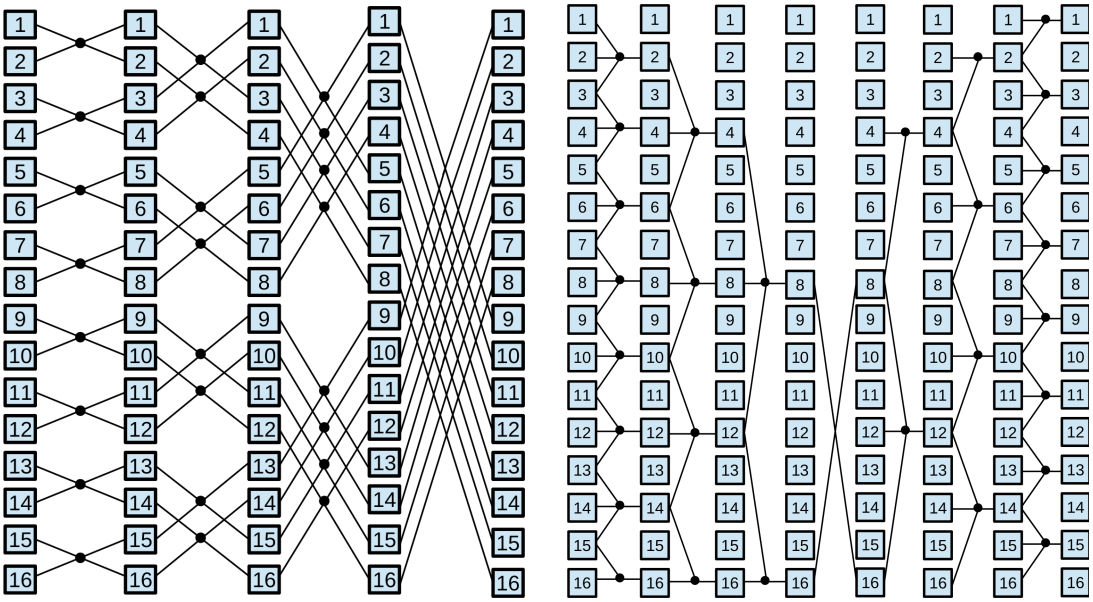
There is a subset of parallel prefix algorithms, called *Index-Digit* (ID) algorithms [82], which have special properties. These algorithms have a number of *Node operators* which remains constant along the computing steps, and the elements that take part in one Node operator are not used by another Node operator in the same step. Additionally, the number of computing steps is equal to n, $K = n$, and the

fan in and *fan out* values of their Node operators are equal to the radix r. These properties also imply that the number of Node operators in each step is equal to $\frac{N}{r}$, and the fact that increasing the radix of the algorithm involves decreasing the number of taken steps. Figure 2.1 shows the Cooley-Tukey algorithm, an Index-Digit algorithm, for $N = 16$ elements, where Figure 2.1 (a) shows the execution with $r = 2$ and Figure 2.1 (b) with $r = 4$. When using radix 2, there are $\frac{N}{r} = \frac{16}{2} = 8$ Node operators and the problem is solved in $K = 4$ steps, as $N = 2^4$. However, when using radix 4, each Node operator works with 4 elements, there are $\frac{N}{r} = \frac{16}{4} = 4$ nodes in each step and the problem is solved in $K = 2$ steps.

An Index-Digit algorithm is formally defined as an algorithm whose data interchange can be modeled as the rearrangement of a data array according to common permutations of the digits of each element index. To this end, a datum, or element, $x(t)$ with index $t = t_n \cdot r^{n-1} + \cdots + t_2 \cdot r + t_1$ is written as $[t_n \cdots t_2 t_1]$. For example, element $x(5)$ of an arbitrary radix-2 data sequence of $N = 16 = 2^4$ elements, is represented as [0101]; whereas $x(1)$ is represented as [0001]. Taking this digit representation into account, it is possible to model the algorithm, as will be seen in the following chapters.

Figure 2.2 shows two algorithms that solve tridiagonal systems. The Wang and Mou algorithm, see Figure 2.2 (a), is based on the Cooley-Tukey pattern, and is an Index-Digit algorithm. The number of computing steps is $K = n$, and the *fan in* and *fan out* values of the Node operator are equal to r. There are $\frac{N}{r}$ Node operators per step, and each element is not shared among several Node operators. The Cyclic Reduction algorithm, Figure 2.2 (b), is a parallel prefix algorithm, but it is not Index-Digit. As can be observed, the number of computing steps is $K = 2 \cdot n - 1$, the *fan in* is 3 in most cases and the *fan out* is always 1. The number of Node operators depends on the taking step and elements can be shared by two different Node operators.

Finally, Table 2.1 shows all the algorithms developed in this work, specifying whether they are Index-Digit or just parallel prefix algorithms. They are analyzed in depth the following sections and chapters.



(a) The Wang and Mou algorithm (b) The Cyclic Reduction algorithm

Figure 2.2: Difference between an Index-Digit algorithm (a) and a Parallel Prefix non-ID algorithm (b)

Operation	Algorithm	Type
FFT	Cooley-Tukey	ID algorithm
	Stockham	ID algorithm
Scan	KS pattern	Parallel Prefix algorithm
	LF pattern	Parallel Prefix algorithm
Tridiagonal System Solver	CR	Parallel Prefix algorithm
	PCR	Parallel Prefix algorithm
	RR-LF pattern	Parallel Prefix algorithm
	RR-KS pattern	Parallel Prefix algorithm
	WM	ID algorithm
	Tree-Partitioning Reduction	Parallel Prefix algorithm
Sorting	Bitonic Merge Sort	Parallel Prefix algorithm
	Bitonic Merge Comb Sort	Parallel Prefix algorithm

Table 2.1: Classification of the algorithms employed in this work

2.2. Fast Fourier Transform (FFT)

This section summarizes the basics about the *Discrute Fourier Transform* (DFT) and the *Fast Fourier Transform* (FFT), based on the content explained in [8] and [117].

The *Discrete Fourier Transform* (DFT) is a highly important operation for many applications, such as image and digital signal processing, filtering, compression or partial differential equation resolution. The DFT changes an N-point input signal into two-point output signals. Specifically, the input signal is decomposed into two output signals, which contain the amplitudes of the component sine and cosine waves. Furthermore, the input signal is in the time domain, whereas the outputs are in the frequency domain.

In the time domain, a signal x consists of N points or samples. In the frequency domain, the real part is written as ReX, and the imaginary part as ImX, and each of these signals are $N/2 + 1$ points long. The transform from the time domain to the frequency domain is called *Forward DFT* (decomposition), also denoted as y, whereas the *Inverse DFT* (synthesis) performs the inverse transform. The equation used to obtain the Forward DFT is the following:

$$y_l = \sum_{i=0}^{N-1} x_i \left[cos(\frac{2\pi}{N} il) - j \ sin(\frac{2\pi}{N} il) \right], \quad 0 \leq l < N \tag{2.1}$$

Applying Euler's formula $e^{jx} = cos(x) + jsin(x)$, it is possible to obtain:

$$y_l = \sum_{i=0}^{N-1} x_i W_N^{il}, \quad 0 \leq l < N \tag{2.2}$$

where $W_N = e^{-j\frac{2\pi}{N}}$. This transform is easily reversed to obtain the time domain signal by:

$$x_i = \frac{1}{N} \sum_{l=0}^{N-1} y_l W_N^{-il}, \quad 0 \leq i < N \tag{2.3}$$

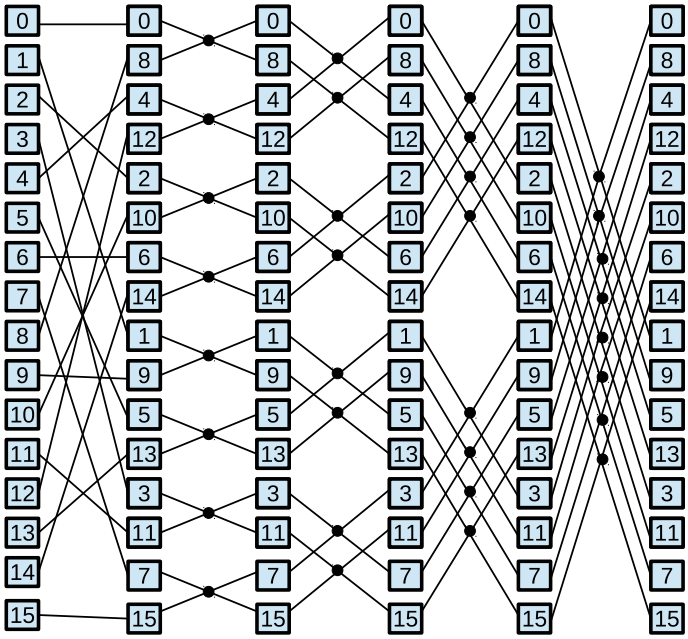
The DFT can be calculated in three different ways: *simultaneous equations*, which requires a tremendous number of calculations; *correlation*, still computing expensive,

or using the *Fast Fourier Transform* (FFT), an algorithm that decomposes a DFT with N samples into N subsignals of a single sample, being hundreds of times faster than the other methods. Specifically, getting a $O(Nlog_2N)$ time instead of the $O(N^2)$ complexity from classical algorithms. This schema follows a divide-and-conquer strategy, and many algorithms can be used, such as *Cooley-Tukey* [67] and *Stockham* [121].

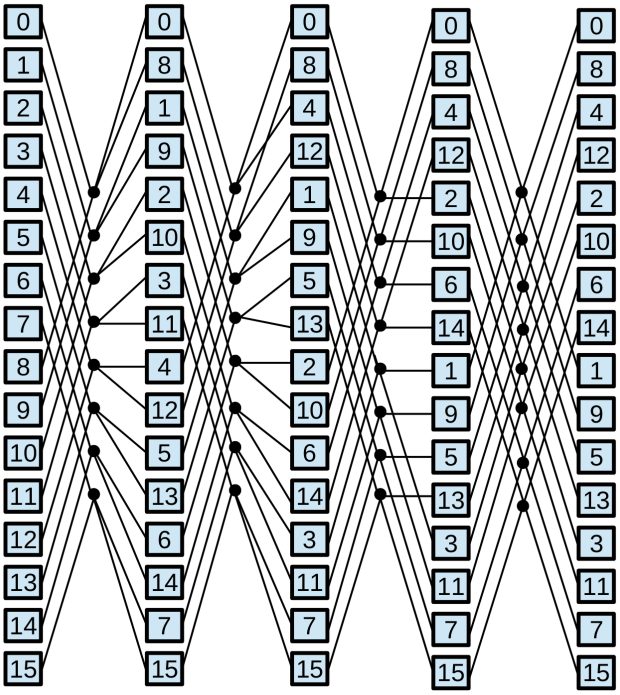
The *Cooley-Tukey* subdivides a signal of size N into two signals of half size, and repeats this procedure recursively. The pattern continues until there are N signals composed of a single point. Each time a signal is broken down into two, an *interlaced decomposition* is performed; i.e., the signal is separated into its even and odd numbered samples. This process can be seen as a reordering of the samples in the signal, where the binary index of the signals are the *reversals* of each other. For example, sample 3 (0011) is exchanged with sample 12 (1100) when considering a signal of size $N = 16$. This rearranging process can be carried out by a bit reversal sorting algorithm. The next step is to find the frequency transform of each single-sample signal. Due to signal properties, the frequency transform of a 1-point signal is equal to itself, thus there is no work involved. The last step is to combine the N signals in the exact reverse order to which the time domain decomposition took place. To combine two signals in time, each signal is diluted with zeros, and then both signals are added. This is achieved by shifting one of the time signals to the right by one sample (the same as convolving the signal with a shifted delta function). In order to perform this combination in frequency, diluting the time domain with zeros corresponds to duplicating the frequency signal, and the time shifting corresponds to multiplying the signal by a sinusoid (the transform of a shifted delta function):

$$y_l = \sum_{i=0}^{N/2-1} x_{2i} W_N^{(2i)l} + \sum_{i=0}^{N/2-1} x_{2i+1} W_N^{(2i+1)l}, \quad 0 \leq l < N \tag{2.4}$$

There are two approaches of this algorithm. If the input is bit-reversed and the output is natural order, then this is called *Decimation in time* (DIT) and, in this case, the multiplication is done before additions. Otherwise, the implementation is called *Decimation in frequency* (DFT), the multiplication is performed after additions and the output is bit-reversed. Figure 2.3 (a) presents an example of the DIT



(a) Cooley-Tukey DIT algorithm



(b) Stockham algorithm

Figure 2.3: Examples of FFT algorithms with $r = 2$ and $N = 16$.

FFT with $N = 16$.

This algorithm is an ID-algorithm with $r = 2$ by default. Thus, each Node operator reads and writes two elements in $K = n$ steps, and there are N/r Node operators per step. There is also an additional phase to compute the bit reversal. In the FFT notation, each Node operator is also known as *butterfly*, due to the shape of the operator in the prefix graph. The radix r of the algorithm can be increased in order to work with more elements by butterfly and reduce the number of computing steps.

Figure 2.3 (b) depicts the *Stockham* pattern which provides an output that is already digit reversed, so there is no need to compute an additional phase. It should be observed that the read stride in each step coincides with the write stride of the previous step. This access pattern is usually more efficient on GPU architectures. The Stockham algorithm is also an ID-algorithm with $r = 2$, which can increase the radix to compute half of the steps, as demonstrated in [82].

There are several FFT libraries for multi-core CPUs, such as *Intel's MKL* [65], the *IPP library* [66], the *Fastest Fourier Transform in the West* (FFTW) [48] or the *Spiral project* [107]. Regarding its GPU implementation, an efficient *Brook+* implementation can be found in [80] and a CUDA implementation in [128]. There are also a number of auto-tuning proposals for *GPUs*, which achieve high performance, such as [89, 90, 134]. Specifically, approaches focused on large 1D FFT on a single coprocessor include [104, 120, 134]. Another proposal for solving this problem in a sparse format is presented in [129]. However, the most widely used and well-known *GPU* implementation is *NVIDIA's CUFFT* [94].

2.2.1. The Real Fourier Transform

There is a specialized version of the FFT which works on real data [8] [18] instead of complex elements. This approach is widely used in audio processing and other fields where the input signal only takes real values. However, it should be observed that the output signal is still complex data. One approach to computing this specialized variant of the FFT is to pack the input in a vector with half of the size, by storing two consecutive real values into a single complex number, and then

use a post-processing phase to unpack the output. This is possible thanks to the symmetry and conjugate property:

$$y_l = \bar{y}_{N-l}, \quad 1 \leq l \leq N/2 \tag{2.5}$$

where $\bar{y}_{N-k}$ is the complex conjugate. Given a complex number, its conjugate is the same number but with the sign reversed $\overline{a+bj} = a - bj$. It should be noted that using this property, half of the signal information is redundant. Thus, if the input signal is packed into:

$$x'_l = x_l + x_{2l+1}j \tag{2.6}$$

then the output signal $[y_0 \dots y_{N/2-1}]$ can be obtained as:

$$y_l = \frac{1}{2}(z_l + \bar{z}_{N/2-l}) - \frac{j}{2}e^{\frac{-2\pi}{N}l}(z_l - \bar{z}_{N/2-l}) \tag{2.7}$$

where z is the complex transform of the signal x'_l. It should be noted that values in the range $[y_1 \dots y_{N/2-1}]$ have an imaginary component, but both y_0 and $y_{N/2}$ are pure real values. Due to the periodicity of z, the case of $y_{N/2}$ can be calculated as:

$$y_{N/2} = \frac{1}{2}(z_0 + \bar{z}_0) + \frac{j}{2}(z_0 + \bar{z}_0) = ReX(z_0) + ImX(z_0)j \tag{2.8}$$

The $[y_{N/2+1} \dots y_{N-1}]$ values are easily obtained applying the symmetry and conjugate property.

2.3. Tridiagonal System Solvers

Solving systems of linear equations with tridiagonal matrices arises in many scientific, engineering and computing problems, this being a highly important component in different fields, such as fluid dynamics, heat conduction, diffusion equations, numerical analysis, ocean models, cubic spline approximations and real-time applications in computer graphics. The Thomas algorithm [122] is the best-known sequential algorithm for solving these systems. Since the 1960s, a wide range of parallel algorithms for solving tridiagonal systems have been developed, among which Cyclic Reduction (CR) [61], Parallel Cyclic Reduction (PCR) [60] and Recursive

Doubling (RD) [119] are the most notable methods. Currently, these algorithms have been also implemented in GPUs, since they are used for scientific computation, providing high computational throughput and large memory bandwidth, being less expensive with rather lower power consumption than CPUs. It should be noted that many applications require solving a number of tridiagonal systems simultaneously.

There are many tridiagonal system solver implementations on GPUs. Most of them solve small problems that can be stored in the GPU shared memory, such as [23, 136], where parallelism is inherent and there is no partitioning overhead. *CUDPP* [98] is another accelerated GPU library that solves small-size tridiagonal systems and other parallel operations.

In [24], the authors first recognized that partitioning is essential for solving large matrices on GPUs, using a hybrid PCR - Thomas algorithm to do so, although this algorithm suffers from a computation overhead. Argüello et al. [3] proposed a split-and-merge method based on the CR algorithm, reducing the overhead from previous proposals. This split-and-merge approach is later refined in [16]. In [52], a partition method based on the SPIKE [109] algorithm is presented. Additionally, a diagonal pivoting method for numerical stability is first introduced in [74]. Combining QR factorization with Given rotations in [123] improved the previous implementation. In [138], a CR-based approach for solving large-problems is also presented. In [77], authors present a novel work-sharing and register blocking-based Thomas solver. Finally, *NVIDIA* implements CUSPARSE [95], a library that uses a hybrid CR-PCR implementation with pivoting for solving large-problem sizes. However, one of the disadvantages of the CUSPARSE implementation is that this preprocessing stage is often extremely slow in comparison to the runtime of the solving phase [40], also suffering from synchronization penalties [45] [78].

A different approach was presented in [82] for small problem sizes. This approach adapts the Wang and Mou algorithm [132] for CUDA-enabled *GPU* architectures. The Wang and Mou algorithm is based on the same Divide-and-Conquer strategy [105] as the SPIKE algorithm; however, in contrast to the SPIKE algorithm, the

diagonalization of each block is performed using the Gaussian elimination method, also reordering the equations in a different way.

A *tridiagonal system* (TS) is composed of N equations E_i, with $i = 1, \cdots, N$ where E_i takes the form: $a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$. The system can also be represented by its coefficient matrix, A. The b_i coefficients constitute the main diagonal of the coefficient matrix, whereas a_i and c_i are known as the lower and upper diagonals, respectively. Thus, $Ax = d$, where x and d are vectors.

$$A = \begin{pmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & 0 & \\ & a_3 & b_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & \ddots & \ddots & c_{N-1} \\ & & & & a_N & b_N \end{pmatrix}$$

In this matrix, a_1 and c_N values are zero. If $|b_i| \geq |a_i| + |c_i|$, $\forall i = 1, \cdots, N$, then the system is known as diagonally dominant. This kind of matrix guarantees numerical stability in most of the algorithms proposed in the literature.

In an iterative system solver, an equation E_i is composed of different unknowns depending on the iteration (step) k in which: $E_i^k \equiv a_i^k x_{i-u} + b_i^k x_i + c_i^k x_{i+u} = d_i^k$ is computed, where u is a function of k. It should be noted that k represents the given step of the computation, whereas K indicates the total number of steps required. For the sake of clarity, an equation E_i^k is represented in this work by a tuple of three numbers $\{i-u, i, i+u\}$ which corresponds to the indices of the unknowns that compose the equation in the step k.

2.3.1. Thomas Algorithm

The classic algorithm for solving tridiagonal systems is the Thomas algorithm [122], which is based on Gaussian elimination. The algorithm comprises two phases,

forward elimination and *backward substitution*. The first phase eliminates the first unknown in each equation (a_i coefficient in E_i equation) by

$$c_i^{k+1} = \frac{c_i^k}{b_i^k - c_{i-1}^{k+1} a_i^k} \tag{2.9}$$

$$d_i^{k+1} = \frac{d_i^k - d_{i-1}^{k+1} a_i^k}{b_i^k - c_{i-1}^{k+1} a_i^k} \quad \text{with} \quad i = 1, ..., N \tag{2.10}$$

The second phase solves the reduced system by back substitution:

$$x_N = d_N^{k+1} \tag{2.11}$$

$$x_i = d_i^{k+1} - c_i^{k+1} x_{i+1}, \quad i = N-1, ..., 1 \tag{2.12}$$

This algorithm is inherently serial, taking $2 \cdot N$ computation steps, since c_i^{k+1}, d_i^{k+1} and x_i depend on the preceding $c_{i-1}^{k+1}, d_{i-1}^{k+1}$ and x_{i+1}.

2.3.2. Parallel Algorithms for Solving Tridiagonal Systems

There are several parallel algorithms for solving tridiagonal systems, but Cyclic Reduction (CR) [61] and Parallel Cyclic Reduction (PCR) [60] are the most popular methods. Additionally, the Wang and Mou algorithm [132] is also a well-known parallel tridiagonal solver.

On the one hand, CR [61] comprises two phases, *forward reduction* and *backward substitution*, each with radix $r = 2$, thus $n = log_2 N$. Forward reduction reduces a system to another with half the number of unknowns, until a 2-unknowns system is reached in $K = n-1$ steps. Even-indexed equations are updated in parallel as linear combination of equations E_i, E_{i-1} and E_{i+1}, deriving a system of only even-indexed

unknowns by

$$a_i^{k+1} = -a_{i-1}^k s_1, \qquad b_i^{k+1} = b_i^k - c_{i-1}^k s_1 - a_{i+1}^k s_2, with \quad s_1 = \frac{a_i^k}{b_{i-1}^k} \tag{2.13}$$

$$c_i^{k+1} = -c_{i+1}^k s_2, \qquad d_i^{k+1} = d_i^k - d_{i-1}^k s_1 - d_{i+1}^k s_2, with \quad s_2 = \frac{c_i^k}{b_{i+1}^k} \tag{2.14}$$

where k denotes the step of the algorithm. In each step of backward substitution, odd-indexed unknowns x_i are solved in parallel by substituting the previously solved x_{i-1} and x_{i+1} values to E_i equation in $K = n$ steps.

On the other hand, PCR [60] is a modification of CR that only has the forward reduction phase, with the same formula and updating mechanism as CR, but reducing each of the current systems to two half-sized systems. For example, for an 8-unknown system, the first step obtains two 4-unknown systems, then next step reduces the two 4-unknown systems to four 2-unknown systems. When each system has 2 unknowns, then it is possible to solve them and the algorithm is finished after $K = n$ steps.

Figure 2.4 (a) and (b) show the CR and PCR algorithms, respectively, for a problem size of $N = 16$ elements, where each $i-$box represents the E_i equation; the x_i-boxes shows the unknowns vector; and the black circles the Node operators. Each Node operator, also known as *reduction* in the TS notation, performs the coefficient updating for an equation, as explained above. Both algorithms are paralell prefix algorithms of radix $r = 2$, but not ID-algorithms. The number of Node operators is not constant along steps and the same element is shared by several Node operators. In the case of CR, the *fan in* of each Node operator is three and the *fan out* is one; the same as in the PCR case. However, the final step of PCR uses a Node operator with both *fan in* and *fan out* equal to two in order to perform the substitution. The number of Node operators in the CR forward phase is $\frac{N}{2^k}, \quad 1 \leq k \leq K$; and $\frac{N}{2^{K-k+1}}, \quad 1 \leq k \leq K$ in the CR substitution phase. In the PCR algorithm, the number of Node operators in each step k is N with $1 \leq k \leq K-1$, except for the final step, where this number is $\frac{N}{2}$.

As can be observed in Figure 2.4, these methods do not allow the problem to be partitioned into independent chunks, as elements that take part in a reduction are used in two different Node operators in the same computing step. In the case of CR, it takes 3 computing steps to perform the forward reduction of the example, and 4 steps for the backward reduction; whereas PCR only needs 4 computing steps for the whole process.

In addition to these algorithms, the Wang and Mou (WM) algorithm, an ID-algorithm with radix $r = 2$ and shown in Figure 2.4 (c), divides the computation into $K = n$ steps, and each Node operator works on triads of equations, labeled *L*eft, *C*enter and *R*ight, represented as:

$$[i]^{k-1} = [\ \underbrace{E^{k-1}_{q\cdot 2^{k-1}}}_{L_i}, \underbrace{E^{k-1}_{i}}_{C_i}, \underbrace{E^{k-1}_{(q+1)2^{k-1}-1}}_{R_i}\ ] \tag{2.15}$$

where $q = \lfloor i/2^{k-1} \rfloor$ and the equation i-th in $k-1$ step is of the type:

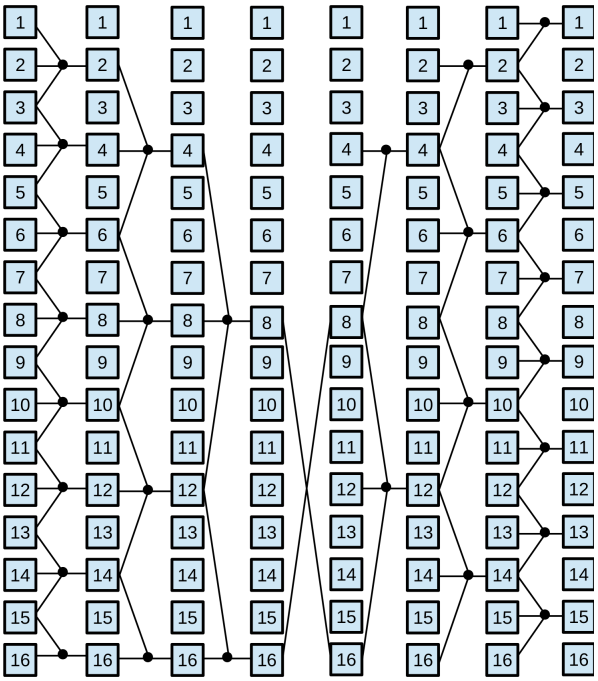
$$E^{k-1}_i = \{a^{k-1}_i x_{q2^{k-1}-1} + b^{k-1}_i x_i + c^{k-1}_i x_{(q+1)2^{k-1}} = d^{k-1}_i\} \tag{2.16}$$

Figure 2.5 depicts the reduction of two elements (triads) that take part in a Node operator. In addition to this, when all elements are stored in the same memory space, it is possible to only work with the Central equation, C_i, instead of storing the whole triad, since L_i and R_i are easily obtained when necessary as follows:

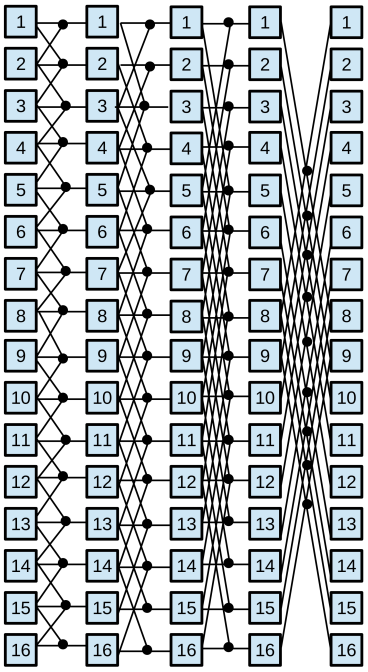
$$L_i = C_a \rightarrow a = 2^k \times \lfloor i/2^k \rfloor \tag{2.17}$$

$$R_i = C_b \rightarrow b = 2^k \times (1 + \lfloor i/2^k \rfloor) - 1 \tag{2.18}$$

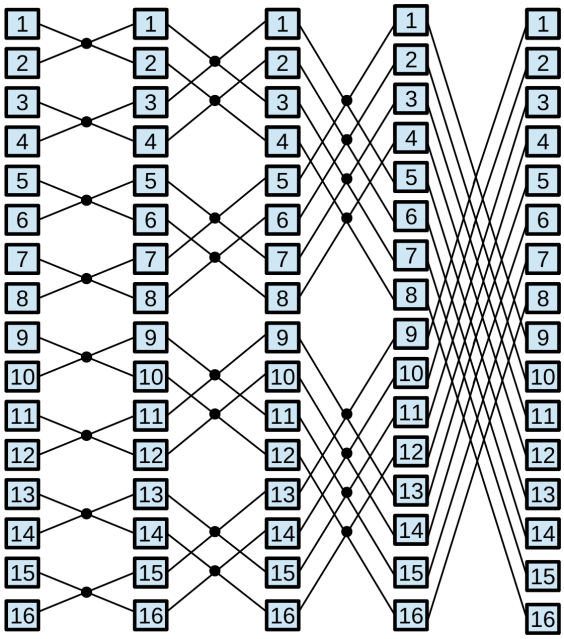
This algorithm is easily partitioned among different independent chunks, if necessary.



(a) Cyclic Reduction



(b) Parallel Cyclic Reduction



(c) Wang and Mou algorithm

Figure 2.4: Patterns for different tridiagonal system solvers with $N = 16$ elements.

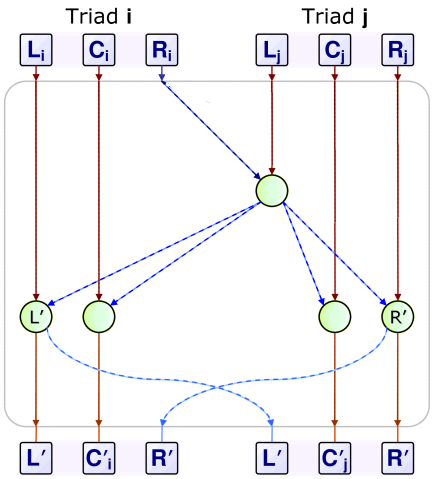
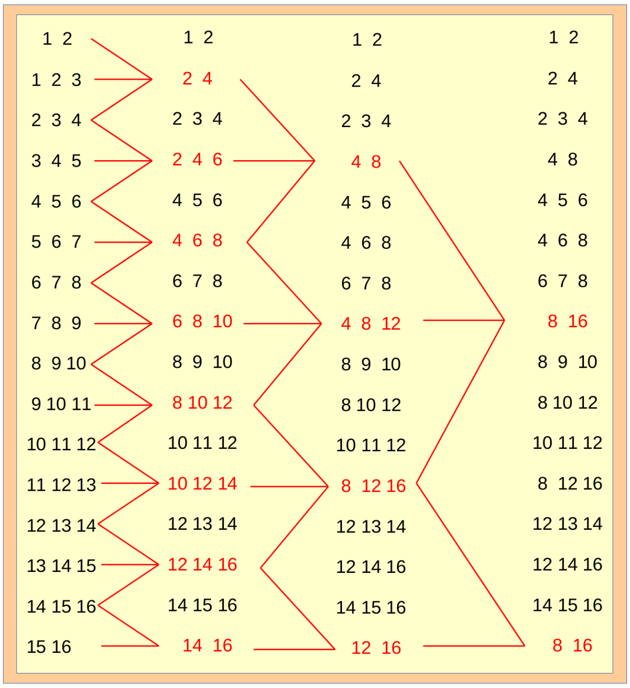


Figure 2.5: Reduction of two triads in the Wang and Mou algorithm

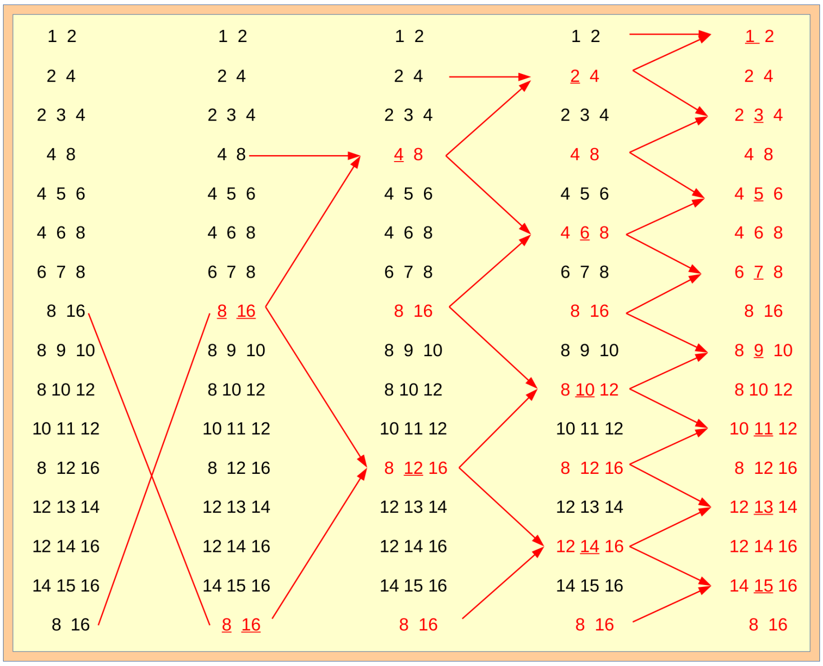
2.3.3. The Partitioning Problem

As introduced above, partitioning the system is crucial for being over memory limits. In order to efficiently solve the problem on distributed platforms, each private-memory system of the distributed platform must process a subset of equations as independently as possible, to avoid communication latency. In this work, each subset of equations, which is computed in a private-memory space, is called a *slice*. However, most of the iterative algorithms cannot be easily partitioned. In the case of the Cyclic Reduction (CR) method, equations that take part in a reduction may belong to different slices. Specifically, the equation E_i^k, at step k, is the result of reducing the $[E_{i-u}^{k-1}, E_i^{k-1}, E_{i+u}^{k-1},]$ equations, with $u = 2^{k-1}$.

Figure 2.6 shows an example of the CR method for $N = 16$ equations. Specifically, Figure 2.6 (a) depicts the forward reduction phase, where the equations shown in bold in each step are the result of reducing three equations from the previous step, and Figure 2.6 (b) shows its substitution phase. As can be observed in Figure 2.7 for a coefficient matrix, the problem cannot be directly partitioned into independent slices (marked with horizontal lines in figure), as equations need other slice equations to be reduced. This reduction schema is the same for the PCR method, although the number of equations which are reduced per step is higher in PCR, compared with CR. Regarding WM, an equation E_i^k, at step k, is the result of reducing the $[E_i^{k-1}, E_{i+u}^{k-1}]$ equations, with $u = 2^{k-1}$.

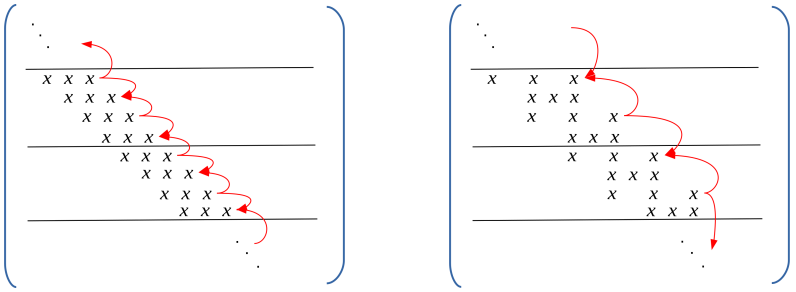


(a) CR forward reduction



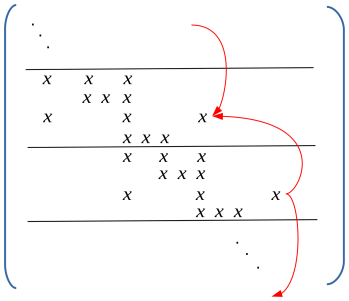
(b) CR backward substitution

Figure 2.6: Cyclic Reduction example for $N = 16$ elements



(a) Step $k = 1$

(b) Step $k = 2$



(c) Step $k = 3$

Figure 2.7: Equation dependencies among slices in the coefficient matrix for the Cyclic Reduction algorithm

2.4. Scan Operator

The *scan* operator [56] is defined as an associative and binary operator $\oplus$ with identity I, where, given an input array of N elements $[a_0, a_1, ..., a_{N-1}]$, it returns

$$[I, a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{N-2}) \ ]$$

The scan operator defined here is an *exclusive* scan, since the a_j element is not taken into consideration for calculating the position j of the result. Otherwise, it would be an *inclusive* scan, although the transformation from one to the other is trivial. Hereinafter, we will use addition as scan operator in our examples.

The algorithm performs N adds for arrays of N elements; thus it possesses an *O(N) work complexity* and the calculation of the i-element requires of the calculation of $i - 1$ elements.

The scan operator is widely used in areas such as the construction of summed area tables [87], stream compaction [114], sorting [56], image filtering [87], Brownian

values generation [106], polynomial evaluation [41] or cryptography [130], among many others.

The patterns here expouned herein are parallel prefix algorithms, whose properties have already been explained. Scan primitive in VLSI adders was proposed by Sklansky in 1960 [116]. Most implementations on GPU are based on either the Kogge-Stone or the Brent-Kung parallel prefix patterns, although also exists another GPU approach based on VLSI adders, which was developed by Han-Carlson in 1987. Figure 2.8 shows a taxonomy of these existing parallel prefix implementations based on VLSI adders.

The analysis of different proposals for the scan operator permits us to classify them in terms of their prefix graph, which enables us to describe the operations carried out on the data.

The following approaches provide a solution focused on just the scan primitive; however, there is a growing trend towards using accelerated libraries that solve this and other parallel operations. In the case of the scan primitive, most of these libraries use hybrid approaches which combine several prefix algorithms with high-efficient CUDA optimization techniques. An example of these accelerated libraries, which solve the scan, are *CUSPARSE* [95], *CUDPP* [98], Merrill's *CUB* [100], *Thrust* [101] and *ModernGPU* [97].

2.4.1. Brent-Kung Pattern

The Brent-Kung pattern [12] [9] reduces the complexity through the use of two balanced binary trees of radix $r = 2$. This algorithm performs $K = 2 \cdot log_2 N$ steps and the work complexity is $O(N)$. Once the input tree is built, the next step is to sweep it to and from the root in two phases: *up-sweep phase* and *down-sweep phase*. Figure 2.9 shows this pattern for $N = 8$ elements. The *up-sweep* phase computes partial sums at internal nodes, whereas the *down-sweep* phase uses previous partial sums for building the scan in place. In the *down-sweep* phase, there also exists a step where values are passed to its child node (dashed line notation in Figure 2.9).

The Brent-Kung pattern is a VLSI area efficient method that uses two balanced binary trees to obtain this efficiency. Blelloch developed this primitive in an efficient

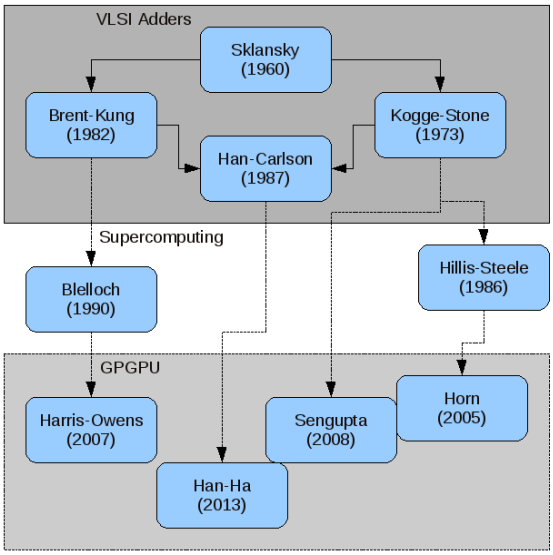


Figure 2.8: Taxonomy of the existing parallel algorithms for scan operator based on VLSI adders.

way for supercomputing in 1990. This pattern was first implemented for GPUs in [56].

2.4.2. Kogge-Stone Pattern (KS)

The Kogge-Stone pattern [72] was used as a parallel carry look-ahead adder. This takes up more area than Brent-Kung adder but this layout has a minimum depth, which increases performance. The work complexity is observed as $O(N \cdot \log_2 N)$ and takes $K = \log_2 N$ steps. Figure 2.10 depicts the KS pattern for $N = 8$ elements. However, this pattern is considered work inefficient in comparison to the serial implementation which is bounded by $O(N)$.

The Kogge-Stone VLSI adder pattern [72] was designed for a small VLSI area, as it is work inefficient. In [58], this algorithm was adapted for supercomputing with $O(N \cdot \log_2 N)$ complexity. This algorithm was demonstrated for GPUs in [63] who used this scan for a non-uniform stream compaction operation as well as for a collision-detection application. A scan for summed-table area generation was subsequently proposed in [57], improving the implementation of [63] by pruning

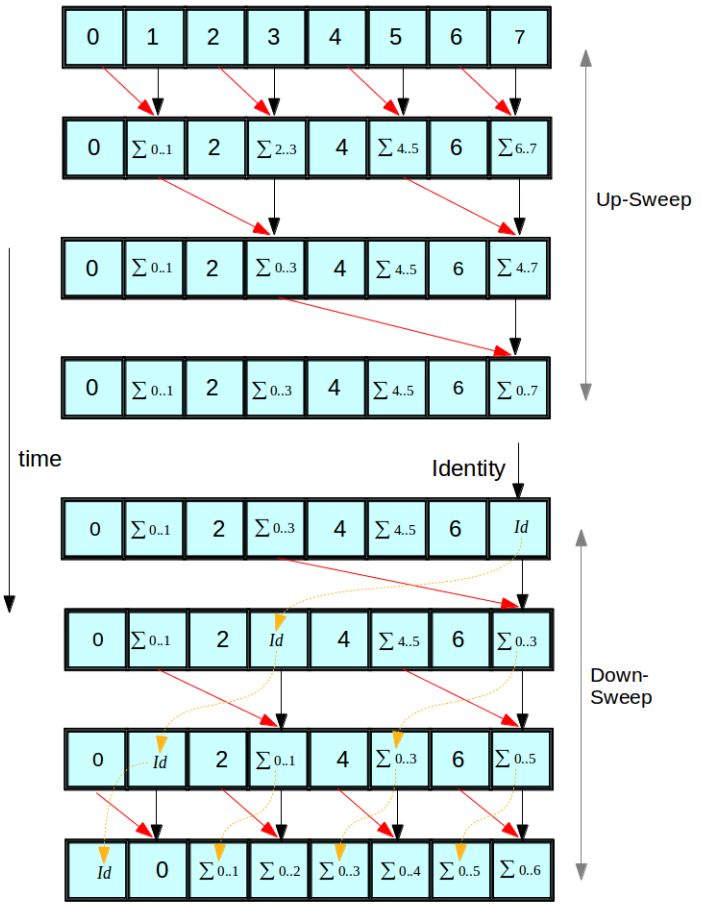


Figure 2.9: Brent-Kung pattern for addition with $N = 8$.

unnecesary work, but still obtaining a $O(N \cdot \log_2 N)$ complexity. The first $O(N)$ implementation was published in 2006 [114], emphasizing the algorithm's depth as a key performance parameter in GPU implementation [112]. The funcionality to combine the Kogge-Stone and Brent-Kung patterns was presented in [85], and a strategy for computing large arrays was introduced in [39].

2.4.3. Han-Carlson Pattern

The Han-Carlson pattern [54], see Figure 2.11, combines both the Kogge-Stone and Brent-Kung methods, seeking a tradeoff between area and time required. This

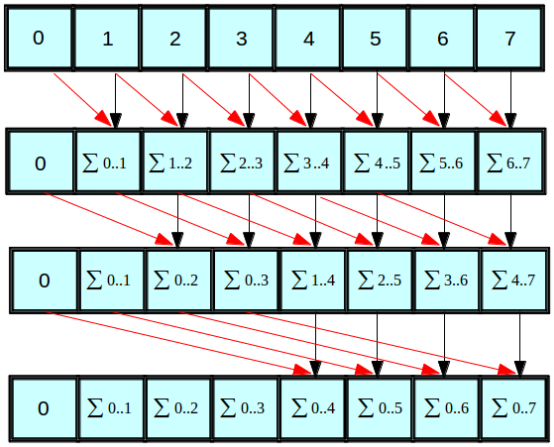


Figure 2.10: Kogge-Stone pattern for addition with $N = 8$.

pattern makes it possible to obtain a tradeoff between area and time required. At the beginning and at the end, it executes Brent-Kunt steps, whereas Kogge-Stone works in the middle of the graph. However, this proposal also offers an $O(N \cdot \log_2 N)$ complexity. In 2013 [53], a GPU work efficient algorithm based on this third pattern was proposed.

2.4.4. Ladner-Fischer Pattern

Additionally, the Ladner-Fischer pattern was introduced in [75] as a parallel solution of Boolean combinational circuits and finite-state transducers, taking $K = \log_2 N$ steps. The algorithm is based on the Brent-Kung reading stage, but computing a block of 2^k adjacent positions for each element in the k step. Unlike other algorithms, the number of read and write operations remains constant over all steps. Figure 2.12 shows this pattern for $N = 8$ elements.

2.5. Sorting Algorithms

Sorting is a computational building block of high importance, it being one of the most studied algorithms due to its impact. Many algorithms rely on the efficiency of sorting routines as the main pillars of their efficiency. For example, sorting is

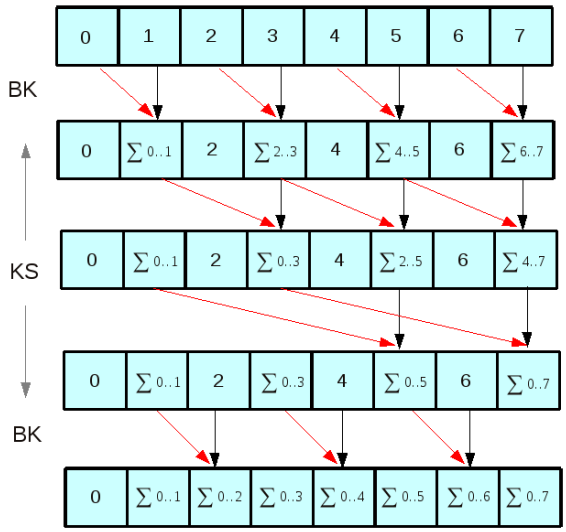


Figure 2.11: Han-Carlson pattern for addition with $N = 8$.

widely used in computer graphics and geographic information systems for building spatial data structures, as well as a basis for solving sparse matrix operations or MapReduce patterns [25]. Sorting is also applied in database queries [50] and for collision detection in physics simulation, among many others.

There are several parallel sorting algorithms, such as Radix sort [137], Mergesort [69], Bitonic sort [7], and Quicksort [59]. Furthermore, many of these algorithms have been developed for GPUs. Radix sort was efficiently implemented on a GPU in [56]. Quicksort algorithm was first implemented on a GPU in [113], being improved in [15] and [83]. A hybrid algorithm that combines Mergesort and Bucketsort [14] was presented in [115]; whereas new implementations based on Radix sort and Mergesort were developed in [110]. There are several *accelerated* libraries that integrate sorting routines inside a set of different algorithms. As example of these libraries, we can find *CUDPP* [98], *CUB* [100] and *ModernGPU* [97], where *ModernGPU* is currently the fastest one on small problem sizes, although all of these libraries were developed focusing on large problem sizes.

Bitonic Merge Sort is a parallel prefix algorithm with radix $r = 2$ for sorting [7]. The classic complexity is of $N \cdot (logN)^2$. Figure 2.13 shows the classic algorithm representation for $N = 16$ where each horizontal line represents a key value, starting at the left end and finishing at the right end. Vertical segments are the Node oper-

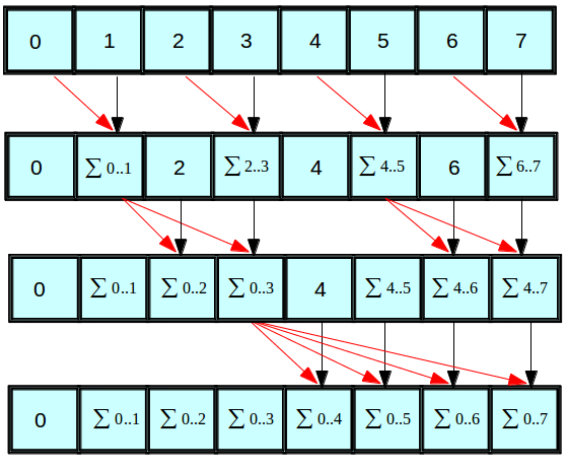


Figure 2.12: Ladner-Fischer pattern for addition with $N = 8$.

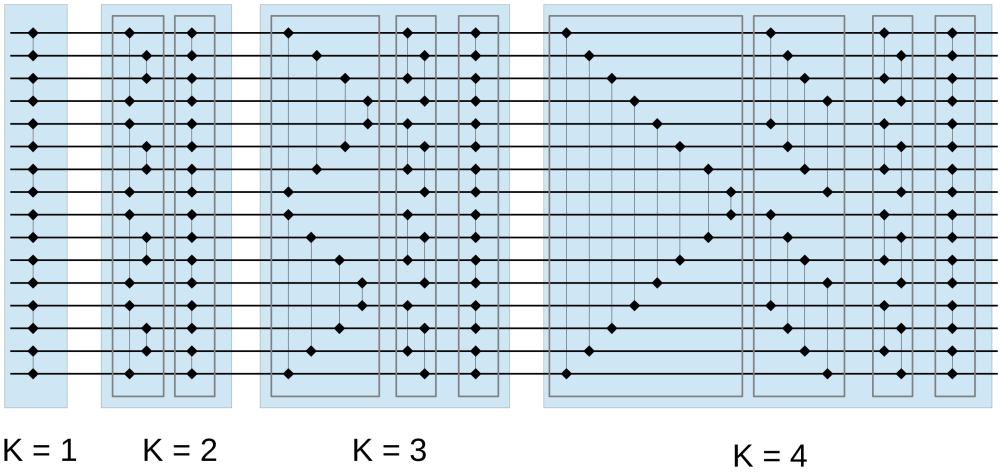


Figure 2.13: Bitonic Merge Sort Algorithm with $N = 16$.

ators, also known as comparators in sorting notation, which make the comparison of the two selected keys, swapping their values if necessary. Both the *fan in* and *fan out* is two in this Node operator. The sorting is processed along $K = log_2 N$ steps (blue boxes) where step k has k internal steps (line rectangles) incurring in $O(N \cdot (logN)^2)$. This process is repeated until a single N-element sorted sequence is obtained.

2.6. CUDA Notation for Paralell Prefix Algorithms

These parallel prefix algorithms can be implemented in any parallel paradigm, although the following chapters are focused on their usage in GPUs, to which this work is devoted. In order to work with these algorithms in CUDA, it is necessary to identify the different parameters that take place in its design. Table 2.2 provides a global summary of the basic parameters employed here. Depending on the strategy followed to compute the different problem sizes, this list of parameters is extended, as well as new relations are established among them in the remaining text.

Each problem has size $N = r^n$, and $G = r^{batch}$ *batches* of the same length are concurrently solved. The batch data are divided among $B = r^b$ threadblocks, and each of these threadblocks executes $L = r^l$ threads. A thread performs the calculation of $P = r^p$ elements stored in private registers and threads within a threadblock have access to $S = r^s$ data stored in shared memory. Here, b may be formed by two coordinates $b = (b_x, b_y)$. In a similar manner, l may be composed of three coordinates (l_x, l_y, l_z). Finally, the l parameter can be related with s and p using $s = p + l$, as all threads within a threadblock usually have a copy in shared memory of the elements stored in their registers. It should be observed that P represents the number of elements stored by each thread in registers, but does not specify the size of the element datatype. For example, one complex element occupies 8-bytes in the FFT algorithm; whereas a single-precision equation occupies 32 bytes in TS, and 4 bytes are used per element in the scan operator; however, all of them are considered as a single element when establishing P in our notation. The same occurs with the S definition: it only considers number of elements but not their size.

One thread is responsible for computing one Node operator in the CUDA implementation, working with as many elements as the *fan in / fan out* values specify. However, if the given architecture allows more than these elements to be stored in registers, without an SM occupancy penalty, it may be interesting to process more elements per thread. There are two options. One option is to process more than one Node operator per thread. It should be observed that this only affects the implementation, both the same r and number of steps taken are maintained for the algorithm, only the number of threads to compute the problem is reduced when increasing the number of Node operators per thread. In this case, the number of Node operators

Parameter	*Definition*
$N = r^n$	Problem size.
$G = r^{batch}$	Number of problems being simultaneously solved.
$P = r^p$	Number of elements stored in registers per thread.
$B = r^b$	Number of thread blocks per stage, where $B = B_x \cdot B_y$
$L = r^l$	Number of threads per thread block, where $L = L_x \cdot L_y \cdot L_z$
$S = r^s$	Number of shared-memory elements per thread block

Table 2.2: Description of the GPU parameters used.

per thread is given by $\frac{P}{max(fan\ in, fan\ out)}$. Another option to force working with more elements per thread is to increase the radix r of the algorithm, where each thread continues working with a single Node operator. This option changes the definition of the algorithm, not only the implementation, as $N = r^n$, decreasing the number of steps taken, K.

As mentioned above, most of the parallel prefix are defined with $r = 2$, and only Index-Digit algorithms can easily extend its radix definition to a higher value. Thus, taking the algorithms used in this work into account, we work with a radix $r = 2$ implementation for all parallel prefix algorithms which are not ID-algorithms. To increase the number of elements to be stored in registers, the first explained option is used. However, the radix r can be increased to a higher value when working with ID-algorithms, if considered desirable, employing the second option.

In the case of an arbitrary N (not power of r), our methodology is easily extended. There are two alternatives. One approach is to compute the the smallest power of r able to compute N, padding those extra locations with the identity for the given operation. This approach may be expensive for some operations and problem sizes; in such case, the following approach is used. If N is not a power of r, then it can be expressed as $r^n + N'$. On the one hand, r^n is computed following the explained methodology. On the other hand, the smallest power of r able to compute N' is executed, and then r^n and N' data are integrated into one step.

Chapter 3

New Parallel Prefix Algorithms

As seen so far, this work is composed of two different perspectives. On the one hand, the algorithmic perspective, which chooses efficient parallel prefix algorithms to solve several common parallel problems in computer science. On the other hand, this work also provides the high performance computing perspective: an efficient CUDA methodology for the selected algorithms.

In this chapter, a number of new parallel prefix algorithms are presented. These new algorithms have been created and designed in this work, being novel to the best of our knowledge, and their aim is to solve certain parallel problems in the most efficient possible way. Although they were designed focusing on the GPU computing model, they can be implemented in any other parallel programming paradigm. In the following chapters, an efficient CUDA tuning methodology is proposed for different parallel prefix algorithms and sizes.

Specifically, two new different algorithms have been designed to solve tridiagonal systems: *Redundant Reduction* (RR) and *Tree-Partitioning Reduction* (TPR); and a new algorithm for sorting, *Bitonic Merge Comb Sort* (BMCS), is also proposed. The work presented in this chapter was originally introduced in [31], [32] and [34].

3.1. Redundant Reduction: A New Algorithm for Solving Tridiagonal Systems

As previously introduced in Chapter 2, solving many tridiagonal systems simultaneously in parallel is critical in many applications, such as combustion and chemical simulation models. However, many solvers were designed to deal with large systems, proving to be inefficient when many problems of small size are processed simultaneously. Furthermore, complex systems are commonly transformed into multiple small independent systems, rendering the original problem more manageable. Thus, it is important to solve small problem sizes as well as multiple problems simultaneously. Accordingly, the aim of this section is to provide an efficient GPU approach to address the solution of many problems simultaneously in a single invocation of the library.

Although the most common parallel algorithms for solving tridiagonal systems are CR, PCR and the Wang&Mou, this work creates and develops a new reduction method for solving small tridiagonal systems, which is better suited to the GPU architecture and performs fewer computing steps. The main idea of this new method is to use a communication pattern well-suited to the GPU architecture. Following this idea, we have created a new operation for reducing two equations, which is called *Redundant Reduction* (RR), introduced in [32].

This operation is combined with a communication pattern from a parallel prefix pattern. Specifically, our proposal uses two different prefix patterns: Kogge-Stone [72] (*RR-KS*) and Ladner-Fischer [75] (*RR-LF*), resulting in our work being capable of surpassing the performance of the state-of-the-art.

3.1.1. The Redundant Reduction Operation

The idea of this new operation is that each Node operator reads two equations, instead of three, to perform the reduction. This implies less accesses to memory, and thus, lower latencies in the execution. Additionally, the whole algorithm (i.e., this operation plus the communication pattern) must be able to compute the overall result in $log_2 N$ steps, and the substitution phase is only one step.

The RR operation performs the reduction over a pair of equations $\{E_i^k, E_j^k\}$ with $i = j - l$ and where the value of l depends on the k step and the communication pattern. With this operation, the reduction of two equations, E_i^k and E_j^k, only updates the coefficients of the E_j^{k+1} equation, with the other equation coefficients remaining constant ($E_i^{k+1} = E_i^k$). Specifically, the following two equations

$$\left.\begin{array}{ll} E_i^k: & a_i^k x_{i-l} + b_i^k x_i + c_i^k x_{i+l} = d_i^k \\ E_j^k: & a_j^k x_{j-l} + b_j^k x_j + c_j^k x_{j+l} = d_j^k \end{array}\right\} \xrightarrow{j=i+l} \begin{array}{llll} a_i^k x_{i-l} + & b_i^k x_i + c_i^k x_{i+l} & & = d_i^k \\ & a_j^k x_i + b_j^k x_{i+l} + c_j^k x_{i+2l} & & = d_j^k \end{array} \tag{3.1}$$

have two common unknowns $\{x_i, x_{i+l}\}$, with two possible reductions. Eliminating either the x_i unknown (called *Reduction A* in our work) or the x_{i+l} unknown (*Reduction B*). In the RR operation, we propose that each E_i^k equation is represented by a complex format with two forms, I_i^k and C_i^k, such that $E_i^k = I_i^k \cup C_i^k$. Each one of these forms is an auxiliary equation, equivalent and representative of E_i^k, but necessary for performing our proposal. I_j^{k+1} is obtained by reducing I_i^k with C_j^k, following the *Reduction B* case, and this intermediate result is also reduced with I_j^k (*Reduction A*). To obtain C_j^{k+1}, firstly I_i^k is reduced with C_j^k by the *Reduction A*, and its result with C_i^k (*Reduction B*). Figure 3.1 shows the Redundant Reduction scheme for a pair of equations in the step k, where each equation is represented by the two forms. In this figure, the I_i^k equation is reduced with the C_j^k using the *Reduction A* and the *Reduction B* possibilites (A,B circles). The resulting equation from the *Reduction A* is processed with C_i^k using the *Reduction B* method, generating C^{k+1}, and the resulting equation from the *Reduction B* is reduced with I_j^k using the *Reduction A* method, obtaining I_j^{k+1}.

After the final step of the algorithm, the second *form* of the last complex equation, $C_N^{log_2 N-1}$, contains the value of the unknown x_1. The other E_i equations, with $i = 1, \cdots, N-1$, will have a two-unknown equation where one unknown is x_1 and the other one is x_{i+1}. At this point, all unknowns can be solved simultaneously in only one step.

Regarding the CUDA implementation of this operation, the Node operator reads two equations, E_i^k and E_j^k, from global memory to registers. After each processing step, the Node operator stores E_j^{k+1} in shared memory for the next step. Since E_j^{k+1}

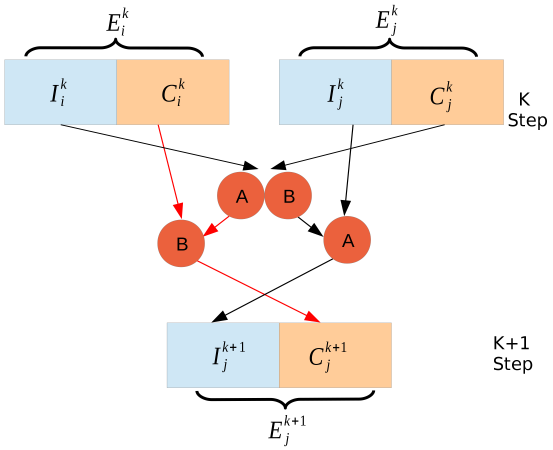


Figure 3.1: Redundant Reduction scheme for E_i^k and E_j^k where A,B circles denote the resulting equation of applying *Reduction A* or *Reduction B*, respectively.

is the only equation updated in the operation, only I_j^{k+1} and C_j^{k+1} forms are stored.

3.1.2. Redundant Reduction Algorithm using the Kogge-Stone Pattern

The solver presented in this section (*RR-KS*) is the result of combining RR with the Kogge-Stone [72] pattern in the reduction step. Figure 3.2 depicts the RR-KS approach for $N = 8$, where each box number represents an E_i equation with its two different forms, I_i and C_i. This algorithm has $log_2 N$ forward steps, each with $N - 2^k$ Node operators ($k = \{0, \cdots, log_2 N - 1\}$) and only one substitution step.

Figure 3.3 shows the pseudocode of our *RR-KS* proposal. In this code, lines 2-5 contain the computation of the first step. As initial equations are specified in a single form, each E_i equation is reduced with the E_j equation by *Reduction A* and *Reduction B*, obtaining the *I form* and the *C form* of the E_{i+1}^1 equation, respectively. The *RR* function updates the new equation coefficients. Its first and second arguments are the equations to be reduced, whereas the third argument represents the resulting equation; finally, the fourth argument indicates the reduction case: *Reduction A* or *Reduction B*. Lines 8-16 compute the rest of steps, according to the explained reduction scheme. Line 20 obtains the x_1 unknown's value, and lines 22-24 perform the parallel substitution for the others unknowns.

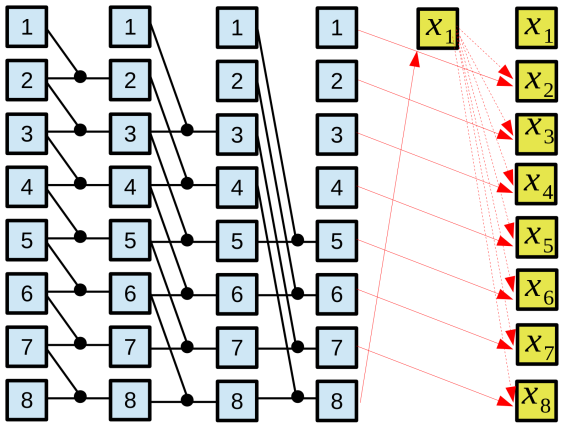


Figure 3.2: Reduction and substitution steps in RR-KS for N=8 equations.

```
1  //First step
2  for i=1 to N-1 in parallel
3          RR(E_i, E_{i+1}, I_{i+1}, optionA)
4          RR(E_i, E_{i+1}, C_{i+1}, optionB)
5  end
6
7  //Other steps
8  for j=2:j<N:j*=2
9          for i=1 to i ≤ N − j in parallel
10           RR(I_i, C_{i+j}, auxA, optionA)
11           RR(I_i, C_{i+j}, auxB, optionB)
12
13           RR(auxB, I_{i+j}, I_{i+j}, optionA)
14           RR(auxA, C_i, C_{i+j}, auxB, optionB)
15         end
16 end
17
18 //Substitution step
19
20 x_1 = C_N.d/C_N.b
21
22 for i=1 to N-1 in parallel
23         solve_unknown(C_i, x_1, x_{i+1})
24 end
```

Figure 3.3: *RR-KS* algorithm pseudocode

As explained, facts that influence GPU performance include coalescence issues in global memory accesses and shared memory bank conflicts. A good feature of *RR-KS* algorithm is the shared memory communication pattern: elements within a warp access consecutive memory directions; hence it does not generate bank conflicts. In a similar manner, data loading from global memory is performed following coalesced patterns, where adjacent threads access adjacent memory elements, minimizing the

number of transactions in global memory. To this end, one different vector is built for each coefficient (obtaining 4 different arrays a, b, c, d). To exploit the coalescing pattern, each thread reads two consecutive elements in a single memory transaction through CUDA (*float2*, *float4*) datatypes, when working on simple precision. Owing to the algorithm's communication pattern, the same equation is read and written by different threads in the same step, with the use of synchronization barriers being necessary. In algorihtms of this type, *memory bound* problems, the global memory bandwidth may become a limiting factor due to the enormous use made of it. In order to deal with this, as will be shown in Section 3.1.4, our implementation for Kepler and Maxwell architectures takes advantage of their read-only data cache, providing extra bandwidth performance owing to its separated pipeline.

3.1.3. Redundant Reduction Algorithm using the Ladner-Fischer Pattern

In this section we propose a second new solver based on our Redundant Reduction operator and the Ladner-Fischer [75] prefix pattern (RR-LF), which has obtained very good performance results for GPU architectures previously [26]. Figure 3.4 depicts the RR-LF approach for $N = 8$ (the substitution step is the same as the one in the *RR-KS* algorithm), where each box number represents an E_i equation with its two different forms. As happens with *RR-KS*, the reduction is performed with 2 equations, sharing the same reduction scheme (see Figure 3.1) but changing the communication pattern. It has $log_2 N$ forward steps, each with $\frac{N}{2}$ Nodes operators.

Regarding its CUDA implementation, it should be noted that the algorithm needs to store two different forms per equation, but unlike the *RR-KS* algorithm, the number of active threads remains constant along steps, skipping the conditional instructions used to control the thread identifier. However, as result of this *non-divergent* pattern, a small rate of shared memory bank conflicts appears in the execution due to the fact that consecutive threads are not accessing to adjacent shared memory banks. Different padding techniques have been implemented to avoid these conflicts, although the index calculation overhead is not worth the small latency generated by conflicts.

Furthermore, Kepler architecture offers the possibility of 8-byte shared memory

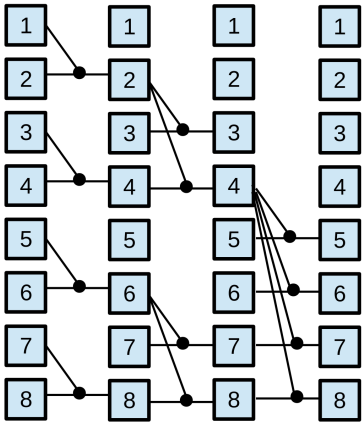


Figure 3.4: Reduction step in *RR-LF* for *N=8* equations.

accesses, as mentioned in Section 1.2.2 of Chapter 1, which significantly reduces the number of bank conflicts with this configuration. In addition to the saving in conditional instructions, this pattern also reduces the number of synchronization barriers, as the same equation is never read and written simultaneously by different threads. Concerning global memory bandwidth, the Kepler and Maxwell implementations also use the benefits of their read-only data cache when reading the coefficients of the input arrays.

3.1.4. Experimental Results for the RR operation in CUDA

In this section, the results of our two proposals on different *NVIDIA* GPU architectures are presented and analyzed. All tests were run in single precision using input arrays in the range $N = \{64, 128, 256, 512, 1024\}$ over 100 iterations for different batch sizes. The test platforms used in our experiments are described in Table 3.1. In our tests, we have used a diagonally dominant system and test data are already on the GPU, thus there are not data transfers during the benchmarks. Additionally, Chapter 4 also presents an implementation of this algorithm under the proposed CUDA tuning methodology of this work.

One of the main requirements for GPU performance is to explicitly achieve the right balance between the high number of simultaneous warps and the proper utilization of the SM shared resources. The number of warps that can be executed by each

	Fermi Platform	Kepler Platform	Maxwell Platform
CPU	Intel Core i7-2600 3.4 GHz	Intel Xeon E5-2660 2.2 GHz	Intel Core i7-2600 3.4 GHz
Memory	8 GB DDR3 1333	64 GB DDR3 1600	8 GB DDR3 1333
OS	Ubuntu 12.04 LTS	CentOS 6.4	Ubuntu 12.04 LTS
Compiler	GCC 4.4.7	GCC 4.4.7	GCC 4.6.3
GPU	GeForce GTX580	Tesla K20	GeForce GTX980
Driver	304.116, SDK 5.0	320.17, SDK 5.0	343.22, SDK 6.5

Table 3.1: Description of the test platforms for the RR algorithms

SM is limited by the amount of shared memory bytes per threadblock and the maximum number of registers per thread. Without considering temporal data storage used by the compiler, our Node operator requires at least $2 \times 2 \times 4 \times sizeof(datatype)$ bytes in registers, owing to 2 equations are read by the Node, each equation has two forms and each form has four coefficients, respectively. However, elements stored in registers do not become a limiting factor in our implementations, as there are enough registers without any case of spilling. On the other hand, the amount of shared memory reserved for each block in the RR operation is $N \times 2 \times 4 \times sizeof(datatype)$, as they have N elements, each with two forms and 4 coefficients per form. Therefore, the maximum size that may be processed with our proposal is $N = 1024$ (which implies 32768 shared memory bytes per threadblock). This limit is given by the maximum shared memory that can be allocated for a single block. In Fermi and Kepler architectures, the size of shared memory per SM is 48 KB per multiprocessor, whereas Maxwell architecture allows 96 KB per SM. Although Maxwell allows up to 96 KB per SM, the threadblock limit remains 48 KB. Larger problems would require a different approach, which will be studied in next chapters.

Figure 3.5 shows a performance comparative on the Fermi Platform between our proposals, the *CUSPARSE* library [95] and *CUDPP* library [98], the most well-known libraries to solve tridiagonal systems on GPU for small problem sizes. These results were taken for $G = 256$ batches. Note that 1024 equations is the maximum number that can be processed per batch due to the shared memory restriction. As can be observed, our two algorithms offer a clear advantage over *CUSPARSE* library, proving to be around three times faster. For problem sizes between $N = 128$

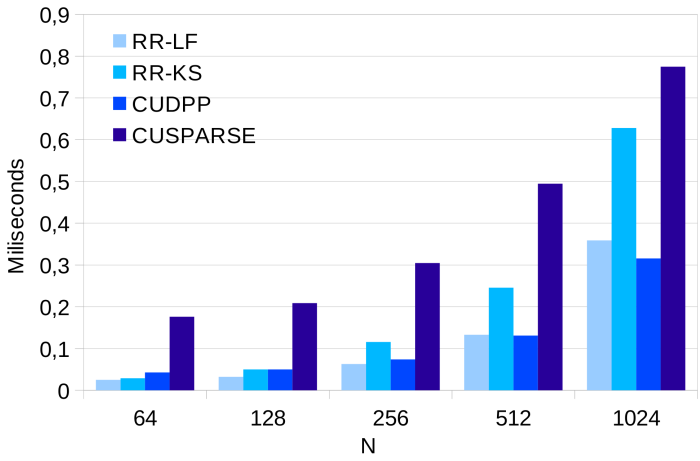


Figure 3.5: Performance results on the Fermi Platform for $G = 256$ *batches*.

and $N = 512$, the *RR-KS* algorithm is, on average, 1.7x faster than *CUSPARSE*. A competitive result is also obtained when $N = 1024$. *RR-LF* provides better performance, achieving 3.25x when $N = 256$. Regarding *CUDPP*, *RR-KS* improves it up to $N = 128$ and *RR-LF* is better when N is lower than 512. In both proposals, the limiting factor for achieving higher block occupancy is the shared memory, since both algorithms need to store two forms per equation. Thus, when N is higher than 256, occupancy decreases and *CUDPP* is faster in these cases. Concerning our proposals, the main advantage of the *RR-LF* algorithm over the *RR-KS* algorithm is the avoidance of several synchronization barriers, since the same equation is never read and written by different threads. As each SM has only 32 SP in Fermi, avoiding synchronization points increases the performance, especially in large threadblocks. Furthermore, *RR-LF* reduces warp divergence, since the number of active threads remains constant along steps and uses fewer synchronization barriers than *RR-KS*.

Figure 3.6 shows the performance evolution along G, the number of batches, for our best proposal, *RR-LF*, with respect to *CUDPP*. Our proposal shows a good improvement up to $G = 256$. The reason is the block occupancy per SM, which is reduced through shared memory requirements increase. Figure 3.7 shows the same analysis over *CUSPARSE*, where our proposal shows better performance for any G value with respect to this *NVIDIA* library.

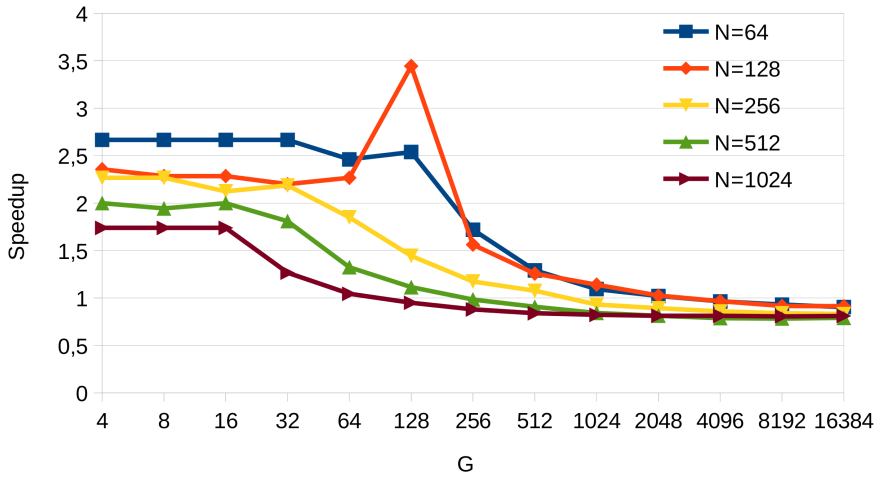


Figure 3.6: *RR-LF* speed-up over *CUDPP* for different G batch sizes on the Fermi Platform.

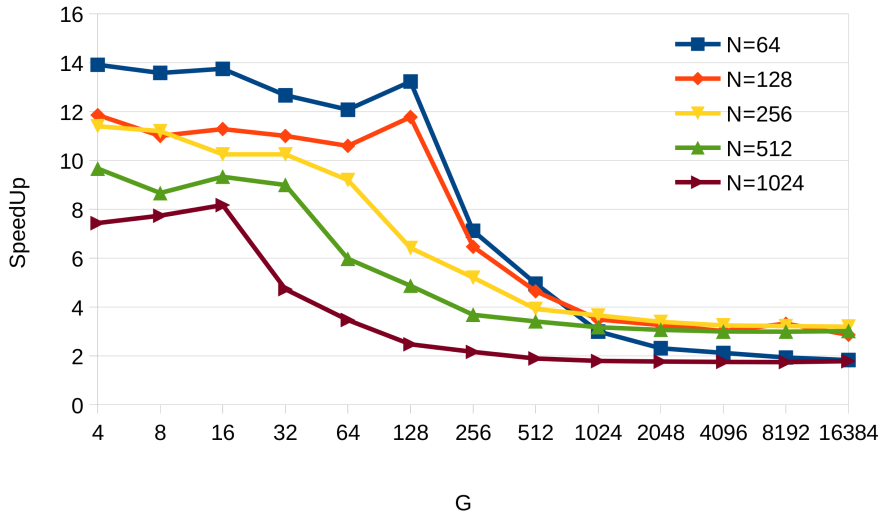


Figure 3.7: *RR-LF* speed-up over *CUSPARSE* for different G batch sizes on the Fermi Platform.

Figure 3.8 shows a global overview on the Kepler Platform when $G = 256$. Firstly, the difference between *RR-LF* and *RR-KS* is less pronounced, as the synchronization penalties are lower in Kepler; but *RR-LF* still achieves the best performance, as shared memory bank conflicts disappear in Kepler with the 8-byte bank

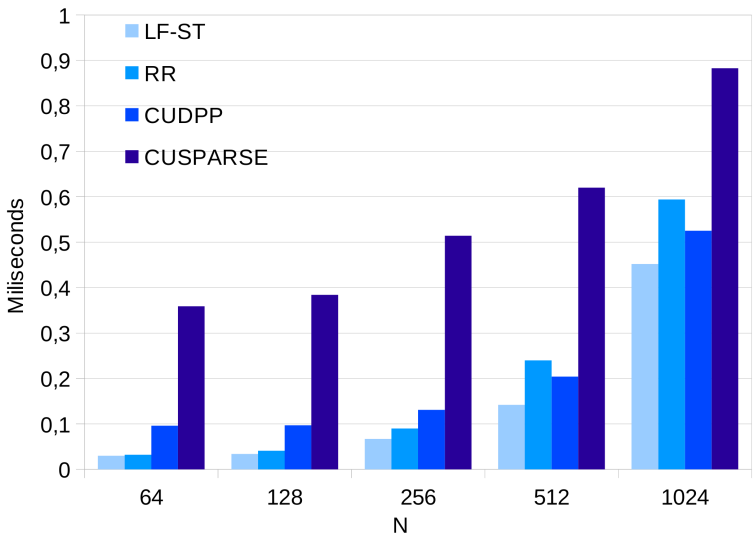


Figure 3.8: Performance results on the Kepler Platform for $G = 256$ *batches*.

access configuration. For this G value, *RR-LF* is always better than *CUDPP* and *CUSPARSE*, being up to 3.2x faster than *CUDPP* and up to 12x over *CUDPP*. In the case of *RR-KS*, it is always faster than *CUSPARSE* (up to 11.21x) but it does not improves *CUDPP* for $N > 256$. Figure 3.9 shows our *RR-LF* proposal speed-up for different batch sizes over *CUDPP* on Kepler, where the best results are obtained for small sizes. Figure 3.10 shows the same analysis with respect to *CUSPARSE*.

The monotonically-decreasing performance is related with the SM block parallelism. Attempting to improve performance by increasing the number of active warps per SM may not give rise to the optimal performance [126]. In our proposals, the need to store two different forms per equation doubles the memory usage in both registers and shared memory. Hence, the increases of N imply doubling the allocated resources, L and S, reducing the number of threadblocks that are executed by each SM. This behavior decreases the block occupancy twice as fast as other algorithms, leading to use a multi-kernel strategy starting from smaller N sizes than other algorithms. In order to study this behaviour, we have performed a kernel profile analysis. Table 3.2 presents the results of profiling our two algorithms on the Kepler Platform. We can see that, although warp occupancy remains constant in general terms, block occupancy decreases when increasing N, as the shared memory become a scarce resource, which consequently reduces the SM block parallelism.

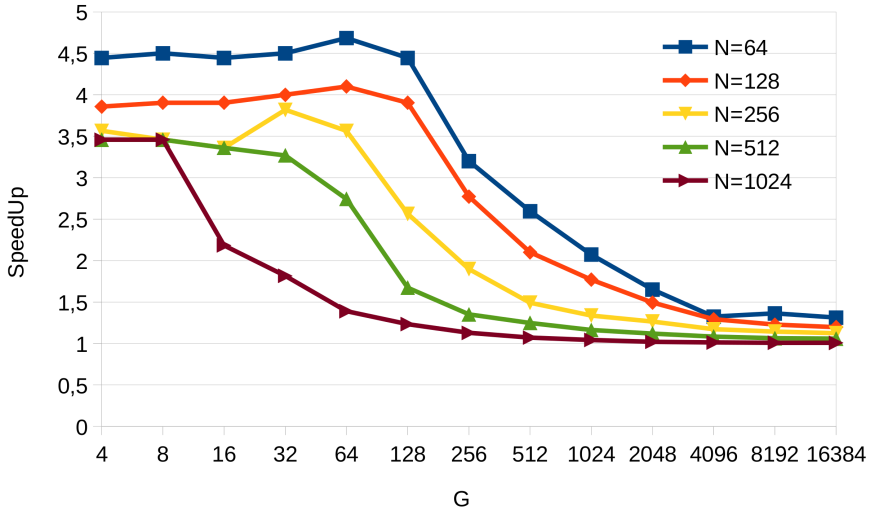


Figure 3.9: *RR-LF* speed-up over *CUDPP* for different G batch sizes on Kepler Platform.

Furthermore, when decreasing G, our competitives are extremely inefficient since they invoke several kernels to perform the reduction. As G increases, the multi-kernel strategy is less penalized. Looking again at Figure 3.10, it is easy to see that certain N sizes sometimes obtain better performance than smaller N values (see $N = 64$ and $N = 128$ when $G = 512$). This behavior is known as *tail effect*, as explained in Section 1.1.1 (Chapter 1): when the GPU launches a grid for a kernel, the grid is divided into waves of threadblocks. The case of $G = 512$, $N = 64$ implies having 2 full waves and another wave at 13% capacity, while $N = 128$ two full waves and another at 85% capacity. Finally, in order to see the effect on performance, we have also tested invoking m kernels, each with $\frac{G}{m}$ problems, but without obtaining satisfactory results.

Finally, Figure 3.11 shows the overall results on the Maxwell Platform when $G = 256$. *RR-LF* also achieves higher performance than *RR-KS*. In this case, the main reason is the fact that Maxwell reduces the number of SPs per SM, in comparison to Kepler; therefore, the synchronization barriers have more impact in Maxwell. As *RR-KS* has more barriers than *RR-LF*, the performance is penalized in Maxwell. Our *RR-KS* is up to 2.8x faster than *CUSPARSE* and up to 1.48x faster than *CUDPP*, although it does not surpasses this library when $N \geq 128$; whereas *RR-LF* outperforms both libraries, being up to 4.9x faster than *CUSPARSE* and 2x

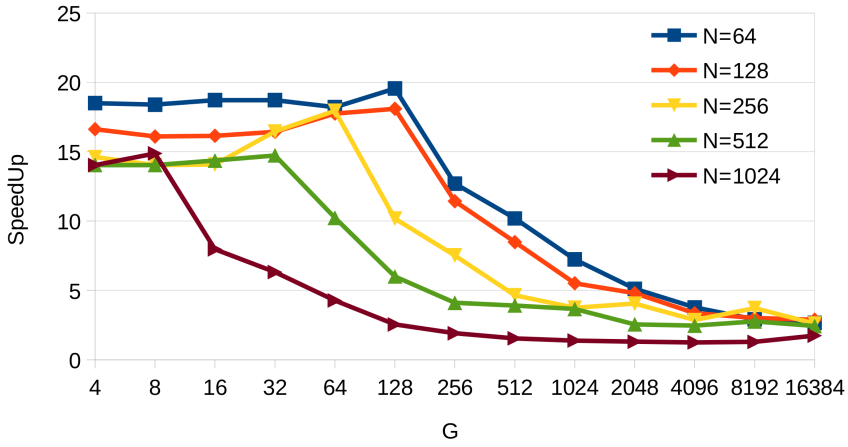


Figure 3.10: *RR-LF* speed-up over *CUSPARSE* for different G batch sizes on Kepler Platform.

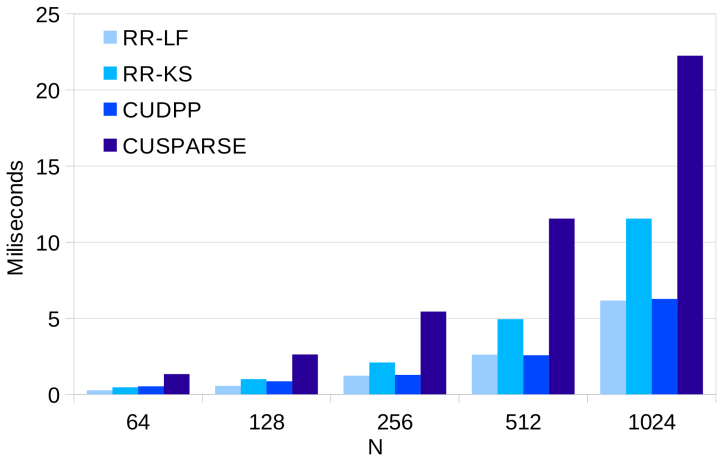


Figure 3.11: Performance results on the Maxwell Platform for $G = 256$ *batches*.

with respect to *CUDPP*. On the other hand, both proposals obtain higher speed-up over *CUDPP* and *CUSPARSE* library than in Kepler, as Figure 3.12 and Figure 3.13 show. As previously mentioned, so far the limiting factor was shared memory, decreasing the block occupancy per SM. However, Maxwell increases shared memory, becoming less limiting in this architecture. Furthermore, increasing the number of

resident blocks per SM also allows occupancy to be increased in our proposals, when it was minimum in previous architectures, thus increasing performance.

In summary, when $N \leq 512$, our *RR-LF* proposal is up to 3.25x faster than the *NVIDIA CUSPARSE* library on the Fermi Platform, up to 2.80x faster on the Kepler Platform, and up to 4.96x on the Maxwell Platform. It also obtained fairly competitive results for $N = 1024$ problem sizes. Even the *CUDPP* library is surpassed, up to 3.5x on the Fermi Platform, 4.7 on the Kepler Platform, and up to 28.9x on the Maxwell Platform for small problem sizes. Shared memory becomes a limiting factor for larger problem sizes, which are addressed later in this text.

3.2. Tree-Partitioning Reduction: A New Algorithm for Solving Tridiagonal Systems

In this section, a new tridiagonal system solver, called *Tree Partitioning Reduction* (TPR), is presented. This method is based on a division of the problem into independent slices to compute large-problem sizes and was originally presented in [34]. The TPR algorithm has two phases: the forward reduction and the backward substitution. In contrast to most iterative solvers, equations that take part in the TPR can be reduced independently in each slice over many steps, facilitating the computation of large systems.

The goal of the forward reduction, which is shown in Figure 3.14, is to compute as many steps in independent slices as possible, where there is no communication among slices, and finally, to integrate all the resulting equations in the lowest number of steps possible. In the backward substitution, unknowns are solved with the equations obtained in the forward reduction.

3.2.1. The TPR Forward Reduction phase

Regarding the forward reduction, the TPR method divides the coefficient matrix A into $M = N/S$ sub-matrices of equal size S, $A = \{A_0, ..., A_{M-1}\}$, where each sub-matrix corresponds to an independent slice. A sub-matrix A_j is composed of

G	N	Proposal	L	S	Warp Occup.	Block Occup.
16384	64	RR-KS	64	2048	50%	100%
		RR-LF	32	2048	25%	100 %
	128	RR-KS	128	4096	68%	68.75%
		RR-LF	64	4096	37%	75%
	256	RR-KS	256	8192	61.30%	31.25 %
		RR-LF	128	8192	37.50 %	37.50%
	512	RR-KS	512	16384	50.10 %	12.50%
		RR-LF	256	16384	37.50%	12.50%
	1024	RR-KS	1024	32768	50%	6.25%
		RR-LF	512	32768	25.10%	6.25%

Table 3.2: Kernel profile analysis of our proposals on the Kernel Platform

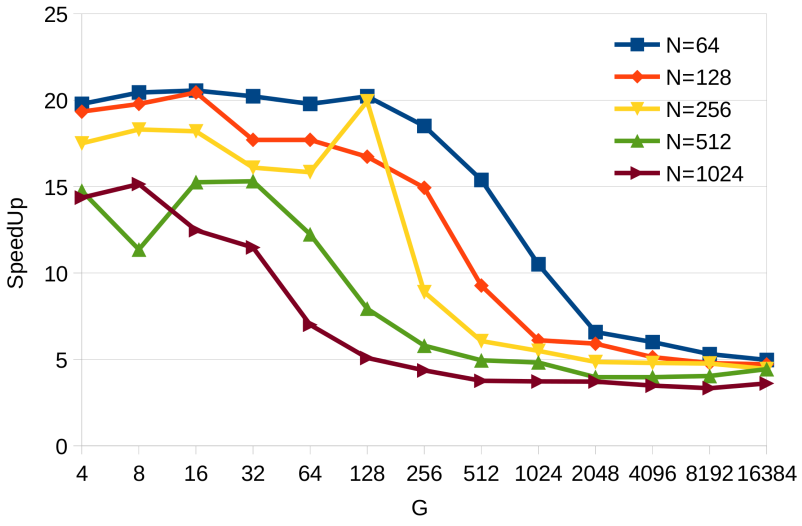


Figure 3.12: *RR-LF* speed-up for different G batch sizes over *CUDPP* on the Maxwell architecture

the following set of equations $\{E_{j\cdot S+1}, \cdots, E_{j\cdot 2\cdot S}\}$. Each row in the sub-matrix corresponds to an equation and is represented by its coefficients; thus, the sub-matrix A_j has the following starting rows ($k = 0$): $\{E^0_{j\cdot S+1} \equiv \{j \cdot S, j \cdot S + 1, j \cdot S + 2\}, \cdots, E^0_{j\cdot 2\cdot S} \equiv \{j\cdot 2\cdot S - 1, j\cdot 2\cdot S, j\cdot 2\cdot S + 1\}\}$. Figure 3.15 depicts the evolution of the coefficient matrix in the TPR forward reduction. The TPR method transforms the starting coefficient matrix, as shown in Figure 3.15 (a), into an equivalent matrix composed of sub-matrices, where the bottom row (equation) of each sub-matrix has

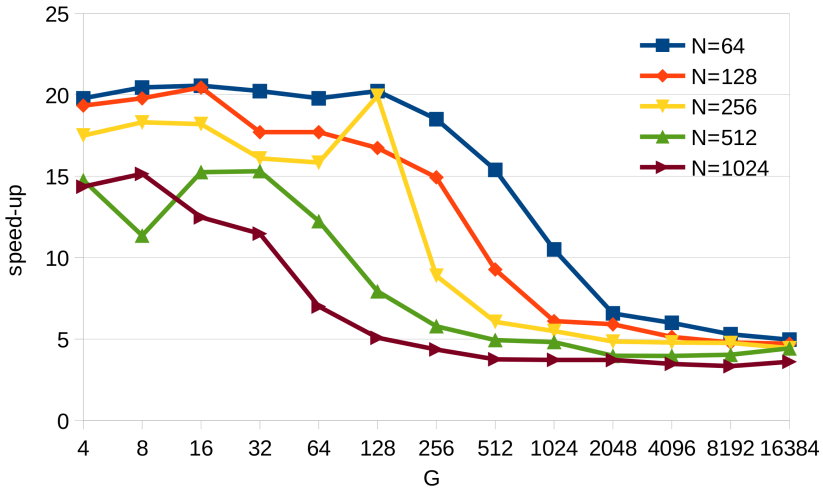


Figure 3.13: *RR-LF* speed-up for different G batch sizes over *CUSPARSE* on the Maxwell architecture

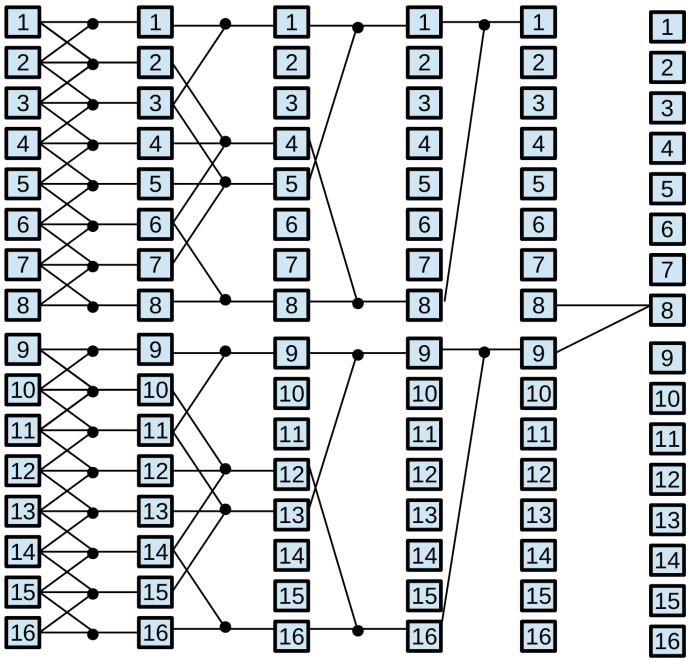
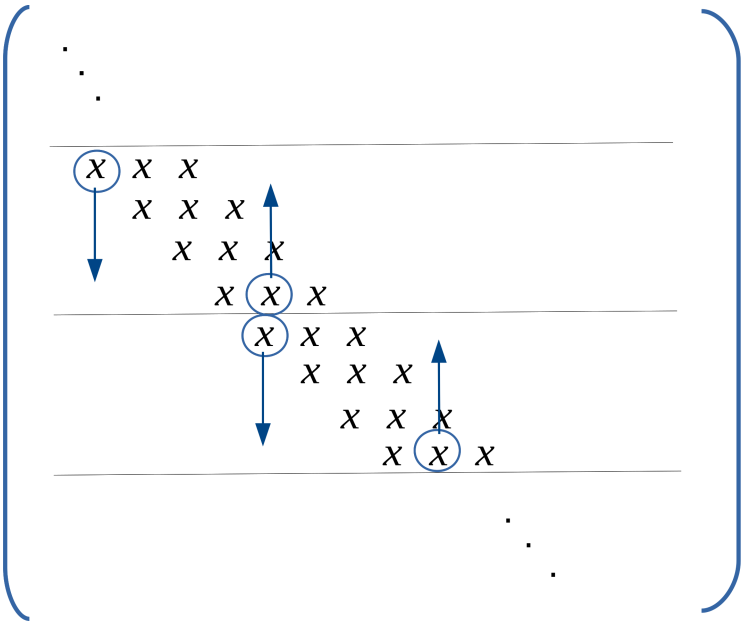
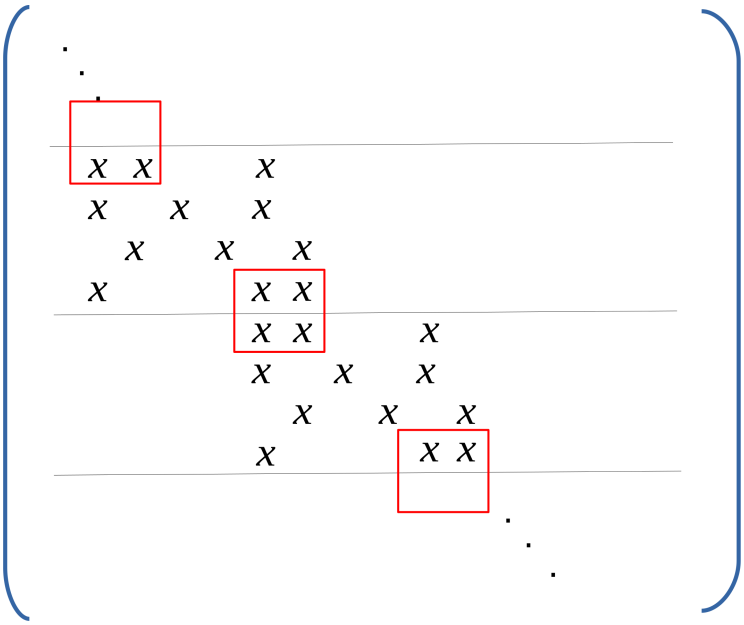


Figure 3.14: Forward reduction phase for $N = 16$ elements in the TPR method

two common columns (unknowns) with respect to the top row of its lower sub-matrix, as represented in Figure 3.15 (b). Thanks to this process, each sub-matrix computes many steps independently, and subsequently, can easily use equations



(a) Coefficient matrix in the starting step



(b) Coefficient matrix after processing sub-matrices

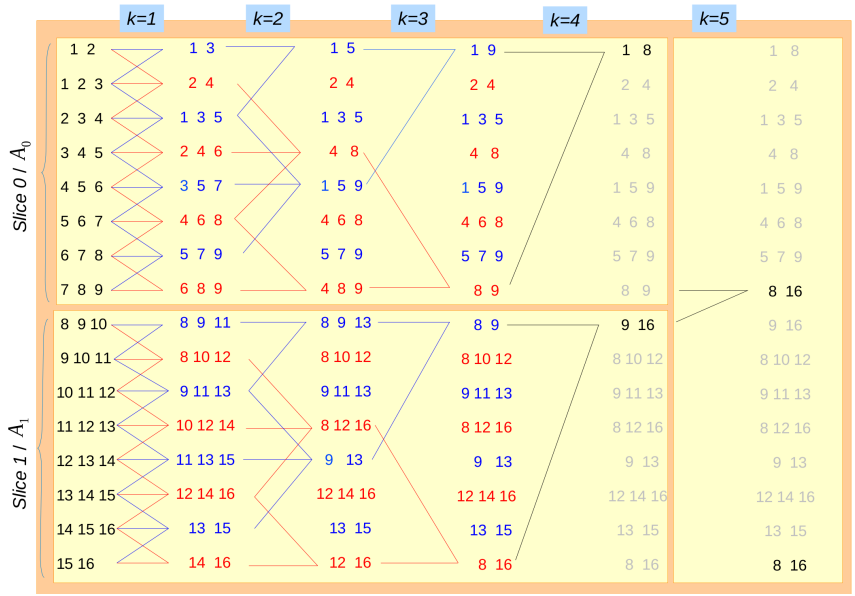
Figure 3.15: Coefficient matrix evolution in the TPR method

from other sub-matrices to build the overall final reduction. This transformation is carried out in $log_2 S + 1$ steps, called *sliced forward reduction*, where the rows of a sub-matrix are independently reduced with other rows from the same sub-matrix. As can be observed, columns from the top rows of each sub-matrix are carried to its bottom rows in order to provide these unknowns to lower sub-matrices. Likewise, columns from the bottom rows of each sub-matrix are carried to its top rows in order to provide these unknowns to upper sub-matrices. After $log_2 S + 1$ steps, the corresponding rows from one sub-matrix work with the same two unknowns as the ones of the corresponding rows from adjacent sub-matrices, and allowing to be reduced in a tree-form reduction that follows the updating schema defined below in Equation 3.2. After the reduction, a tree-form substitution is performed to obtain the value of the unknowns.

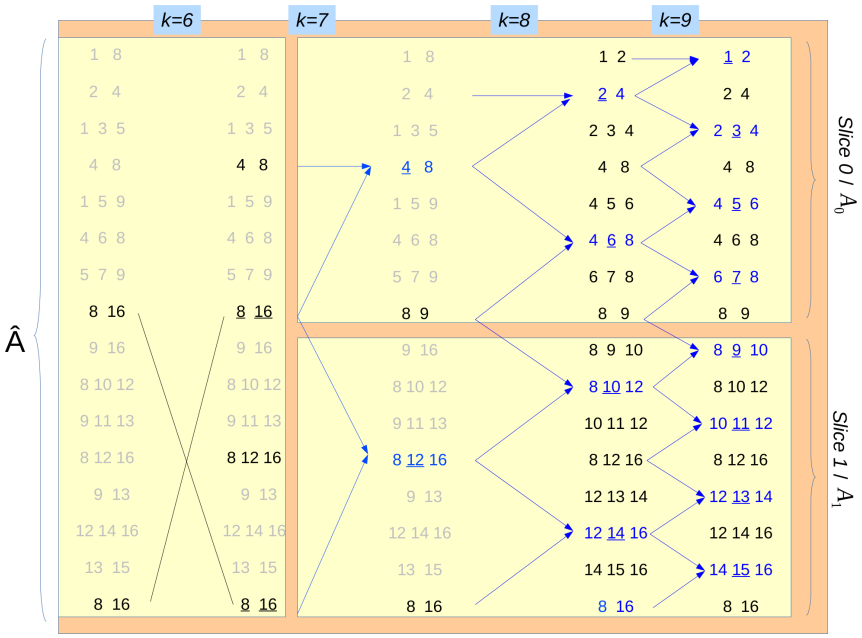
In terms of computation, each sub-matrix represents a slice of the problem to be computed in an independent memory space. Equations of each slice are independently computed through $log_2 S + 1$ iterations, without communication among slices. The suitable E_i equations in each slice are reduced with the E_{i-u} and E_{i+u} equations from the same slice in $log_2 S$ steps, starting with $u = 1$ and where the domain of u shrinks exponentially in each step ($u = 2^{k-1}$). To determine the suitable E_i equations in the step k, each slice j, which represents the A_j sub-matrix, updates the coefficients of the E_i equations that match one of the following index conditions: $i = j \cdot S + 1 + 2 \cdot u \cdot q$ (blue equations in Figure 3.16 (a)) or $i = j \cdot S + 1 + 2^k - 1 + 2 \cdot u \cdot q$ (red equations), for all $q \in \mathbb{N} / \ 0 \leq q \leq \frac{S}{2^k} - 1$. Hence, the suitable equations for each step k are updated as Figure 3.17 shows by

$$\begin{aligned} a_i^{k+1} &= -a_{i-u}^k s_1, \qquad b_i^{k+u} = b_i^k - c_{i-u}^k s_1 - a_{i+u}^k s_2, \quad with \ \ s_1 = \frac{a_i^k}{b_{i-u}^k} \\ c_i^{k+1} &= -c_{i+u}^k s_2, \qquad d_i^{k+u} = d_i^k - d_{i-u}^k s_1 - d_{i+u}^k s_2, \quad with \ \ s_2 = \frac{c_i^k}{b_{i+u}^k} \end{aligned} \tag{3.2}$$

using the $[E_{i+u}^{k-1}, E_i^{k-1}, E_{i-u}^{k-1}]$ equations, if they are in the same private-memory space. Finally, in the step $log_2 S + 1$, the $E_{j \cdot S+1}$ equation in each slice j is updated using $E_{j \cdot 2 \cdot S}$. Figure 3.16 (a) and Figure 3.16 (b) shows the forward reduction and substitution phases of this method, respectively, for $N = 16$ elements, partitioning the system into two sub-matrices of size $S = 8$ elements.



(a) TPR forward reduction



(b) TPR backward substitution

Figure 3.16: Tree Partitioning Reduction example for $N = 16$ elements with $S = 8$

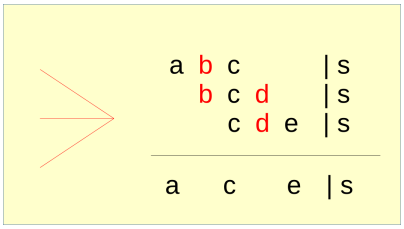


Figure 3.17: Coefficient reductions in the TPR forward reduction phase for a node computation

This sliced forward reduction can also be seen as the transformation of each sub-matrix A_j^k into a half-size matrix A_j^{k+1} composed of the $\{E_i\}$ equations that suit either $i = j \cdot S + 1 + 2 \cdot u \cdot q$ or $i = j \cdot S + 1 + 2^k - 1 + 2 \cdot u \cdot q$, with $0 \leq q \leq \frac{S}{2^k} - 1$. Thus, the number of rows in each sub-matrix is reduced by half in each step. After $log_2 S$ steps, the sub-matrix is composed of only two equations: the top and bottom equations of the initial sub-matrix, $E_{j \cdot S+1}$ and $E_{(j+1) \cdot S}$. The $E_{j \cdot S+1}$ equation is reduced using the $E_{(j+1) \cdot S}$ equation in the step $log_2 S + 1$. Therefore, $log_2 S + 1$ steps are computed independently inside each sub-matrix, but after this point, sub-matrices must be reduced with each other. After $log_2 S+1$ steps, the $E_{(j+1) \cdot S}$ equation (sub-matrix A_j) is reduced with the $E_{(j+1) \cdot S+1}$ equation (sub-matrix A_{j+1}) in the $log_2 S + 2$ step. Let us define $\hat{A}$ as the coefficient matrix which contains the updated equations from all sub-matrices at this point. The remaining steps of the forward reduction phase will work with only M rows of $\hat{A}$, specifically with the $E_{(j+1) \cdot S}$ equations. A conventional tree-form reduction is performed for these M equations of $\hat{A}$ during $log_2 M - 1$ additional steps until a system of two equations is reached; the coefficients are updated in each step following the Equation 3.2. In every step, the odd equations are eliminated by their two adjacent equations, resulting in a half-sized matrix formed of the remaining unsolved equations.

To sum up, the forward reduction phase is composed of three blocks. In the first block, equations of each slice are independently reduced in $log_2 S + 1$ steps. These steps are also known as sliced forward reduction. Then, the bottom equation of each slice is computed with the top equation of its corresponding lower slice in one step. And finally, in the third block, the resulting equations of previous steps are solved in a single overall single matrix, in $log_2 N/S$ steps. At this point, the backward substitution phase is performed.

3.2.2. The TPR Backward Substitution phase

Concerning the backward substitution, a two-unknown system composed of two equations, and received from the forward reduction phase, is solved in its first step. Then, each step of the backward substitution solves the unknown variables in the overall matrix $\hat{A}$ in $log_2 M$ steps, by substituting solutions obtained from the previous step:

$$x_i = \frac{d_i - a_i x_{i-u} - c_i x_{i+u}}{b_i}$$

decreasing u exponentially step-by-step, $u^k = \frac{u^{k-1}}{2}$, whereas the domain of i increases exponentially, as Figure 3.16 (b) shows. Once the M rows of $\hat{A}$ are solved, a subset of the equations calculated in the sliced forward reduction can be reused to solve the remaining unknowns in conjunction with a subset of the original starting equations from the original coefficient matrix A. Specifically, the E_i equations that suit $i\%2 \neq 0$ are taken from A^0; i.e. the E_i^0 starting equations (i rows of A^0); whereas the E_i equations that match $i\%2 = 0$ are the ones calculated after the sliced forward reduction; i.e. the $E_i^{log_2 S+1}$ equations (i rows of $\hat{A}$). After this replacement, the remaining unknowns are calculated by the same substitution process explained above. It should be observed that this computation of the last $log_2 S$ steps can be performed independently in slices, since each slice only needs to know the bottom equation of its upper slice, which does not vary during the final steps, and these steps are known as sliced backward substitution. This algorithm can be implemented efficiently in any parallel programming paradigm, as Chapter 5 shows.

3.2.3. An example of the TPR method

Figure 3.16 shows an example of the TPR method for $N = 16$ equations, where the matrix is divided into $M = 2$ independent sub-matrices of size $S = 8$; i.e. the computation is divided into two slices, as depicted in Figure 3.16 (a). The forward reduction phase is performed in $(log_2 S) + 2 = 5$ steps, as defined above. Specifically, the first $log_2 S$ steps are computed inside each slice, with no communication with other slices. To this end, the suitable E_i equations are reduced with the E_{i-u} and

E_{i+u} equations from the same slice, where u shrinks exponentially in each step: $u = \{1, 2, 4, ...\}$.

The suitable E_i equations, whose coefficients are updated using Eq. 3.2, are determined in each step by the following index conditions $i = j \cdot 8 + 1 + 2 \cdot u \cdot q$ (blue equations in figure) or $i = j \cdot 8 + 1 + 2^k - 1 + 2 \cdot u \cdot q$ (red equations), for all $q \in \mathbb{N}/$ $0 \leq q \leq \frac{8}{2^k} - 1$. Hence, the suitable equations in slice 0 are:

- $k = 1$: $\{E_1, E_3, E_5, E_7\}$ and $\{E_2, E_4, E_6, E_8\}$
- $k = 2$: $\{E_1, E_5\}$ and $\{E_4, E_8\}$
- $k = 3$: $\{E_1\}$ and $\{E_8\}$

Suitable equations in slice 1 are obtained in a similar manner. In $k = 4$, E_1 is updated with E_8 in slice 0, whereas E_9 with E_16 in slice 1. At this point, the sliced forward reduction is completed, and all the computation have been performed independently inside each slice. In $k = 5$, E_8 is updated with E_9, and the $\hat{A}$ single overall matrix is built with $[E_8, E_{16}]$. In this case, $\hat{A}$ represents a 2-unknown system whose unknowns are x_8 and x_{16}; and this can be solved directly in the first step of the backward substitution phase. At this point, the substitution can be performed again in slices for the last 3 steps. It should be observed that slice 1 needs the E_8 equation in its substitutions, but E_8 does not vary since the sliced backward substitution starts.

In the Chapter 5, this algorithm is implemented under the proposed CUDA tuning methodology of this work. Additionally, a numerical stability analysis is also provided.

3.3. Bitonic Merge Comb Sort: A New Algorithm for Sorting

As already introduced, sorting is an important computing operation that takes part in many applications. In order to obtain a high performance implementation for GPUs, it is necessary to consider an algorithm which matches well to GPU

architectures, but also apply different optimization programming techniques. In this section, we present an efficient and portable sorting operator for GPUs, a CUDA implementation of the *Bitonic Merge Sort* (BMS) algorithm [7]. However, in order to achieve the said efficiency, we have developed an algorithmic variant of BMS, called *Bitonic Merge Comb Sort* (BMCS), and we have applied different CUDA optimizations.

3.3.1. A CUDA Implementation for the Bitonic Merge Sort Algorithm

Before introducing our BMCS proposal, let us explain a CUDA implementation for the BMS algorithm. The parallel pattern of this algorithm is easily programmable in GPUs. In an initial implementation, each vertical segment of the Figure 2.13 introduced in Chapter 2 represents a Node operator performed by one CUDA thread. Figure 3.18 contains the CUDA code of the Bitonic Merge Sort. This first implementation is tagged as *BS-naive* in the experimental results. According to our notation, the *fan in* and *fan out* of the Node operator are both two. Each thread processes one Node operator, and each threadblock performs the sorting of a single problem. The code can be divided into four main sections:

- Initialization section (lines 3-6). We define the thread and threadblock identifiers, and memory offsets for load and store operations. Furthermore, registers and shared memory are allocated.
- Load data from global memory (line 8) and first computing step (lines 9-10). We load coalescent data using a 64-bit load to obtain 2 consecutive elements, instead of accessing a single data element per memory request. Since elements loaded by each thread are not shared among other threads, they are directly processed in registers by the *compare* function. Finally, computed data are stored in shared memory for the next step.
- Compute steps of the algorithm (lines 13-25). The loop computes the remaining steps of the algorithm, each step k contains a number of $j = k$ sub-steps. For this, the loop reordes data registers using shared memory and synchronization barriers. For accesses, both memory offsets and strides are calculated

```
1  template<int N>
2   __global__ void Bitonic ( int *  glbData) {
3
4          const int globalId = ..., threadId = ... ;
5          __shared__ int shm[N];
6          int regs[2];
7
8          copy<..>(regs,glbData,...);//loads coalescing data from glbMem to regs
9          compare(regs); //it compares and swaps if necessary
10         copy<..>(shm,regs,...);//stores data in shm
11         __syncthreads();
12
13         for(int k=2;k<N;k*=2){
14                 copy<..>(regs,shm,...);//load data from shm to reg
15                 compare(regs);
16                 copy<..>(shm,regs,...);//store data in shm
17                 __syncthreads();
18
19                 for(int j=k; j>1; j/=2) {
20                         copy<..>(regs,shm,...);//load data from shm to reg
21                         compare(regs);
22                         copy<..>(shm,regs,...);//store data in shm
23                         __syncthreads();
24                 }
25         }
26         copy<..>(glbData,regs,...);//Stores data from regs to glbMem
27 }
```

Figure 3.18: Kernel code for Bitonic Merge Sort algorithm (BS-naive).

using bit masks, binary operators and displacements to be efficient. The internal loop keeps the same structure as the external loop, returning the results of the last internal step in registers.

- Store data to global memory (line 26). The last iteration of the loop stores the results into registers, moving these data from registers to global memory here. Even for smaller problems, coalescence is achieved thanks to the cache hierarchy of the GPU.

3.3.2. Bitonic Merge Comb Sort

The Bitonic Merge Sort algorithm presented above can be improved to achieve high parallelism and efficiency in GPUs. We present an algorithmic variant of Bitonic, Bitonic Merge Comb Sort (BMCS), that matches well to the GPU features, adapting the thread workload to available registers, obtaining coalescing accesses, avoiding bank conflicts, decreasing the number of synchronization barriers and en-

abling the use of shuffle instructions. This work was originally introduced in [31].

Increasing the thread workload, depends on two factors: one is the existence of available registers and the other is that the increase does not reduce the warp occupancy. Profiling the execution, we can find out this trade-off for each CUDA architecture. In our tests, best results were achieved when each thread works with four elements, so we have modified the algorithm accordingly. The classic Bitonic Merge Sort has log_2N external steps, each with log_2N internal steps, where the *fan_in* and *fan_out* are both two, processing two elements per thread. However, increasing it up to four elements per thread implies that each thread reads and writes four elements in the algorithm, so *fan_in* and *fan_out* are both four. The naive algorithm is radix 2, and the number of external steps is log_2N. We propose a hybrid algorithm with $log_2N - 1$ external steps, each with log_4N steps (*Radix 4*). In the case of external steps, we read four consecutive elements from global memory, computing them directly in registers, thus the two first external steps are reduced to one ($log_2N - 1$). Figure 3.19 displays a scheme for a problem of $N = 16$ with 4 threads, where each thread computes a Node operator with $r = 4$. As can be observed in this figure, the number of steps decreases. When working with arrays, as in this case, it is necessary to pay attention to accesses. Highest performance is achieved with accesses where the compiler can derive constant indices in all accesses, placing elements into registers. If the compiler cannot determine the index and it depends on a value determined at run-time, array elements are placed in local memory in a process known as dynamic indexing, as explained in Section 1.2 (Chapter 1). We have carefully designed our code in order to take advantage of the compiler to produce constant indices.

On the other hand, if each thread computes several elements, it is also necessary to specify what elements are accessed in order to avoid bank conflicts or coalescencing problems. Figure 3.19 also shows the distribution of work per thread in terms of efficiency. Despite coalescing accesses, consecutive threads loads consecutive sets of four adjacent elements in its first external step from global memory to registers. The first step is designed on this way to enable the use of customized CUDA datatypes such as *Float2* or *Int4*. These datatypes reduce the number of memory transactions, as well as allowing coalescing accesses. In a similar manner, the last step of the algorithm is designed for computing four adjacent elements in registers, and then

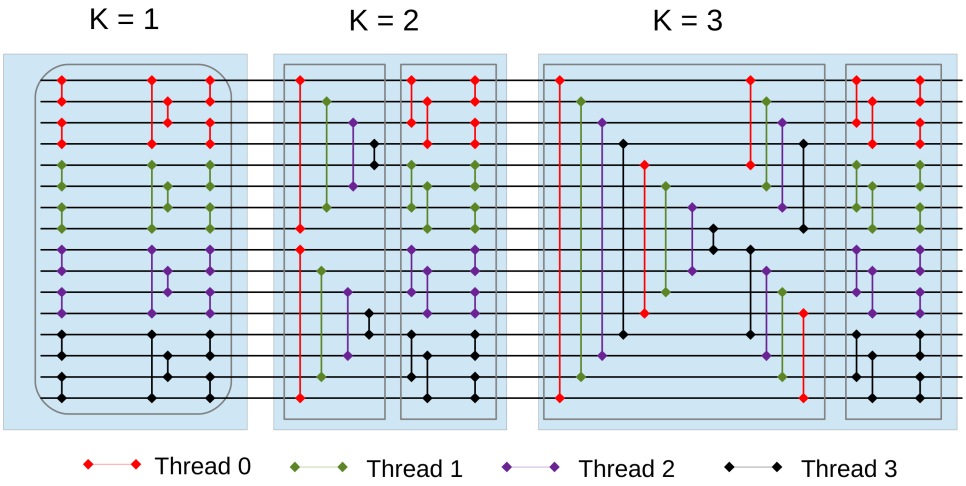


Figure 3.19: Bitonic Merge Comb Sort Algorithm with $N = 16$.

storing the result in global memory using these datatypes. Even for smaller problems coalescence is achieved thanks to the cache hierarchy of the GPU.

Matching of threads with elements is not trivial in the remaining steps. Depending on the k external step, the number of internal steps could be a non-power of two. In this case, a mixed computation of *Radix 2* is required in the first internal step of k, whereas the reamining steps are *Radix 4*. Furthermore, each thread has to operate with the corresponding four elements that allow two steps in the radix 2 approach to be transformed into one step in the radix 4 proposal. This fact can be easily understood in Figure 3.19 when $k = 3$. In the last internal step of $k = 3$, four elements are consecutively written for each thread. If we focus on *thread 0*, first four elements are processed. To obtain this, in the previous internal step, the first, fifth, twelfth and sixteenth elements were computed by *thread 0* and only these four elements allow us to process four consecutive elements in the last step. If there were more previous steps, our algorithm scheduler would choose the corresponding four elements, which would allow four consecutive elements to be processed in the last step. This distribution pattern also avoids shared memory bank conflicts.

Figure 3.20 also shows the algorithm but changing the representation of the Node operator to a $r = 4$ representation. Increasing the radix implies reducing the number of steps, in other words, reducing the number of synchronization barriers. It should

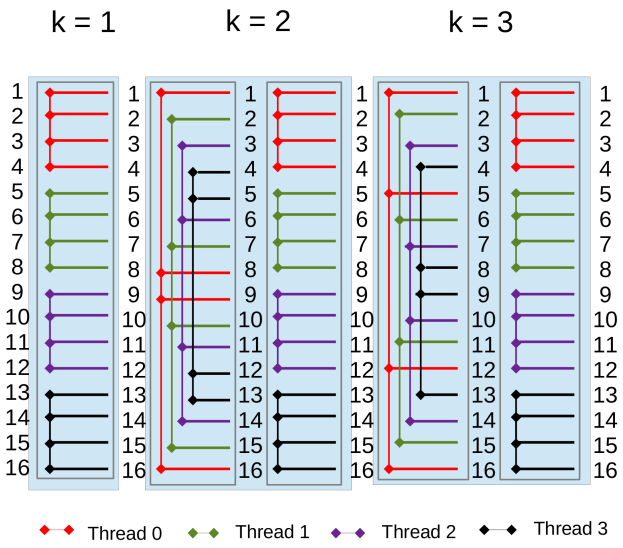


Figure 3.20: Bitonic Merge Comb Sort Algorithm for $N = 16$.

be noted that the classical algorithm has $n + \frac{n(n+1)}{2}$ synchronization barriers, being $N = 2^n$, whereas our proposal has $n-1+\frac{n/2(n/2+1)}{2}$. The decrease in synchronization barriers is especially visible in Fermi CUDA architectures, where each SM has only 32 SPs, but also in current CUDA architectures, as Section 3.3.3 shows.

Finally, it is also possible to reduce the synchronization barriers and memory latency even further. Shuffle instructions allow the exchange of information using registers instead of shared memory, among threads in the same warp, as already mentioned. These instructions free up shared memory to be used for other data or to increase the occupancy; these are faster as they only require one instruction, instead of three (write, synchronize and read). This approach avoids warp-synchronization barriers, although they are limited to the warp scope. In the CUDA implementation of BMCS, the initial steps are computed using shuffle instructions, sorting $fan\_in \times 32$ elements in each warp, and then using shared memory as communication channel among warps. If $N \leq 128$, then there are no synchronization barriers in the execution. Otherwise, this technique saves 9 synchronization barriers.

	Kepler Platform	Maxwell Platform
CPU	Intel Xeon E5-2660 CPU 2.2 GHz	Intel Core i7-2600 3.4 GHz
Memory	64 GB DDR3 1600	8 GB DDR3 1333
OS	CentOS 6.4	Ubuntu 12.04 LTS
Compiler	GCC 4.4.7	GCC 4.6.3
GPU	Nvidia Tesla K20 GPU	Nvidia GeForce GTX980
Driver	340.58, SDK 6.0	343.22, SDK 6.5

Table 3.3: Description of the test platforms for the sorting problem

3.3.3. Experimental Results for BMCS in CUDA

In this section, the results of our proposals on different *NVIDIA* GPU architectures are presented. All tests were run using integers as datatype. All the data reside in the GPU memory at the beginning, so there are no data transfers to CPU during benchmarks. The test platforms used in our experiments are described in Table 3.3. All these algorithms were developed to take advantage of the read-only data cache, which slightly improves global memory read bandwidth. The performance of these experiments is measured in million data processed per second, MData/s. Additionally, many applications need to solve G batch problems in parallel. Therefore, we use a multi-batch execution to compute G problems each time. The size of the batch depends on the input size and is given by the expresssion $G = 2^{24}/N$ in order to use most device memory and exploit the GPU parallelism. Thus, MData/s value is performed using the expression $N \times G \times 10^{-6}/t$.

First, Figure 3.21 depicts a performance comparison of the classical BMS and the BMCS for the Kepler Platform executing several batches in parallel. In order to demonstrate the portion of efectiveness of the algorithm and other optimizations, both proposals were hand-tuned for each architecture. In Chapter 4, this algorithm is implemented under the the proposed CUDA tuning methodology of this work. *BS-Naive* tag refers to a CUDA implementation using the classical BMS algorithm presented in Section 3.3.1, whereas *BS-R4* is the radix-4 BMCS variant introduced in Section 3.3.2 but without using shuffle instructions. Finally, *BS-Comb* is the radix-4 variant introduced in Section 3.3.2, which uses shuffle instructions and obtains the best performance. In general, shared memory is an expensive resource that progresively reduces performance (and warp occupancy per SM) when N increases,

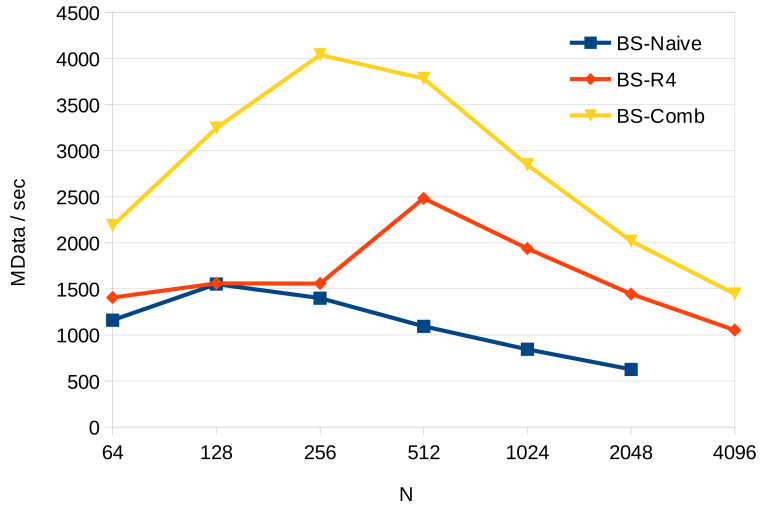


Figure 3.21: Comparison of our proposal optimizations in the Kepler Platform.

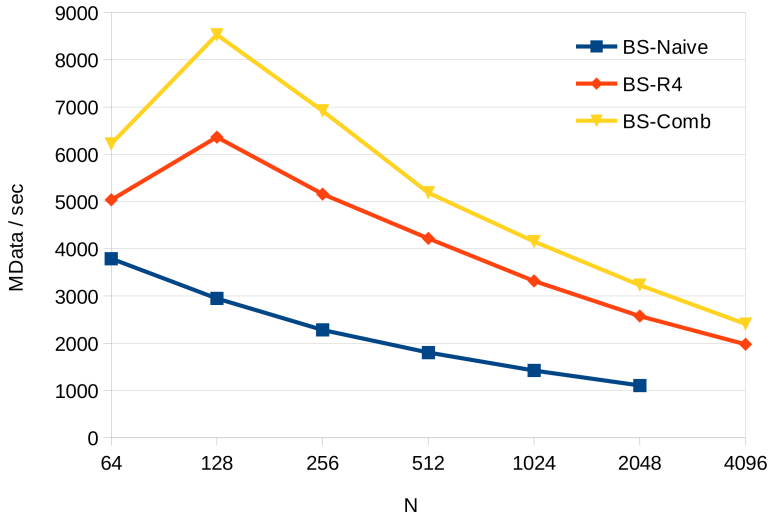


Figure 3.22: Comparison of our proposal optimizations in the Maxwell Platform.

since more data are stored in this memory, becoming the limiting factor. Proposals obtain a peak of performance with $N = 256$ or $N = 512$ as occupancy is maximum, obtaining high performance in their computations, where *BS-Comb* has an improvement of up to 3.45x over *BS-Naive* and 2.6x over *BS-R4*. Figure 4.25 shows the same comparison on the Maxwell Platform, obtaining similar results, where *BS-Comb* is up to 3x faster than *BS-naive* and 1.34x than *BS-R4*.

Regarding the performance of our proposals in the Kepler Platform, Figure 3.23

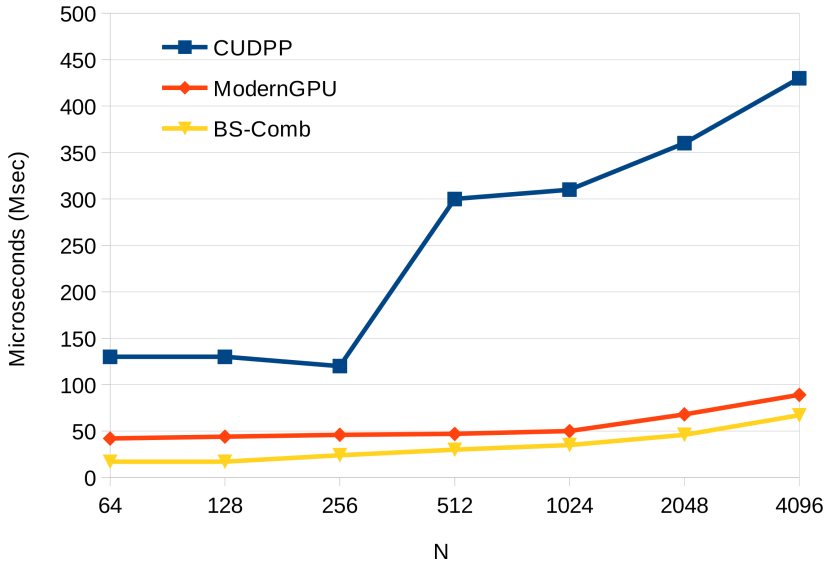


Figure 3.23: Comparison of GPU sorting implementations for one batch in the Kepler Platform.

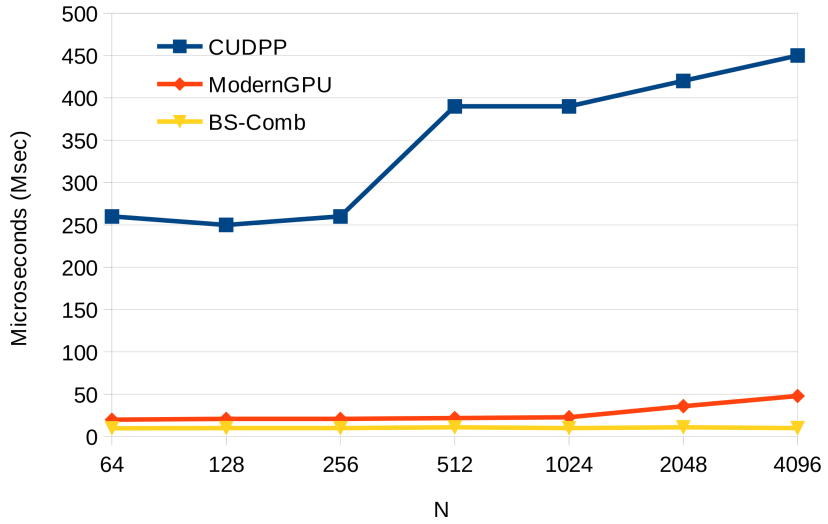


Figure 3.24: Comparison of GPU sorting implementations for one batch in the Maxwell Platform.

compares our proposal results to the *CUDPP* library [98] and the *ModernGPU* library [97]. It should be noted that this comparison is done in terms of execution time for only one batch. *CUDPP* shows the worst results for small problem sizes that can be directly processed in shared memory. Our proposal, *BS-Comb*, shows very competitive results compared to *ModernGPU*, the fastest sorting library for

small problem sizes, to the best of our knowledge. This implementation obtains an improvement of up to 10x over *CUDPP*, and up to 2.6x over *ModernGPU*. On the other hand, Figure 4.27 presents the same comparison on the Maxwell Platform, where results are similar to the Kepler Platform, obtaining up to 40x in comparison to *CUDPP* and up to 4.8x over *ModernGPU*.

Table 3.4 contains the MData/s obtained for the proposal *BS-Comb*. It compares our results to the *CUDPP* and *ModernGPU* ones. Both *ModernGPU* and *CUDPP* are extremely inefficient with problems in which many batches of small size are processed in parallel, since they were designed to solve just one large-size problem. In order to solve G problems of size N, these two libraries have to launch G *light* kernels. Our proposal is up to 2431x faster than *CUDPP* library and up to 1094x over *ModernGPU*. Table 3.4 also shows the MData/s obtained in the Maxwell Platform. Here, performance is higher owing to the Maxwell design. Our proposal is up to 2592x faster than *CUDPP* and up to 1446x than *ModernGPU*.

Finally, Table 3.5 shows different GPU parameters and profiling metrics for the *BS-naive* and *BS-Comb* proposals in the Kepler Platform. The warp occupancy is lower with the *BS-Comb* proposal at smaller sizes, since it uses a $r = 4$ implementation that implies using less threads. As can be observed, registers are not a limiting factor of the SM parallelism, whereas the parallelism is bounded by the number of threads employed and shared memory consumption. For small problem sizes, both the number of threads and shared memory are so low, although there are the maximum number of active threadblocks, the number of warps per threadblock is low and the warp occupancy is also low. However, when the problem size is larger, there are more threads per threadblock, maximizing the warp occupancy. In this case, the shared memory consumption also increases, decreasing the number of active threadblocks.

3.4. Conclusions of the Chapter

This chapter presents the new parallel prefix algorithms developed in this work. Specifically, it analyzes the algorithmic design of two new tridiagonal system solvers, *Redundant Reduction* (RR) and *Tree-Partitioning Reduction* (TPR); and a new al-

		Kepler Platform			Maxwell Platform		
N	G	BS-Comb	ModernGPU	CUDPP	BS-Comb	ModernGPU	CUDPP
64	262144	2188	2	0.9	6219	4.3	2.4
128	131072	3246	3.7	1.8	8529	8.3	4.9
256	65536	4040	7.1	4	6918	16.3	10
512	32768	3783	13.4	7.9	5185	31.4	19.6
1024	16384	2846	25.2	14.7	4151	61.2	35.9
2048	8192	2018	34.4	16.1	3229	66.5	33.4
4096	4096	1444	50.4	16	2406	97.2	48

Table 3.4: MData/s comparison of GPU multi-batch sorting algorithms.

	BS-naive				BS-Comb			
N	S	L	# Registers	W. Occupancy	S	L	# Registers	W. Occupancy
64	256	32	11	25%	256	16	16	25%
128	512	64	11	50%	512	32	16	25%
256	1024	128	11	100%	1024	64	17	50%
512	2048	256	11	100%	2048	128	17	100%
1024	4096	512	11	100%	4096	256	17	100%
2048	8192	1024	11	100%	8192	512	17	100%

Table 3.5: GPU parameters and profiling metrics for our sorting proposals.

gorithm for sorting, *Bitonic Merge Comb Sort* (BMCS).

Additionally, although these algorithms can be implemented on any parallel paradigm, this chapter also presents some naive GPU implementations for *RR* and *BMCS*. The *RR* proposal is up to $3.25x$ times faster than *CUSPARSE*, the state-of-the-art; and *BMCS* outperforms other well-known libraries such as *CUDPP* (up to $10x$) and *ModernGPU* (up to $2.6x$) in the case of solving a single problem, and obtaining a speed-up of several magnitudes in the case of solving several batches simultaneously. In the remaining text, the three algorithms developed on this chapter are accurately implemented for GPUs following a methodology.

Chapter 4

A Tuning Methodology for Small Problem Sizes on a GPU

As explained in the Preface, the aim of this work is to provide programmers with a tuning accelerated library; i.e., an easy way to execute high performance parallel algorithms through simple routines, without the need for accurate knowledge of the language or the target GPU architecture. To do this, we develop a tuning methodology composed of a set of guidelines and *CUDA skeletons* that allows us to easily design any parallel prefix algorithm in CUDA and efficiently execute them on any CUDA GPU architecture.

This chapter presents an original contribution with respect to previous works. A three-phase tuning methodology is presented to efficiently solve parallel prefix algorithms of small size on a GPU. In the first phase, *GPU Resource Utilization Analysis*, presented in Section 4.1, the main features which influence the GPU performance for each algorithm are determined, establishing a number of tuning parameters and a set of premises for performance maximization. In the second phase, *CUDA Kernel Optimization*, algorithms are implemented using parameterized *CUDA skeletons*, explained in Section 4.2. The third phase, *Performance Parameters Tuning*, analyzed in Section 4.3, obtains the suitable values for the performance parameters described in the first phase for each algorithm, depending on the problem size and the target architecture.

This methodology is an adaption of the strategy presented in [82]. In said paper,

the authors present a tuning strategy for ID-algorithms which fit in the shared memory, i.e. small problem sizes. Nevertheless, many parallel prefix algorithms do not match in the Index-Digit description. In this work, we extend that strategy to a larger set of algorithms, the parallel prefix algorithms, which have less uniform and systematic properties, complicating the design of the methodology.

Following this methodology, we have developed three different tridiagonal system solvers (Section 4.4), two scan primitive solvers (Section 4.5) and one sorting operator (Section 4.6), outperforming the state-of-the-art for limited-size problems which fit in the shared memory of the GPU. This work was originally introduced in [27], [28] and [33].

4.1. GPU Resource Utilization Analysis Phase

This phase identifies the problem and the GPU performance parameters, collected in Table 4.1, which maximize the GPU execution throughput of radix-2 parallel prefix algorithms; hence, $r = 2$. Many of them were previously introduced in Chapter 2. Each thread works with at least one pair of elements, thus $p \geq 1$. Considering that all the data stored in registers also have a copy in shared memory to perform the inter-step memory exchanges, $s = p + l$, implies $l \leq s - 1$. Additionally, as the data of each problem fit directly into the shared memory of one threadblock, $n \leq s$, then L_G problems can be simultaneously solved in each block, where $L_G = S/N$. This allows the parallelism to be increased when the number of threads required to compute one problem is low.

Hence, $G = 2^{batch}$ problems of $N = 2^n$ elements are simultaneously processed, being identified by (n, p, s, l, b). However, the following relations among parameters allow the tuple to be expressed with only three parameters, (n, p, s): on the one hand, $s = p + l$ as explained above. On the other hand, the number of thread blocks is given by the batch size, which is only known at runtime, $b = batch - (s - n)$. Consequently,

$$B = \frac{G}{L_G} \quad and \quad L = \frac{N}{P} \times L_G \tag{4.1}$$

are also obtained.

Problem Parameters

$N = 2^n$	Problem size.
$G = 2^{batch}$	Number of problems being solved simulteneously.

GPU Performance Parameters

$S = 2^s$	Number of shared-memory elements per block.
$P = 2^p$	Number of elements stored in registers per thread.
$B = 2^b$	Number of thread blocks executed per GPU
$L = 2^l$	Number of threads that compose a block, where $s = p + l$
L_G	Number of problems being solved per block, where $B = \frac{G}{L_G}$ and $L = \frac{N}{P} \times L_G$
W_{SM}	Number of warps per SM
W^{max}	Maximum number of warps per SM
W_B	Number of warps per thread block
B_a	Number of active thread blocks simultaneously executed per SM
B^{max}	Maximum number of thread blocks per SM
B^r	Number of thread blocks per SM limited by the registers available
B^s	Number of thread blocks per SM limited by the shared memory available

Table 4.1: Description of tuning strategy parameters.

4.1.1. Premises for Performance Maximization

Once performance parameters are established, the values which maximize *GPU* performance need to be assigned. The main features that influence the *GPU* performance are the efficiency of the memory operations and the SM parallelism achieved. Thus, it is crucial to develop coalescing-friendly implementations that expose global-memory load/store efficiency. Thanks to the communication patterns of the parallel prefix algorithms used in this work, as well as the provided skeleton-implementation, the global-memory load/store efficiency achieved for our proposals is very high. Thus, this phase focuses on establishing a balanced ratio between the number of parallel threads and the amount of shared resources that can be assigned to each thread. Actually, the level of parallelism can be determined in terms of the number of thread blocks per SM (*SM block parallelism)*, or the number of warps per SM (*SM warp parallelism*), and a trade-off must be sought between them depending on the problem features.

The number of warps per SM, W_{SM}, is limited by the maximum number of warps per SM, W^{max}, and given by the expression: $W_{SM} = Min(W^{max}, W_B \times B_a)$, where W_B is the number of warps per threadblock ($W_B = L/32$ in current architectures). Additonally, B_a is the number of active threadblocks that are simultaneously executed per SM: $B_a = Min(B^r, B^s, B^{max})$, where B^r the number of threadblocks limited by the registers available in the *SM*, B^s the number of threadblocks limited by shared memory and B^{max} the *SM* threadblock limit of the hardware.

It is not always possible to obtain both $W_{SM} = W^{max}$ and $B_a = B^{max}$; in such a case, our priority is to obtain $W_{SM} = W^{max}$, since it helps to hide latency. It should be noted that W_{SM} can be W^{max}, even when $B_a < B^{max}$. In addition to this, there are cases where achieving $W_{SM} = W^{max}$ is impossible due to an unavoidable resource consumption; in theses cases, the goal of our proposal is to obtain the highest W_{SM} possible. In the case of there being several configurations for reaching a given W_{SM} value, our proposal always selects the one that uses the greatest B_a value. In order to ascertain the ratio between active warps employed per SM by the maximum number of warps enabled per SM, the *warp occupancy* rate is introduced. The *Performance Parameters Tuning* phase will select the performance parameters which maximize the warp parallelism.

4.2. CUDA Kernel Optimization Phase

The flexibility and adaptability of our proposals come from a set of parameterized *CUDA skeletons*. These skeletons are predefined and generic functions that implement common specific patterns of computation and data movements, which can be customized with user-defined code parameters. These skeletons are assigned the task of implementing the computation and data movements behaviors from the parallel prefix algorithms. In addition to this, thanks to the use of skeletons, the body of the kernel remains invariant across different algorithms, simply changing the *Node operator* skeleton for each algorithm.

In [81], the *Butterfly Processing Library for GPUs* (BPLG) is presented. This library is composed of a set of CUDA skeletons, which are also called *building blocks*. All these skeletons have common features:

- The use of templates that allows generic programming and *template metaprogramming*, performing many optimizations at compile time (such as reducing code complexity or avoiding temporal registers for function calls).

- Knowing additional information at compile-time, such as the problem size or the number of threads, makes it possible to perform private thread data reordering using register renaming. It is possible to take advantage of fully unrolling static loops avoiding dynamic addressing of register arrays.

- All functions have been designed to operate in any GPU memory space hierarchy.

Owing to this parameterizable design, the *BPLG* skeletons are written as templates, and receive the optimal values of the performance parameters as template arguments. In the code, a table is built for each problem size and architecture, specifying the performance parameter values for each case. At compile-time, all kernel combinations from the table are generated and, once the problem size and target architecture are defined in the execution, the corresponding skeleton is automatically loaded at runtime. There is no need to manually choose any configuration, since the suitable skeleton is automatically chosen in runtime for the given problem size and architecture. Thus, the tuning process is transparent for the user.

Based on this procedure, this work extends the existing *BPLG* skeletons with a set of improvements:

- *Efficient index calculation avoiding non-uniform access. Static Indexing* represents the fact that constant indices are derived by the compiler in all accesses to an array, placing elements directly into registers, and it is the most efficient way to reference an array. However, when the compiler cannot resolve indices to constants, it places them into local memory, with the consequent performance loss (*dynamic indexing*). Indices must be determined by the compiler and must not depend on a value determined at runtime. In this case, if all threads within a warp access the same index (*uniform access*), performance is fairly high thanks to the GPU cache system. Otherwise, if the threads of a warp access elements using different indices, it is called *non-uniform indexing*,

this being the worst scenario. Our *BPLG* skeletons have been carefully designed to take advantage of static indexing whenever possible (loop unrolling or using constants at compile time).

- *Hybrid communication strategy inside a block.* Shuffle instructions allow information to be shared among threads in the same warp using registers instead of shared memory. The use of these instructions is faster than shared memory communication, although they are limited to the warp scope. The parameter W represents the number of threads employed per warp in our strategy ($W = 32$). In our proposal, a hybrid strategy is provided: the largest possible number of steps are computed with shuffle instructions, and then, shared memory is used to obtain the final result among different warps. Depending on the communication pattern of the parallel prefix algorithm, different approaches are proposed for each algorithm.

- *Specialized skeletons.* *BPLG* skeletons were originally developed with generic templates; i.e., naive implementations use a single skeleton for each operation, where data types and performance parameters are not specified in code, but they are passed as template arguments in the compilation process. Nevertheless, the specialization of these operations is usually more efficient than a parameterizable process. This entails, having different customized instances for each skeleton, where each instance directly implements a concrete data type and P, and subsequently, the desired version of that skeleton is chosen at compile-time. Using this specialization, it is possible to work directly over customized datatypes in registers, such as *float2*, *float4* and *int4*.

In addition to the use of these skeletons, each algorithm can benefit from different CUDA optimizations, depending on its characteristics.

Archit.	Warps per block	Regs per thread	Shared memory per block	Warp occupancy (W_{SM}/W^{max})	SM blocks (B_a)
Kepler cc 3.5	1	128	3072	25%	16
	2	64	3072	50%	16
	4	32	0	100%	16
	4	40	0	75%	12
	4	**32**	**3072**	**100%**	**16**
	4	32	4096	75%	12
	8	32	6143	100%	8
	8	32	8192	75%	6
	16	32	12285	100%	4
	16	32	16384	75%	3
	32	32	24576	100%	2
Maxwell cc 5.0	1	64	2048	50%	32
	2	32	0	100%	32
	2	40	0	75%	24
	2	**32**	**2048**	**100%**	**32**
	4	32	4096	100%	16
	8	32	8192	100%	8
	16	32	16384	100%	4
	32	32	32768	100%	2

Table 4.2: Performance parameters which maximize the number of warps and blocks per SM

4.3. Performance Parameter Tuning Phase

After having identified the GPU performance parameters in the first phase, and having developed the kernels for each algorithm in the second one, the suitable values for the performance parameters need to be found. The choice of these values takes into consideration the algorithm features, the problem size and the target architecture. Keeping Section 4.1.1 in mind, Table 4.2 shows some possible configurations for a given W_B entry, which obtain the maximum number of threadblocks and warps per SM, although there are other possible configurations that do not ap-

pear. This analysis is performed for Kepler and Maxwell architectures, with *compute capabilities 3.5* and *5.0*, respectively. As in the previous phase, the *Performance Parameter Tuning* phase depends on the algorithm features. Hence, this table will be used later to configure the (s, p, l) values of each proposal in Section 4.4, Section 4.5 and Section 4.6. It is important to note the bold row of 4 warps per block on Kepler architecture, and the bold row of 2 warps per block on Maxwell platforms, since they achieve both the maximum block parallelism and warp occupancy when working with these parameters values. Henceforth, we are assuming single-precision floating points to calculate these tunable values.

While the first phase of our strategy is common to all algorithms, the second and third phases are specific to each one. The second phase applies different *CUDA* optimizations to each method, and the third phase chooses the suitable performance parameters for each algorithm. In Section 4.4, Section 4.5 and Section 4.6, each algorithm is analyzed and improved with the applicable optimizations, the technique commented previously in Section 4.2, *Hybrid communication strategy inside a block*, is explained in greater detail, and the optimal performance parameters are chosen. Finally, our proposals are compared against a non-skeleton implementation and a naive *BPLG* implementation (which uses previous skeleton implementations) in the experimental results in order to demonstrate the efficiency of our new skeletons implementations.

It should be noted that these parameter values can be theoretically obtained for any architecture by simply constructing a similar table as the one shown in Table 4.2 for the supported sizes and architectures. In terms of CUDA code, these values are defined in a static table and easily loaded and sent to kernels at compile-time, thus the tuning procedure has no time penalty in our approach.

4.4. Tridiagonal System Solvers under a three-phase methodology

This section presents the tridiagonal systems solvers [28] [33] used in our library and their implementation following our three-phase methodology.

4.4.1. Cyclic Reduction Tridiagonal System Solver (BPLG-CR-TS Algorithm)

CR comprises two phases: *forward reduction* and *backward substitution*. Its reduction schema is shown in Section 2.3.2. As explained previously, forward reduction reduces a system to another with half the number of unknowns, until a 2-unknowns system is reached. In each step of backward substitution, odd-indexed unknowns x_i are solved in parallel, by substituting the previously computed values x_{i-1} and x_{i+1} into the equation E_i. The forward reduction is performed in $log_2 N$ steps (see Figure 4.1 (a)), whereas the backward substitution needs $log_2 N - 1$ additional steps.

Figure 4.2 presents the CR forward reduction code using *BPLG* skeletons:

- Initialization section (lines 3-7). Registers and shared memory are allocated using the customized *Float4* data type in order to represent equations, reducing the number of memory transactions.

- Load data from global memory (lines 8-9). Instead of accessing a single data element per memory request, a 128-bit load to obtain 4 consecutive elements is used. Function $copy < X, p >$ loads X elements by p times, thus $copy < 3, p >$ is used in this case, since each thread works with 3 equations.

- First step of the algorithm (lines 10-11). The first computing step is performed.

- Remaining computing steps in forward reduction (lines 16-26). Both offsets and strides are efficiently calculated using bit masks, binary operators and displacements. First, it stores equations from previous step into shared memory. After a synchronization barrier, each thread loads new equations, from shared memory to registers, for the corresponding computing step and their reduction is performed in registers.

- Backward substitution phase (lines 28-41). The loop loads the equation necessary to solve the corresponding unkown in global memory. Even for smaller problems, coalescence is achieved thanks to the cache hierarchy of the *GPU*.

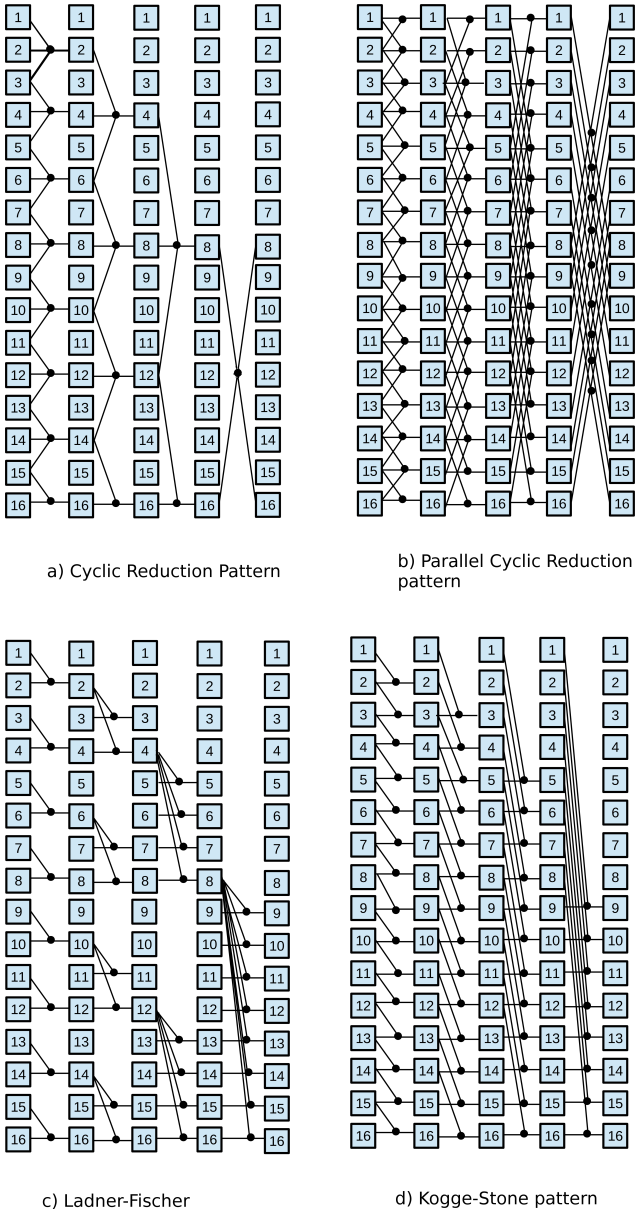


Figure 4.1: Parallel Prefix Patterns for $N = 16$.

CUDA Kernel Optimization phase: BPLG-CR-TS

A *BPLG-CR-TS-Naive* implemention uses previous *BPLG* skeletons [81] to test its performance. Regarding the *Hybrid communication strategy* in our new proposal, the pattern of this algorithm complicates the use of shuffle instructions, as the same equation is shared among several threads, giving rise to a dependency among warps.

```
1  template<int N, int p, int S> __global__ void
2  BPLG_CR(const float* __restrict__ src, float* dstX)
3          Float4 reg[p*3];
4          __shared__ Float4 shm[N > p ? S : 1];
5          //Load data from global mem to reg
6          copy<3,p>(...);
7          //First computing step
8          compute<p,MixR>(reg);
9          for(int accR=MixR; accR < N/2 ; accR*=2) {
10                 //Obtains strides and offsets
11                 int readOffset = ..., readStride = ... ;
12                 int writeOffset = ..., writeStride = ... ;
13                 //Reg-> Shm -> Reg
14                 if(accR>MixR) __syncthreads();
15                 copy<1,p>(shm+writeOffset, writeStride,reg,...);
16                 __syncthreads();
17                 copy<3,p>(reg,shm+readOffset,readStride,...);
18                 compute<p>(reg); //Computation in registers
19         }
20         //Thread 0 solves x_N and x_N/2
21         if(!threadId){
22                 ...
23         }
24         //Backward substitution
25         for(int j= N/2, num_threads=2; j>1; j/=2, num_threads*=2) {
26                 __syncthreads();
27                 int readOffset = ...,  writeOffset = ...;
28                 if(threadId<num_threads)
29                 {
30                         copy<1,p>(reg, shm+readOffset,..); //copy from
                               Shm the corresponding equation
31                         float value= ... ; // Solving unknown
32                         copy<1,p>(dstX+writeOffset, ... ); //Update the
                               unknown value in Glb Memory
33                 }
34         }
```

Figure 4.2: Forward Reduction code for CR tridiagonal algorithm using BPLG.

In order to address this, shuffle instructions are applied in the final steps of the reduction phase, and in the initial steps of the substitution phase. Specifically, the selected steps are those where all Node operators are located in the same warp. In the remaining steps, shared memory is used for communication. Thus, there are no synchronization barriers along the shuffle steps.

Unknowns can be stored either in global memory, reusing the independent-terms vector, or in shared memory. On the one hand, since the substitution phase has $log_2 N - 1$ steps, multiple accesses are performed; thus, global memory latency could be a limiting factor. On the other hand, shared memory consumption could suppose a disadvantage, as the SM block parallelism might be reduced. Selecting which memory is better for the substitution phase depends on the *CUDA* architecture. In

Section 4.4.4, *BPLG-CR-TS-GM* represents the use of global memory when storing the unknowns vector, whereas *BPLG-CR-TS-SH* tags the case of shared memory usage.

Performance Parameter Tuning phase: BPLG-CR-TS

As explained in Section 4.1.1, the objective is to maximize the warp parallelism. Firstly, it should be pointed out that each element is an equation composed of 4 coefficients (4×4 bytes in the case of single precision). With $P = 2$ ($p = 1$), each thread stores 2 equations (8 coefficients) in registers, but since it needs a third equation to perform the reduction, it takes this auxiliary equation from another thread. Thus, it is important to take into account this extra register consumption. The (s, p, l) values for the Kepler architecture are explained in Table 4.3. If $n \leq 6$, the computation can be performed exclusively with shuffle instructions ($N \leq P \times W$), without shared memory. Keeping in mind Table 4.2, the optimal configuration, which maximizes both threadblock and warp parallelism, is achieved with $l = 7$ and fewer than 32 registers per thread in Kepler architecture (the 3rd row in Table 4.2). Taking into consideration additional variables and index calculation, p must be equal to 1 in order to consume less than 32 registers per thread. Finally, since each problem is performed with up to $W = 32$ threads, and $l = 7$, then $L_G > 1$ in this case.

When $n = 7$, it is possible to avoid shared memory communications, using $p = 2$ ($N \leq P \times W$), but this consumes more than 32 registers. Thus, there are three different options: the first case, using $l = 6$ with either $(7, 1, 6)$ or $(0, 2, 6)$ (2nd row in Table 4.2), which achieves 50% of warp occupancy; the second option is $(0, 2, 7)$ (4th row in the table), achieving 75% of warp occupancy and $B_a = 12$; and finally, $(8, 1, 7)$ (6th row in the table), which obtains 75% of warp occupancy and $B_a = 12$. Hence, both $(0, 2, 7)$ and $(8, 1, 7)$ show the maximum parallelism, choosing $(0, 2, 7)$ in this case to avoid the shared memory communications.

When $n > 7$, shared memory is needed for storing the problem data; thus $s = n$. When s occupies more than 3072 bytes, as in the the case of $n > 7$, neither the maximum warp parallelism nor the maximum block parallelism is possible, choosing the configurations that maximize the warp occupancy (6th, 8th and 10th row in

Problem size	*(s,p,l) values*
Kepler Platform	
$n \leq 6$	$(0, 1, 7)$
$n = 7$	$(0, 2, 7)$
$n > 7$	$(n, 1, n-1)$
Maxwell Platform	
$n \leq 6$	$(0, 1, 6)$
$n = 7$	$(0, 2, 6)$
$n > 7$	$(n, 1, n-1)$

Table 4.3: Description of tridiagonal tuning parameters.

Table 4.2). To maximize warp parallelism, $p = 1$ is always established; thus, $l = s-1$, since $s = p + l$. Shared memory becomes the limiting performance resource in this approach, but still achieving a warp occupancy of 75%.

A similar reasoning can be applied in Maxwell architecures, following Table 4.2, which obtains the tuning values shown in Table 4.3.

4.4.2. Parallel Cyclic Reduction Tridiagonal System Solver (BPLG-PCR-TS Algorithm)

PCR is a modification of CR that performs more Node operators per step but, instead, the substitution phase is solved in a single step, using the same formulas and updating mechanism as CR (see Figure 4.1 (b)). In fact, PCR is a variant of CR that only changes the number of reductions performed per step, but not the reduction method itself.

A straightforward implementation of Figure 4.1 (b) with previous *BPLG* skeletons is called *BPLG-PCR-TS-Naive* and its code has almost the same structure that of the CR, as shown in Figure 4.3. After the reduction phase, results are stored in shared memory for the substitution phase, which requires half as many Node operators as the reduction phase. Finally, the value is stored directly in global memory.

```
template<int N, int p, int S> __global__ void
BPLG_PCR ( const float* __restrict__ src, float* dstX ) {
        Float4 reg[p*3];
        __shared__ Float4 shm[N > p ? S : 1];
        //Load data from global mem to shm
        copy<1,p>(...);
        __syncthreads();
        //Load data from shm to reg
        copy<3,p>(...);
        //First computing step
        compute<p,MixR>(reg);
        for(int accR=MixR; accR < N/2 ; accR*=2) {
                //Obtains strides and offsets
                int readOffset = ..., readStride = ... ;
                int writeOffset = ..., writeStride = ... ;
                //Reg-> Shm -> Reg
                if(accR>MixR) __syncthreads();
                copy<1,p>(shm+writeOffset, writeStride,reg, ...);
                __syncthreads();
                copy<3,p>(reg,shm+readOffset,readStride, ...);
                compute<p>(reg); //Computation in registers
        }
        copy<1,p>(...);
        _syncthreads();

        //Substitution phase
                ...
}
```

Figure 4.3: Code for the PCR tridiagonal algorithm in BPLG.

CUDA Kernel Optimization phase: BPLG-PCR-TS

Two different techniques have been developed for this algorithm in order to take advantage of its characteristics and communication pattern: the *Efficient Allocation* and the *Equation-warp matching* techniques.

In the *BPLG-PCR-TS-Naive* version, each thread executes one Node operator per step, storing three equations in registers. However, following our three-phase methodology, this pattern needs to be redesigned in order to achieve $l \leq s - 1$. The same equation is shared among different Node operators, as shown in Figure 4.1 (b). Keeping this in mind, it is possible to store 4 equations per thread instead of 6, applying an *Efficient Allocation* technique. The reduction in the number of equations decreases the register usage, leading to more active threadblocks per SM if registers are the limiting factor in occupancy. The key is to rearrange the reading indices per thread in each step, in order to work with nested Node operators. This recalculation can be effectively performed using displacements and bit masks avoid-

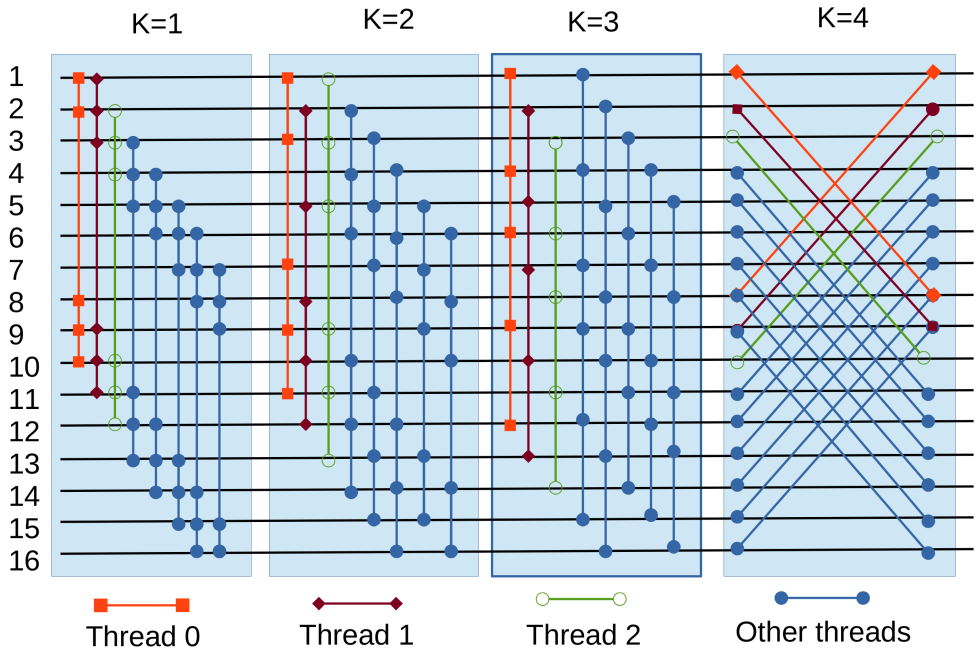


Figure 4.4: Operator nodes allocation for the PCR algorithm with $N = 16$.

ing non-uniform accesses, as the thread offset between equations is a power of two. Figure 4.4 shows the distribution of two Node operators per thread when $N = 16$, without using the *Efficient Allocation* strategy and where each vertical line represents the work per thread. As it can be seen, consecutive threads load consecutive sets of adjacent equations and each thread needs 6 equations per stage. In contrast, Figure 4.5 shows the *Efficient Allocation* technique when $N = 16$.

The second technique, *Equation-warp matching*, allows the efficient utilization of shuffle instructions. Applying shuffle instructions directly in PCR is not trivial. PCR has a collaborative communication pattern in which warps have to always share their equations with other warps, avoiding an independent computation inside the warp. Each computation in a Node operator entails reading equations from other Node operators. If these Node operators are located in the same warp, shuffle instructions can be used for exchanging them; otherwise, they have to take equations from another warp through the shared memory. Hence, we have agglomerated threads to minimize the communication among warps, reordering the equation-warp matching. With this technique, it is now possible to reach a point where there is no

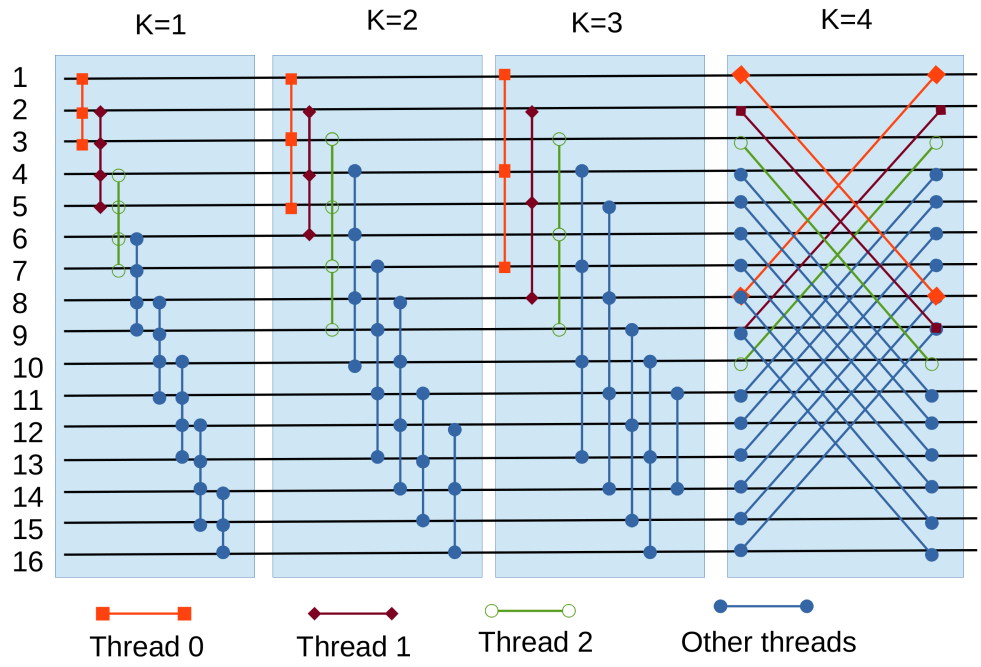


Figure 4.5: Operator nodes for the PCR algorithm with $N = 16$ using the *Efficient Allocation* strategy.

dependence among warps (at the final computing steps, enabling the use of shuffle in these steps and in the substitution phase). Figure 4.6 shows this technique for $N = 16$ with 4 warps, where the gray-line rectangles represent warps, the numbered boxes are threads and the dashed boxes indicate dependencies between threads. For the sake of simplicity and a more compact representation of the example, it is assumed that the warp size is composed of 4 threads. In $k = 1$, in order to reduce the first and last equation of each warp, it is necessary to access the last and the first thread from the previous and the next warp, respectively. When reaching $k = 3$, there is no dependence among warps, thus a shuffle computation is possible. Specifically, the mapping that assigns Node operators to warps is now expressed by

$$Mapping(NO_i) = \frac{ROR(NO_i, k-1)}{W} \quad with \;\; 0 \leq i \leq N/P \tag{4.2}$$

where NO_i is the Node operator *id*, W the warp size and ROR moves $k-1$ bits of NO_i that falls off the least-significant bit up to the most-significant bit in the word; bits moved out the right-hand end are rotated back into the left-hand end.

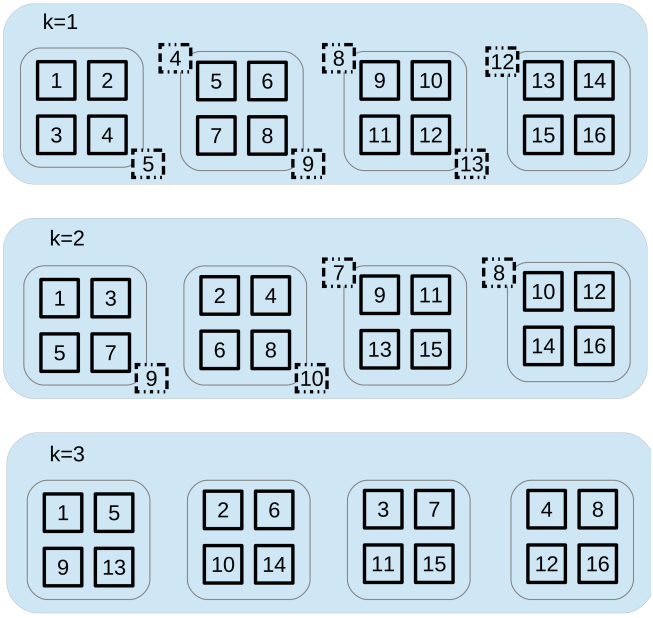


Figure 4.6: PCR dependences when applying the *Equation-warp matching* with $N = 16$.

Performance Parameter Tuning phase: BPLG-PCR-TS

The same tuning values as the ones explained for CR in Table 4.3 are obtained for PCR. However, in comparison to CR, each thread takes two auxiliary equations from other threads, instead of one, having to store them in additional registers. In Kepler, when $n \leq 6$, the tuple $(s, p, l) = (0, 1, 7)$ is obtained with $L_G > 1$, without shared memory consumption. With $n = 7$, $(s, p, l) = (0, 2, 7)$ is set up, and $(s, p, l) = (n, 1, n - 1)$ is established in the remaining cases, obtaining 75% warp occupancy for each size. If $n > 7$, it is not possible to achieve the maximum parallelism due to shared memory consumption, which is the limiting factor here. Maxwell parameters are also outlined in the Table 4.3.

4.4.3. Ladner-Fischer Tridiagonal System Solver (BPLG-LF-TS Algorithm)

In Section 3.1, a new Node operator for reducing tridiagonal systems was presented, called *Redundant Reduction* (RR). Unlike CR and PCR, each Node operator reads only 2 equations, instead of 3, when performing the reduction. This new Node operator has to be combined with a communication pattern in order to build the parallel prefix algorithm. In this chapter, this new solver is implemented under our tuning methodology. Here, the Ladner-Fischer pattern was chosen (see Figure 4.1(c)), as its communication pattern combined with RR has shown the best performance.

Figure 4.7 presents the reduction phase of the algorithm, using *BPLG* skeletons:

- Initialization section (lines 3-5). As previously, registers and shared memory are allocated. In contrast to previous algorithms, this proposal needs to store two forms of each equation, doubling the shared memory requirements and the amount of registers.
- Load data from global memory (lines 7) and first mixed computing step (line 9). Each node loads 64-bit coalescing data, obtaining 2 consecutive elements. These data are stored in the *form1* registers. Then, the mixed computing step performs the first reduction. As initial equations are specified in a single form, the mixed *compute* function copies this result to the *form2* registers, obtaining the *I form* and the *C form* of each equation.
- Remaining computing steps (lines 10-25). First write in shared memory does not need a previous synchronization barrier, as no element has been previously read from shared memory. Then, each Node operator stores the two equations in their two forms. Remaining writes only store the modified equation in shared memory.

CUDA Kernel Optimization phase: BPLG-LF-TS

An implementation of the algorithm that uses previous *BPLG* skeletons is tagged as *BPLG-LF-Naive*. Regarding the *Hybrid communication strategy* of this proposal,

```
template<int N, int p, int S> __global__ void
BPLG_LF ( const float* __restrict__ src, float* dstX ) {
        Float4 form1[2*p], form2[2*p];
        __shared__ Float4 shm[N > p ? S : 1];
        __shared__ Float4 shm2[N > p ? S : 1];
        //Load data from global mem to registers
        copy<2,p>(form1,...);
        //First computing step
        compute<p,MixR>(form1,form2);
        for(int accR=MixR; accR < N ; accR*=2) {
                //Obtains strides and offsets
                ...
                //Reg-> Shm -> Reg
                if(accR>MixR) __syncthreads();
                if(accR==MixR)
                        copy<2,p>(shm+writeOffset,writeStride,form1);
                        copy<2,p>(shm2+writeOffset,writeStride,form2);
                else
                        copy<1,p>(shm+writeOffset, writeStride,form1);
                        copy<1,p>(shm2+writeOffset, writeStride,form2);
                __syncthreads();
                copy<2,p>(form1,shm+readOffset,readStride);
                copy<2,p>(form2,shm2+readOffset,readStride);
                compute<p>(form1,form2); //Computation in registers
        }
        //Substitution
        ...
}
```

Figure 4.7: Code for LF tridiagonal algorithm in BPLG.

shuffle instructions are used for small problem sizes ($N \leq P \times W$) to avoid shared memory communications. As N grows, a hybrid strategy is followed, combining both shuffle and shared memory communication: firstly, $P \times W$ equations are reduced with shuffle instructions in the early steps. Considering k as the current step of the execution, where $k = \{0, 1, .., n-1\}$, the communication through shared memory are applied in the k-steps that meet $2^k > P \times W$.

Performance Parameter Tuning phase: BPLG-LF-TS

The RR reduction operation uses two representatives for each element; thus, each element stores two equations of 4 coefficients each ($2 \times 2 \times 4 \times 4$ bytes per thread, in case of *floats* and $P = 2$). Hence, this condition must be taken into account when selecting the performance parameters based on Table 4.2. As the number of registers must not exceed 32, considering registers employed for additional variables, p must be strictly 1. The same performance values as in previous algorithms are obtained,

shown in Table 4.3, although the achieved parallelism is different.

In Kepler architecture, three cases are defined. First, as $P \times W$ elements can be performed exclusively with shuffle instructions, shared memory is not used when $n \leq 6$; thus, $(s, p, l) = (0, 1, 7)$ with $L_G > 1$. In the case of $n = 7$, there are two possibilites, using the hybrid communication strategy with $p = 1$ or shuffle instructions exclusively with $p = 2$. Using $(0, 2, 7)$ achieves a warp occupancy of 56%, whereas $(8, 1, 7)$ only reaches 38% (these entries do not appear in Table 4.2). For the remaining sizes, shared memory is needed and more than 3072 shared memory bytes are consumed. To obtain the highest warp parallelism possible, $(s, p, l) = (n, 1, n-1)$ is utilized when $n > 7$.

In Maxwell architecture, three cases are also considered. In the first case, $(s, p, l) = (0, 1, 6)$, with $L_G > 1$. In the second case, when $n = 7$, $(s, p, l) = (0, 2, 6)$ is established. Otherwise, shared memory becomes the performance limiting factor, and $(n, 1, n - 1)$ is employed.

4.4.4. Experimental Results for Tridiagonal System Solvers with Small Problem Sizes

In this section, the results of our proposals on different CUDA GPU architectures are presented and analyzed. All the tests were run in single precision, and all the data reside in the GPU device memory at the beginning of each test, so there are no data transfers to CPU during the benchmarks to prevent interactions with other factors in the study. The test platforms used in our experiments are described in Table 4.4, composed of a Kepler Platform and a Maxwell Platform. The kernels were executed setting the *cudaFuncCachePreferShared* cache configuration flag in the Kepler platform, so it is possible to use up to 48 *KB* of shared memory, but only 16 KB of *L1* cache are available. In the case of the Maxwell architecture, although each SM contains 96 KB of shared memory, each block can only use up to 48 KB. Different driver versions were also tested with little impact on performance.

Firstly, we compare our three-phase strategy implementations against the same algorithm implemented directly without any *BPLG* skeleton, in order to see the impact of the *BPLG* library on the results, and also against previous *BPLG* skeletons

	Kepler Platform	Maxwell Platform
CPU	Intel Xeon E5-2660 CPU 2.2 GHz	Intel Core i7-2600 3.4 GHz
Memory	64 GB DDR3 1600	8 GB DDR3 1333
OS	CentOS 6.4	Ubuntu 14.04 LTS
Compiler	GCC 4.4.7	GCC 4.8.4
GPU	Nvidia Tesla K20 GPU	Nvidia GeForce GTX980
Driver	367.57, SDK 8.0	384.90, SDK 8.0

Table 4.4: Description of the test platforms

implementations (*Naive version*). We then compare the results of our library with respect to the *CUDPP* [98] and *CUSPARSE* [95] libraries, and a tridiagonal system solver based on the Wang&Mou algorithm (BPLG-WM-TS) [82]. The performance of the experiments for solving tridiagonal systems is measured in million rows processed per second, MROWS/s, using diagonally dominant systems, the same metric and assumptions as [74] does. The number of concurrent problems depends on the input size and is given by the expression $G = 2^{24}/N$. Thus, the MROWS/s value is obtained using the expression: $N \cdot G \cdot 10^{-6}/t$.

BPLG-CR-TS Results

Concerning the Cyclic Reduction algorithm, Figure 4.8 shows a comparison among different implementations on the Kepler Platform. A CR implementation without using *BPLG* (*CR-TS* proposal in graphics) is also considered. The performance of *BPLG-CR-TS-Naive* achieves an improvement of up to 3.7x over the *CR-TS* implementation. However, better results are obtained using *BPLG-CR-TS-GM* and *BPLG-CR-TS-SH*, which apply the optimizations explained in Section 4.4.1. The difference between them is the place where the unknown array is stored: global memory or shared memory. As expected, *BPLG-CR-TS-GM* performs better due to shared memory constraints on Kepler, obtaining an improvement of up to 13x over the *CR-TS* implementation and 3.6x over *BPLG-CR-TS-Naive*. When $n < 7$, this implementation can execute $B_a = 16$ active blocks per SM with 100% of warp occupancy. In the case of $n = 7$, it achieves $B_a = 10$ blocks but warp occupancy of 63% due to register consumption. The occupancy is lower if N grows due to shared

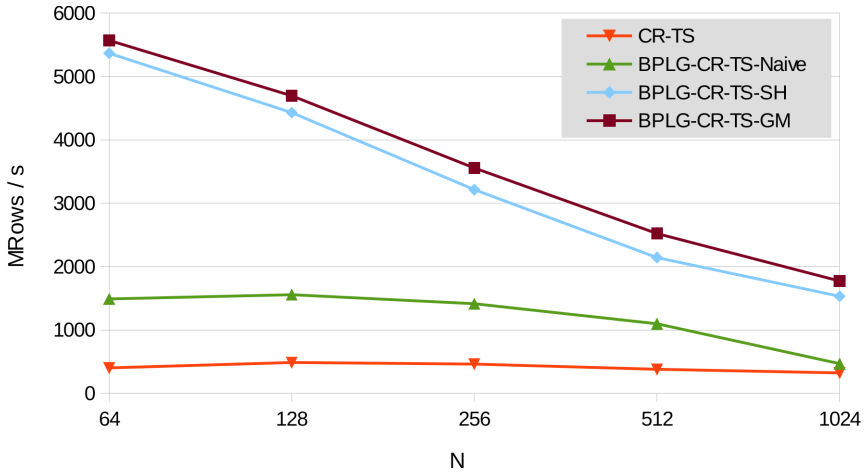


Figure 4.8: MRows/s comparison of the CR tridiagonal proposals in the Kepler Platform.

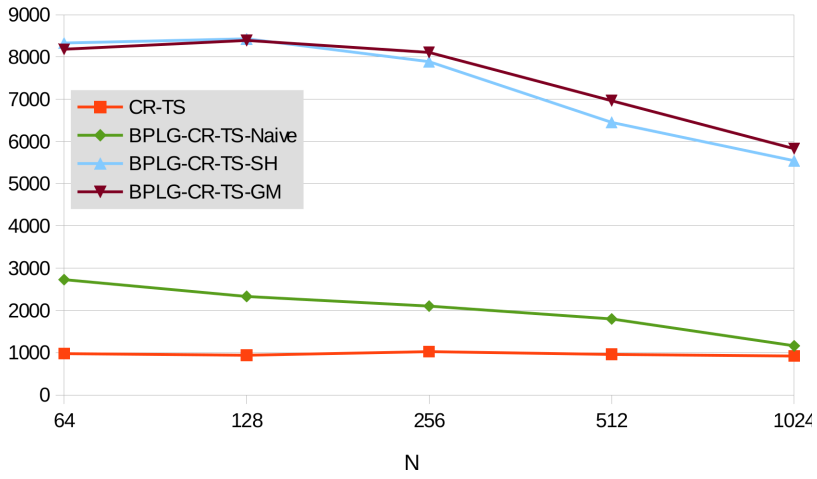


Figure 4.9: MRows/s comparison of the CR tridiagonal proposals in the Maxwell Platform.

memory requirements and thread block size. When $n = 8$, $B_a = 12$ blocks and warp occupancy of 75% are reached. In the case of $n = 9$, $B_a = 6$ blocks and warp occupancy of 75% are obtained and, finally, $B_a = 3$ blocks and warp occupancy of 75% when $n = 10$.

On the other hand, very similar results are obtained on the Maxwell architecture solving the unknowns vector in global memory or in shared memory, as shown in Figure 4.9. The new Maxwell architecture provides 96 KB per SM of shared memory

(although only 48 KB can be used within a block). Thus, the unknown array is stored in shared memory without much penalty. When $n \leq 6$, this implementation achieves 100% block and warp occupancy. In the case of $n = 7$, the register consumption is higher than 32 registers due to $p = 2$, thus $B_a = 20$ blocks and warp occupancy of 63% are obtained. For the remaining cases, the following occupancies are reached: $B_a = 16$ blocks and warp occupancy of 100% when $n = 8$, $B_a = 8$ blocks and warp occupancy of 100% in the case of $n = 9$ and $B_a = 4$ blocks and warp occupancy of 100% if $n = 10$. These final approaches obtain an improvement of up to 8.9x over *CR-TS* and up to 5x over *BPLG-CR-TS-Naive*.

BPLG-PCR-TS Results

Figure 4.10 shows the performance of the Parallel Cyclic Reduction algorithm in the Kepler Platform. An implementation of this algorithm without using *BPLG* is denoted as *PCR-TS* in Figure 4.10. As can be observed on the Kepler Platform, the naive approach (*BPLG-PCR-TS-Naive*) offers a clear advantage over *PCR-TS*, achieving up to 6x of improvement; whereas on the Maxwell Platform, it reaches a remarkable 5.30x. Concerning our proposals, *BPLG-PCR-TS* uses the *Efficient Allocation* and *Equation-warp matching* strategies. On the Kepler Platform, if $n < 7$, it obtains $B_a = 12$ concurrent blocks and 75% of warp parallelism due to the use of 37 registers per thread. When $n = 7$, our proposal achieves $B_a = 9$ active blocks per SM and 56% warp occupancy, owing to $p = 2$. In the remaining cases, $B_a = 12$ blocks and 75% warp occupancy are achieved when $n = 8$; if $n = 7$ then $B_a = 6$ blocks and 75% warp occupancy are obtained, and $B_a = 3$ blocks with 75% warp occupancy in the case of $n = 10$, being up to 1.36x times faster than *BPLG-PCR-TS-Naive*.

On the Maxwell Platform, Figure 4.11, our proposal has a speed-up of up to 1.19x over *BPLG-PCR-TS-Naive*. For non-shared memory implementations, it obtains $B_a = 24$ blocks and 75% warp occupancy when $n \leq 6$; and $B_a = 18$ blocks and 56% when $n = 7$. In the remaining cases, it always achieves 75% warp occupancy, where the number of blocks $(12, 6, 3)$ for $n = (8, 9, 10)$, respectively.

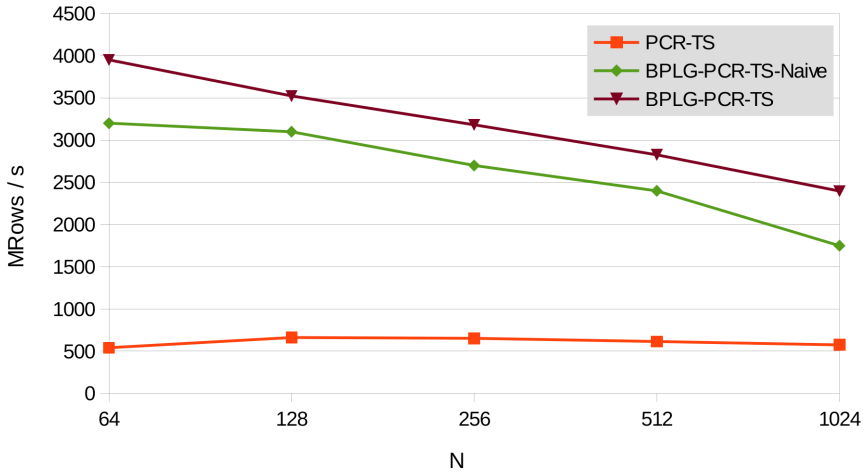


Figure 4.10: MRows/s comparison of the PCR tridiagonal implementations in the Kepler Platform.

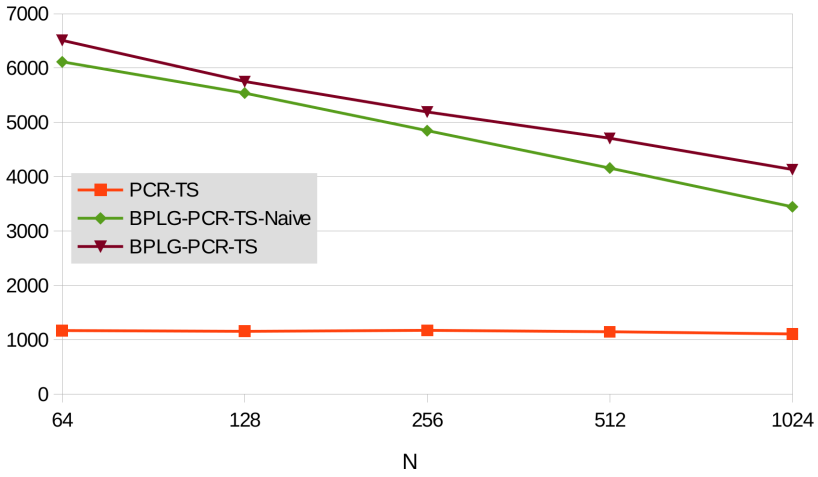


Figure 4.11: MRows/s comparison of the PCR tridiagonal implementations in the Maxwell Platform.

BPLG-LF-TS Results

Figure 4.12 shows the performance of our Ladner-Fischer approaches on the Kepler Platform. Firstly, an implementation of LF without *BPLG*, tagged as *LF-TS* in graphics, is provided. Then, *BPLG-LF-TS-Naive* is also implemented, achieving an improvement of up to 2.95x over *LF-TS* in Kepler, and up to 2.05x in the Maxwell version. *BPLG-LF-TS* achieves the best improvement, being up to 7.60x better than *LF-TS* and up to 2.76x with respect to *BPLG-LF-TS-Naive* in the

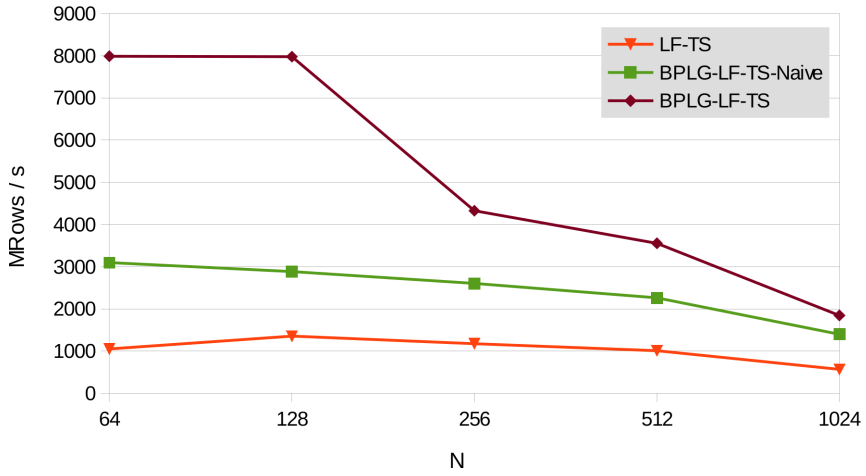


Figure 4.12: MRows/s comparison of LF tridiagonal implementations in the Kepler Platform.

Kepler architecture.

In Maxwell, Figure 4.13, up to 3.07x and 2.1x speed-ups were obtained, respectively. This algorithm consumes a high amount of shared memory, twice that consumed with other tridiagonal systems solvers. However, when $N \leq (W \times P)$, there is no shared memory consumption, as exchanges can be performed in registers with shuffle instructions using our *Hybrid communication strategy*. That is the reason why in Figure 4.12 there is a decrease in performance after $n = 7$ ($N = 128$), since occupancy decreases due to shared memory restrictions. Table 4.5 shows the occupancies achieved for both platforms. The limiting factor in all problem sizes is shared memory. In contrast to Kepler, throughput in Maxwell does not decrease along N, despite shared memory consumption. For example, with floats, when $n = 10$, a number of 2048 equations of 4 coefficients are stored (32768 bytes) in shared memory. In this case, Kepler obtains 25% warp occupancy due to shared memory limitations, but in Maxwell, thanks to its 96 KB of shared memory, warp occupancy rises to 50%, it also being able to allocate more active blocks.

Overall Results

Finally, Figure 4.14 and Figure 4.15 provide a global overview, and a comparison with respect to *CUSPARSE* [95] and *CUDPP* [98] libraries, and a *BPLG* implemen-

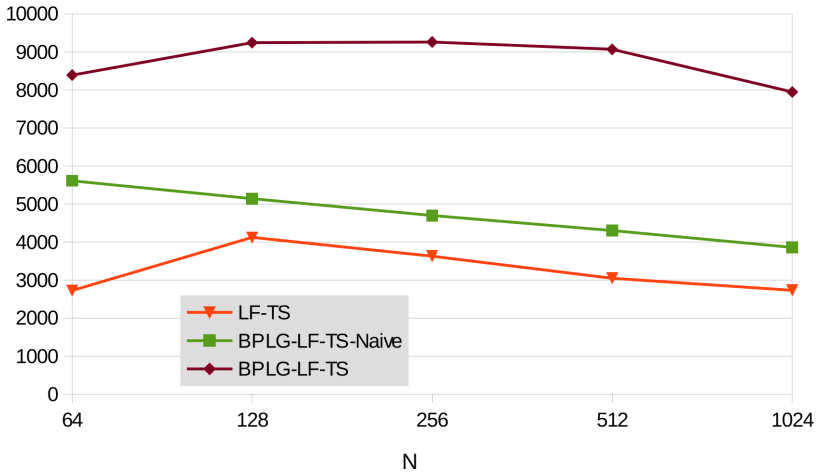


Figure 4.13: MRows/s comparison of LF tridiagonal implementations in the Maxwell Platform.

	Kepler Platform		Maxwell Platform	
n	Block Parallel.	Warp Occup.	Block Parallel.	Warp Occup.
6	12	75,00%	24	75,00%
7	9	56,00%	18	56,00%
8	6	38,00%	8	50,00%
9	3	38,00%	4	50,00%
10	1	25,00%	2	50,00%

Table 4.5: BPLG-LF-TS occupancies

tation of Wang&Mou. All these libraries are outperformed by our proposals. The *CUSPARSE* library launches several kernels to solve the batch problem; therefore, the global memory bandwidth becomes a limiting factor. Only for larger problems does it amortize the cost of the multi-kernel approach. Regarding *CUDPP*, it assigns a single tridiagonal system problem to each threadblock and some threads will be idle.

The Kepler results are shown in Figure 4.14, where *BPLG-LF-TS* achieves the best results among our proposals for $N \leq 512$, obtaining up to 9.31 x of speed-up over *CUSPARSE*, up to 9.37x over *CUDPP* and up to 2.01x over *BPLG-WM-TS*. For larger problems, $N = 1024$, the shared memory becomes a limiting factor since it has to store two forms per equations, and *BPLG-PCR-TS* obtains the maximum

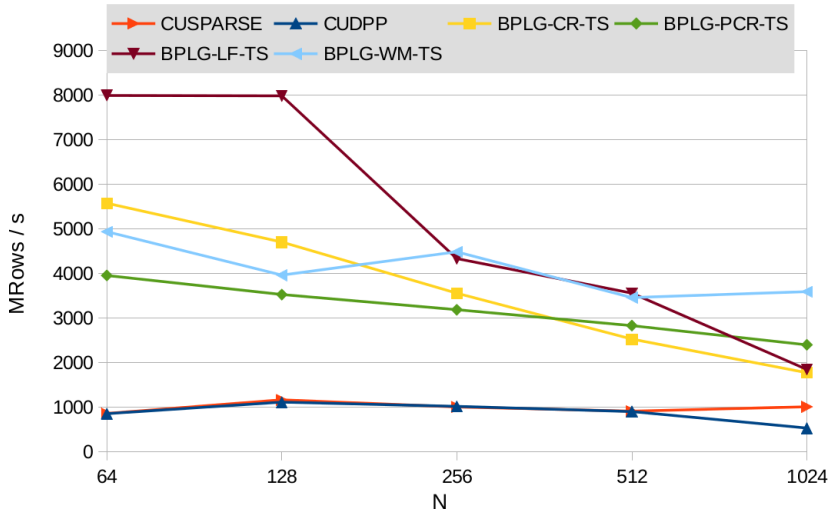


Figure 4.14: Comparison of BPLG tridiagonal solvers performance in the Kepler Platform.

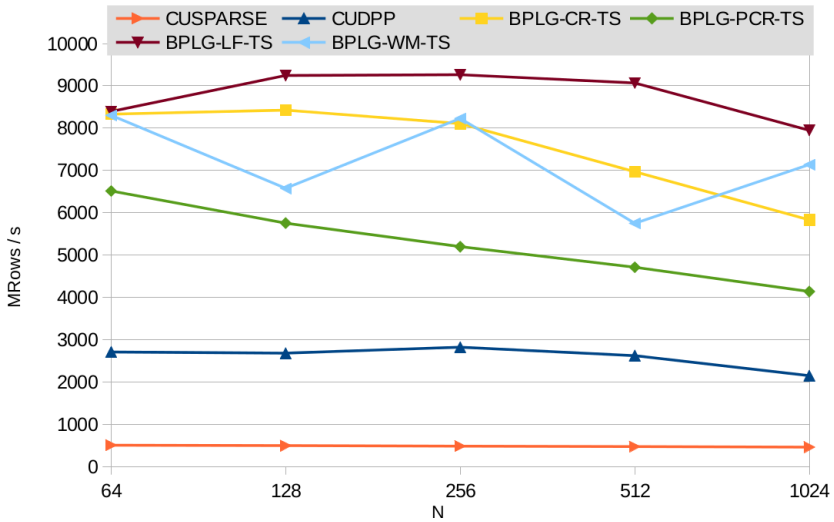


Figure 4.15: Comparison of BPLG tridiagonal solvers performance in the Maxwell Platform.

speed-up, being 4.6x faster than *CUSPARSE*, and 4.63x faster than *CUDPP* in the worst-case scenario. *BPLG-CR-TS* also outperforms the *CUSPARSE* and *CUDPP* libraries, being up to 6.49x and 6.53x faster, respectively. Nevertheless, none of our proposals surpasses *BPGL-WM-TS* for this size.

Figure 4.15 presents a similar performance comparison on the Maxwell Platform.

In Maxwell, our *BPLG-LF-TS* proposal is the best one, being up to 19.28x faster than *CUSPARSE*, presenting a speed-up of 3.7x over *CUDPP* and up to 1.57x over *BPLG-WM-TS*. In the case of *BPLG-PCR-TS*, our proposal is up to 12.9x faster than *CUSPARSE* and 2.4x faster than *CUDPP*, but it does not outperform *BPLG-WM-TS*; whereas *BPLG-CR-TS* has a speedup of up to 12.94x over *CUSPARSE*, up to 2.4x over *CUDPP*. As the main limitation of our approaches is shared memory, the new Maxwell's shared-memory increment helps to increase occupancy in our proposals, increasing the performance gap between our solvers and the state-of-the-art ones.

Efficiency of the Performance Parameters Tuning Phase

In order to check the effectiveness of our strategy, this section compares the performance achieved by using the proposed performance parameters, against the performance achieved by an exhaustive search of performance parameters for the Kepler platform. Table 4.6 demonstrates the success of our strategy criteria for *BPLG-LF-TS*, where the maximum performance achieved empirically is shown in colored cells, whereas the performance resulted by using our strategy is highlighted for each N size. Please note that there are some configurations that are not allowed due to shared memory consumption limitations or sizes greater than $p + l$. In this case, the performance parameters proposed were $(s, p, l) = (0, 1, 7)$ when $n \leq 6$; $(s, p, l) = (0, 2, 7)$ when $n = 7$; and $(s, p, l) = (n, 1, n-1)$ in the remaining cases. As can be observed, all tuples proposed in the *Performance Parameter Tuning phase* of the algorithm match with the results obtained empirically.

4.5. Scan Primitive under a three-phase methodology

In this section, two different parallel prefix algorithms [28] are implemented to compute the scan under our three-phase methodology.

(p,l) \ n		$n=6$	$n=7$	$n=8$	$n=9$	$n=10$
$p=1$	$l=5$	4890	x	x	x	x
	$l=6$	7716	4688	x	x	x
	$l=7$	***8152***	4725	***4326***	x	x
	$l=8$	8052	4245	3838	***3554***	x
	$l=9$	7878	2169	2127	2054	***1839***
	$l=10$	4690	x	x	x	x
$p=2$	$l=5$	5591	5737	x	x	x
	$l=6$	7079	7940	3153	x	x
	$l=7$	7241	***8033***	3087	2812	x
	$l=8$	6693	7463	1799	1700	1554
	$l=9$	6680	6523	x	x	x
	$(l=10$	4800	4992	x	x	x

Table 4.6: Performance comparison of different performance parameters values for BPLG-LF-TS in MRows / s

4.5.1. Scan operator using the Ladner-Fischer pattern (BPLG-LF-SC Algorithm)

This proposal is our own adaptation of the *Ladner-Fischer* parallel prefix algorithm (see Figure 4.1 (c)) for performing scan operations. Solutions are obtained after $log_2 N$ steps, and unlike with other scan algorithms, here the number of read and write operations remains invariant over all steps; i.e. the number of active threads remains constant along steps. Figure 4.16 contains the code structure with *BPLG* skeletons for the *BPLG-LF-Naive* algorithm, considering $p = 1$. The code can be divided into four main sections:

- Initialization section (lines 3-4). This allocates registers and shared memory.
- Load data from global memory (lines 5-6) and first computing step (line 7). This section of code changes with respect to tridiagonal solvers, as elements are not shared by different nodes in the first step, so they can be directly loaded from global memory to registers, and computed in registers.

```
1  template<int N, int p, int S> __global__ void
2  BPLG_LF_scan ( const float* __restrict__ src,  float* dst ) {
3          float reg[p*2];
4          __shared__ float shm[N > p ? S : 1];
5          //Load data from global memory to regs
6          copy<2,p>(...);
7          compute<p,MixR>(reg);
8          for(int accR=MixR; accR < N ; accR*=2) {
9                  //Obtains strides and offsets
10                 ...
11                 //Reg-> Shm -> Reg
12                 if(accR>MixR) __syncthreads();
13                 copy<1,p>(shm+writeOffset, writeStride,reg);
14                 __syncthreads();
15                 copy<2,p>(reg,shm+readOffset,readStride);
16                 compute<p>(reg); //Computation in registers
17         }
18         copy<1,p>(dst+glbWPos,reg,...); //Storing data in Glb
               memory
19         copy<1,p>(dst+glbWPos,shm+shmOffset+...);
20 }
```

Figure 4.16: Kernel code for the LF-scan algorithm in BPLG.

- Remaining computing steps (lines 8-17). The loop reorders data registers using shared memory and performs the computation in each step.
- Storing results to global memory (lines 18-19). Figure 4.1 (c) shows that the last step of the algorithm processes the second half of data, so threads have the second half of final results stored in registers, after the last iteration. Thus, $N/2$ elements are directly stored from registers into global memory. The remaining elements are moved from shared memory.

CUDA Kernel Optimization phase: BPLG-LF-SC

In Ladner-Fischer, unlike other algorithms, there are not two separate phases for reading and writing operations: elements read by one thread are never overwritten by another thread in the same iteration. Hence, this proposal saves an extra synchronization barrier in each step. However, this pattern can produce shared memory bank conflicts, so it is important to define the pair of elements which are accessed by each thread. Accessing adjancent pairs of elements per thread generates a high percentage of bank conflicts, since some banks are addressed several times while others are never addressed. Specifically, this proposal takes bank conflicts into account looking for the stride among thread elements which reduces these conflicts. To

this end, each thread works with Node operators whose elements are separated by threadblock's size positions. Previous implementations for this algorithm achieved a 5% rate of bank conflicts on average (*shared memory reply overhead* in profiler), but this proposal allows to be reduced to 0.1%.

Regarding the *Hybrid communication strategy inside a block*, shuffle instructions are used to compute the scan in each warp, performing the whole scan in chunks of $P \times W$ elements. After this, each warp saves its partial sum in shared memory (the final element of each chunk), and this process is repeated over the values stored in shared memory. This repetition is currently performed by only one warp, as 32 warps were used at most in the chunk computation. Thus, at most there are 32 elements in shared memory per problem. After computing the scan in shared memory, each warp adds its corresponding element from shared memory to all elements in the chunk, obtaining the final result.

Performance Parameter Tuning phase: BPLG-LF-SC

It should be noted that the *Hybrid communication strategy* stores $S = 32 \times L_G$ elements per threadblock in shared memory, which means, 32 elements per problem being solved. It is also important to emphasize that s is different to $p + l$ here, thanks to the shuffle approach explained previously, where only a small portion of elements are stored in shared memory.

Table 4.7 shows the two tuning cases in the Kepler platform. Looking at Table 4.2, obtaining the maximum warp and block parallelism is possible with $l = 7$ and fewer registers per thread than 32. Taking into account auxiliary variables and index calculation, p must be less than or equal to 2. In addition to this, shared memory consumption must be lower than 3072 bytes. As each problem stores 32 elements in shared memory (128 bytes), the number of concurrent problems per threadblock, L_G, must be less than or equal to $\frac{3072}{128} = 24$. Considering $S = 32 \times L_G$ and $L = L_G \times \frac{N}{P}$, then $s = 5 + (l + p - n)$ and $(s, p, l) = ((14 - n), 3, 7)$ in order to achieve maximum warp and block parallelism. If $n > 9$, the previous tuple cannot be applied since $14 - n$ must be at least 5, as well as the fact that more than 128 threads are needed for solving each problem when $p = 2$. In that case, $L_G = 1$, thus $L = \frac{N}{P}$ and $(s, p, l) = (5, 2, n - 2)$.

Problem size	*(s,p,l) values*
Kepler Platform	
$n \leq 9$	$((14-n), 2, 7)$
$n > 9$	$(5, 2, n-2)$
Maxwell Platform	
$n \leq 8$	$((13-n), 2, 6)$
$n > 8$	$(5, 2, n-2)$

Table 4.7: Description of the LF scan tuning parameters.

Table 4.7 also shows the tuning values for the Maxwell architecture, where the same reasoning was followed to maximize the parallelism. Using 2 warps per block ($l = 6$), fewer than 32 registers per thread and no more than 2048 bytes of shared memory, then 32 active blocks and 100% warp occupancy are achieved. Thus, applying $p = 2$ and the previous premises, $(13 - n, 2, 6)$ is built when $n \leq 8$, with $L_G \leq 16$. The second case is focused on $n \geq 9$, obtaining $(s, p, l) = (5, 2, n - 2)$.

4.5.2. Scan operator using Kogge-Stone pattern (BPLG-KS-SC Algorithm)

A second proposal for the scan algorithm using our three-phase methodology is analyzed here, considering the Kogge-Stone pattern as the communication pattern of the algorithm. This algorithm requires $log_2 N$ steps, as can be observed in Figure 4.1 (d). The number of Node operators per step decreases by a factor of 2^k, introducing divergence into the final warp of each threadblock. By contrast, this pattern reduces bank conflicts since adjacent threads access adjacent shared memory banks. As each element is used by several threads, it is necessary to have a two-phase loading process, using shared memory for loading one element, instead of working directly in registers, as Figure 4.17 depicts.

```
template<int N, int p, int S> __global__ void
BPLG_KS_scan ( const float* __restrict__ src,  float* dst ) {
        float reg[p*2];
        __shared__ float shm[N > p ? S : 1];
        //Load data from global mem to regs
        copy<1,p>(reg,src+...,);
        copy<1,p>(shm+...,reg);
        __syncthreads();
        if(threadId<(blockDim.x-1))
                copy<1,p>(reg+1,shm+...);
        compute<p,MixR>(reg);
        for(int accR=MixR; accR < N ; accR*=2) {
                //Obtains strides and  offsets
                ...
                //Reg-> Shm -> Reg
                if(accR>MixR) __syncthreads();
                if(threadId < (blockDim.x - accR/2))
                        copy<1,p>(shm+writeOffset, writeStride,reg, ...);
                __syncthreads();
                if(threadId < (blockDim.x-accR))
                        copy<2,p>(reg,shm+readOffset,readStride, ...);
                        compute<p >(reg); //Computation in registers
        }
        copy<1,p>(dst+glbWPos,reg,...);
        copy<1,p>(dst+glbWPos,shm+shmOffset+...);
}
```

Figure 4.17: Kernel code for KS scan algorithm in BPLG.

CUDA Kernel Optimization phase: BPLG-KS-SC

Regarding the *Hybrid communication strategy inside a block* logic, this is very similar to the Ladner-Fischer one, but using the $shfl\_up$ instructions instead of $shfl$ instructions, due to the Kogge-Stone communication pattern structure.

Performance Parameter Tuning phase: BPLG-KS-SC

As occurred in Section 4.4.2, here each element is also shared by two different Node operators. Thus, although each Node operator works with P elements, it only has $\frac{P}{2}$ elements stored in its own registers, taking the remaining elements from other threads. This can be expressed as $P' = \frac{P}{2}$, obtaining $L = L_G \times \frac{N}{P'}$, since each element is shared by two Node operators in each computing step.

Since the same *Hybrid communication strategy* as LF is performed here, this algorithm applies the same reasoning as in the previous one. Table 4.8 contains the two distinguished cases for Kepler. The first one considers $n \leq 8$, with $(s, p, l) =$

$(13-n, 2, 7)$, achieving the maximum warp and block parallelism. In higher sizes, the tuple is updated to $(s, p, l) = (5, 2, n-1)$. Regarding Maxwell, the same reasoning is followed in Table 4.8, obtaining $(s, p, l) = (12-n, 2, 6)$ when $n \leq 7$, and $(s, p, l) = (5, 2, n-1)$ when $n \geq 8$.

4.5.3. Experimental Results for the Scan Primitive with Small Problem Sizes

This section presents the results of our strategy implementations for the scan primitive in the Kepler and Maxwell architectures from Table 4.4. As in the case of tridiagonal system solvers, we have compared our implementations against both non-*BPLG* and *BPLG*-naive implementations. Finally, each proposal is compared with respect to the *CUDPP* [98], *Thrust* [101], *ModernGPU* [97] and *CUB* [100] libraries. In the case of scan algorithms, the performance is expressed in million elements processed per second, MDATA/s, following the expression $N \cdot G \cdot 10^{-6}/t$ where $G = 2^{21}$.

BPLG-LF-SC Results

Figure 4.18 and Figure 4.19 depict the performance of our different versions for the Ladner-Fischer scan algorithm. Specifically, Figure 4.18 shows the performance on the Kepler Platform, where *BPLG-LF-SC-Naive* starts with fairly good performance, achieving a high occupancy (64 active warps and 16 active blocks per SM). As N increases, this occupancy decreases (64 warps but only 2 blocks when $N = 2048$) due to a high shared memory requirement (8192 bytes per block). On the other hand, as the amount of shared memory remains invariant in the *BPLG-LF-SC* implementation independently of N, the performance is also constant, with the global memory bandwidth being the limiting factor of our implementation. The *BPLG-LF-SC* proposal achieves an improvement of up to 2.31x with respect to *BPLG-LF-SC-Naive*, and up to 3.54x over the non-*BPLG* implementation. The *BPLG-LF-SC* always obtains a 100% warp occupancy, achieving $B_a = 16$ active blocks when $n \leq 9$ and $B_a = 8$ active blocks if $n = 10$. The decrease in the number of active blocks is unavoidable when $l = n - p$.

Problem size	*(s,p,l) values*
Kepler Platform	
$n \leq 8$	$((13-n), 2, 7)$
$n > 8$	$(5, 2, n-1)$
Maxwell Platform	
$n \leq 7$	$((12-n), 2, 6)$
$n > 7$	$(5, 2, n-1)$

Table 4.8: Description of the KS scan tuning parameters.

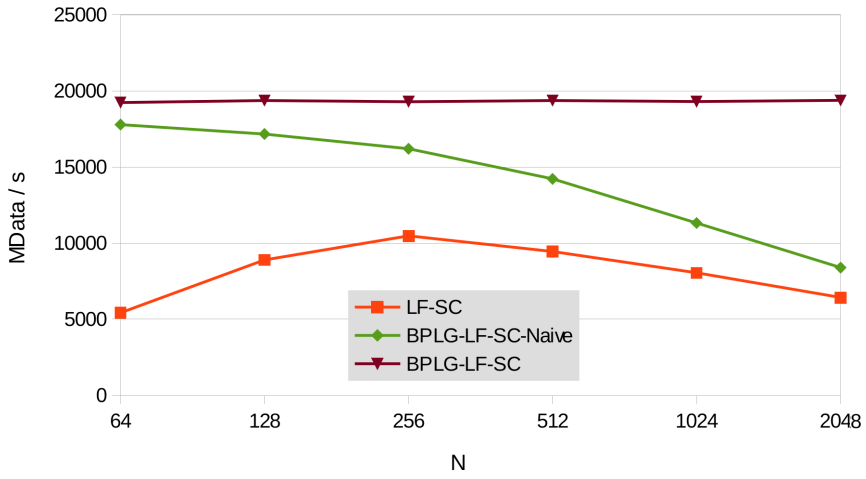


Figure 4.18: MData/s comparison of BPLG-LF scan implementations in the Kepler Platform.

The Maxwell architecture (see Figure 4.19) increases shared memory to 96 KB per SM, increasing the block occupancy in these proposals. This fact means that the performance difference between proposals is less pronounced in this architecture. The performance of *BPLG-LF-SC-Naive* and *LF-SC* increases due to the computational power of Maxwell. However, *BPLG-LF-SC* performance does not rise, due to the bandwidth, which is the limiting factor, but it still obtains up to 1.45x of improvement over *BPLG-LF-SC-Naive* and 1.57x over *LF-SC*. Here, 100% of warp parallelism is achieved in all cases, but the maximum threadblock parallelism cannot be obtained when $n > 8$, due to $l = n - p$.

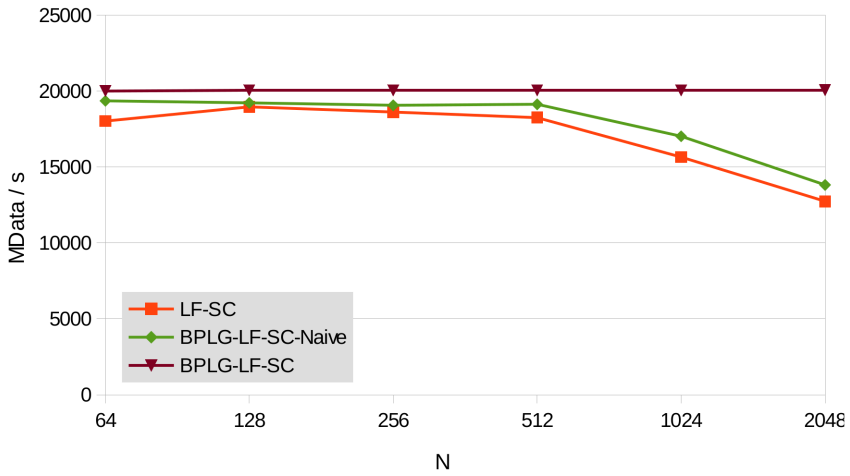


Figure 4.19: MData/s comparison of BPLG-LF scan implementations in the Maxwell Platform.

BPLG-KS-SC Results

On the other hand, Figure 4.20 and Figure 4.21 show a performance comparison of our Kogge-Stone scan proposals on both platforms. Figure 4.20 depicts results on the Kepler Platform, where *BPLG-KS-SC-Naive* shows poor performance as N increases. Furthermore, as can be observed, neither *KS-SC* nor *BPLG-KS-SC-Naive* reaches $N = 2048$ as $L = 1024$ at most in current architectures and, although they work with two elements per thread ($P = 2$), it uses $P' = 1$ in these implementations. On the other hand, *BPLG-KS-SC* implementation consumes very little shared memory, and this amount is constant regardless of N. Specifically, each problem is solved with 32 elements in shared memory, as explained.

On the Kepler Platform, *BPLG-KS-SC* is up to 4.03x faster than *BPLG-KS-SC-Naive* and 4.94x faster than *KS-SC*. Empirically, it uses 27 registers per thread, thus solving 8 elements per thread was the correct choice. On the Maxwell Platform, Figure 4.21, it is important to notice that *BPLG-KS-SC-Naive* performance increases in Maxwell owing to two reasons: more block occupancy due to 96 KB of shared memory per SM, and the cache hierarchy in global memory accesses. These two features reduce the performance impact of *BPLG-KS-SC* implementation, with an improvement of up to 2.11x over *BPLG-KS-SC-Naive* and 2.53x over *KS-SC*.

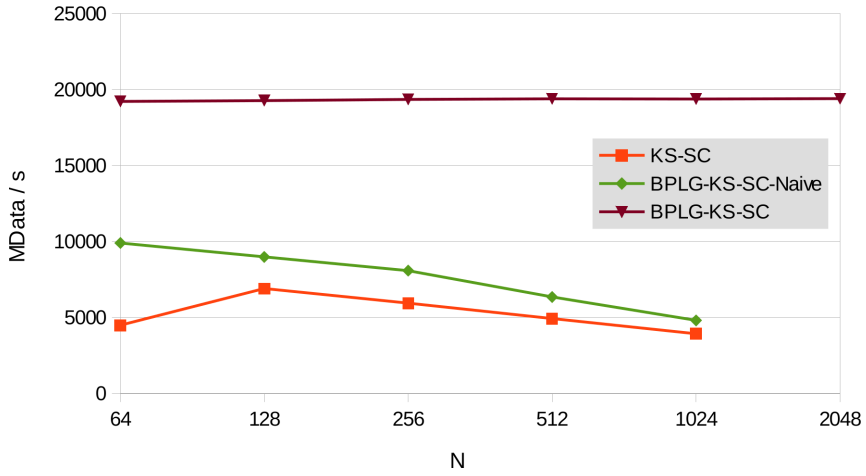


Figure 4.20: MData/s comparison of BPLG-KS scan implementations in the Kepler Platform.

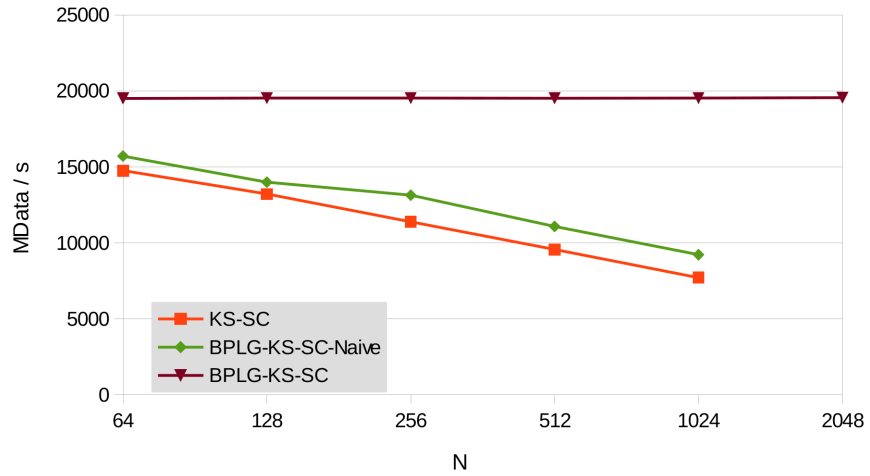


Figure 4.21: MData/s comparison of BPLG-KS scan implementations in the Maxwell Platform.

Overall Results

This section gives a global overview of our best proposals with respect to *CUDPP* [98], *Thrust* [101], *ModernGPU* [97] and *CUB* [100] libraries on both Platforms. Although the most representative scenario of our proposal lies in solving several problems simultaneously in one single invocation ($G > 1$), only *CUDPP* supports this feature. *Thrust, ModernGPU* and *CUB* do not implement a *multi-batch* scan,

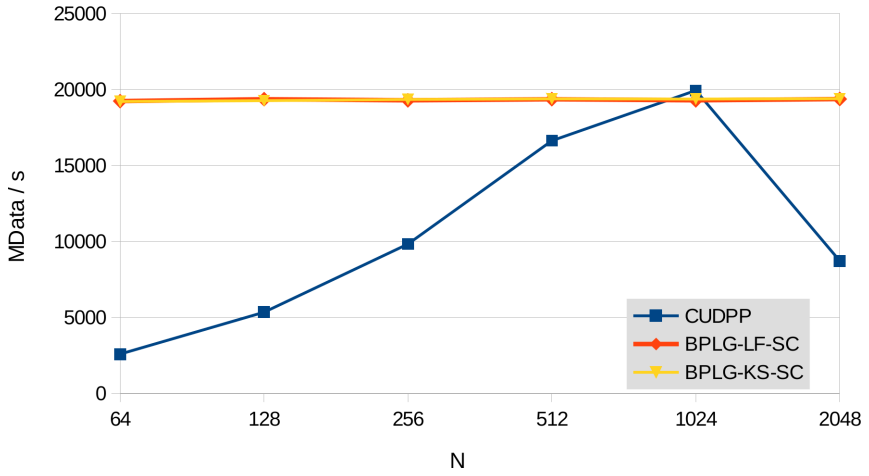


Figure 4.22: MData/s performance comparison of BPLG scan proposals in the Kepler Platform

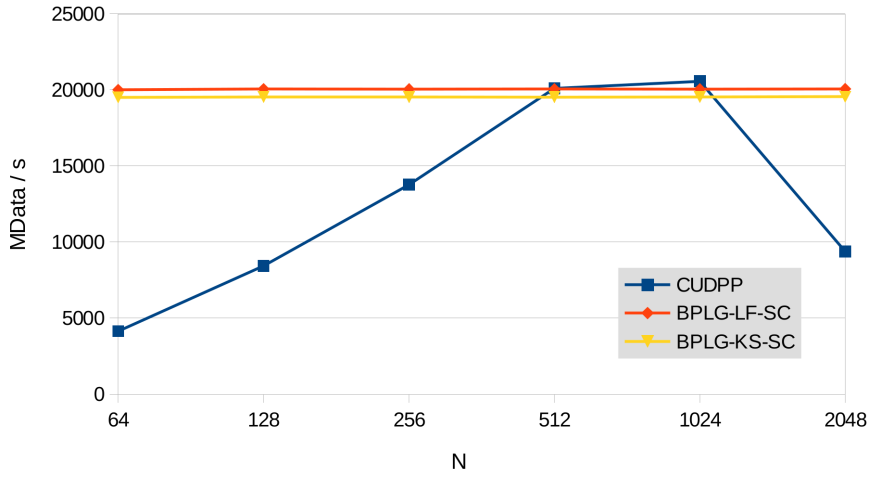


Figure 4.23: MData/s performance comparison of BPLG scan proposals in the Maxwell Platform

invoking the library several times. Figure 4.22 and Figure 4.23 show the performance results against *CUDPP*. However, *Thrust*, *ModernGPU* and *CUB* libraries are not shown in the graph due to scale imperceptibility when $G = 2^{24}/N$.

On the Kepler Platform, Figure 4.22, *BPLG-LF-SC* obtains an improvement of up to 7.44x over *CUDPP*, whereas *BPLG-KS-SC* has a very similar improvement, being up to 7.43x faster than *CUDPP*. Our proposals, with very similar performance, are 1447x, 610x and 1512x, on average, faster than *Thrust*, *ModernGPU* and *CUB*,

respectively, as they do not support a multi-batch scan and the several invocations performed penalyze throughtput. Executing them for the case of $G = 1$, a single invocation in all libraries, *ModernGPU* is the fastest one, but our proposals still surpass *Thrust* (up to 4.23x), *CUB* (up to 1.04x) and *CUDPP* (up to 1.49x).

On the Maxwell Platform, Figure 4.23, *BPLG-LF-SC* is up to 4.84x faster than *CUDPP* and *BPLG-KS-SC* obtains an improvement of up to 4.72x over *CUDPP*, again similar results are obtained for both proposals. With respect to *Thrust*, *ModernGPU* and *CUB*, which do not support a multi-batch scan execution, our proposals are, respectively, 823x, 355x and 813x times faster on average.

Efficiency of the Performance Parameters Tuning Phase

This section compares the performance achieved by using the proposed performance parameters, against the performance achieved by an exhaustive search of performance parameters for the Kepler Platform. Table 4.9 shows this information for *BPLG-LF-SC* on Kepler. For this algorithm, the values proposed were $(s, p, l) = ((14 - n), 2, 7)$ when $n \leq 9$ and $(s, p, l) = (5, 2, n - 2)$ in the remaining cases. In this case, all the proposed values obtain the maximum performance possible. Taking this empirical analysis into account, we can conclude that the premises outlined in Section 4.1.1 for performance maximization obtain the best performance parameters from the searching space, demonstrating the effectiveness of our proposal.

4.6. Sorting under a three-phase methodology (BPLG-BMCS Algorithm)

This proposal is the adaptation of the BMCS sorting algorithm presented in Section 3.3.2 to our three-phase methodology [27]. Figure 4.24 presents an implementation of the BMCS algorithm using *BPLG* skeletons. The code can be divided into four main sections:

(p,l) \ n		$n=6$	$n=7$	$n=8$	$n=9$	$n=10$	$n=11$
$p=1$	$l=5$	6458	x	x	x	x	x
	$l=6$	11959	11876	x	x	x	x
	$l=7$	17637	17356	17137	x	x	x
	$l=8$	17321	16653	16644	16564	x	x
	$l=9$	16723	16336	16274	16142	16023	x
	$l=10$	13771	13785	13869	14173	14280	14239
$p=2$	$l=5$	12021	12011	x	x	x	x
	$l=6$	17269	17276	17211	x	x	x
	$l=7$	***19172***	***19179***	***19205***	***19189***	x	x
	$l=8$	19123	19129	19123	19172	***19161***	x
	$l=9$	19010	19030	18934	17968	19149	***19150***
	$l=10$	17458	17423	17767	17736	18180	18327
$p=3$	$l=5$	15399	16021	16028	x	x	x
	$l=6$	18580	18600	18602	18714	x	x
	$l=7$	18342	18379	18379	18197	182890	x
	$l=8$	18408	18410	18416	18410	118237	18330
	$l=9$	18331	18382	18376	18381	18388	18194
	$l=10$	17464	17730	17723	17679	17801	18002

Table 4.9: Performance comparison of different performance parameters values for BPLG-LF-SC in MData / s

- Initialization section (lines 3-4). Allocates registers and shared memory.

- Load data from global memory (lines 5-6) and first computing step (line 8). Loads coalescent data using a 64-bit load to obtain 2 consecutive elements instead of accessing a single data element per memory request. Then, elements are directly processed in registers by the *compute* function, which compares and swaps values.

- Computing steps of the algorithm (lines 9-29). The loop computes the remaining steps of the algorithm, with its internal steps. To this end, the loop loads the corresponding data into registers using shared memory and synchronization barriers. The synchronization barrier in line 13 can be avoided in the first iteration, as the data are already in registers. The same behaviour occurs in

```
1  template<int N, int p, int S> __global__ void
2  BPLG_Bitonic ( const int* __restrict__ data) {
3          int reg[p*2];
4          __shared__ int shm[N > p ? S : 1];
5          //Load data from global memory to registers
6          copy<2,p>(reg, data+...);
7          //First computing step
8          compute<p,MixR>(reg);
9          for(int accR=MixR; accR < N ; accR*=2) {
10                 //Obtains strides and offsets
11                 int readOffset = ..., readStride = ... ;
12                 //Reg-> Shm -> Reg
13                 if(accR>MixR) __syncthreads();
14                 copy<2,p>(shm+2*threadId, 1,reg, ...);
15                 __syncthreads();
16                 copy<2,p>(reg,shm+readOffset,readStride, ...);
17                 //Computation in registers of first internal step
18                 compute<p,MixR>(reg);
19                 //Remaining internal steps
20                 for(int j=accRad; j>1; j/=2) {
21                         int readOffset = ..., readStride = ... ;
22                         int writeOffset = ..., writeStride = ... ;
23                         if (j<accRad) __syncthreads();
24                         copy<2,p>(shm+writeOffset, writeStride,reg, ...)
                               ;
25                         __syncthreads();
26                         copy<2,p>(reg,shm+readOffset,readStride, ...);
27                         compute4<p-1>(reg);
28                 }
29         }
30         copy<2,p>(data,reg,...);
31 }
```

Figure 4.24: Kernel code for the BMCS algorithm using BPLG skeletons.

the internal loop on line 23, where results are returned in registers.

- Store data to global memory (line 30). The final iteration of the loop stores the results into registers; thus, the final result is moved from registers to global memory using 64-bit stores, reducing the number of memory transactions.

This algorithm uses a radix-2 expression for the external steps, whereas a radix-4 implementation is used in the internal steps, as explained in Section 3.3.2. In each external step, an increased sub-sequence of the problem is solved in several internal steps. The size of this sub-sequence is doubled in each iteration, along $log_2 N - 1$ iterations. Thus, some sub-sequence sizes are power of 2 ($r = 2$) and others are power of 4 ($r = 4$). In order to integrate both radix, the implementation works with radix $r = 2$ and considers $N = 2^n$. Furthermore, each thread may work with two Node operators of $fan\ in = 2$, or with one Node operator of $fan\ in = 4$, depending

on the sub-sequence size. When the sub-sequence size is not a power of 4, the mixed-radix *compute* function is the responsible of determining whether the first step is performed with $r = 2$ (each thread works with 2 Node operators of $fan\ in = 2$) or with $r = 4$ (each thread works with a single Node operator of $fan\ in = 4$). The remaning steps are always performed with $r = 4$ (*compute4* function in the code) and each thread works with a single Node operator of $fan\ in = 4$.

4.6.1. CUDA Kernel Optimization phase: BPLG-BMCS

The effectiveness of BMCS has been demonstrated in Section 3.3.3. Here, BMCS is here adapted to *BPLG*, providing specialized kernels for different datatypes and p values, with the specific code for each case that exploits the maximum parallelism. For example, specialized kernels that enables the use of customized data types as *Float2* or *Int4* has been implemented, reducing the number of memory requests and improving performance. This approach has been also optimized with the *Hybrid communication strategy inside a block* where initial steps are computed using shuffle instructions, sorting $p \times warpSize$ elements in each warp; whereas the other steps use shared memory as a communication channel between warps. If $N \leq 128$, then there is no synchronization barrier in the execution.

4.6.2. Performance Parameter Tuning phase: BPLG-BMCS

Using our notation, this algorithm uses $P = 4$ ($p = 2$), then $S = L \times 4$. The use of $P = 4$ with integers does not surpass the established limit of 32 registers per thread, and enables the communication with shuffle instructions exclusively when $n \leq 7$.

In Kepler architectures, when $n \leq 7$, the values proposed for (s, p, l) are $(0, 2, 7)$, as they obtain both maximum warp occupancy and maximum number of active threadblocks per SM, where $L_G > 1$. When $8 \leq n \leq 9$, the following tuple is obtained $(s, p, l) = (9, 2, 7)$, using performing communications across shared memory, but still achieving both maximum warp and threadblock occupancy. When $9 < n \leq 12$, the tuple $(s, p, l) = (n, 2, n - 2)$ is employed, where $L_G = 1$, obtaining the maximum warp parallelism, but decreasing the number of active threadblocks

Problem size	*(s,p,l) values*
Kepler Platform	
$n \leq 7$	$(0, 2, 7)$
$8 \leq n \leq 9$	$(9, 2, 7)$
$n > 9$	$(n, 2, n-2)$
Maxwell Platform	
$n \leq 7$	$(0, 2, 6)$
$n = 8$	$(8, 2, 6)$
$n > 8$	$(n, 2, n-2)$

Table 4.10: Description of the *BPLG-BMCS* sorting tuning parameters.

when increasing n, due to the high shared memory requirements.

In Maxwell architectures, similar values are obtained. When $n \leq 7$, $(s, p, l) = (0, 2, 6)$, achieving both maximum warp occupancy and maximum number of active threadblocks per SM, with $L_G > 1$. When $n = 8$, then the tuple $(s, p, l) = (8, 2, 6)$ is obtained, using shared memory but still achieving both maximum warp and threadblock occupancy. For the remaining sizes, $9 \leq n \leq 12$, the tuple $(s, p, l) = (n, 2, n-2)$ is used, obtaining 100% warp parallelism but decreasing the number of active threadblocks progressively owing to shared memory consumption. Table 4.10 shows these values for both architectures.

4.6.3. Experimental Results for Sorting with Small Problem Sizes

In this section, we present the results of our proposals. All tests were run using integers as datatype. All the data initially reside in the GPU memory, so there are no data transfers to CPU during benchmarks. The test platforms used in our experiments are described in Table 4.4. The performance of these experiments is measured in million data processed per second, MData/s. Many applications need to solve G batch problems in parallel; therefore, we used a batch execution to compute G problems each time. The size of the batch depends on the input size and is

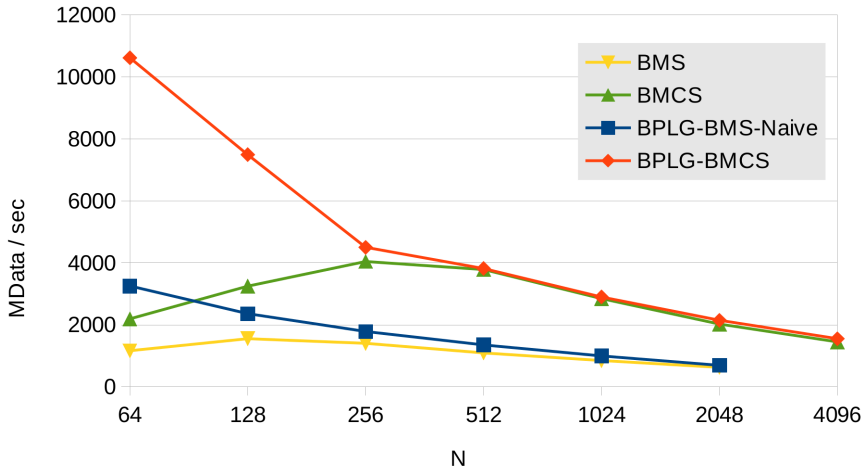


Figure 4.25: Comparison of our proposals in the Kepler Platform.

given by the expression $G = 2^{24}/N$. Thus, MData/s value is performed using the expression $N \times G \times 10^{-6}/t$.

Firstly, Figure 4.25 depicts a performance comparison among our implementations in the Kepler platform. The *BMS* tag refers to an optimized Bitonic Merge Sort implementation, whereas *BMCS* represents the implementation of our algorithmic variant with shuffle communications presented in Section 3.3.2. *BPLG-BMS-Naive* denotes a naive implementation using the previous BPLG skeletons. *BPLG-BMCS* is the approach proposed in this chapter. In general, while shared memory is not an expensive resource, *BPLG* skeletons are much better due to the explained features of the library. Until $N = 256$, *BPLG-BMCS* runs faster than *BMCS*, as each threadblock executes several batches in parallel, which guarantees a high occupancy, being up to $5.4x$ faster than *BMCS*, and $3.8x$ with respect to *BPLG-BMS-Naive*. As N increases, the number of batches per block is reduced in *BPLG-BMCS*, and performance is very similar with *BMCS*. In the case of *BMCS*, the peak of performance is achieved with $N = 256$ and $N = 512$, as the occupancy is maximum with these values. In problem sizes that are larger than $N = 512$, both *BMCS* and *BPLG-BMCS* can only execute one problem per threadblock, owing to resource consumption. Even in this case, *BPLG-BMCS* is slightly better than *BMCS*. As demonstrated, *BPLG* skeletons offer a simple way of programming, obtaining the same, or higher, performance as with other complex-optimized verbose kernels for

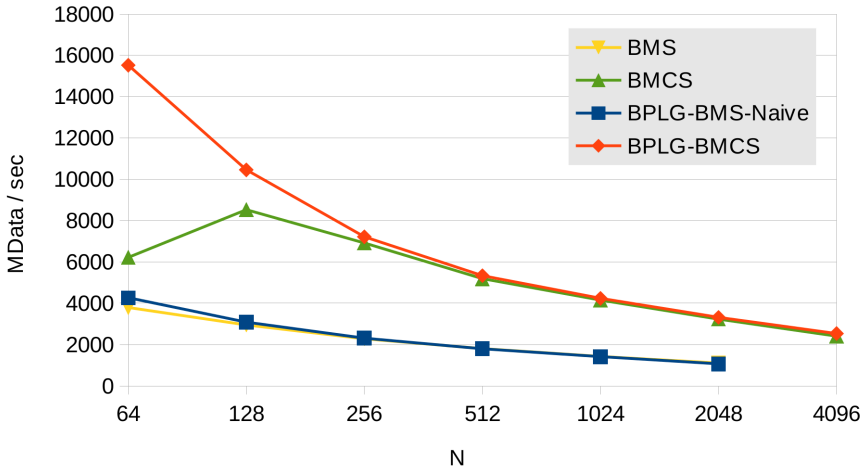


Figure 4.26: Comparison of our proposals in the Maxwell Platform.

the same task, such as *BMCS*. However, in both implementations, shared memory becomes a limiting factor.

Figure 4.26 shows the same comparison on the Maxwell platform, maintaining the same nomenclature. The MData/s achieved on this platform is higher, which can be ascribed to the fact that Maxwell presents a power-efficient performance which provides a higher delivered performance per CUDA core than Kepler owing to its new datapath organization, new improved instruction scheduler, new memory hierarchy and bandwidth, obtaining a higher number of active blocks per Streaming Multiprocessor. The architecture doubles the number of blocks per SM, up to 32 blocks (double than Kepler), although the available shared memory per block remains the same. Owing to this behaviour, *BMCS* occupancy is maximum with $N = 128$ in Maxwell. In this platform, *BPLG-BMCS* is up to $2.63x$ faster with respect to *BMCS* and up to $3.76x$ over *BPLG-BMS-Naive*, when small sizes; although this speed-up is reduced for all proposals when larger sizes, due to the shared memory consumption penalty.

Figure 4.27 compares *BPLG-BMCS* with respect to the *CUDPP* [98], *CUB* [100] and *ModernGPU* [97] libraries, where *ModernGPU* is the reference library for sorting small problem sizes. It should be noted that this comparison is made in terms of execution time for only one batch. *CUDPP* shows the worst results for small prob-

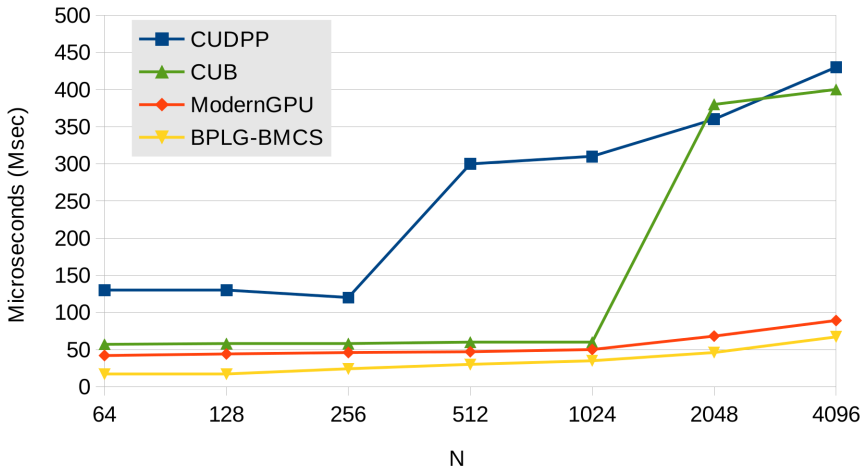


Figure 4.27: Comparison of GPU sorting implementations for one batch on the Kepler Platform.

lem sizes that can be directly processed in shared memory. Our proposal, *BPLG-BMCS*, provides highly competitive results compared to *ModernGPU*, obtaining an improvement of up to $10x$ over *CUDPP*, up to $8.26x$ over *CUB* and up to $2.6x$ over *ModernGPU*. On the other hand, Figure 4.28 presents the same comparison on the Maxwell Platform. Results are similar to the Kepler Platform, obtaining up to $40x$ in comparison to *CUDPP*, up to $20.9x$ over *CUB* and up to $4.8x$ over *ModernGPU*.

Table 4.11 compares our *BPLG-BMCS* to *CUDPP*, *CUB* and *ModernGPU*, which prove to be extremely inefficient with problems where many *batches* of small size are processed in parallel, as they were designed for solving just one large-size problem. In order to solve G problems of size N, these libraries have to launch G *light* kernels. Our proposal is up to $11.79x$ faster than *CUDPP* library, up to $7.58x$ over *CUB* and up to $5.31x$ over *ModernGPU* on the Kepler Platform. Table 4.11 shows the MData/s obtained in the Maxwell Platform. The MData/s achieved are higher owing to Maxwell design. Our proposal is up to $6.47x$ faster than *CUDPP*, up to $5.35x$ over *CUB* and up to $3.61x$ than *ModernGPU*.

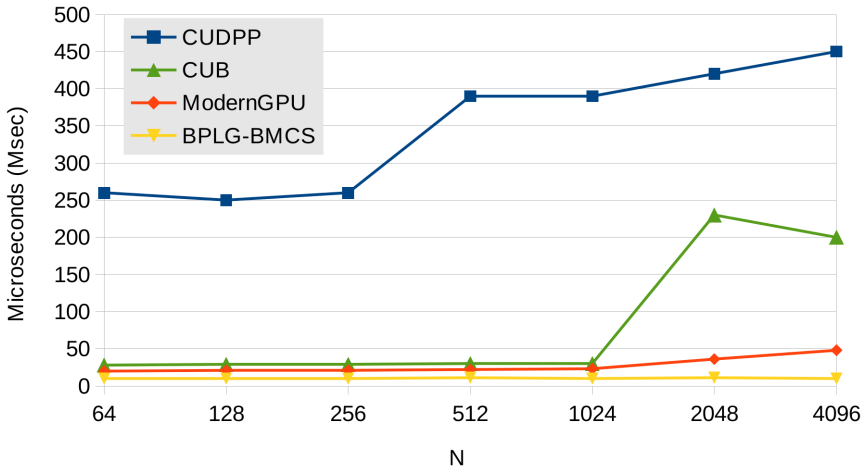


Figure 4.28: Comparison of GPU sorting implementations for one batch on the Maxwell Platform.

		Kepler Platform				Maxwell Platform			
N	*G*	BPLG-BMCS	ModernGPU	CUDPP	CUB	BPLG-BMCS	ModernGPU	CUDPP	CUB
64	262144	10614	2	0.9	1.4	15521	4.3	2.4	2.9
128	131072	7492	3.7	1.8	2.7	10457	8.3	4.9	6.1
256	65536	4496	7.1	4	5.3	7218	16.3	10	12.3
512	32768	3817	13.4	7.9	10.2	5333	31.4	19.6	23.5
1024	16384	2897	25.2	14.7	20.4	4234	61.2	35.9	43.1
2048	8192	2144	34.4	16.1	5.5	3314	66.5	33.4	24.2
4096	4096	1549	50.4	16	10.4	2534	97.2	48	40.1

Table 4.11: MData/s comparison of GPU multi-batch Sorting Algorithms in the Maxwell Platform.

4.7. Conclusions of the Chapter

This chapter provides a three-phase tuning methodology for parallel prefix algorithms of small size on a GPU (whose size fits in the GPU shared memory). In the first phase, *GPU Resource Utilization Analysis*, the GPU performance parameters are identified, and a set of performance premises are established. In the *CUDA Kernel Optimization* phase, the algorithms are implemented using CUDA skeletons. Finally, in the *Performance Parameters Tuning* phase, the suitable values for the GPU performance parameters are obtained for each problem size and GPU architecture.

Following this methodology, three different tridiagonal system solvers (*BPLG-CR-TS*, *BPLG-PCR-TS* and *BPLG-LF-TS*) have been developed, as well as two scan operators (*BPLG-LF-SC* and *BPLG-KS-SC*) and a sorting operator (*BPLG-BMCS*). In the case of tridiagonal solvers, the three proposals are tested on two different GPU architectures, outperforming both *CUDPP* and *CUSPARSE*, the state-of-the-art, performing especially well *BPLG-LF-TS*. Both scan proposals have a similar performance, outperforming *CUDPP* in most cases. Regarding *BPLG-BMCS*, surpasses *CUDPP*, *CUB* and *ModernGPU*, the state-of-the-art, being up to $11.79x$, $7.58x$ and $5.31x$, respectively. It should be observed that this methodology is especially well suited to solve several batches simultaneously, and the proposed values for the GPU performance parameters proposed by the methodology match very well with those obtained empirically.

Chapter 5

A Tuning Methodology for Parallel Prefix Algorithms on a GPU: Medium and Large Problem Sizes

In this chapter, the previous tuning methodology for small problem sizes is extended to support medium and large problem sizes; i.e., problems that do not fit in the CUDA shared memory, but which still can be stored in the global memory a single GPU. This is done by partitioning the computation into several stages (*multi-stage strategy*). As was introduced in Chapter 2, Index-Digit algorithms are a subset of parallel prefix algorithms which have special features. Thus, owing to these special features, this tuning methodology is initially specialized for ID-algorithms, and then generalized to the superset of the parallel prefix algorithms.

In the case of ID-algorithms, this methodology is tested for the Wang & Mou tridiagonal system solver (WM); whereas the methodology focused on general parallel prefix algorithms is tested for the scan primitive and the Tree-Partitioning Reduction tridiagonal system solver (*TPR*). This work was originally introduced in [28], [34] and [36]. In [36], we also presented a FFT proposal (MS-ID-FFT) under the ID methodology, in collaboration with other authors and based on the *Stockham* algorithm [121].

5.1. A two-phase Methodology for Index-Digit Algorithms

Similar to [82], this tuning methodology for ID-algorithms also has two phases. In the first phase, we need to determine the main features which influence the GPU performance for these problem sizes and establish a set of theoretical performance premises. Based on these premises, a number of tunable parameters is obtained and, for each algorithm, the optimal values are chosen. In the second phase, CUDA kernels are built with CUDA skeletons and previous values, and the suitable performing kernel version is chosen at compile-time with the corresponding tuning parameters (according to the problem size and target architecture).

When working with problem sizes that are bigger than one threadblock's shared memory capacity but which still fit into the global memory of a single GPU, data interchange is performed via global memory, and different options for synchronizing threadblocks can be considered:

- *Multi-Stage Strategy.* In this case, the work is divided into several kernels; i.e., into several stages. Here, each kernel invocation acts as a global synchronization. Threadblocks from each kernel write their partial results into global memory. As a synchronization mechanism for this data interchange, another kernel is launched with its corresponding threadblocks. These new threadblocks build new partial results using the previous kernel data from global memory. This strategy significantly increases the global memory bandwidth requirements.

- *Dynamic parallelism.* Using dynamic parallelism, a kernel directly from the GPU can spawn other kernels. Its main objective is to reduce the overhead of starting and synchronizing kernels. Even considering the added flexibility, the generated code tends to run slower due to the relocatable device code generation and the use of local memory, global memory accessible only by the thread that declares it, used as a stack [135]. This approach is suitable for problems that require mesh refinement (such as finite element methods) using a dynamic work distribution.

- *Persistent threads.* This is a decentralized sleeping strategy. Each threadblock sets a flag when it reaches the intra-block barrier, executing an infinite loop until a master threadblock changes the flag value. When all flags are set, the master block resets them and all threadblocks continue execution. This allows threadblocks to be synchronized in a single kernel. A kernel uses, at most, as many threadblocks as can be concurrently scheduled on the SM. Thus, this strategy synchronizes global memory using a single kernel and a constant number of threadblocks. In many cases, the use of persistent threads on GPUs results in performance losses [51]; nonetheless, it has been successfully applied in some optimized libraries, such as *CUB* [100], as it presents low memory contention.

In this work we have used the multi-stage strategy as the synchronization mechanism among threadblocks. This is the same technique used in other libraries, such as CUFFT [94] or the proposal presented in [23]. Despite the increased global memory requirements, if the data exchanges are properly optimized and the workload is properly balanced among the GPU resources, the multi-stage strategy is quite efficient, as this work demonstrates.

5.1.1. GPU Resources Utilization Analysis Phase

The mapping of $G = r^{batch}$ data sequences of size $N = r^n$ is identified with a 5-tuple of the form (n, p, s, l, b). In our multi-stage proposal of ID-algorithms, each problem is computed by dividing it into a set of m stages, where each stage executes several steps. Each stage executes a kernel which assigns a part of its corresponding problem to different threadblocks. As introduced in Chapter 2, b is formed by two coordinates $b = (b_x, b_y)$, where r^{b_x} represents the number of threadblocks used per each problem, while r^{b_y} represents the number of problems being simultaneously executed on that kernel in a batch mode. Thus, $B_x \cdot B_y$ threadblocks process the whole batch. Furthermore, each thread performs the computations associated to the Node operator. Data are stored in private registers in order to achieve high performance, as register files have lower access latency and higher bandwidth than shared memory. Finally, threads from the same threadblock exchange data before the next computing step through shared memory. Specifically, our multi-stage proposal is

based on only three parameters (n, p, b_x) given that $s = n - b_x$, as all the data stored in registers also have a copy in shared memory to perform the intra-block memory exchanges; and $b_y = batch$ is given by the batch size, which is only known at runtime. In our proposal, l consists of three coordinates (l_x, l_y, l_z), where the second and third coordinates, (l_y, l_z), are optional. The l parameter can be related with s and p using $s = p + l$.

Premises for Performance Maximization

In this work, large problems are computed over several kernels. When computing several kernels, new parameters influence performance. For example, the number of invoked kernels, the number of steps processed by kernel and the number of elements processed by each kernel. Considering these factors and attempting to improve performance, we define the following premises:

1. *The minimization of the number of stages, m.* Global memory data exchanges are slower than using other memories, such as shared memory, despite implicitly utilizing the L1 and L2 caches. In the multi-stage strategy, data interchange via global memory is the only method for sharing information among kernels, as $n > s$. In addition to this latency, each kernel invocation implies an overhead, even for empty kernels. Thus, the number of stages (kernels) in the multi-stage strategy needs to be minimized. In our proposal, the number of stages is given by the following expression:

$$m = \left\lceil \frac{n}{s} \right\rceil \tag{5.1}$$

   In order to minimize this expression, s must be as large as possible. Each kernel invocation executes as many problem steps as the shared memory allows. Thus, each kernel processes several chunks of S elements (one chunk per threadblock). Subsequent kernels will merge elements among chunks until the final result is obtained.

2. *Balancing warp and block parallelism.* As previously explained, the level of GPU parallelism can be supported in terms of the number of threadblocks

per *SM* (*SM* block parallelism), or the number of warps per *SM* (*SM* warp parallelism):

a) *The maximization of block parallelism in each stage* in order to keep processing the maximum amount of simultaneous threadblocks per *SM* (16 in the case of *Kepler* and 32 in the case of *Maxwell*-based *GPUs*). In fact, the *GPU* hardware is able to provide highly satisfactory performance even at lower occupancies (low *SM* warp parallelism) [126, 127].

b) *The maximization of warp parallelism in each stage.* This premise is focused on increasing the number of warps per *SM*.

Our proposal attempts to strike a balance between the maximization of warp and block parallelism. In order to increase this parallelism, we need to limit the factors that reduce the SM parallelism, such as the number of registers used by each thread or the amount of shared memory required by threadblock.

3. *Increasing the computational load per thread.* Both r and P parameters are closely related. Note that r is a feature of the algorithm which represents the number of elements computed in each Node operator. However, if the target architecture allows more than r elements to be stored in registers, without SM occupancy penalization, it may be interesting to process more Node operators per thread, without modifying the base r of N. In this case, each thread processes P elements in $\frac{P}{r}$ radix-r nodes. Increasing either P or r means processing more elements per thread. This influences the number of steps taken and the number of threads which process a problem, and reduces the number of synchronization barriers. Thus, larger values obtain higher performance. Nevertheless, their increase may also require too many registers per thread, resulting in local memory spilling and the minimization of parallelism. In this work, $\frac{P}{r}$ radix-r nodes are easily integrated in a single radix-P node, reducing the number of steps taken, thanks to the ID-algorithm features.

Combining all of these premises is not easy. Firstly, s needs to be increased in order for there to be fewer stages (Premise 1). This is fundamental since it avoids launching several kernels, synchronizations and reads/writes to global memory. This increment entails more elements being processed by a single kernel. However, the increase of s may decrease the SM block parallelism (Premise 2.a). Each SM has a fixed

amount of shared memory partitioned among the threadblocks, thus the amount of shared memory required by each threadblock limits the SM block parallelism. In order to achieve Premise 2.b, keeping the number of threadblocks constant, l must be raised. Due to the equation $s = p + l$, there are two options: either decreasing p and keeping s constant or increasing s and keeping p constant. In the former option, block parallelism remains the same, reducing the workload per thread (which implies more steps, loop iterations and shared memory accesses), whereas the opposite is true for the latter. In addition to this, each SM also has a register file partitioned among threads. Decreasing register consumption implies better warp occupancy. However, p should be high if Premise 3 is to be achieved. The maximum number of concurrent warps and threadblocks per SM depends on the architecture.

In the case of *Kepler*, the total amount of registers per *SM* is 65536 and the amount of shared memory per *SM* is 48 KB, enabling up to 16 concurrent threadblocks and 64 concurrent warps. The number of registers used by each thread is assigned at compile time. In hardware with *CUDA* capabilities 2.x or 3.0, no more than 63 registers can be assigned to the same thread. Hardware with *CUDA* capabilities 3.5 supports up to 255 registers per thread. If the kernel requires more registers than those supported by the architecture, local memory spilling will be generated. This means using global memory for placing values instead of registers, paying the penalty of global memory latency.

Regarding *Maxwell GPUs*, the architecture has 96 KB of shared memory per SM and can use up to 48 KB per threadblock, with the register file size remaining constant. It enables up to 32 concurrent threadblocks and 64 concurrent warps.

Our main objective in the optimization of these algorithms is to find a trade-off between premises for each problem on each architecture in order to achieve the highest possible performance.

5.1.2. CUDA Kernel Optimization Phase: String Operators and Mapping Vector

This section describes the use of mapping vectors based on the Index-Digit representation [46]. The mapping vector is a compact representation of the data distri-

bution on the system memory hierarchy. A mapping vector divides the Index-Digit representation into different fields which are used to assign resources of the CUDA GPU (e.g. threadblock, thread or registers) to the specific data item to be treated by the GPU. At the beginning and the end of the algorithm, data reside in global memory; however, data are moved among different resources in the GPU during the execution. The string operator provides a precise description of the data reordering, being useful in the design and optimization of different algorithms. Furthermore, the string operator makes it possible to obtain an Index-Digit representation in each step of the algorithm. Further information about string operator properties can be found in [30].

Data sequences are stored in the *GPU*'s global memory with a consecutive data distribution according to the following mapping vector:

$$[\ t_{n+batch} \cdots t_{n+1}\ t_n \cdots t_1\ ] \tag{5.2}$$

This means that data with size N will be stored consecutively in global memory. Hence, the first data sequence of the batch will start at location 0, the second at location N, and the i-th problem of the batch at location $i \times N$.

The mapping vector for data on the *SM* resources that we consider is

$$[\ \underbrace{t_{n+batch} \cdots t_{s+1}}_{b}\ \overbrace{\underbrace{t_s \cdots t_{l+1}}_{p}\ \underbrace{t_l \cdots t_1}_{l}}^{s}\ ] \tag{5.3}$$

Firstly, this means that each threadblock $i = [t_{n+batch} \cdots t_{s+1}]$ processes S items of data which are stored in shared memory, and secondly, thread $j = [t_l \cdots t_1]$ within a threadblock processes P items of data where datum $[t_n \cdots t_{l+1}\ t_l \cdots t_1]$ is stored on the register $[t_s \cdots t_{l+1}]$ of thread j. Note that consecutive data belong to different threads.

However, the data distribution of a problem could change depending on the implementation design for a given target architecture. For instance, the previous example can be also expressed with the following mapping vector:

$$[\ \underbrace{t_{n+batch}\cdots t_{s+1}}_{b}\ \overbrace{\underbrace{t_s\cdots t_{p+1}}_{l}\ \underbrace{t_p\cdots t_1}_{p}}^{s}\ ] \tag{5.4}$$

In this case, each thread $j = [t_s \cdots t_{p+1}]$ within a threadblock, processes P consecutive data stored in registers.

Figure 5.1 depicts an example of mapping the data to the GPU resources when $s = 9$, $p = 4$ and $b_x = 2$ for the case $r = 2$ and $n = 11$. Each threadblock receives a set of elements, $2048/4 = 512$, which are stored in shared memory, and evenly distributed to the registers among 32 threads ($l = 9 - 4$). The mapping vector for this example would be:

$$[\ \underbrace{t_{batch+11}\cdots t_{12}}_{b_y}\ \underbrace{t_{11}t_{10}}_{b_x=2}\overbrace{\underbrace{t_9\cdots t_5}_{l=5}\ \underbrace{t_4\cdots t_1}_{p=4}}^{s=9}\ ] \tag{5.5}$$

For example, element $1041 = [\cdots \underbrace{10}_{b_x}\ \underbrace{00001}_{l}\ \underbrace{0001}_{p}]$ is processed by thread 1 ($l = 00001$) in block 2 ($b_x = 10$) and stored in register 1 ($p = 0001$) of that thread.

Two types of operators are defined, which correspond to computations and data permutations [4], respectively. All of these are formally defined in Table 5.1, wherein the modified digits are underlined. To write the expressions of the string operators we follow the convention of composing operators from left to right. For example, in the string operator $\phi_1\phi_2$, we first execute ϕ_1 and then, ϕ_2. First, we define the operator that represents the computations.

Definition 1. *The Node operator, Υ_i^r, with $1 \leq i \leq n$ where $n = log_r N$, reads those sets of r data items whose position differs precisely in their i-th digit, performs an operation over them and writes r results.*

Depending on the operation, each Node function will be defined with its own behavior for each algorithm. This specialization only affects the implementation details, but not the methodology design. In general, to simplify the notation when using the basic radix-2 algorithm, the expression of this operator will be referred to simply as Υ_i instead of Υ_i^2. Furthermore, in order to clarify the explanation, we

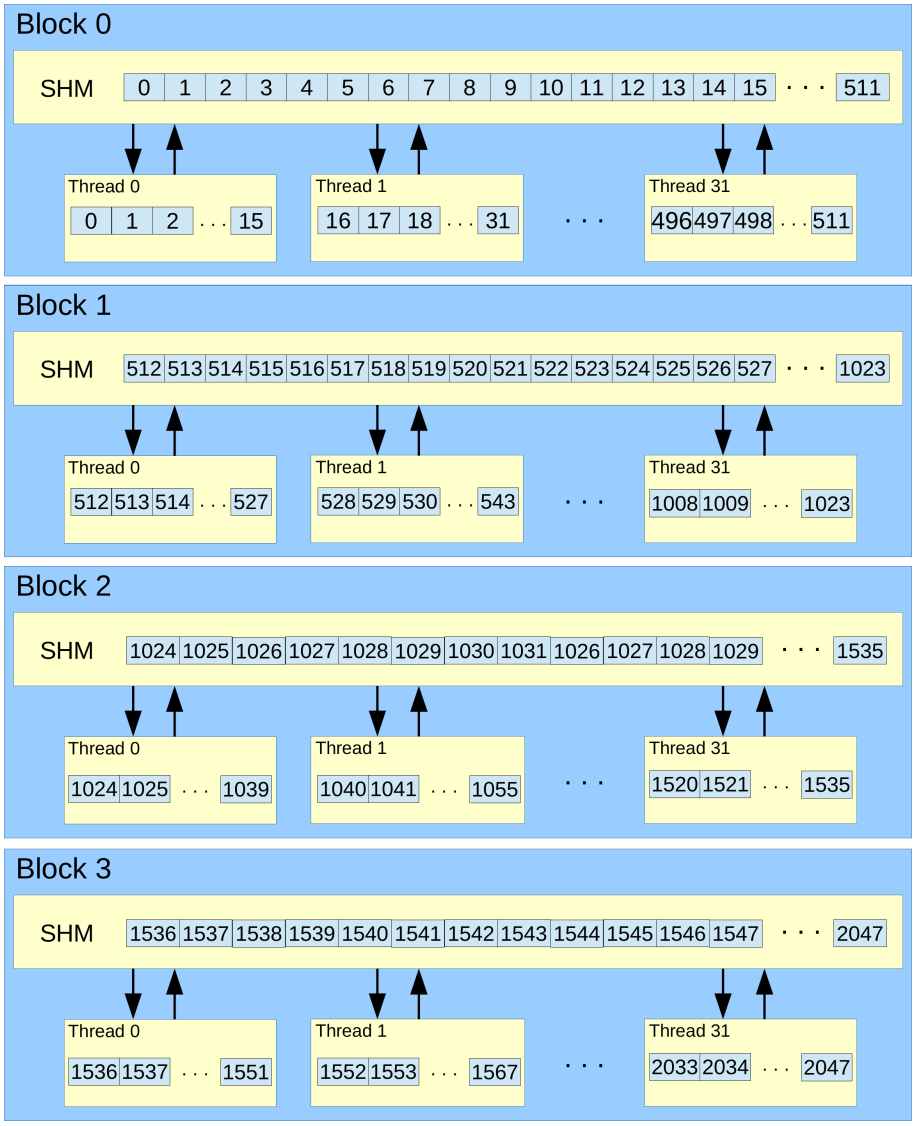


Figure 5.1: Data mapping with $r = 2$, $n = 11$, $s = 9$, $p = 4$ and $b_x = 2$.

keep the Index-Digit representation with $r = 2$.

The second type of operators represents data permutations.

Definition 2. *The perfect unshuffle operator $\Gamma_{i,j}$, $i \geq j$, performs a cyclic shift to the right between the i-th and j-th digits of the Index-Digit representation of the data.*

We also define a generalization of this operator, $\Gamma^m_{i,j}$. Instead of performing a single cyclic shift to the right, it will perform m consecutive shift operations, such

as $\Gamma^2_{i,j} = \Gamma_{i,j}\Gamma_{i,j}$. For instance, $\Gamma^2_{7,2}[\ t_8\ t_7\ t_6\ t_5\ t_4\ t_3\ t_2\ t_1\ ] = [\ t_8\ \underline{t_3\ t_2\ t_7\ t_6\ t_5\ t_4}\ t_1\ ]$.

Definition 3. *The general unshuffle operator $\Gamma_{i,j,k,l}$, $i \geq j \geq k \geq l$, is similar to the previous definition, however it is applied to two digit subfields $\{i \dots j\}$ and $\{k \dots l\}$ of the Index-Digit representation.*

Therefore, data in the range $\{t_{j-1} \cdots t_{k+1}\}$ remain unmodified. For instance, $\Gamma_{8,6,2,1}[\ t_8\ t_7\ t_6\ t_5\ t_4\ t_3\ t_2\ t_1\ ] = [\ \underline{t_1\ t_8\ t_7}\ t_5\ t_4\ t_3\ \underline{t_6\ t_2}\ ]$.

Definition 4. *The digit reversal operator $\rho_{i,j}$, $i \geq j$, performs the reversal of the digits between the i and j-th digit of the Index-Digit representation of the data.*

For instance, the digit reversal of the sequence $\rho_{7,2}[\ t_8\ t_7\ t_6\ t_5\ t_4\ \ t_3\ t_2\ t_1\ ] = [\ t_8\ \underline{t_2\ t_3\ t_4\ t_5\ t_6\ t_7}\ t_1\ ]$. This operator coincides with its inverse.

Once the algorithm expression is generated with the operators, obtaining the code is quite straightforward. Permutation operators are easily implemented using different strides and offsets when transferring data from different memory spaces. Computation operators are implemented directly from their definition. The implementation makes extensive use of template functions (CUDA skeletons) to create several optimized versions, depending on the problem size and the target architecture. Different tables are built, where each problem size represents an entry indicating both how to split the problem over the number of kernels and the optimized performance parameters for each kernel. The library chooses the entry depending on the problem size and target architecture, and kernels are then built with these parameters at compile time, via template metaprogramming. Hence, the user does not have to generate it. Most of the function calls, register loops and redundant move operations will be fully optimized at compile time. Thus, this approach provides generality and usability, generating well performing kernels with little effort, as can be seen in the performance evaluation section (see Section 5.3).

Operator	*Definition*
Node	Υ_i^r, with $1 \leq i \leq n$, computes r data elements whose index differs in the i-th digit.
Perfect Unshuffle	$\Gamma_{i,j}[t_n \cdots t_1] = [t_n \cdots t_{i+1} \underline{t_j t_i \cdots t_{j+1}} t_{j-1} \cdots t_1]$
General Unshuffle	$\Gamma_{i,j,k,l}[t_n \cdots t_1] =$ $[t_n \cdots t_{i+1} \underline{t_l t_i \cdots t_{j+1}} t_{j-1} \cdots t_{k+1} \underline{t_j t_k \cdots t_{l+1}} t_{l-1} \cdots t_1]$.
Digit Reversal	$\rho_{i,j}[t_n \cdots t_1] = [t_n \cdots t_{i+1} \underline{t_j t_{j+1} \cdots t_i} t_{j-1} \cdots t_1]$.

Table 5.1: Description of string operators.

5.2. Multi-Stage Index-Digit Tridiagonal System Solver Algorithm (MS-ID-TS)

There are several reasons for solving tridiagonal systems of large size [74]: (*i*) a set of small systems can be expressed as a single large problem by joining their matrices, (*ii*) solving a large problem is the most difficult case to implement, since there is no independence to compute slices of the problem separately, in addition to which the common shared-memory is limited. In the case of GPU programming, there is also another strong reason: (*iii*) although there are a great number of GPU-based solvers for small problem sizes, only few implementations can handle large problem sizes.

Our *MS-ID-TS* proposal is based on the Wang and Mou algorithm [132] and solves large problem sizes efficiently. The computation is divided into n steps, and follows a pattern similar to *Cooley-Tukey*, but excluding the initial bit-reversal stage.

Each Node operator operates on triads of equations, labeled *L*eft, *C*enter and *R*ight, represented as:

$$[i]^{t-1} = [\ \underbrace{E^{t-1}_{q \cdot 2^{t-1}}}_{L_i}, \underbrace{E^{t-1}_{i}}_{C_i}, \underbrace{E^{t-1}_{(q+1)2^{t-1}-1}}_{R_i}\ ] \tag{5.6}$$

First, the middle term of the equation R_i reduces the first term of L_j. The middle term of the new equation in L_j is used to reduce the final term of L_i and C_i. On the other hand, the final term in R_i reduces the middle term of the original

L_j, and then, the new equation in L_j reduces the first term of R_j and C_j. At the end, both left-hand equations will be identical (see L'); the same is true for both right-hand equations (see R'). After n computation steps, the solution x_i can be immediately computed by dividing the second term of C_i by its independent term. This is the basic computation in the case of radix-2, but higher radix versions can be used. Therefore, each element in a Node operator is a triad of equations that requires 3×16 bytes of storage. However, when dealing with adjacent equations, there is a property which means that the whole triad need not be stored, just a single equation, as the two others are easily obtained from adjacent equations. Specifically, the left- and right-hand equations are equal to two of the center equations. In step k, the left- and right-hand equations of $[i]$ can be obtained as follows:

$$L_i = C_a \rightarrow a = 2^k \times \lfloor i/2^k \rfloor$$
$$R_i = C_b \rightarrow b = 2^k \times (1 + \lfloor i/2^k \rfloor) - 1$$

Hence, each element is represented by a single central equation and is stored in a *float*4 data type, since its right- and left-hand equations are easily obtained from the central equations of other elements. This property only arises in the first stage of the algorithm, where adjacent equations are stored in a common memory space; whereas in the remaining stages, triads need to be stored for each equation, since the central equations used for calculating the right- and left-hand equations could be placed into another memory space. Henceforth, an element is formed by one equation in the first stage, thanks to the adjacent property, but by 3 equations (a triad) in the remaining stages.

Figure 5.2 shows the reason why this property cannot be applied in several stages. It shows the *MS-ID-TS* proposal for $n = 4$, $p = l = 1$ and $b_x = 2$, where each number-box represents an element. The computation is divided into two stages; the first stage processes the first and second steps, while the second stage performs the third and fourth steps, using four threadblocks in each step. In the first stage, adjacent elements are stored in the corresponding shared memory of each threadblock, so only one equation per element is stored. However, it is easy to observe how this behavior changes in the second stage, as each threadblock works with non-

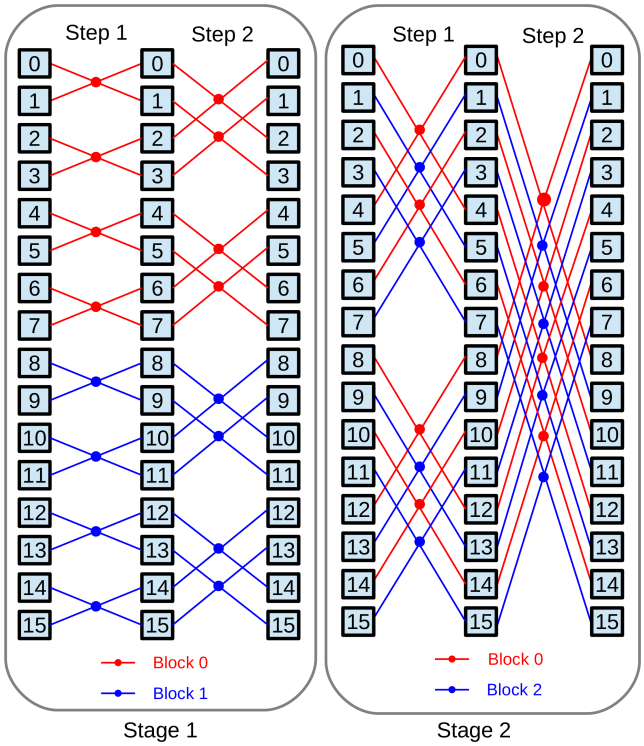


Figure 5.2: Distribution of the *ID-LD-TS* proposal with two stages for $N = 16$, $p = l = 1$ and $b_x = 2$

adjacent elements, storing the whole triads for each element. For example, *element 4* is solved by threadblock 0 in the second stage (although it was processed by threadblock 1 in previous stage). Accordingly, the right-hand equation of *element 4*, R_4, corresponds with the central equation of *element 7*, C_7, in the first step of the second stage. However, *element 4* cannot access *element 7*, since they are in different threadblocks: *element 4* is stored in threadblock 0, whereas *element 7* is contained in threadblock 3. This fact forces us to store the corresponding three equations of each triad for all elements.

Changing the access pattern (i.e., changing the current data distribution among threadblocks) to another pattern where adjacent elements are placed together in the same threadblock (working with portions of consecutive elements) would imply increasing the number of stages (with its corresponding latency penalty). This new communication pattern guarantees processing the maximum number of steps per stage, thus minimizing the number of stages. For non-initial stages, we have

preferred to launch a small number of kernels which store whole triads, instead of launching more kernels whose elements are single equations. We justify this decision on the basis that the new GPU architectures (and it is highly likely that future architectures too) increase their shared memory space, which is beneficial for this implementation.

Due to this limitation, it has been necessary to differentiate the s parameter depending on its being in the first stage or in subsequent ones. Thus, s is split into s_1 and s_2, as data size is not the same for a single equation (first stage) as for a triad (remaining stages). Therefore, s_1 is used to represent the elements in the first stage, where it is not necessary to store triads, and s_2 is used in remaining stages, taking into account that elements are represented by triads of equations, $3 \times float4$. Note that both s_1 and s_2 refer to the number of stored elements, irrespective of their size. Thus, the first kernel's shared memory can hold more elements than other kernels shared memory, as its elements are much lighter than the elements in the remaining stages, $s_1 > s_2$. In our proposal, the first stage computes $\lfloor s_1/p \rfloor$ steps and the remaining stages will compute $w = \lfloor \frac{(n-s_1)}{p \cdot (m-1)} \rfloor$ steps per stage. In order to take advantage of the data type used for representing the elements, s_1 should be as large as possible, since more steps can be performed in Stage 1 in comparison to other stages, while using the same amount of shared memory thanks to the adjacency property.

Moreover, instead of having only $s_2 = r^w$ elements per threadblock in the remaining stages, our implementation stores s_2 elements of the same problem in each threadblock, where the s_2 value is defined to maximize GPU parallelism, as explained below. Thus, each threadblock computes several sets of r^w dependent elements from the same problem until s_2 is fulfilled. This increases warp parallelism, performing more work in each threadblock. There is dependence among elements of the same set (computing w steps implies r^w elements), but sets are independent from each other. As each set operates separately without needing information from the other sets, the number of performed steps is still w, and all threads in a threadblock working on the same set have the same l_y-identifier.

In order to better understand the mapping vector design, the CUDA implementation steps of our *MS-LD-TS* algorithm are analyzed below. At the beginning of computation, there is a table which determines the number of stages and steps

processed by stage (kernel) for each problem size. Likewise, there is another table which specifies s, l and, thus, the radix employed for each problem size.

1. Each thread loads P elements from global memory into registers. In the first stage, these elements are adjacent, benefiting coalescence. In the remaining stages, each thread loads triads of equations following the corresponding pattern.
2. Compute the first step using radix-P or a mixed radix.
3. Exchange data through shared memory. Equations are stored as *float4* elements in shared memory. Except in the first stage, there are three shared memory buffers, one for each triad.
4. Compute the following step.
5. If no step remains in that stage, then the result is written to global memory in last stage or triads are stored into global memory in the remaining stages.
6. There are two possibilities:

   a) If all the required stages have been completed, the algorithm ends.
   b) Otherwise, the next kernel reads triads from global memory, using the corresponding offset between their elements, restarting the process of this list from Point 1.

5.2.1. MS-ID-TS Mapping Vector

In our proposal, p is mapped to the lower part of the Index-Digits to ensure the global memory coalescence. Each component of the equation is stored in a different array, and each thread accesses four consecutive elements from up to four equations using $float4$ data type. In the first stage, the mapping vector of data on the GPU resources is as follows:

$$[\ \underbrace{t_{n+batch}\cdots t_{n+1}}_{b_y}\overbrace{\underbrace{t_{n}\cdots t_{l+p+1}}_{b_x}\ \underbrace{t_{l+p}\cdots t_{p+1}}_{l}\underbrace{t_{p}\cdots t_{1}}_{p}}^{n}\ ] \tag{5.7}$$

In order to determine the (p, s_1, l) tuple, two factors need to be considered: on the one hand, using the largest shared memory possible is advisable in order to compute the maximum number of steps in the first stage as explained above; on the other hand, it is also important to fulfill the three stated premises, analyzing each target architecture and finding a trade-off.

Following Premise 1, the number of stages should be minimized and is determined by s. In Kepler, $s_1 = 11$ implies only 1 active threadblock and 25% warp occupancy per SM; $s_1 = 10$ involves 3 active threadblocks and 38% warp occupancy, whereas $s_1 = 9$ generates 6 active threadblocks and 38% warp occupancy. Lower s_1 values underexploit shared memory, as registers would be the limiting factor of occupancy. Thus, $s_1 = 9$ is selected in order to achieve Premise 2, making it possible to solve $n \leq 18$ problem sizes with only $m = 2$ stages. The same reasoning is applied to Maxwell, choosing $s_1 = 9$. This involves 8 active threadblocks and 32 active warps per SM. However, taking into account that s_2 stores at least $n - s_1$ elements, and each element occupies 48 bytes in the second stage, then this involves $s_2 \leq 9$, owing to hardware limitations, and the second kernel occupancy would be very low when $n > 16$. In order to avoid this, $s_1 = 10$ is utilized when $n > 16$. Although some occupancy is lost in Stage 1, performance will be improved in the second stage, obtaining better global performance in the whole application. For example, note that executing $n = 17$ on Kepler with $s_1 = 9$ (and $p = 2$) in the first stage implies the following mapping vector in the second stage:

$$[\ t_{17+batch} \cdots t_{18} \overbrace{t_{17} \cdots t_{10}\ \underbrace{\underbrace{t_8 \cdots t_3}_{l}\ \underbrace{t_2 t_1}_{p}}_{s_2=8}}^{n}\ ] \tag{5.8}$$

Obtaining only 4 concurrent threadblocks and 8 active warps per SM in the second stage. Nevertheless, using $s_1 = 10$ and $s_2 = 7$ involves the following mapping vector:

$$[\ t_{17+batch} \cdots t_{18} \overbrace{t_{17} \cdots t_{11} \underbrace{\underbrace{t_7 \cdots t_3}_{l} \underbrace{t_2 t_1}_{p}}_{s_2=7}}^{n}\ ] \tag{5.9}$$

with 8 concurrent threadblocks and 8 active warps per SM in the second stage. Despite slightly reducing the number of concurrent threadblocks in the first stage, global performance is enhanced. Additionally, thanks to using $s_1 = 10$ in large problem sizes, up to $n \leq 19$ sizes can be solved in only 2 stages. Maxwell provides up to 96 KB per SM, delaying this parameter update until $n > 17$, as it achieves a higher occupancy than Kepler at the same shared memory consumption. Once both s_1 and s_2 have been established, and taking into account that $p = 2$ according to [82], the following tuples are obtained for Kepler: $(p, s_1, l) = (2, 9, 7)$ when $n \leq 16$, and $(p, s_1, l) = (2, 10, 8)$ when $16 < n \leq 19$. Regarding Maxwell, these values are $(p, s_1, l) = (2, 9, 7)$ when $n \leq 17$ and $(p, s_1, l) = (2, 10, 8)$ otherwise.

In the remaining stages, the mapping vector is:

$$\begin{aligned}[\underbrace{t_{n+batch} \cdots t_{n+1}}_{b_y} \underbrace{t_n \cdots t_{b_x+l_y+p+1}}_{l_x} \\ \underbrace{t_{b_x+l_y+p} \cdots t_{b_x+l_y+1}}_{p} \underbrace{t_{b_x+l_y} \cdots t_{b_x+1}}_{l_y} \underbrace{t_{b_x} \cdots t_1}_{b_x}\ ]\end{aligned} \tag{5.10}$$

In this case, the use of p as of the $b_x + l_y + 1$ Index-Digits also ensures global memory coalescence. In the remaining stages, the performance parameters used are given by $p = 2$, $s_2 = max(6, n - s_1)$ and $(l_x, l_y) = ((n - s_1 - p), (s_2 - l_x - p))$. In the case of $s_2 = 6$, this ensures having 3072 shared memory bytes per block, not limiting block parallelism in either Kepler or Maxwell architectures and executing several sets per threadblock, in the case of small problem sizes. When $n > (6 + s_1)$, a single set of elements is processed by each threadblock, consuming as much shared memory as necessary. Regarding the distribution of threads in each threadblock, $l_x = (n - s_1 - p)$, L_x of them collaborate in the same set of equations that have dependencies between each other, as explained above, while $l_y = (s_2 - l_x - p)$ represents the fact that there are L_y sets of equations of the same problem being

independently solved in the threadblock. Therefore, our implementation uses L_x for working on the same set, whereas L_y sets of the same problem are solved in parallel in that threadblock. Finally, B_x thread blocks work on the same problem, whereas B_y blocks work on different problems in batch mode.

The string operator used in the first kernel is as follows:

$$\prod_{i=1}^{s_1/p-1} \left[\Upsilon_1^p \Gamma^p_{(i+1)\cdot p, i\cdot p+1, p, 1}\right] \Upsilon_1^p \Gamma^p_{s_1,1} \tag{5.11}$$

In the case of ($s_1 \; mod \; p \neq 0$), an extra step is needed. For the remaining kernels, the same expression is used but an offset of $l_y + b_x + 1$ digits is applied in sub-indexes, as its mapping vector is different, executing $\frac{n-s_1}{p}$ steps.

For example, for $n = 14$, the data mapping vector in the first stage would be

$$[\underbrace{\cdots t_{15}}_{b_y} \overbrace{\underbrace{t_{14}t_{13}t_{12}t_{11}}_{b_x} \underbrace{t_{10}t_9t_8t_7t_6t_5t_4t_3}_{l} \underbrace{t_2t_1}_{p}}^{n}] \tag{5.12}$$

and in the second stage

$$[\underbrace{\cdots t_{15}}_{b_y} \overbrace{\underbrace{t_{14}t_{13}t_{12}t_{11}t_{10}}_{l_x} \underbrace{t_9t_8}_{p} \underbrace{t_7t_6}_{l_y} \underbrace{t_5t_4t_3t_2t_1}_{b_x}}^{n}] \tag{5.13}$$

5.3. Experimental Results for ID-Algorithms with Medium-Large Problem Sizes

In this section, the results of the tridiagonal system solver are presented and analyzed.The test data are already on the GPU, thus there are no data transfers during the benchmarks. The experiments are run in single precision. Table 5.2 describes the test platforms used in our experiments. The first two platforms have similar features, presenting a Kepler GPU architecture, whereas the third platform has a Maxwell GPU architecture.

	Kepler K20 Platform, Kepler K40 Platform	*Maxwell Platform*
CPU	Intel Xeon E5-2660 2.2 GHz	Intel Core i7-2600 3.4 GHz
Memory	64 GB DDR3 1600	8 GB DDR3 1333
OS	CentOS 6.4	Ubuntu 12.04 LTS
Compiler	GCC 4.4.7	GCC 4.6.3
GPU	*Nvidia Tesla K20, Nvidia Tesla K40*	*Nvidia GeForce GTX980*
Driver	340.58, SDK 6.0	343.22, SDK 6.5

Table 5.2: Description of the test platforms

In the case of tridiagonal system solver, performance is measured in million rows processed per second, MROWS/s. Therefore, the MROWS/s value is obtained using the expression $N \cdot r^{batch} \cdot 10^{-6}/t$. In these tests, data are initialized using diagonally dominant equation systems. Our *MS-ID-TS* proposal considers problem sizes with $8 \leq n \leq 19$. For dealing with larger sizes, more GPU memory would need to be used, as explained later in the text. Our proposal is then compared with respect to *CUSPARSE* [95], the state-of-the-art for medium and large problem sizes, since *CUDPP* [98] does not support these sizes.

Table 5.3 and Table 5.4 present the profiler analysis for the Kepler K20 Platform and the Maxwell Platform, respectively. Each column contains the values obtained for the two executed kernels. Firstly, it should be noted that the $(p, s_1, l) = (2, 9, 7)$ tuple was established on Kepler in Section 5.2 when $n \leq 16$, as can be seen in Table 5.3, obtaining 768.6 MRows/s when $n = 16$. From $n = 17$, the tuple $(p, s_1, l) = (2, 10, 8)$ is used for the reasons explained in Section 5.2 (decreasing block parallelism on kernel 1, but increasing global performance since block parallelism is increased on kernel 2), achieving 664.2 MRows/s. Table 5.3 shows global performance for both cases, $s_1 = 9$ and $s_1 = 10$, highlighting the best result. However, Table 5.4 shows 1554.2 MRows/s when $n = 16$ and 1623.4 MRows/s when $n = 17$. In this case, the same $(p, s_1, l) = (2, 9, 7)$ tuple is being used for both cases, as Maxwell architecture provides up to 96 KB per SM, delaying $s_1 = 10$ until $n = 17$, as it maintains a higher occupancy than Kepler at the same shared memory consumption. With $n = 18$, 1556 MRows/s are achieved in Maxwell platform, since the new $(p, s_1, l) = (2, 10, 8)$ tuple is being employed, achieving 1.29x with respect to the $s_1 = 9$ implementation. Finally, it should be observed that greater occupancy is achieved on Maxwell architectures than on Kepler architectures at the same level

n	*MRows/s*	L	*Reg*	Sh. mem. bytes	Occup (%)
12	690.6	128,16	56,66	8192,3072	36,25
13	826.2	128,16	56,66	8192,3072	36,25
14	888.4	128,16	56,66	8192,3072	36,25
15	792.6	128,16	56,66	8192,3072	36,25
16	768.6	128,32	56,68	8192,6144	36,13
17	660.3	128 ,64	56,68	8192,12288	36,13
	664.2	256,32	59,68	16384,6144	36,13
18	645	128,128	56, 68	8192,24576	36,13
	687	256,64	59,66	16384,12288	36,13
19	620	256,128	59,61	16384,24576	36,13

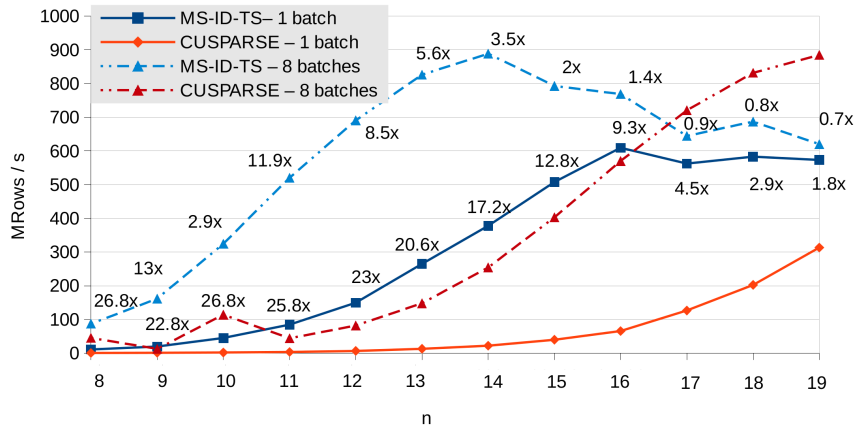
Table 5.3: Complex MS-ID-TS kernel performance and profiler analysis (Kepler K20 Platform)

n	*MRows/s*	L	*Reg*	Sh. mem. bytes	Occup (%)
12	1202.1	128,16	55,63	8192,3072	48,33
13	1130.9	128,16	55,63	8192,3072	48,33
14	1349.3	128,16	55,63	8192,3072	48,33
15	1508.8	128,16	55,63	8192,3072	48,33
16	1554.2	128,32	55,64	8192,6144	48,16
17	**1623.4**	128,64	55,65	8192,12288	48,16
	1516	256,64	59,59	16384,12288	48,16
18	1204	128,128	55,59	8192,24576	48,16
	1556	256,64	59,65	16384,12288	48,16
19	1250.4	256,128	59,63	16384,24576	48,13

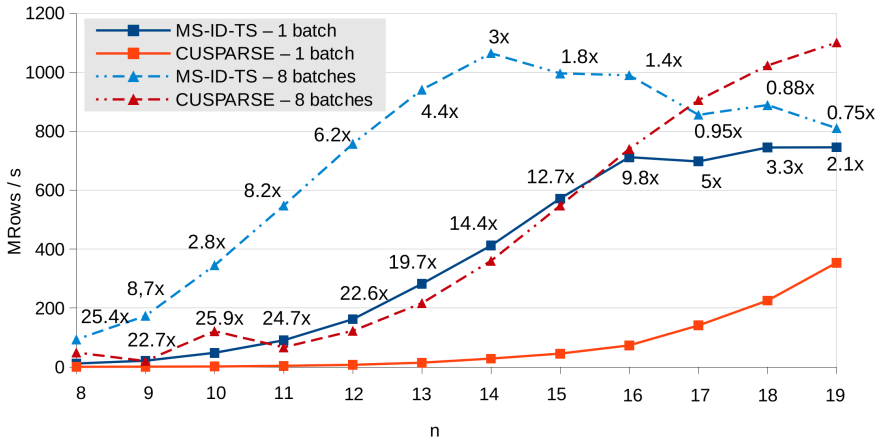
Table 5.4: Complex MS-ID-TS kernel performance and profiler analysis (Maxwell Platform)

of shared memory consumption due to the increased SM shared memory size in Maxwell, as explained above.

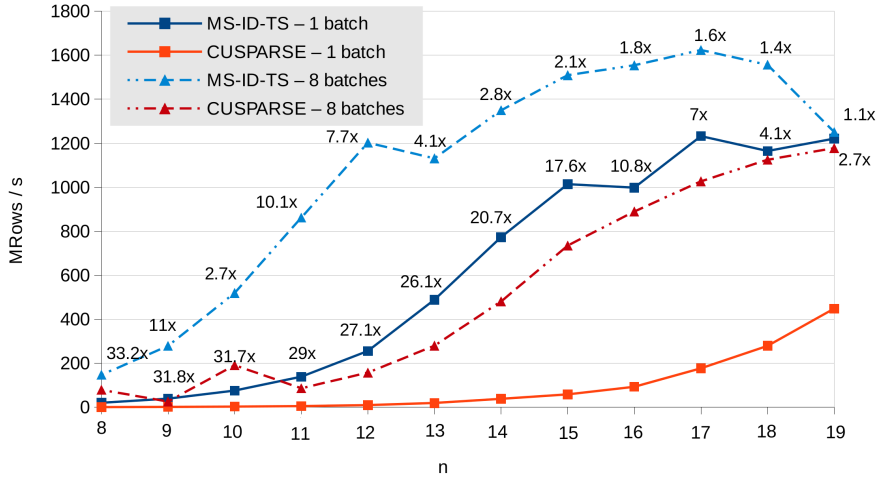
Figure 5.3 shows the results of executing a single problem (solid lines) and multiple batches (dashed lines) on the three Platforms. The performance comparison with respect to the *CUSPARSE* library for one batch is very similar on all three



(a) Kepler K20 Platform



(b) Kepler K40 Platform



(c) Maxwell Platform

Figure 5.3: Performance comparison of $MS-ID-TS$ proposal

platforms. The performance growth of *CUSPARSE* is very slow, as it launches 10 kernels. However, the performance growth in our solver is immediate. From $n = 16$ on Kepler Platforms; and from $n = 17$ on the Maxwell Platform, performance begins to decrease, as was expected owing to the replacement of $s_1 = 9$ by $s_1 = 10$. In Section 5.2 and Tables 5.3 and 5.4, we have justified the peaks in $n = 16$ (on Kepler) and $n = 17$ (on Maxwell). As having more elements requires more shared memory, then the fixed amount of shared memory, optimized in our implementation, is not sufficient and needs to be increased. This increase in shared memory leads to reduced occupancy and a loss of performance, obtaining those peaks. In the case of eight batches, the speed-up with respect to *CUSPARSE* is more modest, as storing triads from all batches in global memory consumes much more bandwidth. As more batches are introduced, more GPU parallelism is exploited. Furthermore, more global memory operations are issued, and L1/L2 cache behavior will determine the location of peaks for each batch execution. In all cases, the occupancy in the second kernel is lower, especially when dealing with large problem sizes, where the shared memory becomes a limiting factor.

In the case of the Kepler K20 Platform and one batch, our solver obtains up to 26.8x improvement over *CUSPARSE*, being 16.37x times faster on average. With eight batches, this proposal is, on average, 4.41x faster. On the Kepler K40 Platform, it is up to 25.9x faster for one batch, 15.7x on average; whereas it provides up to 8.7x of speed-up when processing eight batches simultaneously. Additionally, an improvement of up to 33.2x is achieved with one batch on Platform 3, 20.14x on average, while a speed-up of up to 11x with eight batches, 4x on average.

5.4. A three-phase Methodology for Parallel Prefix Algorithms

Index-Digit algorithms are a subset of parallel prefix algorithms, whose communication pattern facilitates their representation with mapping vectors and string operators. However, as previously mentioned, not every parallel prefix algorithm suits in the definition of ID-algorithm, and a general methodology is required for them. To do this, the 3-phase tuning methodology exposed in Chapter 4 for small

problem sizes is here extended to medium and large problem sizes.

As in the previous case, the *GPU Resources Utilization Analysis* phase identifies the performance parameters which influence on the GPU throughput, and establish a set of theoretical performance premises. In the *CUDA Kernel Optimization* phase, the corresponding CUDA kernels are built with CUDA skeletons. Finally, the optimal performance parameter values for the developed kernels are obtained in the *Performance Parameter Tuning* phase. Since the morphology of parallel prefix algorithms hinders the representation thereof in terms of string operators and mapping vectors, the number of phases slightly varies.

Specifically, we analyze this methodology for the scan primitive based on the Ladner-Fischer pattern, and the Tree-Partitioning Reduction solver for tridiagonal systems, explained in Chapter 3.

5.4.1. GPU Resources Utilization Analysis

When working with parallel prefix algorithms, we establish $r = 2$ as radix. Table 5.5 collects the performance parameters identfied. Thus, our strategy considers the case of simultaneously executing $G = 2^{batch}$ problems of N elements each, where $N = 2^n$.

In Chapter 2, these parameters where introduced, but not the distribution of the problem and execution were not explained in terms of them. Problems are solved using $B = 2^b$ threadblocks per kernel, which can be scheduled into a two-dimensional distribution $B = B_x \cdot B_y$, $(b = b_x + b_y)$. In our strategy, B_x represents the number of threadblocks used for computing each problem, whereas B_y represents the number of problems being simultaneously executed on that kernel. Each threadblock comprises $L = 2^l$ threads, which can be distributed as $L = L_x \cdot L_y$, $(l = l_x + l_y)$. L_x represents the number of threads per threadblock working on the same problem, whereas L_y the number of problems being solved in that threadblock. Each thread works with $P = 2^p$ elements of the problem in private registers, whereas all threads into a thread block can access to $S = 2^s$ elements in shared memory. There are cases where the data stored in registers have no copy in shared memory. For example, when using *shuffle* instructions (intra-warp communication via registers), shared memory is not

Problem Parameters

$N = 2^n$	Problem size.
$G = 2^{batch}$	Number of problems being solved simulteneously.

GPU Performance Parameters

$S = 2^s$	Number of shared-memory elements per block.
$P = 2^p$	Number of elements stored in registers per thread.
$B = 2^b$	Number of thread blocks executed per GPU, where $B = B_x \cdot B_y$.
$L = 2^l$	Number of threads that compose a block, where $L = L_x \cdot L_y$ and $S \leq P \cdot L$
m	Number of stages (kernels) invoked to compute a problem

Table 5.5: Description of the performance parameters for parallel prefix algorithms.

needed for exchanging information inside a warp. In that case, shared memory is only used for exchanging data among different warps; thus, $s \leq p+l$. When working with several kernels, all previously defined parameters use a superscript number to identify the referred kernel. For example, B_x value of kernel 1 is represented by B_x^1, whereas B_x value of kernel 2, by B_x^2.

As in the case of ID-algorithms, the methodology focuses on solving large-size problems, partitioning the problem among several threadblocks (*multi-stage* strategy). The exchange of information between threadblocks is performed through global memory. Threadblocks write their information in global memory, and after using any global synchronization mechanism, the remaining threadblocks will be able to read this information from global memory.

However, in contrast to the ID methodology, it should be noted that parallel prefix algorithms work with radix $r = 2$, thus all parameters are expressed in terms of a power of two. This specification slightly varies the GPU Resources Utilization Analysis phase.

Premises for Performance Maximization

Our tuning strategy is based on a set of premises that aim at obtaining different GPU performance parameters which maximize the execution throughput. These premises were explained in Section 5.1.1, and are adapted to work with parallel prefix algorithms, where the radix is $r = 2$, as summarized below.

Premise 1. *Balancing warp and block parallelism.* Higher parallelism obtains better performance since it hides latency from functional units and the access to memory. However, this parallelism is limited by the amount of common resources shared in an SM. More threads per threadblock implies less resources in each threadblock. In the GPU, the level of parallelism can be supported in terms of the number of blocks per SM (*SM* block parallelism), or the number of warps per SM (*SM* warp parallelism), so our strategy attempts to strike a balance between the maximization of both:

1. *The maximization of SM block parallelism in each stage* in order to keep processing the maximum amount of simultaneous blocks per SM (16 in the case of *Kepler* and 32 in the case of *Maxwell*-based *GPUs*). Factors that limit SM block parallelism are the number of registers used by each thread and the amount of shared memory required by a threadblock.
2. *The maximization of SM warp parallelism in each stage.* This premise is focused on increasing the number of warps per SM, allowing the SM to hide latency among warps when one stalls.

In order to balance warp and block parallelism, it is necessary to limit the factors that decrease the SM parallelism, such as the number of registers used by each thread or the amount of shared memory required per threadblock.

Premise 2. *Increase the computational load per thread.* The number of elements processed by each thread, P, influences the number of computing steps per stage and the number of threads that process a problem. A larger P delivers higher performance, as there are fewer shuffle exchanges and more elements are processed in each iteration. Nevertheless, the increase of P may also require too many registers per thread, reducing the block parallelism or generating memory spilling (high-latency memory usage when there are no registers available). In contrast to the ID methodology, increasing P does not reduce the number of computing steps, as r remains constant.

Premise 3. *Maximization of SM occupancy and minimization of global memory communications.* The multi-stage strategy involves the invocation of m kernels. The number of steps processed by each kernel influences others, thus it is important to find the optimal distribution that maximizes global throughput.

5.4.2. CUDA Kernel Optimization

Thanks to the modularity and generality of our CUDA skeletons, they can easily be extended to other algorithms and designs with no effort. This phase uses our set of CUDA skeletons developed so far, extending them with the corresponding needs of each algorithm. Sections 5.5 and 5.6 give greater details about their implementation.

5.4.3. Performance Parameter Tuning

In order to balance warp and block parallelism, it is necessary to limit the factors that decrease the SM parallelism, such as the number of registers used by each thread or the amount of shared memory required per block. Table 5.6 summarizes different GPU performance configurations in order to maximize SM warp and block parallelism in Kepler and Maxwell architectures. It is easy to see that increasing warp parallelism reduces block parallelism, and vice versa. However, there are configurations, marked as a bold row in the table, that maximizes both types of parallelism, as Premise 1 indicates. In Sections 5.5 and 5.6, the optimal configuration for each case is analyzed.

5.5. Scan Primitive based on Ladner-Fischer

As previously introduced, there were three typical parallel patterns for computing the scan primitive: the *Brent-Kung pattern*, the *Kogge-Stone pattern* and the *Han-Carlson pattern*. However, when computing large problem sizes in GPUs, the Brent-Kung offers the worst performance. In [56], the authors present a CUDA implementation of this pattern, whose performance is limited by the existence of two phases, doubling the number of synchronization barriers and the presence of warp divergence in some stages. They use a multi-stage strategy in which the problem is partitioned in different threadblocks, and each threadblock needs to cooperate with each other. With this in mind, the proposal uses a recursive processing: after each threadblock computes its own scan, it saves the accumulative addition of the whole block in an auxiliary array. This procedure is repeated in different levels until the auxiliary array can be computed by a single threadblock. After this, each element of

Archit.	Warps per block	Regs per thread	Shared mem per block	SM warp occupancy	SM number of blocks
Kepler cc 3.7	1	256	7168	25%	16
	2	128	7168	50%	16
	4	**64**	**7168**	**100%**	**16**
	8	64	14336	100%	8
	16	64	28672	100%	4
	32	64	49152	100%	2
Maxwell cc 5.0	1	64	2048	50%	32
	2	32	0	100%	32
	2	40	0	75%	24
	2	**32**	**2048**	**100%**	**32**
	4	32	4096	100%	16
	8	32	8192	100%	8
	16	32	16384	100%	4
	32	32	32768	100%	2

Table 5.6: Performance parameters per SM on Kepler Platforms with compute capability 3.7

the auxiliary array at one level is added to all elements of the corresponding threadblock in the previous level. This strategy is also similar to the method implemented in [39].

In [85], which combines Brent-Kung and Kogge-Stone patterns, the execution of large-problem sizes is improved. In order to produce fewer intermediate values, the number of threadblocks is set to a constant C value before execution. In the first stage, each threadblock computes the reduction for an input dataset, which produces C intermediate values to be stored in global memory; in other words, there are C accesses to global memory. Each threadblock iterates over a loop (*cascade approach*), processing a chunk of data in each iteration, using shared memory for communication between iterations. The second stage reads back these C values and produces their scan. Finally, the third stage reads all of the input elements again, performs their scan and updates them with the results of the second stage. Hence, the total amount of global memory required is $3N + 3C$. The value for C has to be small enough to compute the scan but large enough to obtain the maximum warp occupancy per SM. Later, this implementation is improved in [53] by eliminating redundant global memory accesses in the first and last threadblocks, obtaining $(3N + 3C) - (\frac{2N}{C} + 3)$ accesses to global memory. Our multi-stage scan

proposal, *Scan Single-GPU Problem (Scan-SP)*, is based on this cascade strategy.

5.5.1. CUDA Kernel Optimization: Scan-SP

Based on current large-size scan problems solvers, data are divided into several data blocks. The reduction value of each data block is computed, stored in an auxiliary array and then all elements in the auxiliary array are scanned. Data blocks compute then their local scan and add the corresponding value from the auxiliary array to their elements, completing the overall scan. This procedure is summarized in Figure 5.4.

From the CUDA perspective, the execution is divided into three stages (kernels). As it can be seen in Figure 5.4, firstly, the N elements are divided into chunks, where each threadblock, represented by a color, processes one chunk. Thus, B_x^1 threadblocks work on the same problem, and $B_y^1 = G$ problems are being solved simultaneously. The first stage (*Chunk Reduce*) computes the reduction for each chunk (light colour square in figure). Note that reduction primitive means writing the cumulative sum for all elements into the last element, whereas the remaining elements are not modified. The result of each reduction is stored in an auxiliary array in global memory. Taking into account the fact that this is a memory-bound problem in current GPU architectures, storing one element per chunk and computing the scan later again is preferable to writing all elements in global memory twice. As each chunk writes its reduction in the auxiliary array, there are $G \cdot B_x^1$ elements in this array. Thus, a second stage (*Intermediate Scan*) computes the scan of these B_x^1 values for each problem using the Ladner-Fischer (LF) pattern, explained in Chapter 2, in a single threadblock. Finally, a third stage (*Scan+Addition*) performs the local scan for each chunk, adding the corresponding element from the global memory auxiliary array, processed in the previous kernel, to all elements in its chunk. Note that the number of elements per problem processed in Stage 1 and Stage 3 is N, whereas it is B_x^1 for Stage 2. As Stage 1 and Stage 3 input sizes are equal and these stages share a very similar computational core, each problem is partitioned into the same number of chunks and both stages use the same amount of SM resources. Thus, $B_x^1 = B_x^3 = B_x^{1,3}$.

In our strategy, each thread reads P elements from global memory using the *int4*

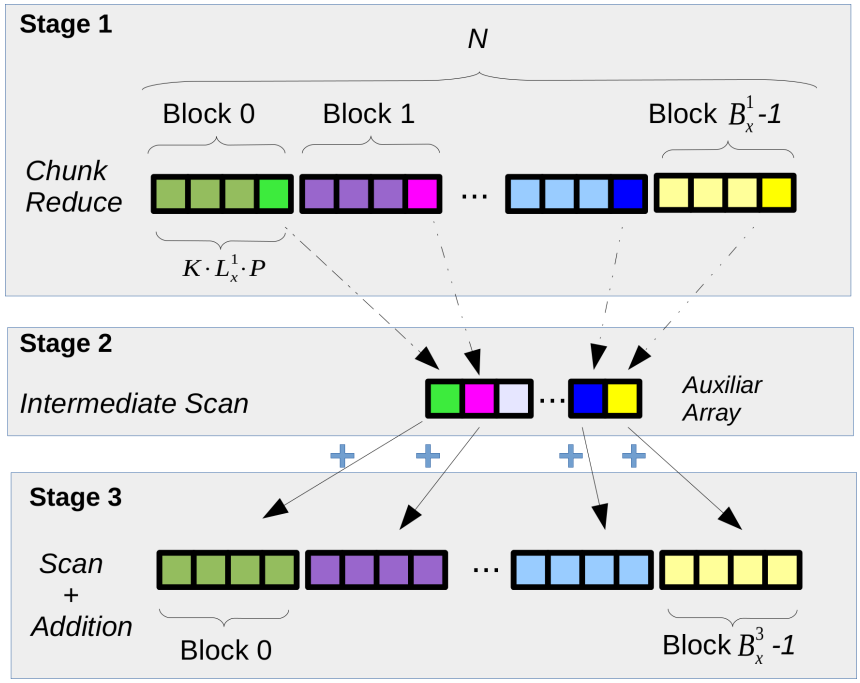


Figure 5.4: Three kernel execution for the scan primitive when $G = 1$ problems.

customized data type, facilitating coalescence and reducing memory transactions. These four elements are computed by each thread in registers, as the top part of Figure 5.5 shows for $L_x = 4$ and $P = 4$. For example, if P is equal to 8, then two loads from global memory are performed by each thread and two 4-elements scans are computed.

The L_x threads are grouped into warps; therefore, each warp computes $P \times warpSize$ elements ($warpSize = 32$ currently, although $warpSize = 4$ in Figure 5.5 for clarity). Hence, after the initial scan of P elements in a single step (red values in the figure), each warp computes $warpSize$ elements using shuffle instructions and the Ladner-Fischer communication pattern. Once the shuffle-scan is performed, each thread adds the corresponding value to its four elements. Using the exclusive scan saves an extra communication step, otherwise each thread would have to send the inclusive result to its neighbor, instead of directly adding the value stored in its register. Computing the exclusive scan is fast; the initial value is subtracted from the scanned value. Finally, the last element of the $P \cdot warpSize$ data sequence is stored in shared memory in order to share this partial sum with other warps. Hence, shared memory has as many elements as warps - at most 32 in current architectures. A

single warp will repeat this process over the 32 partial sums stored in shared memory in order to build the final result of the $P \cdot L_x$ elements. Note that, thanks to the use of shuffle instructions, $S \leq 32$ ($s \leq 5$).

In addition to this computational flow, this implementation also follows a cascade approach [53]. Each threadblock executes K iterations, where each iteration computes $L_x \cdot P$ elements, as explained in the previous paragraph. Thus, each threadblock computes $K \cdot L_x \cdot P$ elements; i.e., the chunk size is equal to $K \cdot L_x \cdot P$ elements. Once one iteration has computed $L_x \cdot P$ elements, the last one is passed to the next iteration, adding this value to all $L_x \cdot P$ elements of that iteration. Figure 5.6 shows this approach, which avoids launching an excessive number of threadblocks, and allows thread information to be reused, generating fewer instructions and also using fewer temporal values. After K iterations, the scan of $K \cdot L_x \cdot P$ elements has been computed. In the case of Stage 1 (Chunk Reduction), the last element is written in the auxiliary array to be passed to Stage 2. As the chunk size is a power of two, K is also a power of two.

As already mentioned, there are G problems being simultaneously solved in each kernel. In Stage 1 (Chunk Reduction) and Stage 3 (Scan+Addition), all threads in a block work on the same chunk, i.e. on the same problem ($L_y^{1,3} = 1$), thus $L^{1,3} = L_x^{1,3} \cdot 1$. In Stage 1, each chunk writes its reduction in an auxiliary array; thus, the number of chunks sets up the number of elements per problem to be processed in Stage 2, $B_x^{1,3}$. In the Intermediate Scan kernel, the same block must process elements from different problems, otherwise warp occupancy would be much too low, as Stage 2 executes much fewer elements. Therefore, all elements which come from the same problem have the same L_y^2 identifier, so $L_y^2 > 1$, $B_y^2 = G/L_y^2$ and $B_x^2 = 1$ in Stage 2.

All these operations are efficiently implemented using our CUDA skeletons, which are carefully designed to attain high levels of efficiency in CUDA architectures. They are designed with templates, enabling the generation, at compile time, of tuned kernels according to the more suitable (s, p, l, K) tuple for the specific GPU architecture in which they are to be executed. The compile-time generation allows the use of generic programming and template metaprogramming, reducing code complexity, avoiding temporal registers for function calls and taking advantage of fully unrolling static loops and avoiding the dynamic addressing of register arrays.

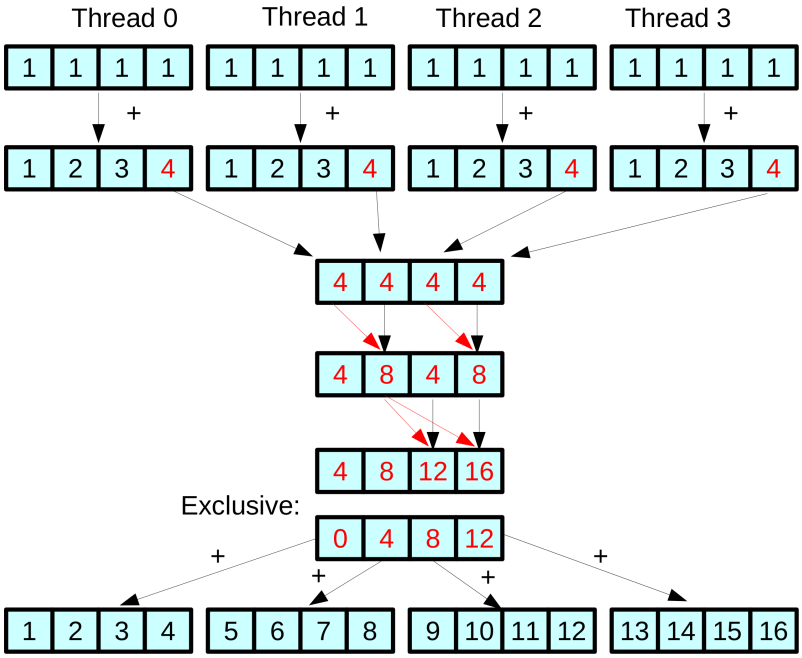


Figure 5.5: Scan computation in one warp, considering *warpSize=4*, $P = 4$ and $L_x = 4$.

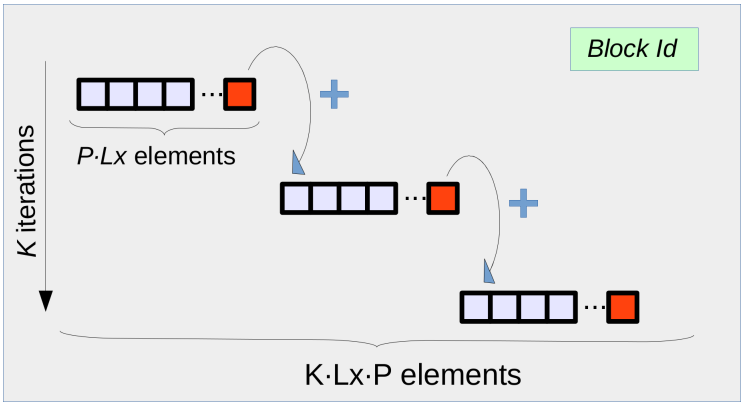


Figure 5.6: Cascade approach computation.

5.5.2. Performance Parameter Tuning: Scan-SP

The values of (s, p, l, K) parameters determine the performance of the execution in a GPU. The set of premises previously defined determines their appropriate values, which maximize execution performance.

Regarding Premise 1, and in order to balance warp and block parallelism, it is

necessary to limit the factors that decrease the SM parallelism, such as the number of registers used by each thread or the amount of shared memory required per block. Focusing on Kepler architectures, and keeping in mind Table 4.2, it is easy to see that the configuration marked as a bold row in the table maximizes both types of parallelism: work with 4 warps, less than 7168 shared memory bytes and less than 64 registers per thread.

With respect to Premise 2, the value of p must be as high as possible, without exceeding 64 registers per thread. Considering integers, each element is stored in a single 32-bit register, thus $p \leq 6$. Following Premise 2, and also considering that auxiliary variables and index calculation consume many registers, $p = 3$ is defined.

The K parameter is related with Premise 3 in our strategy. The number of elements per problem to be processed in Stage 2 is determined by B_x^1, which is the same as the number of chunks, $B_x^1 = \frac{N}{K^1 \cdot L_x^1 \cdot P^1}$, where L_x^1 and P^1 are constant values. On the one hand, K^1 must be small in order to have a large number of elements in Stage 2 and exploit GPU parallelism. On the other hand, K^1 must be large in order to have fewer chunks and reduce the number of global memory transactions (reads and writes from/to global memory auxiliary array).

Since $B_x^1 = B_x^3$, and both Stage 1 and Stage 3 use the same amount of SM resources, $K^1 = K^3$. On the other hand, as the number of elements to be processed in Stage 2 is low, and in favor of exploiting the SM block parallelism for this Stage, $K^2 = 1$, increasing the number of blocks as much as possible in Stage 2. Therefore, it is necessary to calculate the optimal value of K^1, which depends on the total number of elements being processed, $N \cdot G$. To do so, our strategy considers that the total number of threadblocks processed in Stage 2 must be greater than the maximum number of threadblocks executed per SM; i.e., 16 for Kepler architecture. As the number of elements processed in Stage 2 is $[G \cdot \frac{N}{K^1 \cdot L_x^1 \cdot P^1}]$ and each threadblock executes $P^2 \cdot L_x^2 \cdot L_y^2$ in Stage 2, then Premise 3 establishes:
$[G \cdot \frac{N}{K^1 \cdot L_x^1 \cdot P^1} / (P^2 \cdot L_x^2 \cdot L_y^2) \geq 16]$.

Note that $L = L_x \cdot L_y$, and $L_y^{1,3} = 1$. Then, K can be defined as:

$$1 \leq K^1 \leq \frac{G \cdot N}{16 \cdot P^1 \cdot P^2 \cdot L^1 \cdot L^2} \tag{5.14}$$

Thus, Equation 5.14 establishes the searching space for K^1 that seeks a trade-off between maximizing SM occupancy and minimizing global memory communications.

Premises 1, 2 and 3 determine the (s, p, l) performance parameters and trim the subspace to find the K parameter, creating the (s, p, l, K) tuple to be passed to the skeleton-based kernels in a single GPU environment. The optimal parameter K depends on the execution (G and N values) and other factors that are difficult to predict (such as the CUDA memory system management). Thus, once the (s, p, l) is determined using previous premises, all possible K values that meet Eq. 5.14 are tested.

5.6. Tridiagonal System Solver based on the Tree Partitioning Reduction

From a GPU point of view, CR and PCR algorithms are easily implemented in CUDA when the problem fits in the shared memory of one threadblock, as seen in Chapter 4. Otherwise, when the problem is bigger than the shared memory capacity, the computation needs to be divided among several threadblocks, which hinders the CUDA implementation. Regarding the Wang&Mou efficiency, this algorithm is easily partitioned among different threadblocks for solving large problem sizes, as seen above. The main drawback of this algorithm is the need to use three equations per element (triads). Although the triads can be easily generated from other equations, storing only one equation per element when the whole problem can be stored in the same memory space, when distributing the equations among different threadblocks, the elements are not stored in the same shared-memory space and it is necessary to work with (and store) whole triads, decreasing the global performance due to memory bandwidth limitations.

In this section, an efficient CUDA implementation of our Tree-Partitioning Reduction algorithm (TPR) is presented under our three-phase methodology.

5.6.1. CUDA Kernel Optimization: *TPR*

As stated in Chapter 3, the *TPR* method can be implemented in any parallel and distributed programming paradigm. In order to show its efficiency for parallel platforms, this work provides an efficient implementation for GPU accelerators, since they currently play a huge role accelerating applications.

Our CUDA implementation of *TPR* divides the execution into three stages (kernels). The first kernel, Stage 1, is responsible for performing $log_2 S + 1$ steps of the forward reduction, as Figure 5.7 represents, where each slice (sub-matrix) is computed in one threadblock. After $log_2 S + 1$ steps, the last equation of each slice uses the first equation of the next slice, thus communication among threadblocks is needed. In order to do this, a second kernel, Stage 2, is launched, working as a global synchronization barrier among threadblocks. In Stage 2, each problem is represented by as many equations as the number of slices the first stage had ($M = N/S$). Stage 2 computes the last $log_2 M$ steps of the forward reduction, and the first $log_2 M$ steps of the backward substitution. Finally, Stage 3 computes the remaining steps of the backward substitution in slices of S equations, where each slice is again solved by a threadblock, as shown in Figure 5.8. It should be noted that each slice needs the last equation of the upper slice to perform its substitutions.

Specifically, the first kernel is invoked with ($B_x = N/S,\ B_y = G$) blocks, and its pseudo-code is shown in Figure 5.9. Considering floating point single precision elements, each element is composed of four 4-byte elements, requiring 16 bytes of storage, which can be stored in a $float4$ datatype (line 3). In the case of double precision, it would be represented by a $double4$ datatype. Each thread performs a Node operator, and although each Node works with three elements, these elements are shared by two different Nodes; thus each thread loads $P = 2$ elements in its own registers and takes the third element from shared memory (lines 7-11). Please observe that there are S Node operators in the first step, but there are $L = S/P$ threads; considering $P = 2$, each thread has to compute two Node operators in the first step (lines 13-19). In the following steps, the number of Node operators shrinks exponentially; thus, it is necessary to control the thread *id* to know which threads must perform the reduction (lines 26,31). Please observe that there are cases where the Node operator only computes two elements. In these cases, the identity equation

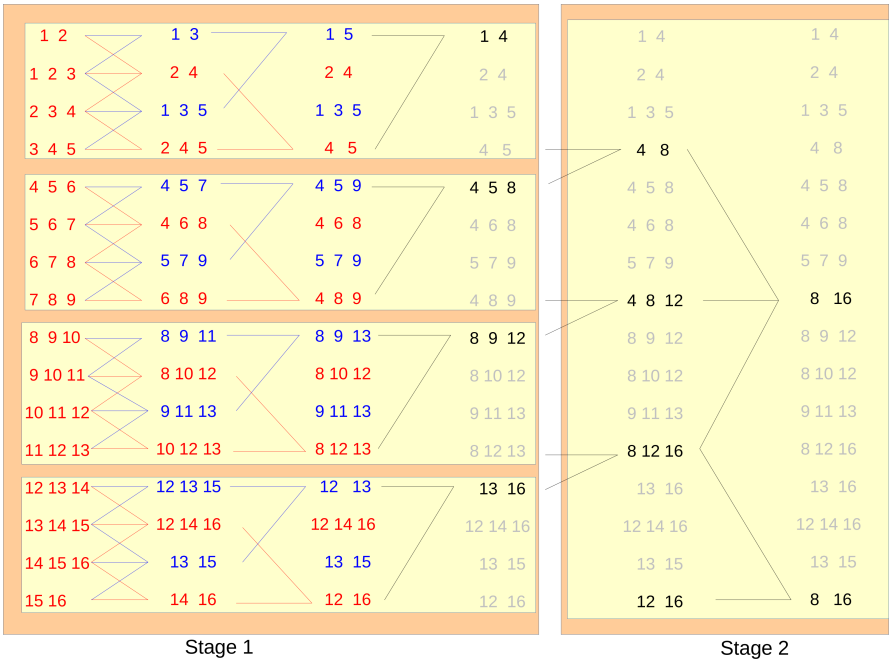


Figure 5.7: *TPR* forward reduction.

is assigned to the third element to avoid influencing in the computation and giving rise to produce branch divergence. Finally, the E_i equations with $i\%2 = 0$ are stored in global memory, overwriting their previous values (line 38), whereas the E_i equations with $i\%2 \neq 0$ remain constant in global memory. Additionally, the bottom equation of each slice is stored in an auxiliary buffer for the next stage (lines 35-36). It should be noted that the size of this buffer corresponds to G problem times $B_x - 1$ slices per problem (the first slice of each problem can skip this storage action, since its equation is not used in future steps).

To optimize the communication among threads, the pseudo-code of Figure 5.9 can be improved with the use of shuffle instructions during the last four steps. Considering $W = 32$ threads/warp in current architectures, where each thread collaborates with one element, implies four steps of enterely warp communication.

Regarding the second kernel (Stage 2), each problem needs as many elements as slices it had in the previous stage. As this number can be low, each threadblock can compute several problems. In the first step, each element from the auxiliary buffer is reduced with its corresponding equation, as Figure 5.7 (a) shows. Then, a

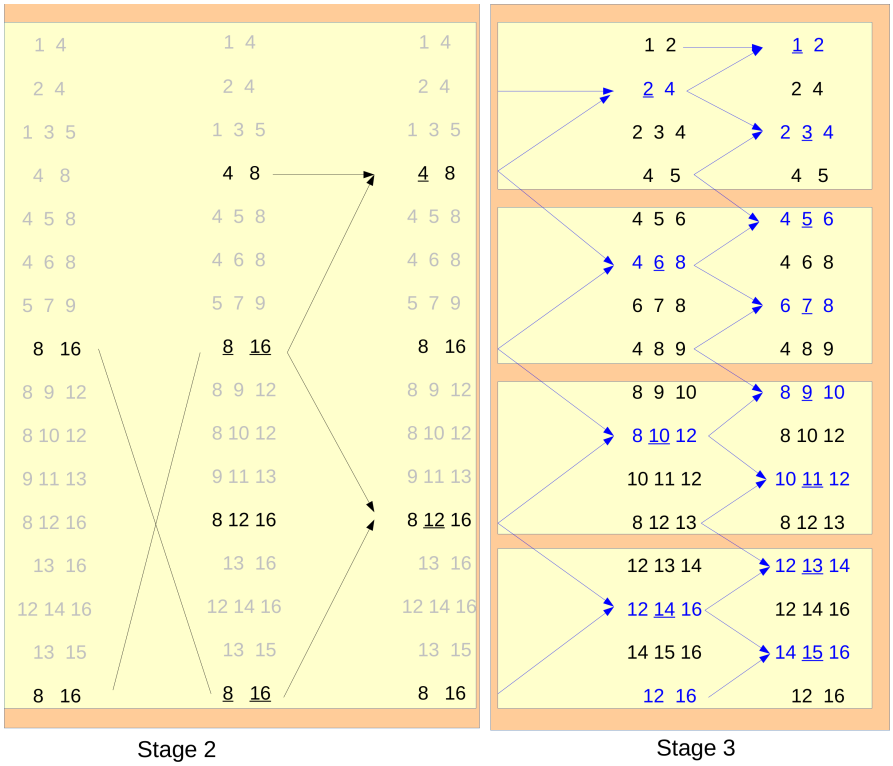


Figure 5.8: *TPR* backward substitution.

conventional reduction is applied, until the unknowns $x_{N/2}$ and x_N can be solved; after which, the backward substitution starts.

Finally, the third kernel (Stage 3) performs the remaining substitutions that did not take place in the Stage 2. The number of threadblocks and threads per block is the same as in the first kernel, dividing the problem in the same number of slices. Observing Figure 5.7 (b), each slice needs the last element of its upper slice.

5.6.2. Performance Parameter Tuning: TPR

Considering devices with compute capability 5.0, Table 5.6 shows different configurations and the corresponding parallelism achieved. The row in bold represents the configuration which maximizes both warp and block occupancy. However, it is not always possible to use this configuration, since the resource consumption limits these occupancies and it is necessary to maximize these values within the available

```
template<int N> __global__ void
BPLG_\textit{TPR}_Stage1(const float* __restrict__ src, float* bufferAux){
        Float4 reg[3];
        __shared__ Float4 shm[N];
        //Obtain id, offsets and strides
        ...
        //Load data from global mem2reg, reg2shm
        copy<2>(reg,src+strideId,...);
        copy<2>(shm+strideSHM, reg, ...);
        __syncthreads();
        copy<1>(reg+2,shm+strideSHM+offset,...);

        //First compute step
        Float4 aux[3];  //second node comp. in first step
        copy<1>(aux,shm+strideSHM-1,..);
        copy<1>(aux+1,reg,..);
        copy<1>(aux+2,reg+1,..);
        compute<2,MixStep>(reg);
        compute<2,MixStep>(aux);

        for(int accR=MixR; accR < N ; accR*=2) {
                __syncthreads();
                //Obtains strides and offsets
                ...
                //Reg-> Shm
                if(threadId<numThreads)
                        copy<1>(shm+writeOffset, reg,..);
                __syncthreads();
                numThreads/=2;
                //Shm-> Reg
                if(threadId<numThreads)
                        copy<3>(reg,shm+readOffset,..);
                        compute<2>(reg); //Computation in registers
        }
        if(threadId==1)
                copy<1>(bufferAux+offset,reg+1,..);
        copy<1>(reg+1,shm+strideSHM+1,..);
        copy<1>(src+strideId+1,reg,..);
}
```

Figure 5.9: Forward Reduction code for the *TPR* tridiagonal algorithm using BPLG.

Problem size	*Kernel 1,3*	*Kernel 2*
$n \leq 18$	$L = 64$	$L = (L_x, L_y) = (\frac{N}{S \cdot 2},\ max(1, \frac{64}{L_x}))$
$n = 19$	$L = 128$	$L = (L_x, L_y) = (\frac{N}{S \cdot 2},\ 1)$

Table 5.7: Description of tuning parameters, where $S = P \cdot L$ and $P = 2$.

resources.

As explained previously, a problem of size N is solved by partitioning the data into $M = N/S$ slices of size S. The first and third kernel solve this problem with $B_x = N/S$ threadblocks of $L = S/P$ threads, whereas the second kernel solves N/S

elements with $\frac{N}{S \cdot P}$ threads within a single threadblock. In the case of solving G problems simultaneously, $B_y = G$ is used. In order to improve the warp occupancy and, as such, performance, each threadblock of the second kernel computes L_y problems, resulting in an invocation of $B = G/L_y$ blocks.

Regarding Premise 2, the use of $P = 2$ already implies employing 40 registers per thread; thus, higher P values would consume a huge amount of registers, resulting in inefficiency. Therefore, $P = 2$ must be used, and S is expressed as $S = 2 \cdot L$. It should be noted that the configuration marked in the row in bold cannot be applied to this case due to the register consumption; thus, an alternative configuration, which maximizes the occupancy as much as possible, must be found when consuming 40 or a higher number of registers per thread.

In the case of storing the unknowns in global memory, the amount of shared memory bytes per block (floats) is calculated as $S{\cdot}4\ coef/eq{\cdot}4\ bytes = 2{\cdot}L{\cdot}4{\cdot}4\ bytes$, as each equation is composed of 4 coefficients. For the first and third kernel, looking at Table 5.6, the row of $L = 64$ threads, 40 registers per thread and up to 2560 shared memory bytes per threadblock, maximizes both the warp occupancy and the number of active thread blocks per SM, between all other possibilities that consume 40 registers or more, following Premise 1. This configuration implies solving $N/128$ elements per problem in the second kernel.

With respect to Premise 3, and in order to maximize the global performance, each threadblock works with $L = 64 = L_y \cdot L_x$ threads in the second kernel, with the following configuration $L_x = \frac{N/128}{2}$ and $L_y = max(1, \frac{64}{L_x})$. It should be noted that the case of $n = 19$ is the only one in which the value of S is higher than 128. Otherwise, the number of threads in the second kernel would result in more than 1024 threads per threadblock. In this case, S is the minimum value higher than 64 that allows the execution of the second kernel (fewer or equal to 1024 threads for current architectures). Table 5.7 summarizes the tuning parameter values for each $N = 2^n$ value.

In the case of storing the unknowns in shared memory, the amount of shared memory bytes per thread block is $S \cdot 4\ coef/eq \cdot 4\ bytes + S \cdot 4\ bytes = 2 \cdot L \cdot (4 \cdot 4 + 4)\ bytes$. Due to the register consumption, the same warp and block parallelism is achieved, as in the global memory case, thus the tuning values are the same.

5.7. Experimental Results for Parallel Prefix Algorithms with Medium-Large Problem Sizes

This section shows the performance results of our scan and *TPR* proposals under our 3-phase methodology. Table 5.8 describes the platforms employed in this analysis, Kepler architecture for the scan proposal and Maxwell architecture for the *TPR* proposal.

5.7.1. Scan Primitive

In this section, our tuning strategy's performance is compared with state-of-the-art libraries, such as *CUDPP* [98], *ModernGPU* [97], *Thrust* [101], *LightScan* [79] and *CUB* [100]. For this algorithm, we have tested our proposal in the Kepler Platform described in Table 5.8. All data elements are integers, and they were in GPUs memory prior to the GPU execution. Regarding the number of problems and their size, $N \leq 268,435,456$ ($n \leq 28$) is established. In all cases, the K^1 parameter for the given configuration is set with the value which maximizes performance. This is obtained empirically for each problem size from the search space proposed in premises, whereas the employed (s, p, l) parameters are the ones obtained in Section 5.4.1.

Figure 5.10 shows a performance comparison with respect to state-of-the-art libraries, where the number of problems solved is *G=1*. Our strategy relies on a massive parallelism for exploiting the *Streaming Multirpocessors* (SMs) of the GPU, therefore our strategy performance is not very impressive if the total number of elements being simultaneously executed is low, *G=1* in this case. Here, GPU computational power is underused, especially for Stage 2. Nonetheless, our proposal is still very competitive, being up to $1.34x$ faster than *CUDPP* and $41.19x$ against *Thrust*. It also surpasses *LightScan* for some datasets, being up to $1.39x$ faster for those cases. However, it does not outperform *CUB* and *ModernGPU*.

Figure 5.11 shows the performance achieved with *Scan-SP* when computing $G = 2^{28}/N$ batches. Although the most representative scenario of our proposal lies in solving several batches simultaneously, only *CUDPP* supports this feature

	Platform: Kepler Architecture	Platform: Maxwell Architecture
CPU	Xeon E5-2620 v2 (2.10 GHz, 6 cores) x2	Intel Core i7-2600 3.4 GHz
Memory	64 GB	8 GB
GPU	Nvidia Tesla K80	Nvidia GeForce GTX980
Driver	375.51, SDK 8.0	384.9, SDK 8.0

Table 5.8: Description of the test platform

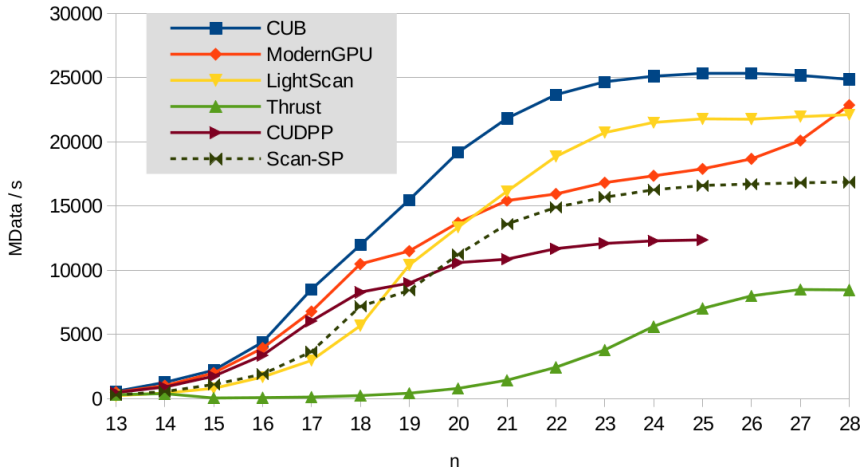


Figure 5.10: Performance analysis for the scan primitive when $G = 1$ problems.

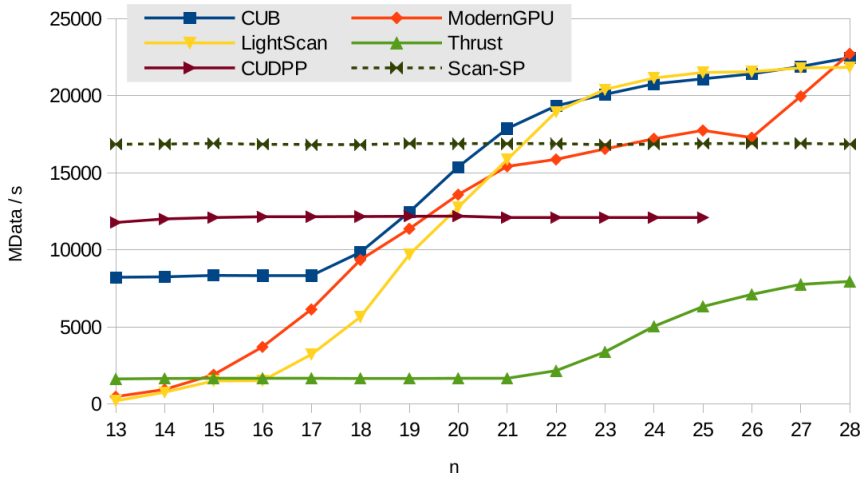


Figure 5.11: Performance analysis for the scan primitive with G problems.

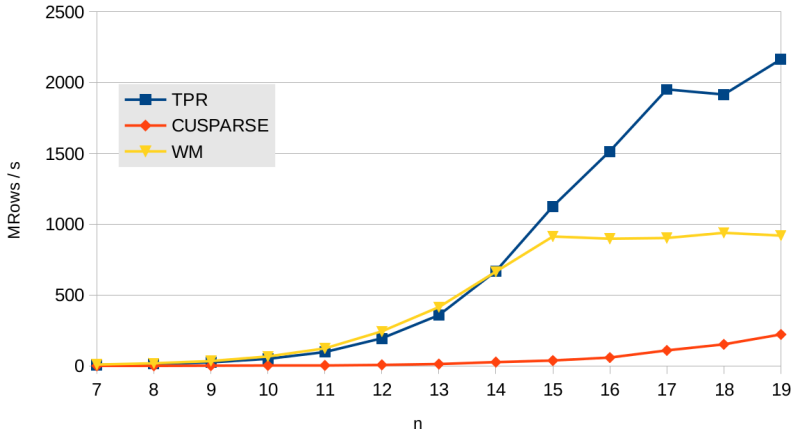
with its *multiScan* function. Thrust provides a segmented operation, but it forces the carrying of an additional flag array, reducing performance. Also, a segmented scan can be implemented with *CUB* following [112], modifying the datatype and

extending the sum operator with an additional condition. However, better performance has been obtained invoking the non-segmented function G times for $n > 21$ in the case of Thrust, and $n > 17$ in CUB. For fairness, we use the option that achieves the best performance for each data point. In the case of *ModernGPU* and *LightScan* libraries, the corresponding function is also invoked G times. Our proposal is on average $1.39x$ faster than *CUDPP*, $7.3x$ against *Thrust*, $5.11x$ with respect to *ModernGPU*, $1.31x$ faster than *CUB* and $8.87x$ against *LightScan* under such scenario. It can be observed how performance increases in *Thrust, ModernGPU, CUB* and *LightScan* libraries in line with the rise in N (increasing N implies lower G, reducing the number of invocations).

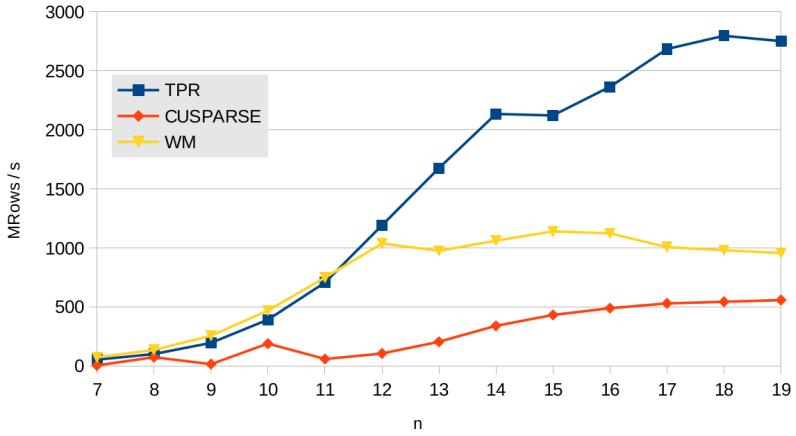
5.7.2. Tridiagonal Systems

In this subsection, a study of the performance achieved with the CUDA implementation of the *TPR* method and our 3-phase tuning methodology is analyzed and compared with other solvers for the Maxwell platform. Specifically, our proposal is compared with respect to the *CUSPARSE* [95] library and the previous Wang&Mou approach presented in Section 5.2. Here, we should stress that the performance results are measured in million rows computed per second, MROWS/s, using a diagonally dominant system which ensures numerical stability (*Toeplitz matrix* with row [-1 2 -1]). The number of batch problems being simultaneously solved in parallel, G, is studied for $G = 1$, $G = 8$ and $G = 64$, whereas the problem size range goes from $N = 128$ to $N = 524288$. Thus, the MROWS/s value is performed using the expression $N \cdot G \cdot 10^{-6}/t$.

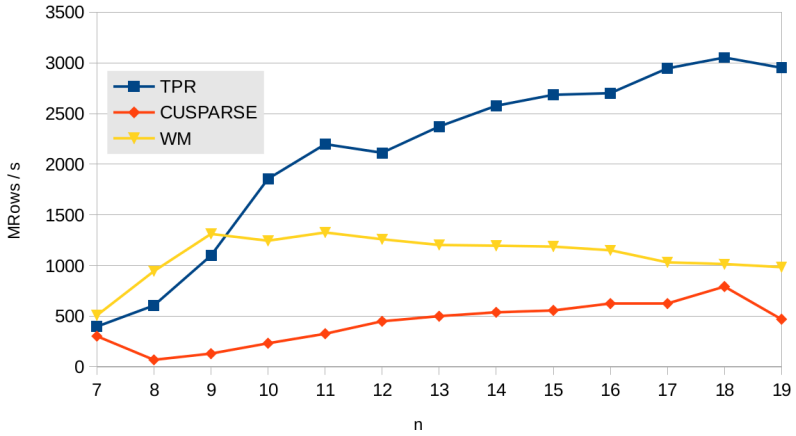
Figure 5.12 shows a global overview and a comparison with respect to *CUSPARSE* and our previous *WM* implementation. In the case of a single problem being solved ($G = 1$), Figure 5.12 (a), the *TPR* method outperforms the *CUSPARSE* library up to 30.16x for all problem sizes, being 22.03x times faster on average. Regarding *WM*, *TPR* surpasses *WM* from $N \geq 32768$ values, being 1.22x faster on average, although this speed-up is higher considering large problem sizes, being up to 2.35x in the case of $N = 524288$. As explained in Section 5.2, this Wang and Mou implementation has to store 3 equations per element for solving large problem sizes, saturating global memory bandwidth when there are many ele-



(a) When $G = 1$ problem.



(b) When $G = 8$ problems.



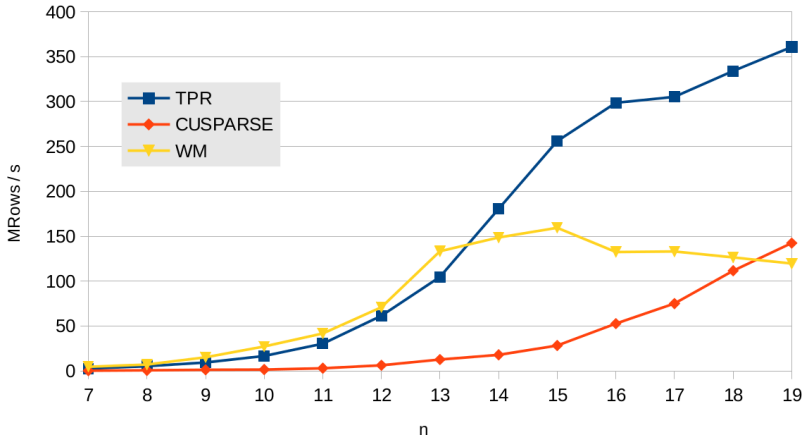
(c) When $G = 64$ problems.

Figure 5.12: Overall FP32 performance comparison of the *TPR* method

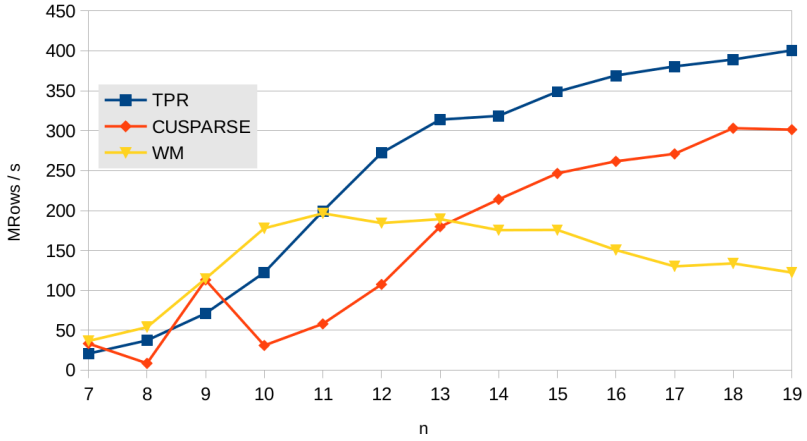
ments. Although *TPR* has more computing steps and invokes three kernels instead of two, it performs a better access to global memory and reduces the use of shared memory, being especially notable when solving large problem sizes. Figure 5.12 (b) depicts the same comparison but solving 8 problems simultaneously ($G = 8$). The *TPR* method again surpasses *CUSPARSE* by up to 13.28x for all cases, being 7.04x faster on average. With respect to *WM*, as the number of batches has been increased, there are more elements being processed, thus there are more global memory transactions in the execution, saturating the global memory bandwidth for a smaller problem size. In this case, *TPR* outperforms *WM* from $N \geq 4096$, being 1.64x times faster on average and up to 2.88x in the best case. In contrast to the previous case, performance stops increasing at $N = 524288$. This lack of scalability can be explained by two factors: firstly, as Table 5.7 shows for this problem size, L increases and the achieved GPU occupancies drop and, secondly, the global memory bandwidth is saturated. Finally, Figure 5.12 (c) depicts the case of $G = 64$ batch problems. In this case, our approach is up to 5.53x faster than *CUSPARSE* on average, and up to 8.95x in the best case. With respect to *WM*, our solver achieves $1.9x$ on average, and up to $3x$ in the case of $N = 524288$.

Additionally, Figure 5.13 depicts the same performance analysis in the case of double precision. It is easy to see how performance drops in comparison to FP32; this is due to the fact that this Maxwell platform has 128 FP32 CUDA cores but just 4 FP64 ALUS per SM, as well as the memory consumption is doubled. In the case of a single batch, our approach is 7.48x faster than *CUSPARSE* on average, and up to $10.44x$ in the case of $N = 1024$. With respect to *WM*, $1.38x$ on average and up to 3.02x. When solving $G = 8$ batches, 1.98x on average compared with *CUSPARSE*, and up to $4.42x$ in the best case. Using *WM*, the improvement is $1.7x$ on average, being up to 3.27x faster for $N = 524288$. In the case of $G = 64$, our approach is, on average, 1.93x faster than *CUSPARSE* and up to 3.41x than *WM*. It should be pointed out that the huge memory consumption of *WM* does not allow us to execute $N = 524288$ in the target architecture.

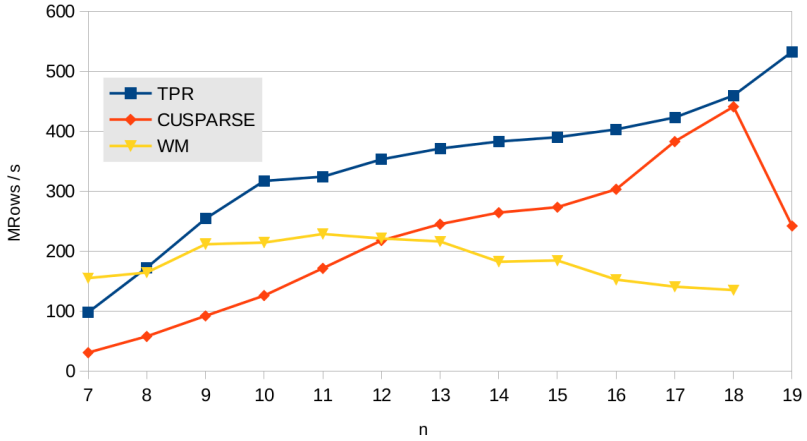
Depending on the application in which the solver is being executed, the numerical stability may be essential or may have a minor role. It is impossible to provide a general solver suitable for all the applications, some of them require solving one problem, and others need to solve several problems simultaneously. The same is



(a) When $G = 1$ problem.



(b) When $G = 8$ problems.



(c) When $G = 64$ problems.

Figure 5.13: Overall FP64 performance comparison of the *TPR* method

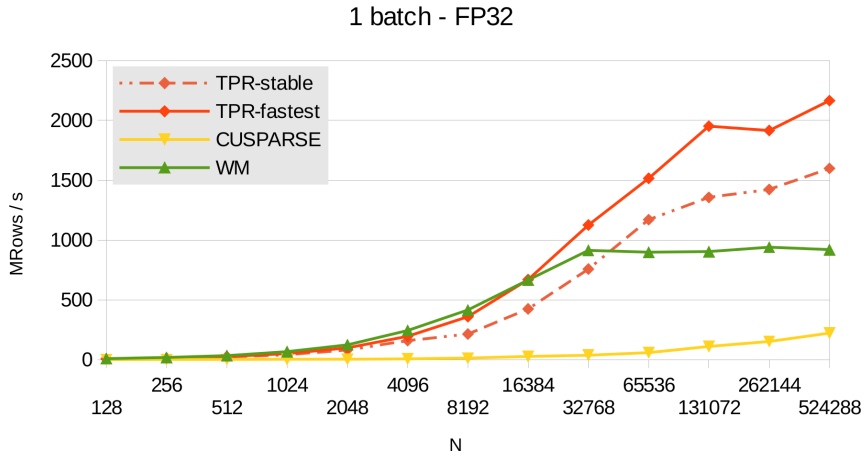


Figure 5.14: Performance comparison of two different TPR configurations: performance vs numerical stability, executing 1 batch in simple precision.

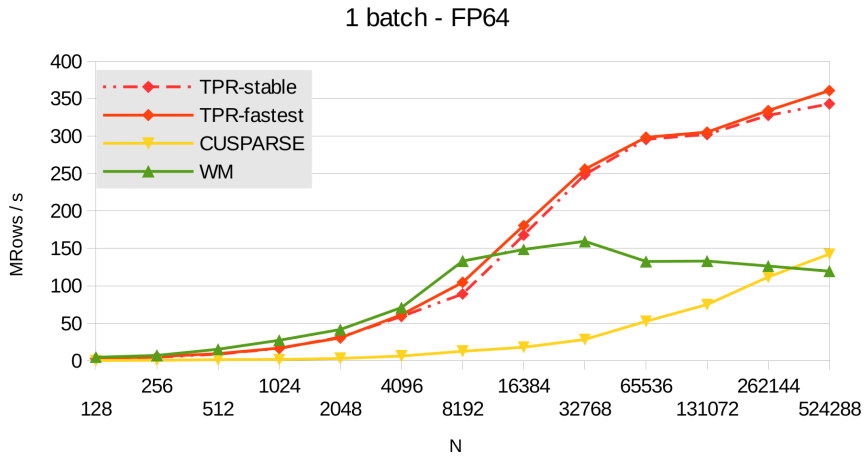


Figure 5.15: Performance comparison of two different TPR configurations: performance vs numerical stability, executing 1 batch in double precision.

true for the numerical stability and the execution time. Our proposal allows the user to choose the rate performance / stability to be employed, depending on the target application, as well as the number of problems to be solved.

The chosen slice size, S, determines the numerical stability in the TPR method. Larger slice sizes allow more equations to participate in the reduction phase of Stage 1, increasing the numerical stability. On the other hand, smaller slice sizes limit this reduction to a reduced number of equations. In the previous performance analysis, the given results are based on the performance configuration which achieves the best

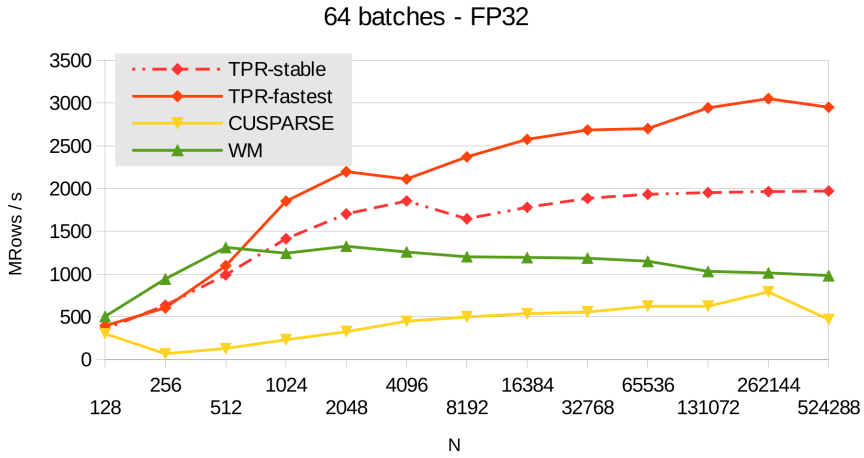


Figure 5.16: Performance comparison of two different TPR configurations: performance vs numerical stability, executing 64 batches in simple precision.

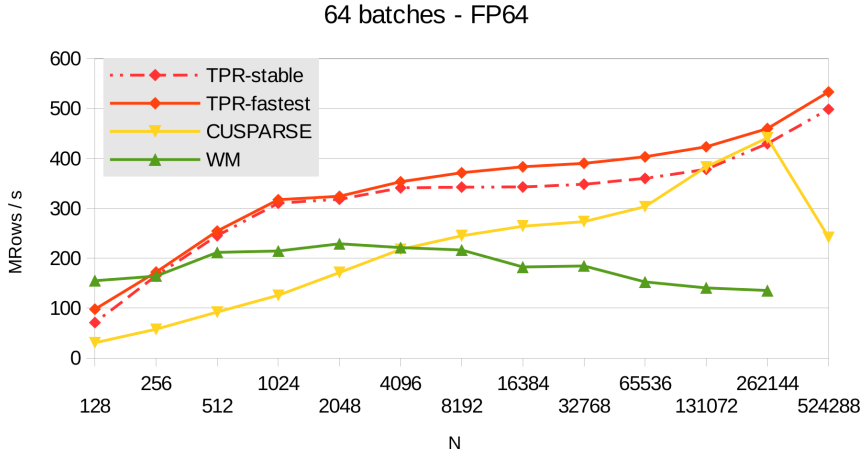


Figure 5.17: Performance comparison of two different TPR configurations: performance vs numerical stability, executing 64 batches in double precision.

execution times. However, if strong numerical stability is required, said configuration can be chosen to achieve the maximum numerical stability possible (basically by increasing the slice size). Figure 5.14 and Figure 5.15 show a performance comparison in the case of $G = 1$ in simple and double precision, respectively, for two configurations of our proposal: the one which minimizes the execution time (*TPR-fastest* in graphics) with respect to the one which maximizes the numerical stability (*TPR-stable* in graphics). Specifically, the results presented under the *TPR-stable* approach were taken using $S = 2048$. On average, the *TPR-fastest* approach obtains a speedup of 1.33x with respect to the *TPR-stable* approach. The same analysis is

N	TPR-stable	TPR-fastest	Thomas Sequential
128	5.70E-007	1.40E-006	2.10E-006
256	0.00E+000	2.20E-006	5.70E-006
512	8.40E-007	2.40E-005	4.40E-005
1024	0.00E+000	3.90E-007	2.30E-004
2048	2.00E-007	1.70E-007	9.30E-004
4096	9.90E-007	6.70E-006	1.10E-002
8192	4.00E-007	2.50E-005	3.10E-001
16384	2.00E-006	8.80E-005	1.2E+000
32768	7.40E-006	2.50E-004	2.3E+000
65536	3.00E-005	5.90E-004	3.7E+000
131072	1.20E-004	3.20E-002	5.4E+000
262144	4.80E-004	3.20E-003	7.9E+000
524288	1.90E-003	5.50E-001	1.1E+001

Table 5.9: Relative error of the two FP32-TPR configurations for a Topletz matrix

Matrix	Condition number	Description
1	4.41E+04	Each matrix entry randomly generated from a uniform distribution on [-1,1] (denoted as U(-1,1))
2	1.00E+00	A Toeplitz matrix, main diagonal is 1e8, off-diagonal elements are from U(-1,1)
3	3.52E+02	gallery('lesp',512) in Matlab: eigenvalues which are real and smoothly distributed in the inversal approximately [-2*512-3.5,-4.5]
4	2.75E+03	Each matrix entry from U(-1,1), the 256th lower diagonal element is multiplied by 1e-50
5	1.24E+04	Each main diagonal element from U(-1,1), each off-diagonal entriechosen with 50% probability either 0 or from U(-1,1)
6	1.03E+00	A Toeplitz marix, main diagonal entries are 64 and off-diagonal entries are from U(-1,1)
7	9.00E+00	inv(gallery('kms',512,0.5)) in Matlab: Inverse of a Kac-Murdock-Szego Toeplitz
8	9.87E+14	gallery('randsvd',512,1e15,2,1,1) in Matlab: A randomly generated matrix, condition number is 1e15, 1 small singular value
9	9.97E+14	gallery('randsvd',512,1e15,3,1,1) in Matlab: A randomly generated matrix, condition number is 1e15, geometrically distributed singular values
10	1.29E+15	gallery('randsvd',512,1e15,1,1,1) in Matlab: A randomly generated matrix, condition number is 1e15, 1 large singular value
11	1.01E+15	gallery('randsvd',512,1e15,4,1,1) in Matlab: A randomly generated matrix, condition number is 1e15, arithmetically distributed singular values
12	2.20E14	Each matrix entry from U(-1,1), the lower diagonal elements are multiplied by 1e-50
13	3.21E+16	gallery('dorr',512,1e-4) in Matlab: An ill-conditioned, diagonally dominant matrix
14	1.14E+67	A Toeplitz matrix, main diagonal is 1e-8, off-diagonal element are from U(-1,1)
15	6.02E+24	gallery('clement',512,0) in Matlab: All main diagonal elements are 0; eigenvalues include plus and minus 511, 509, ... ,1
16	7.1E+191	A Toeplitz matrix, main diagonal is 0, off-diagonal elements are from U=(-1,1)

Table 5.10: Matrix types used in the numerical evaluation from [74]

performed in Figure 5.16 and Figure 5.17 for $G = 64$, where *TPR-fastest* is 1.32x faster than *TPR-stable*. Table 5.9 shows the relative error of the previous configurations for the FP32 execution, when executing the Toeplitz matrix described in the introduction of this section and whose unknowns have the value 1.0 as solution.

Table 5.11 and Table 5.12 show a numerical-stability analysis for the different 16 input matrices of size 512 proposed in [74], whose description is shown in Table 5.10, in simple and double precision (using the *TPR-stable* configuration). This analysis compares the achieved stability with respect to other solvers accuracy, using the Thomas algorithm as a baseline, as in other works [74]. Although Table 5.9 shows

Matrix	TPR	WM	CUSPARSE
1	4.20E-006	8.80E-006	3.80E-006
2	2.60E-009	2.90E-009	8.60E-010
3	8.70E-008	8.20E-008	4.40E-008
4	2.80E-006	5.30E-006	2.90E-006
5	NAN	NAN	1.10E-006
6	1.60E-007	1.60E-007	4.00E-008
7	1.00E-007	9.20E-008	1.80E-007
8	7.90E-007	1.50E-006	1.80E-007
9	4.70E+005	4.70E+005	4.70E+005
10	4.40E+013	4.40E+013	4.40E+013
11	6.10E+000	2.00E-005	3.90E-006
12	4.90E-007	4.70E-007	3.80E-007
13	9.10E-001	1.10E+001	1.10E+001
14	NAN	NAN	NAN
15	NAN	NAN	NAN
16	NAN	NAN	NAN

Table 5.11: Relative errors for FP32

a poor stability for this algorithm with large-problem sizes, it is quite stable for $N = 512$. The relative error for a solution $\hat{x}$ is calculated from the following equation, where x is the solution of the baseline solver:

$$\frac{\|\hat{x} - x\|_2}{\|x\|_2} \tag{5.15}$$

It should be noted that these matrices were chosen to test the robustness of solvers, thus the accuracy of valid solutions varies greatly. In most cases, our proposal produces stable results, similar to the ones achieved by CUSPARSE. Please, observe it is not possible to compare directly these results with the ones obtained in [74], since (i) the vector d has been randomly generated, (ii) the baseline solver is a sequential version of Thomas instead of a Matlab solver, (iii) the CUDA SDK and drivers are different, and (iv) the relative error formula is slightly different.

Matrix	TPR	WM	CUSPARSE
1	1.30E-014	7.70E-015	5.20E-015
2	1.20E-016 1	.20E-016	3.90E-017
3	1.60E-016	2.40E-016	8.20E-017
4	1.00E-014	2.00E-014	6.00E-015
5	NAN	NAN	1.20E-015
6	1.30E-016	1.30E-016	9.50E-017
7	3.60E-016	2.10E-016	3.40E-016
8	4.40E-015	5.50E-015	4.30E-015
9	4.70E+005	4.70E+005	4.70E+005
10	4.40E+013	4.40E+013	4.40E+013
11	6.00E+000	1.60E-015	7.30E-015
12	7.60E-010	3.90E-016	7.80E-016
13	8.50E-001	1.20E-009	7.40E-001
14	1.00E+000	5.40E-014	8.50E-015
15	NAN	NAN	NAN
16	NAN	NAN	NAN

Table 5.12: Relative errors for FP64

5.8. Conclusions of the Chapter

In this chapter, the previous tuning methodology for small problem sizes is extended to support medium and large problem sizes; i.e., problems that do not fit in the CUDA shared memory, but still can be stored in the global memory a single GPU. It distinguishes two methodologies: one for parallel prefix algorithms, a 3-phase methodology, and another one for Index-Digit algorithm (a subset of parallel prefix algorithms), composed of two phases.

In the case of ID algorithms, the methodology has two phases: the *GPU Resources Utilization Analysis* phase and the *CUDA Kernel Optimization* phase. In the first phase, the GPU performance parameters are identified, and a set of performance premises are established. In the second phase, the algorithms are assigned to the GPU resources thanks to a compact representation of the data distribution, taking previous premises into account. Specifically, this methodology is applied to the Wang and Mou (WM) tridiagonal system solver, and tested on three different platforms. The Wang and Mou (WM) proposal (MS-ID-TS) outperforms the state-of-the-art, *CUSPARSE*, being especially competitive and achieving a speed-up of up to $33.2x$.

In the more general case of parallel prefix algorithms, the methodology is com-

posed of three phases: the *GPU Resources Utilization Analysis*, the *CUDA Kernel Optimization* and the *Performance Parameter Tuning*, which slighly varies from the ID methodology. This proposal is tested for the scan primitive using the Ladner-Fischer pattern, and the Tree-Partitioning Reduction (TPR) method for solving tridiagonal systems in a Maxwell platform. For the scan, our approach is focused on solving several batches simultaneously, as explained in the text, only surpassing the state-of-the-art for a set of problem size values. However, in the case of tridiagonal systems, our proposal surpasses both *CUSPARSE* and *WM* implementations for a single batch and multi-batch execution, being up to $30.16x$ faster. Additionally, we also present a study about the performance with double precision. Regarding numerical stability, two versions of TPR are presented: one focused on achieving the fastest performance and the other one focused on improving accuracy, both being highly competitive for the overall applications.

Chapter 6

Parallel Prefix Algorithms on Multiple-GPU systems: Dealing with Extremely Large Problem Sizes

So far, all the proposals were designed to be executed in a single GPU. However, the use of several GPUs, *multiple-GPU programming*, can be motivated by two cases: scalability and memory limits. In the first case, although the dataset can fit in the memory of a single GPU, employing several GPUs can improve throughput. The second case refers the size of dataset is larger than the memory available in a single GPU, and several GPUs are used.

In this chapter, the tuning methodology presented previously is extended to work with parallel prefix algorithms and several GPUs. Specifically, this multiple-GPU methodology is applied to two different algorithms, a scan operator and a tridiagonal system solver. This work was originally introduced in [35], where the tuning methodology was introduced to compute the scan primitive in Multiple-GPU environments, and [37], which presented the tuning methodology for solving tridiagonal systems in a Multiple-GPU platform.

6.1. A Tuning Methodology for Parallel Prefix Algorithms on Multiple-GPU Environments

As introduced in Chapter 2, there are two types of multiple-GPU environments: a Multi-GPU environment, where multiple devices are connected to the same computing node; and a Multi-Node environment with multiple GPUs per computing node and several computing nodes connected by a low-latency bus. In the Multi-GPU environment, the communication through GPUs is performed by high-speed interfaces, such as PCI-e or NVLink. In the Multi-Node environment, MPI routines are employed for communicating nodes. The previously introduced 3-phase methodology is extended in this section to cover these environments.

A typical NVIDIA Multi-Node environment is shown in Figure 6.1, where each computing node has several GPUs. The same node is composed of 4 GPUs grouped into two PCI-e networks. Each PCI-e network contains a CPU with two GPUs. The number of GPUs inside one computing node that are being used in the execution is represented by $W = 2^w$. In addition to this, $Y = 2^y$ is the number of PCI-e networks being employed in each computing node; whereas $V = 2^v$ shows the number of used GPUs connected to the same PCI-e network. Finally, $M = 2^m$ is the number of computing nodes being used. The first three parameters can be related as $w = y + v$. For example, an execution over Node 0 in figure, which uses the 4 GPUs available, implies $W = 4$, $Y = 2$, $V = 2$ and $M = 1$. Using only the GPU 0 and GPU 2 would involve $W = 2$, $Y = 2$, $V = 1$ and $M = 1$. In a Multi-Node configuration, $M = 2$ when using Node 0 and Node 1 with $W = 4$, $V = 2$ and $Y = 2$. It should be noted that W, Y, V and M values are defined by the tuning strategy, and are limited by the hardware distribution.

It should be observed that the multiple-GPU computation means solving one problem by several GPUs, but there could be partial communication among devices, or no data exchange between them. Based on this, two cases can be scheduled:

- *Batch Parallelism.* Each GPU computes a subset of G problems entirely; i.e., each problem is solved in just one GPU. Figure 6.2 depicts this distribution.
- *Problem Parallelism.* All GPUs participate in solving a portion of each problem, as shown in Figure 6.3.

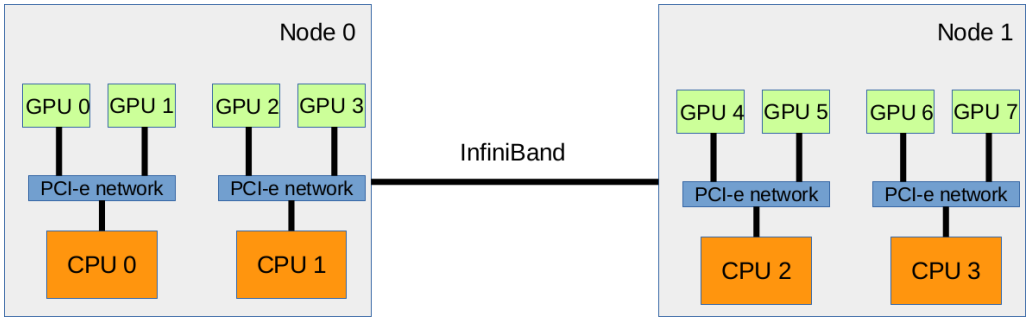


Figure 6.1: Multi-GPU topology within a Multi-Node environment.

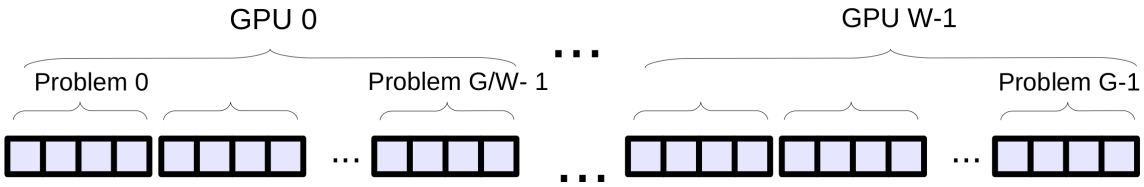


Figure 6.2: Multiple-GPU computation with no communication among GPUs

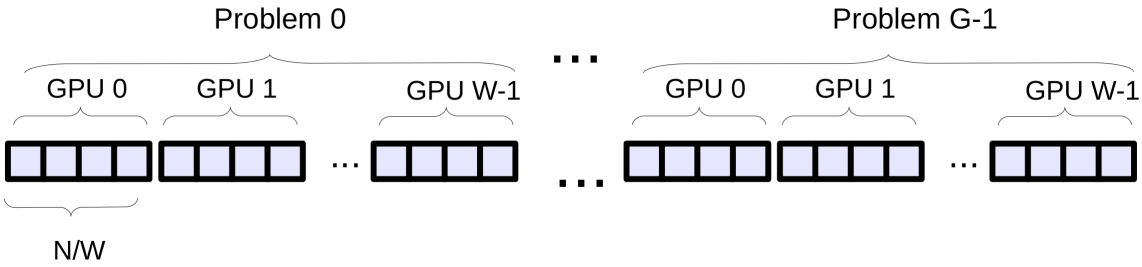


Figure 6.3: Multiple-GPU computation with communication among devices

Depending on the communication case, the performance varies hugely. The first case is trivial, as each part of the problem runs independently in each GPU; however, the second case is more challenging, since its efficiency depends on how data movement is designed among GPUs. Hence, this work covers both cases in order to demonstrate the importance of the data exchange design.

The first phase of our methodology, *the GPU Resource Utilization Analysis*, identifies new factors that influence on GPU performance in these environments. The success of our tuning methodology resides in representing the performance factors in terms of GPU resources, that can be modeled as parameters whose performance should be maximized. In the case of multiple-GPU environments, other factors, related with the hardware node configuration, influence on performance. Previous

performance parameters are extended with these new factors, as Table 6.1 shows.

The three performance premises presented in previous chapters are summarized below:

- **Premise 1**. *Balancing warp and block parallelism.*
- **Premise 2**. *Increase the computational load per thread.*
- **Premise 3**. *Maximization of SM occupancy and minimization of global memory communications.*

A forth premise for the Multiple-GPU environment is now included:

- **Premise 4**. *Prioritizing High-Bandwidth Communications.* Memory-bound problems scale very well when the number of GPUs rises; thus, the number of participating GPUs should be as high as possible. However, it is necessary to pay attention to how these GPUs are connected, since communication latency should be reduced, as well as the amount of data to be transferred from/to each GPU. This premise defines the kind of communications that should be prioritized depending on the target environment.

Premise 4 distinguishes between several scenarios depending on the problem size and the hardware distribution. If there is no communication among devices, batch parallelism, the number of participating GPUs must be maximized. Thus, W, V, M must be maximized as much as the hardware allows, but maintaining GPUs with enough data to exploit parallelism. The idea is to reduce the memory bandwidth limitation, but ensuring a good occupancy rate. If there is communication among GPUs, problem parallelism, there are two possibilities that depends on how participating GPUs are connected. On the one hand, if the participating GPUs in each problem belong to the same PCI-e network, the communication overhead is very low, since the computation is performed inside the same node and there is no communication among nodes. Hence, W, V, M must be maximized as much as the hardware allows. On the other hand, when the participating GPUs do not belong to the same PCI-e network, the computation can be distributed either along several PCI-e networks inside the same node, communicating via host memory and CUDA

Problem Parameters

$N = 2^n$	Problem size.
$G = 2^g$	Number of problems being solved simulteneously.

GPU Performance Parameters

$S = 2^s$	Number of shared-memory elements per block.
$P = 2^p$	Number of elements stored in registers per thread.
$B = 2^b$	Number of thread blocks executed per GPU, where $B = B_x \cdot B_y$.
$L = 2^l$	Number of threads that compose a block, where $L = L_x \cdot L_y$ and $S \leq P \cdot L$

Node Performance Parameters

$Y = 2^y$	Number of PCI-e networks employed per node.
$V = 2^v$	Number of GPUs being executed within the same PCI-e network.
$W = 2^w$	Number of GPUs used per node, where $W = Y \cdot V$
$M = 2^m$	Number of nodes.

Table 6.1: Description of tuning strategy parameters.

API, or across several nodes via InfiniBand and MPI. Empirically, if the amount of data is low, the communication via host memory performs better than via MPI, as MPI introduces a considerable overhead. Therefore, W, V must be maximized. Nevertheless, the computation of a huge amount of data performs better through several nodes via MPI - RDMA, since the MPI latency remains constant. Thus, W and M must be maximized.

The (s, p, l) parameters obtained previously for each case remain constant. However, Premise 3 can be slightly modified for each algorithm, taking the communication parameters into account to maximize the SM occupancy.

Regarding the second phase of our methodology, *CUDA Kernel Optimization*, the CUDA skeletons seen so far are not modified. The communication between nodes is mostly invoked in the host side. In the case of belonging to the same PCI-e, inter-GPU accesses can be performed from kernels, but thanks to the *Unified Virtual Addressing* (UVA) is transparent to CUDA skeletons, as explained previously.

When establishing the optimal values for the performance parameters in the third phase of the methodology, *Performance Parameter Tuning*, the new Premise 4 is considered to calculate the optimal value of the performance parameters for each architecture, kernel and problem size.

6.2. A Multiple-GPU Strategy for the Scan Operator

In Section 5.5 of Chapter 5, a tuning methodology for the scan operator was presented for medium and large problems in a single GPU. In addition to the performance parameters exposed in Table 6.1, the parameter K represented the number of iterations in the cascade approach for this proposal, and whose optimal value was bounded by

$$1 \leq K^1 \leq \frac{G \cdot N}{16 \cdot P^1 \cdot P^2 \cdot L^1 \cdot L^2} \tag{6.1}$$

where the superscript number identified the referred kernel.

This section presents three different approaches for the scan operator when working in multiple-GPU environments, depending on the batch parallelism or the problem parallelism case. Solving the batch parallelism case is trivial, simply executing the single-GPU strategy through several GPUs, since there is no communication among GPUs. This approach is called *Multi-GPU Batch Parallelism*. However, the problem parallelism case requires collaboration among GPUs, and is studied under the *Multi-GPU Problem Scattering* and *Multi-GPU Problem with Prioritized Communications* approaches.

6.2.1. Multi-GPU Batch Parallelism (MBP)

In this proposal, each GPU computes a subset of G problems entirely; i.e., each problem is solved in just one GPU, as Figure 6.2 represents. The first $\frac{G}{W}$ problems are solved by GPU 0, next $\frac{G}{W}$ problems by GPU 1, and so on through W GPUs. This approach is tagged as *Scan-MBP* proposals in the Section 6.3.

As each GPU processes a set of independent problems, there is no communication, or cooperation, among GPUs. In terms of performance, this case spends time on GPU communication routines, improving the global throughput. Regarding the Multi-GPU implementation, $\frac{G}{W}$ problems are assigned to each GPU, copying their data into the corresponding GPU global memory. Each GPU has a stream associ-

ated, executing the multi-stage proposal seen in Chapter 5 over its $\frac{G}{W}$ problems.

In the case of the Multi-Node environment, there is no MPI-communication instructions during execution, since there is no communication among GPU devices. The same code as the employed one in the Multi-GPU proposal is used here, running the execution through $M \cdot W$ GPUs on M nodes.

6.2.2. Multi-GPU Problem Scattering (MPS)

This approach is labeled as *Scan-MPS* proposal in the results. All GPUs participate in solving a portion of each problem, as Figure 6.3 shows for a Multi-GPU environment. Each problem is solved by W GPUs, where each GPU computes a portion of the problem ($\frac{N}{W}$ elements). If there are G problems being simultaneously solved, then each GPU works with G portions of N/W elements. This approach is bounded by GPU-communication bandwidth in most cases.

Figure 6.18 shows the schema of this approach. In Stage 1 (Chunk Reduction), the N/W elements of each problem are divided into chunks of $K^1 \cdot L_x^1 \cdot P^1$ size for each GPU, as explained in Section 5.5, performing the chunk reduction. Note that problems are now divided among W GPUs, thus the number of chunks per problem in a Multi-GPU environment is $W \cdot B_x^1$, and each chunk is computed by one block. The resulting element of each chunk is stored in an auxiliary array of $G \cdot W \cdot B_x^1$ elements. Stage 2 (Intermediate Scan) performs the scan of these values. At this point, there are two options: either a single GPU performs the auxiliary-array scan for all problems in its memory space, or several GPUs participate in this task, with each GPU performing the scan of a set of entire problems in its memory space. This depends on the node topology performance (if all GPUs are connected in the same PCI-e network or it is necessary to transfer through host memory), but empirically, executing this second kernel on a single GPU has better performance than splitting its execution into several GPUs. Finally, Stage 3 (Scan+Addition), which uses the same data distribution as Stage 1, performs the local scan over its chunks, also adding the corresponding elements from the auxiliary array to them.

In a Multi-Node environment, all GPUs participate in solving each problem, thus there is communication among $M \cdot W$ GPUs. As there are G problems simultaneously

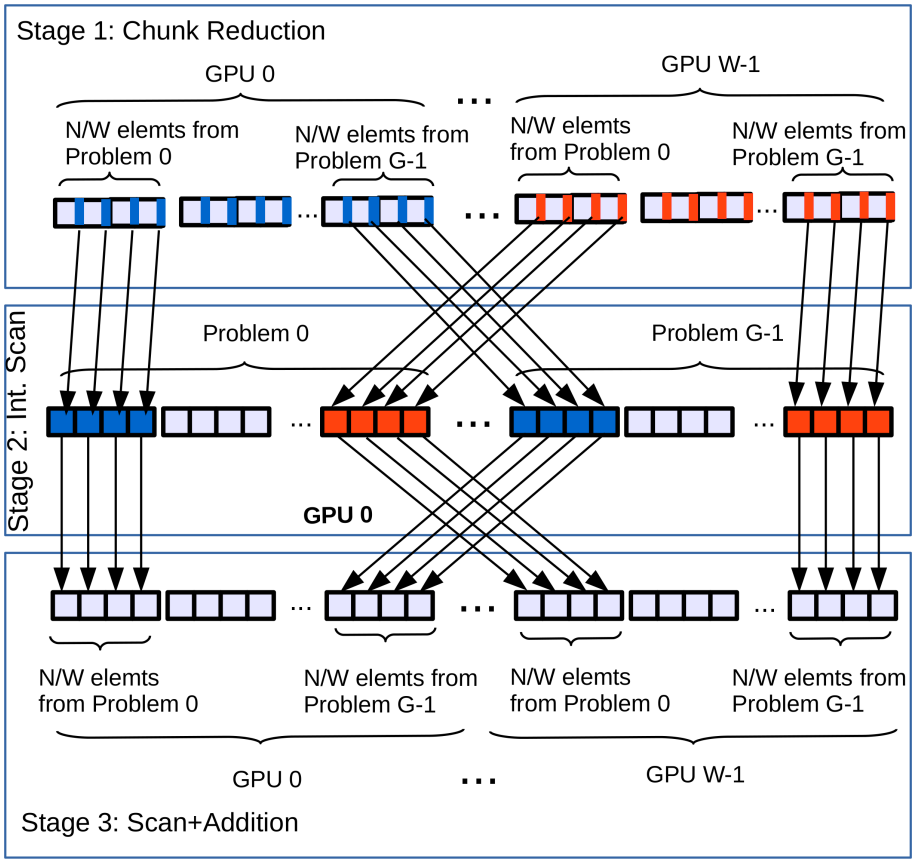


Figure 6.4: Multi-GPU Problem Scattering on a Multi-GPU environment.

executed, each GPU works with G portions of $\frac{N}{M \cdot W}$ elements, and the number of chunks per problem is $M \cdot W \cdot B_x^1$. Basically, the same approach as that shown in Figure 6.18 is used, using the code shown in Figure 6.5. Firstly, each node divides these data among its W GPUs. One GPU in the system acts as a *master process* (GPU 0), allocating an additional array for processing the second stage on its device memory. After synchronizing all MPI processes, the first stage is executed, whose goal is to calculate the chunk reductions that are passed to the second stage. To do so, these values are collected from all GPUs by the master process with an *MPI_Gather* instruction. The master process computes the second stage in its memory and returns the resulting values to the corresponding GPUs through an *MPI_Scatter* instruction. Finally, each GPU executes the third stage, and the result is collected from GPUs to the node. Please, note that data transfers in the same node are performed using the MPI API, although if they are on the same PCI-e bus,

peer-to-peer transfers are automatically used by the CUDA-aware MPI library.

6.2.3. Multi-GPU Problem with Prioritized Communications (MP-PC)

The *Multi-GPU Problem with Prioritized Communications* proposal can be considered as a sub-case of the Multi-GPU Problem Scattering proposal, where the intra-node PCI-e network communications are prioritized. This approach is tagged as *Scan-MP-PC* proposal in the results, and Figure 6.6 shows this schema for a Multi-GPU environment.

This approach basically focuses on the *Multi-GPU Problem Scattering* approach but taking advantage of the PCI-e networks within the compute node. When two GPUs in the same node are not connected to the same PCI-e network, memory transfers are performed through host memory, losing a good deal of performance. In order to minimize this loss, the work is partitioned into the GPUs which belong to the same PCI-e network. Thus, V GPUs of each PCI-e network work on G/Y problems, partitioning each problem into V portions of size N/V. Communication is only performed among the V GPUs of the same PCI-e network node, whereas other PCI-e GPUs work on their problems, as it can be seen in Figure 6.6 for $V = 2$, $W = 8$, $Y = 4$ and $G = 12$. In terms of performance, this proposal improves on the *MPS* proposal by avoiding memory copies through the host.

Regarding the Multi-Node version, each node solves several problems, and these problems are solved only by that node. Specifically, these problems are computed by V GPUs connected to the same PCI-e network. As there is no communication among nodes, just inside each node, this approach runs the same code as the Multi-GPU version, but being executed through several computing nodes. There is no MPI communication in this proposal.

6.2.4. Performance Maximization of Scan Approaches

Considering Premise 4, the desirable approach is the *Scan-MBP* proposal, as there is no communication among GPUs. However, it is not always possible to

```
1  for each GPU in Node
2    cudaMemcpyAsync(d_data[i],node_data,stream[i]);
3    cudaMalloc(scan_array);
4  if(Node=0 & GPU=0)
5    cudaMalloc(scan_array_master);
6  Reduce(d_data,scan_array,N/(W*M),stream);
7  for each GPU in Node
8    cudaStreamSynchronize(stream[i]);
9  MPI_Gather(scan_array_master, scan_array);
10 if (Node=0 & GPU=0)
11   Scan(scan_array_master,stream[0]);
12 cudaStreamSynchronize(stream[0]);
13 MPI_Scatter(scan_array, scan_array_master);
14 ScanAdd(d_data,scan_array,N/(W*M),stream);
15 for each GPU in Node
16   cudaMemcpyAsync(node_data,d_data[i],stream[i]);
17   cudaStreamSynchronize(stream[i]);
```

Figure 6.5: Pseudo-code of Scan-MPS in a Multi-Node environment.

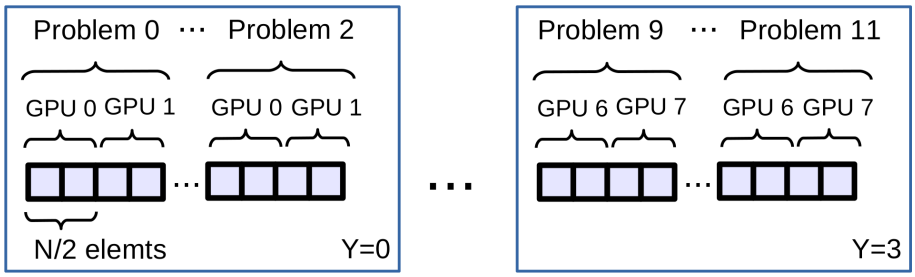


Figure 6.6: 12 problems being solved by 4 different PCI-e networks with 2 GPUs each.

use this approach, for example when each problem is larger than the memory of a single GPU. In such case, the best option is to prioritize the communication of GPUs connected to the same PCI-e, using either the *Scan-MPS* or *Scan-MP-PC* proposals. The specific optimal values for the Node Performance parameters depends on the given hardware, but following Premise 4 can be easily obtained for each case.

Irrespective of the approach employed, (s, p, l) parameters remain constant from the analysis of Section 5.5, since they maximize performance in each GPU. Nevertheless, K^1 may vary slightly as it has an indirect bearing on the performance of the other GPUs. Premise 3 justifies the fact of maximizing K^1 with Equation 6.1. With several GPUs, a large K^1 will generate a low number of chunks and it implies fewer elements to be written in GPU 0 global memory from other GPUs. Equation 6.1 from Premise 3 is extended with the following equations when working

in Multi-GPU or Multi-Node environments:

$$\frac{N}{K^1 \cdot L_x^1 \cdot P^1} \geq M \cdot W \tag{6.2}$$

$$\frac{N}{K^1 \cdot L_x^1 \cdot P^1} \geq V \tag{6.3}$$

The number of chunks, generated by splitting N into portions of $K^1 \cdot L_x^1 \cdot P^1$ elements, must be equal or greater than the number of GPUs employed in the case of MPS and MP-PC proposals, using Equation 6.2 and Equation 6.3, respectively, in order to ensure each GPU processes at least one chunk.

6.3. Experimental Results for the Scan Primitive with Extremely-Large Problem Sizes

In this section, our tuning strategy's performance is compared with state-of-the-art libraries, such as *ModernGPU* [97], *Thrust* [101], *LightScan* [79] and *CUB* [100]. The performance results were taken on the platform described in Table 6.2. All data elements are integers, and they were in GPUs memory prior to the GPU execution. In all cases, the K^1 parameter for the given (W, V, M) configuration is set with the value which maximizes performance. This is obtained empirically for each problem size from the search space proposed in premises, whereas the (s, p, l) parameters employed are those obtained in Section 5.5. Regarding the number of problems and their size, $N \leq 268,435,456$ ($n \leq 28$) with $G = 2^{28}/N$ is established for the proposals with partial communication among GPUs, *Scan-MPS* and *Scan-MP-PC*; whereas $N \leq 33,554,432$ ($n \leq 25$) with $G = 2^{26}/N$ is used for the proposal with no communication among GPUs, *Scan-MBP*.

6.3.1. Multi-GPU Environment

Performance results for the *Multi-GPU Problem Scattering* approach (*Scan-MPS*) are considered in Figure 6.7 where 2^{28} data are solved, divided into $G = 2^{28}/N$

	TSUBAME KFC Node
CPU	Xeon E5-2620 v2 (2.10 GHz, 6 cores) x2
Memory	64 GB
GPU	*4x Nvidia Tesla K80 (8 GPUs), 2 PCI-e networks*
Driver	375.51, SDK 8.0
Inter-node connection	InfiniBand FDR

Table 6.2: Description of a computing node in the test platform

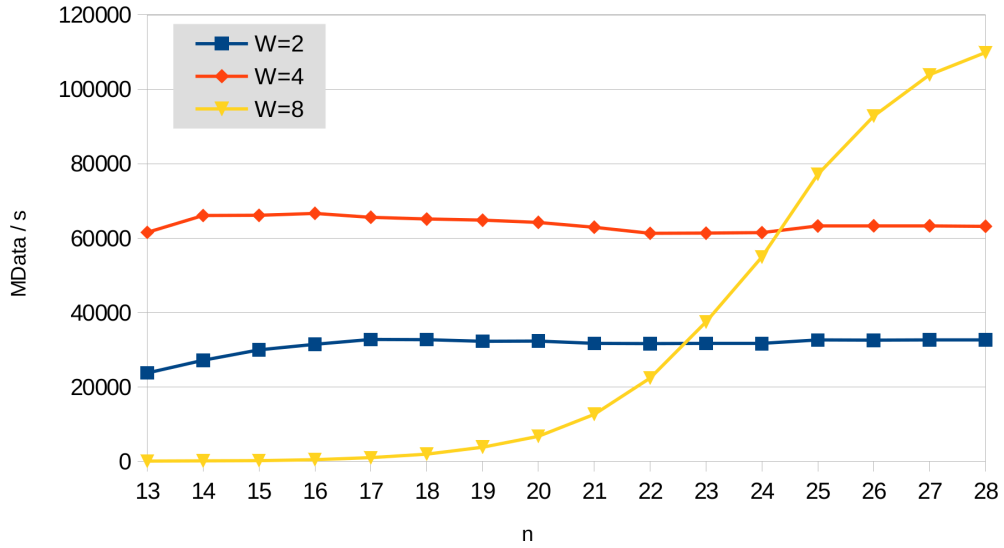


Figure 6.7: Performance analysis for the Multi-GPU Problem Scattering approach (Scan-MPS proposal) where $G = 2^{28}/N$.

batches for each $N = 2^n$ problem. It should be noted that this platform has 2 PCI-e networks, each one with 4 GPUs; thus, W can be configured as $1 \leq W \leq 8$, as well as $V \leq 4$ and $Y \leq 2$. According to Premise 4, if $W \leq 4$, then $V{=}W$ in this case, since the throughput would scale along GPUs ($W{=}2,4$) due to the absence of host memory communications. However, when $W{=}8$, host memory transactions are used, as the node configuration only allows 4 GPUs connected to the same PCI-e. This explains why performance drops so markedly when $W{=}8$: there are G problems being executed simultaneously where each auxiliary array is written by 8 GPUs through host memory. As fast as N grows and G decreases, the number of auxiliary arrays being written is also reduced, raising performance. This analysis also shows that the GPU communication penalty is very low with P2P, demonstrating that the *Multi-GPU Problem Scattering* strategy works well when N cannot be stored in a

single GPU and P2P API can be used.

Figure 6.8 depicts the *Multi-GPU Problem with Prioritized Communications* approach (*Scan-MP-PC*) with $G = 2^{28}/N$. There are communications among GPUs but performed with the P2P API, as each problem can be stored in V GPUs of the same PCI-e network. Note that having *Y=2* in each node does not make sense with only *W=2* GPUs: if they are placed in different PCI-e networks, it represents the trivial case where no communication among GPUs is involved; otherwise, if they are connected to the same PCI-e network, it is the case of *Scan-MPS* proposal. Each node has 2 PCI-e networks with 4 GPUs connected to each network; thus we propose *W=4* and *V=2* for one test, and *W=8*, and *V=4* for a second test. As each problem is solved by V GPUs, when the number of problems, G, is lower than the number of PCI-e networks, Y, the number of PCI-e being used has to be reduced. In Figure 6.8, *n=28* is not shown since it is solved by a single PCI-e network.

Figure 6.9 depicts a performance comparison with respect to state-of-the-art libraries, where the number of problems solved is *G=1*. Our strategy relies on a massive parallelism for exploiting GPU SMs; therefore, our strategy performance is not very impressive if the total number of elements being simultaneously executed is low, *G=1* in this case. It should be noted that having only 1 batch, the *Scan-MP-PC* proposal is executed on a *V=1* PCI-e network, which is the same as executing the *Scan-MPS* proposal. In this case, GPU computational power is underused, especially for Stage 2. Nonetheless, our proposal is still very competitive, being on average (averaging the speedup obtained for each data point) $1.21x$ faster than *CUDPP*, $7.8x$ against *Thrust*, $1.31x$ against *ModernGPU*, $1.31x$ with respect to *LightScan* and $1.04x$ against *CUB*. Please observe that each N is solved with the $(W, V) > 1$ parameters (attached in Figure 6.9) which achieve the best performance. Multi-GPU proposals cannot be competitive for small problem sizes when *G=1*, since the computation time is lower than the communication latency among GPUs. Our proposal running in a single GPU, called Scan Single-GPU Problem *Scan-SP*, is also shown in figure in order to compare the performance with the other libraries. As said, our proposal is focused on solving simultaneously many problems, thus the performance for the case of *G=1* is not very outstanding.

Figure 6.10 shows the performance achieved with *Scan-MP-PC*, our best proposal for $G = 2^{28}/N$ batches for each N value, as well as *Scan-SP*. A *Log10* performance

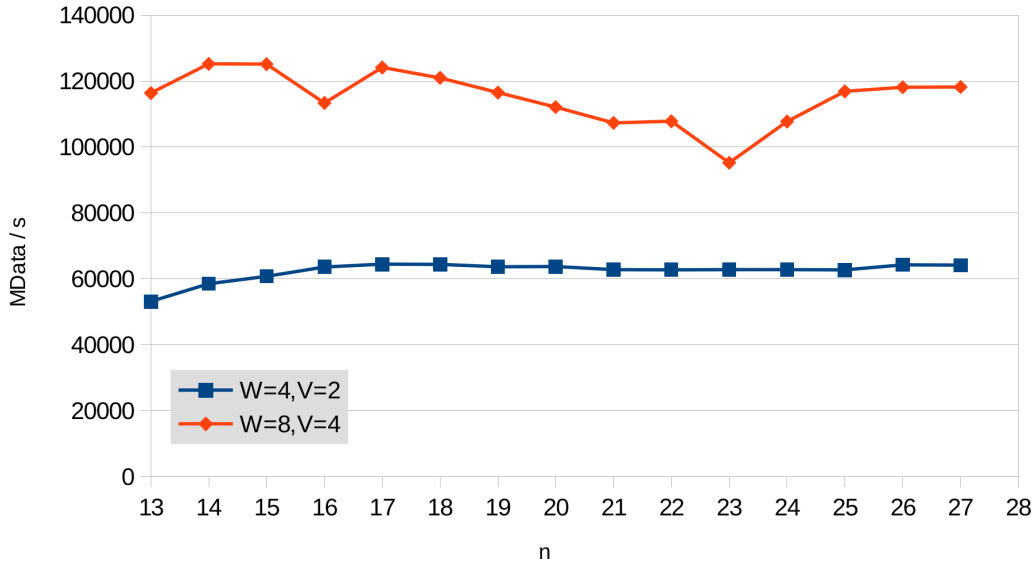


Figure 6.8: Performance analysis for the Multi-GPU Problem with Prioritized Communications approach (Scan-MP-PC proposal) where $G = 2^{28}/N$.

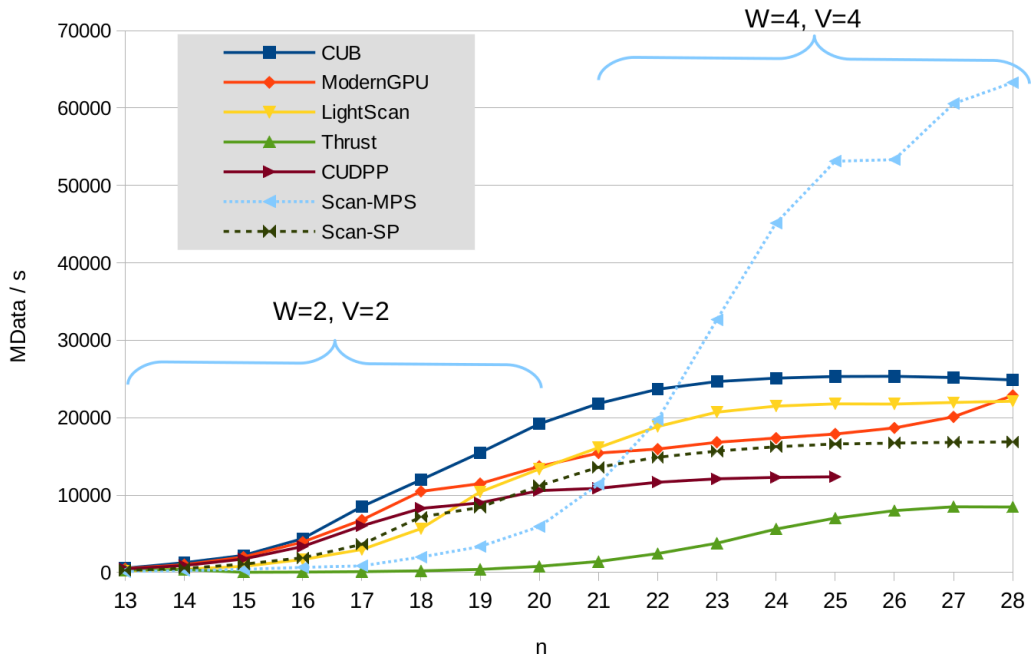


Figure 6.9: Performance analysis for our best Multi-GPU proposal when $G = 1$.

scale has been adopted for readability. Although the most representative scenario of our proposal lies in solving several batches simultaneously, only *CUDPP* supports this feature with its *multiScan* function. *Thrust* provides a segmented operation,

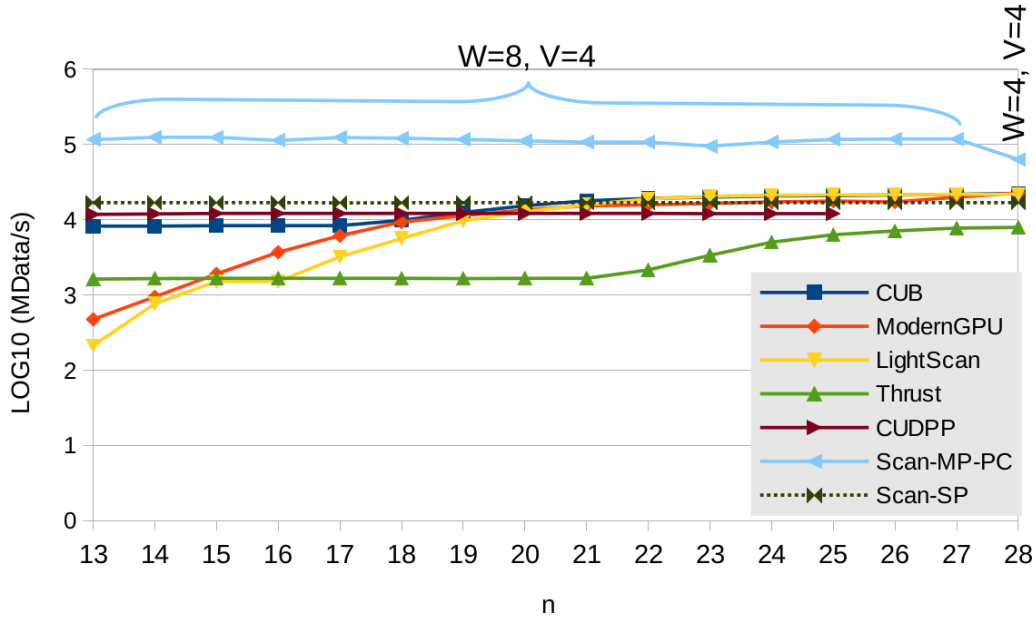


Figure 6.10: Performance analysis for our best Multi-GPU proposal when $G = 2^{28}/N$ problems.

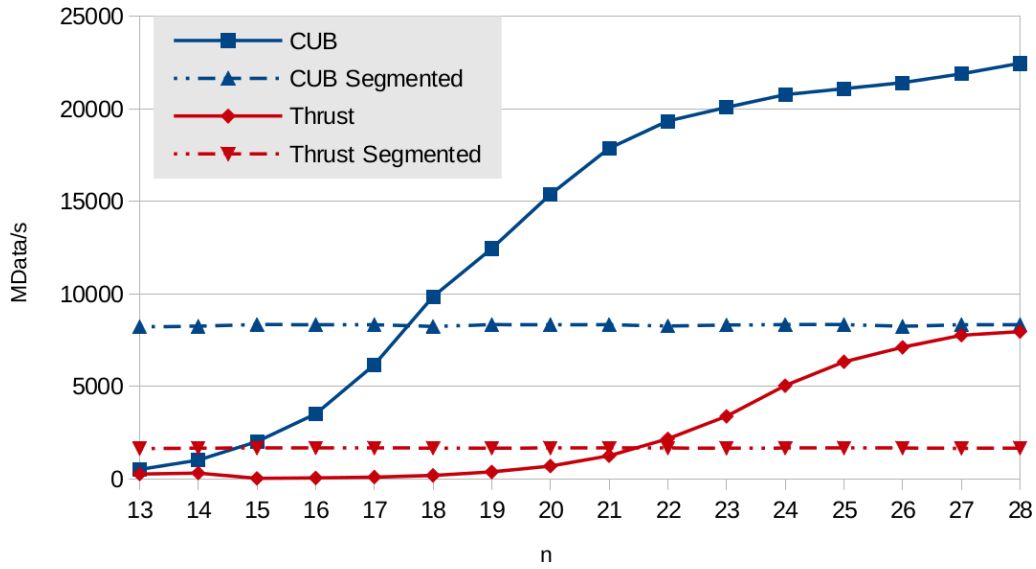


Figure 6.11: Comparison of *CUB* and *Thrust* libraries under a segmented execution when $G = 2^{28}/N$ problems.

but it forces the carrying of an additional flag array, reducing performance. Also, a segmented scan can be implemented with *CUB* following [112], modifying the datatype and extending the sum operator with an additional condition. However,

better performance has been obtained invoking the non-segmented function G times for $n > 21$ in the case of *Thrust*, and $n > 17$ in *CUB*, as Figure 6.11 depicts. For the sake of fairness, we use the option that achieves the best performance for each data point. In the case of *ModernGPU* and *LightScan* libraries, the corresponding function is also invoked G times. All competing libraries are executing in a single GPU, since none of them provides a Multi-GPU support. Our Multi-GPU proposal is on average $9.48x$ faster than *CUDPP*, $49.81x$ against *Thrust*, $33.77x$ with respect to *ModernGPU*, $8.92x$ faster than *CUB* and $58.44x$ against *LightScan* under such scenario. It can be observed how performance increases in *Thrust*, *ModernGPU*, *CUB* and *LightScan* libraries in line with the rise in N (increasing N implies lower G, reducing the number of invocations). Specifically, when *G=32768* problems with *n=13*, our proposal is $245.54x$ times faster with respect to *ModernGPU*, $71.36x$ faster than *Thrust*, $14.28x$ against *CUB* and $549.79x$ with respect to *LightScan*. However, when *G=8* and *n=25*, this speedup is decreased to $6.59x$ for *ModernGPU*, $18.5x$ for *Thrust*, $5.55x$ for *CUB* and $5.44x$ for *LightScan*. Please, note that performance drops when $n = 28$, as *G=1* and only one PCI-e network is used.

In the case of integers and the test platform described, *CUB* and *Thrust* do not solve problems larger than $n > 28$, and *CUDPP* cannot solve problems larger than $n > 26$. Only *ModernGPU* and *LightScan* support the same problem sizes as our library, up to $n = 31$. Please, observe that larger problem sizes would cause integer overflow. With fairness in mind, previous results show problem sizes up to $n = 28$ in order to represent the performance of the highest number of competitors. In the case of floating point simple precision, *CUB* and *Thrust* solve problem sizes smaller than $n < 24$, *CUDPP* for $n < 25$, and *ModernGPU* for $n < 27$; whereas our library can solve problem sizes up to $n = 30$. Additionally, thanks to the problems being distributed through several GPUs, several batch problems can be allocated in memory at the same time in our library, while most of the competitors can only allocate one single batch problem at a time for the said sizes.

Finally, Figure 6.12 shows the performance of the *Multi-GPU Batch Parallelism* approach (*Scan-MBP*) when solving 2^{25} data. Specificially, the results are taken using $W = 8$ GPUs until $n = 23$, since larger n will cause $G < W$, ensuring at least one batch per GPU. Then, $W = 4$ GPUs are employed for $n = 24$ and $W = 2$ for $n = 25$. In this approach, $n = 26$ is not shown as it will be solved by only one

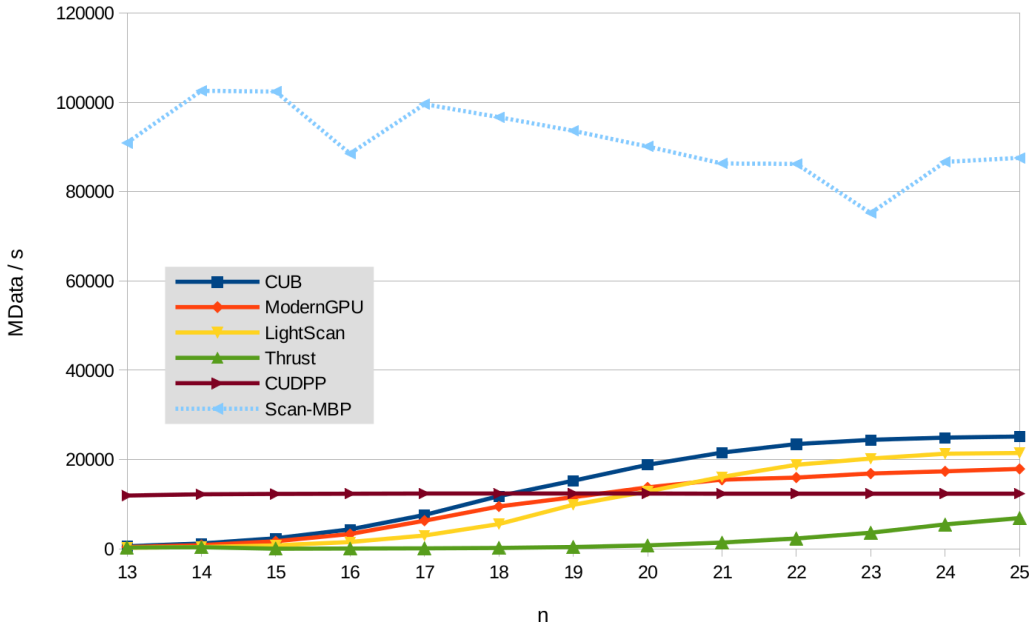


Figure 6.12: Performance analysis for the Multi-GPU Batch Parallelism approach (Scan-MBS proposal) where $G = 2^{26}/N$.

GPU, this being the case of our single-GPU multi-stage proposal; thus, a $G = 1$ analysis makes no sense in this approach. This proposal is up to $160x$ times faster for small sizes with respect to *CUB*, up to $224x$ against *ModernGPU*, up to $4265x$ over *Thrust*, up to $8.4x$ with respect to *CUB* and up to $251x$ over *LightScan*, all of them in a single GPU. Considering all sizes, on average, the speed-up achieved is $28x$, $38x$, $661x$, $7.42x$ and $75x$, respectively.

In both *Scan-MPS* and *Scan-MP-PC*, there is communication among GPUs, since each problem is solved by several GPUs. Thanks to this GPU collaboration, it is possible to solve large problem sizes. Partitioning the problem in several GPUs, *Scan-MBP*, where each problem is solved by a single GPU, increases the performance greatly since GPU communication overhead is avoided, although it is less challenging. Our approaches based on communication allow us to solve larger problem sizes, representing a novelty with respect to the state-of-the-art, but this feature is complemented with the non-communication approach to solve many shorter problems sizes simultaneously.

6.3.2. Multi-Node Environment

The performance when involving several computing nodes, where there is no communication among them, can be easily predicted. However in this environment, the *Multi-GPU Problem Scattering* proposal performs inter-node communications through MPI instructions, adding extra complexity to our model as well as new latency that affect to global performance. OpenMPI 1.8.5 with CUDA-aware and RDMA support are employed, and GPUs are connected through the same PCI-e network inside the computing node. In this section, the *Multi-GPU Batch Parallelism* approach is not included, since there is no communication among nodes and the same performance as the one obtained in the Multi-GPU environment is achieved.

Different combinations of M and W can be used to compute the scan in a Multi-Node environment. Depending on the amount of data to be processed, the correct choice is key to obtaining the maximum performance. For example, in the case of using 8 GPUs in total, there are several M, W possible combinations ($M \times W = 8$). In our experiments, the best performance is achieved with $M = 2, W = 4$, obtaining the same performance results as $M = 4, W = 2$ at high N sizes, whereas $M = 8, W = 1$ obtains the worst results. This is due to the fact that MPI communications introduce an additional overhead in execution, thus the strategy would be to minimize the number of computing nodes as far as possible, maximizing the use of GPUs connected to the same PCI-e network in each node. However, as soon as the amount of data grows, the performance difference among different combinations is reduced. In the case of 2^{13} elements being solved per problem, the configuration $M = 2, W = 4$ is $1.48x$ faster than the configuration $M = 8, W = 1$, whereas in the case of 2^{28} elements, this speed-up is only $1.03x$. This is due to the fact that, empirically, the MPI overhead is almost constant in spite of the amount of data, while GPU computation time is proportional to data size. It should also be noted that K^1 is a factor that has a bearing on global performance and must be small enough to have at least as many chunks as GPUs, $\frac{N}{K^1 \cdot L_x^1 \cdot P^1} \geq M \cdot W$.

Figure 6.13 depicts a performance study of our best Multi-Node proposal in comparison to state-of-the-art libraries for the *Multi-GPU Problem Scattering* approach, outperforming all of them. On average, it is $8.51x$ faster than *CUDPP*,

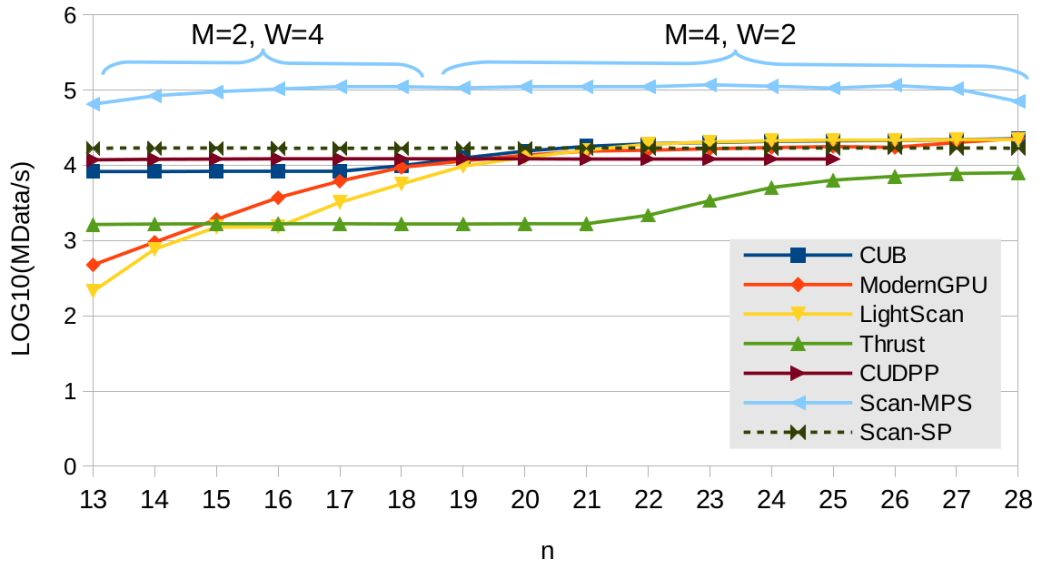


Figure 6.13: Performance analysis for our best Multi-Node proposal for $G = 2^{28}/N$ problems.

$43.82x$ against *Thrust*, $24.85x$ in comparison to *ModernGPU*, $7.7x$ with respect to *CUB* and $41.2x$ for *LightScan*, when simultaneously solving $G = 2^{28}/N$ problems. In the case of low N, these speedups are greater for the libraries with no batch support: $50.37x$ for *Thrust*, $88.31x$ for *ModernGPU*, $10.13x$ for *CUB* and $109.12x$ for *LightScan* in the case of $n = 14$. However, they are smaller for high N values, since the number of memory transactions decreases with G: $8.85x$ for *Thrust*, $3.1x$ for *ModernGPU*, $3.13x$ for *CUB* and $3.22x$ for *LightScan* in the case of $n = 28$.

Finally, Figure 6.14 shows a breakdown of times spent on each problem size for $M = 2$ computing nodes of $W = 4$ GPUs each, executing 2^{28} elements split into $G = 2^{28}/N$ batches for each problem size. Since MPI communications are introduced in the execution, there is an additional overhead which remains almost constant independently of the amount of data to be processed. MPI barriers sometimes increase their time, as they are blocking collectives; thus the time of the collective in each MPI process also depends on how long the process has waited for the others. Note that the time spent on *MPI_Gather* and *MPI_Scatter* collectives is reduced when G is also decreased, since the number of elements to be processed by Stage 2 is also lessened and the MPI collectives work with fewer elements.

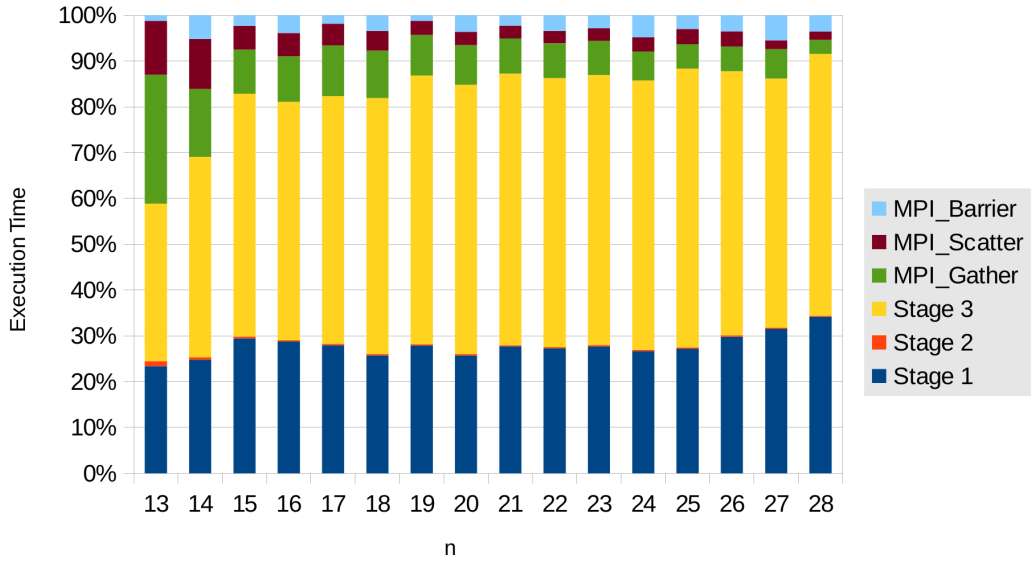


Figure 6.14: Breakdown of times spent on M=2 and W=4 for $G = 2^{28}/N$ problems.

6.4. A Multiple-GPU Strategy for Index-Digit Algorithms on Multiple-GPU Environments

The goal of this section is to extend the 2-phase tuning methodology for ID-algorithms, based on previous good results, in order to tackle large problem sizes. In favor of demonstrating the proposal's efficiency, our approach has been analyzed for a tridiagonal system solver. Specifically, the Wang&Mou solver developed in the Chapter 5 for a single GPU is used here.

6.4.1. A Two-phase Tuning Methodology

As a reminder, the ID methodology developed so far was composed of two phases. In the first phase, the *GPU Resources Utilization Analysis*, the main factors that influence on the GPU performance are identified and a set of theoretical performance presmises are established. Then, during the *CUDA Kernel Optimization* phase, the kernels are modeled with *string operators* and *mapping vectors*. They are also built with CUDA skeletons, and the suitable values are obtained for the performance parameters found out in previous phase and sent to each kernel.

Problem Parameters

$N = r^n$	Problem size.
$G = r^g$	Number of problems being solved simulteneously.

GPU Performance Parameters

$S = r^s$	Number of shared-memory elements per block.
$P = r^p$	Number of elements stored in registers per thread.
$B = r^b$	Number of thread blocks executed per GPU, where $B = B_x \cdot B_y$.
$L = r^l$	Number of threads that compose a block, where $L = L_x \cdot L_y$ and $S \leq P \cdot L$

Node Performance Parameters

$Y = r^y$	Number of PCI-e networks employed per node.
$V = r^v$	Number of GPUs being executed within the same PCI-e network.
$W = r^w$	Number of GPUs used per node, where $W = Y \cdot V$
$M = r^m$	Number of nodes.

Table 6.3: Description of the performance parameters for ID-algorithms in Multiple-GPU.

Table 6.3 shows the performance parameters identfied in the *GPU Resources Utilization Analysis* phase of the multiple-GPU methodology. All of them have already been explained but, in contrast to general parallel prefix algorithms (Table 6.1), they are built in base of a generic r radix, as mentioned previously. As a novelty, when working with several GPUs, our proposal uses $W = r^w$ GPUs within a node. This node has $Y = r^y$ PCIe root networks, and each PCIe network has $V = r^v$ GPUs connected (enabling P2P access among these GPUs), so $w = y + v$.

The theoretical performance premises identified in the Chapter 5 were:

- **Premise 1.** *The minimization of the number of stages.*
- **Premise 2.** *Balancing warp and block parallelism.*
- **Premise 3.** *Increasing the computational load per thread.*

As was the case in Section 6.1 with general parallel prefix algorithms, it is necessary to introduce a new premise related with the Multiple-GPU communication. To simplify the methodology, Premise 4 is the same as the one explained in Section 6.1.

- **Premise 4**. *Prioritizing High-Bandwidth Communications.* The number of participating GPUs should be as high as possible; but paying attention to how

these GPUs are connected. This premise defines the kind of communications that should be prioritized depending on the target environment, prioritazing the approaches with no communication among devices. In case of needing communication, the low-latency communication mechanishms have to be selected.

The second phase, *CUDA Kernel Optimization*, employs the same CUDA skeletons as previously, but extending the arguments with the new performance parameters related with the communication hardware. Also, the *mapping vectors* slightly vary for each algorithm, introducing the new communication parameters, as will be seen in Section 6.5.

6.5. A Multiple-GPU Strategy for a Tridiagonal System Solver

The use of several GPUs can be caused due to two reasons: (Case 1) Each problem can be stored in a single GPU memory, but performance scales very well when using several GPUs to solve several independent problems; and (Case 2) data are distributed among several GPUs due to memory space limitations. Below, the *Batch Parallelism* and the *Problem Parallelism* cases are analyzed for the Wang&Mou tridiagonal system solver in both a Multi-GPU and a Multi-Node environment. The *Multi-GPU Batch Parallelism* (MBP) approach represents the case of Batch Parallelism; whereas the *Multi-GPU Problem Scattering* (MPS) and *Multi-GPU Problem with Prioritized Communications* (MP-PC) represent the case of Problem Parallelism.

6.5.1. Multi-GPU Batch Parallelism (MBP)

In the *Multi-GPU Batch Parallelism* approach, each GPU processes $\frac{G}{W}$ entire problems, as Figure 6.15 depicts. As each GPU processes a set of independent problems, there is no communication, or cooperation, among GPUs. In terms of performance, this case spends no time on GPU communication routines, improving

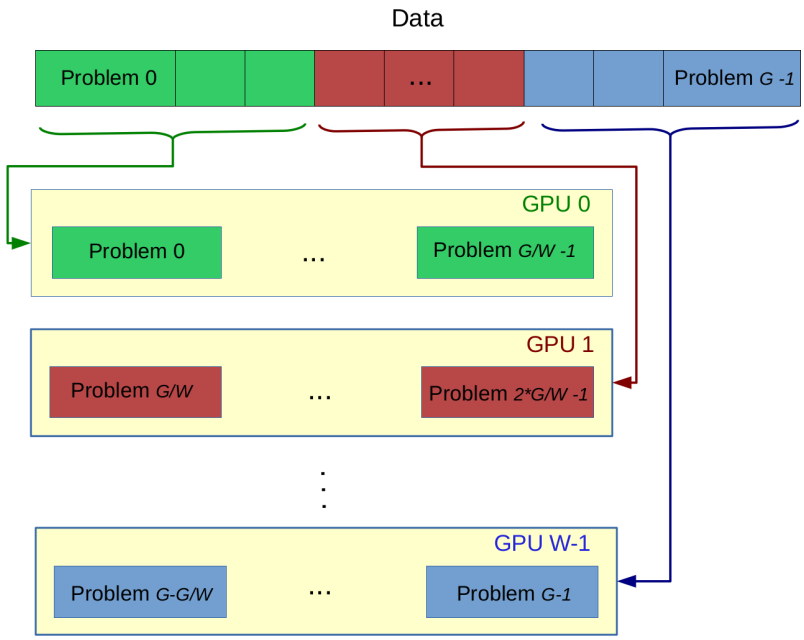


Figure 6.15: Multi-GPU approach with W GPUs for solving G problems of N elements: Each GPU solves G/W entire problems of N elements.

the global throughput.

The computation can be performed in a Multi-GPU environment or in a Multi-Node environment. It should be noted that there are G problems of N elements being solved; i.e. $G \times N$ data. In the Multi-GPU environment, G problems are distributed among W GPUs, and each problem is solved by a single GPU, thus there is no communication (or synchronization points) among GPUs. Specifically, G/W problems are independently computed by each GPU in two stages, launching two kernels for that. It should be noted that launching several kernels is due to synchronizing threadblocks. Firstly, threadblocks of each GPU write their partial results into its global memory. Then, the second kernel is launched, acting as a synchronization among threadblocks of that GPU. The new kernel threadblocks build the final result using previous kernel's data from its global memory. Despite raising global memory requirements, if data exchanges are properly optimized and the workload is properly balanced among the GPU resources, the latency generated by the communication between kernels via global memory can be efficiently hidden.

In the CUDA adaption of the Wang&Mou proposal, as explained previously, each

```
for each i in GPUs
{
  cudaSetDevice(i);
  cudaMalloc((void**) &d_data[i],..);
  cudaMalloc((void**) &triads[i],..);
  cudaStreamCreate(&stream[i]));
  cudaMemcpyAsync(d_data[i],&h_data[i*stride],..,stream[i]);
}
for each i in GPUs
{
  cudaSetDevice(i);
  cudaStreamSynchronize(stream[i]); //Timing On
}
for each i in GPUs
{
  cudaSetDevice(i);
  WM(d_data[i],size, triads, stream[i]);
}
for each i in GPUs
{
  cudaSetDevice(i);
  cudaStreamSynchronize(stream[i]); //Timing Off
}
for each i in GPUs
{
  cudaSetDevice(i);
  cudaMemcpyAsync(h_data[i*stride],d_data[i],..,stream[i]);
}
...
for each i in GPUs
{
  cudaSetDevice(i);
  cudaStreamSynchronize(stream[i]);
  cudaStreamDestroy(stream[i]);
  cudaFree(...);
}
```

Figure 6.16: Pseudocode for the MBP invocation in the Multi-GPU approach

element that takes part in the Node operator is a triad of equations, i.e. 3×16 bytes in the case of floats, a huge consumption compared to other algorithms. However, there is one property when dealing with adjacent equations that allows us to store the central equation per element, as the two others are easily obtained from adjacent equations. Hence, each element can be implemented in the first stage as a *float4* data type, just 16 bytes. The N elements of each problem are partitioned into chunks of $S = P \cdot L$ elements each. It should be observed that the adjacency property only arises in the first stage of the algorithm, where adjacent equations are stored in a common memory space; whereas in the second stage, triads need to be stored for each equation, since the central equations used for calculating the right- and left-hand equations might be placed into another memory space. Thus, shared memory

```
int rank, nprocs;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
for each GPU i in Node
{
  cudaSetDevice(i);
  cudaMalloc((void**) &d_data[i],..);
  cudaMalloc((void**) &triads[i],..);
  cudaStreamCreate(&stream[i]));
  cudaMemcpyAsync(d_data[i],&h_data[i*stride],..,stream[i]);
}
MPI_Barrier(MPI_COMM_WORLD); //Timing On
for each GPU i in Node
{
  cudaSetDevice(i);
  cudaStreamSynchronize(stream[i]);
}
for each GPU i in Node
{
  cudaSetDevice(i);
  WM(d_data[i],size, triads, stream[i]);
}
for each GPU i in Node
{
  cudaSetDevice(i);
  cudaStreamSynchronize(stream[i]);
}
MPI_Barrier(MPI_COMM_WORLD); //Timing Off
for each GPU i in Node
{
  cudaSetDevice(i);
  cudaMemcpyAsync(h_data[i*stride],d_data[i],..,stream[i]);
}
...
for each GPU i in Node
{
  cudaSetDevice(i);
  cudaStreamSynchronize(stream[i]);
  cudaStreamDestroy(stream[i]);
  cudaFree(...);
}
MPI_Finalize();
```

Figure 6.17: Pseudocode for the MBP invocation in the Multi-Node approach

use is doubled in the second stage, reducing its SM occupancy. As the second stage needs the whole triads, these triads are exchanged from the first stage to the second one via global memory. As first kernel makes a better use of shared memory, having as many steps as possible in first kernel will increment global performance. Additionally, the number of steps taken is $log_r N$. Thus, increasing r, i.e. increasing P, reduces the number of steps, involving fewer shared memory communications and synchronization barriers. It should be noted that increasing P also implies raising

the number of registers used per thread, leading in poor performance if the use is too high.

Regarding the Multi-GPU environment, Figure 6.16 shows how to invoke kernels with several GPUs in the Multi-GPU approach. Device memory buffers are allocated for each GPU, initializing the corresponding values from host memory, and creating one stream for each GPU. The use of streams enables asynchronous memory transfers as well as synchronizing GPUs. Each stream executes G/W problems with the two-stage strategy explained above. Finally, both GPU buffers and streams are released for each GPU.

The Multi-GPU environment is limited by the number of GPUs in the computing node, and by the memory of the single node. When the execution needs either more GPUs or more memory, several nodes have to participate in a Multi-Node environment, thus data are distributed among $M \cdot W$ GPUs; i.e., M computing nodes with W GPUs each node. In this case, the communication among nodes and the data distribution is performed using MPI routines, as Figure 6.17 shows. Firstly, MPI processes are created and data structures are allocated and initialized for each GPU in each node. G/M problems are distributed among the M nodes, and then these G/M problems are partitioned among W GPUs in each node. To measure the execution time, nodes and GPUs have to be synchronized, thus the MPI barrier is first executed, and then the GPUs of each node synchronize their streams. Each GPU in the node executes $G/(M \cdot W)$ problems, and then, data are gathered from GPUs. Finally, MPI and CUDA resources are released.

Please observe that the number of problems being solved, G, must be equal to or greater than the number of GPUs employed in the execution. This means, greater than W in the case of the Multi-GPU approach, and greater than $M \cdot W$ in the case of the Multi-Node approach.

Please note that data are distributed in a different way, thus their mapping vector varies. In the Multi-GPU environment:

$$\begin{array}{l} [\ \underbrace{t_{batch+n}\cdots t_{b_y+n+1}}_{w}\ \underbrace{t_{b_y+n}\cdots t_{n+1}}_{b_y} \\ \overbrace{\underbrace{t_n\cdots t_{l_y+l_x+p+1}}_{b_x}\ \underbrace{t_{l_y+l_x+p}\cdots t_{l_x+p+1}}_{l_y}\ \underbrace{t_{l_x+p}\cdots t_{p+1}}_{l_x}\ \underbrace{t_p\cdots t_1}_{p}}^{n}\ ] \end{array} \tag{6.4}$$

whereas the Equation 6.4 is represented in the Multi-Node environment as follows:

$$\begin{array}{l} [\ \underbrace{t_{batch+n}\cdots t_{b_y+n+w+1}}_{m}\underbrace{t_{b_y+n+w}\cdots t_{b_y+n+1}}_{w}\ \underbrace{t_{b_y+n}\cdots t_{n+1}}_{b_y} \\ \overbrace{\underbrace{t_n\cdots t_{l_y+l_x+p+1}}_{b_x}\ \underbrace{t_{l_y+l_x+p}\cdots t_1}_{s}}^{n}\ ] \end{array} \tag{6.5}$$

6.5.2. Multi-GPU Problem Scattering (MPS)

In the *Multi-GPU Problem Scattering* approach, each problem is solved by all the GPUs available. Specifically, in a Multi-GPU environment, each problem is solved by W GPUs, where each GPU computes a portion of the problem ($\frac{N}{W}$ elements). If there are G problems being simultaneously solved, then each GPU works with G chunks of N/W elements as Figure 6.18 shows. In terms of performance, this approach is bounded by GPU-communication bandwidth in most cases.

In this case, the code for invoking kernels involves synchronization among GPUs, as Figure 6.19 shows. In the first kernel, each GPU reads and writes from/to its buffers. However, in the second kernel, each GPU reads triads from any buffer, thus the computation among GPUs needs to be synchronized to ensure data coherence. Thanks to the UVA, there is no need to specify the memory space source of each buffer, just working with the variable names. Additionally, if the GPUs are not connected to the same PCI-e bus, it is not possible to use the P2P API; i.e., GPUs cannot directly access to other GPU buffers. In that case, before invoking the second kernel, it would be necessary that each GPU copies the triads to be used to

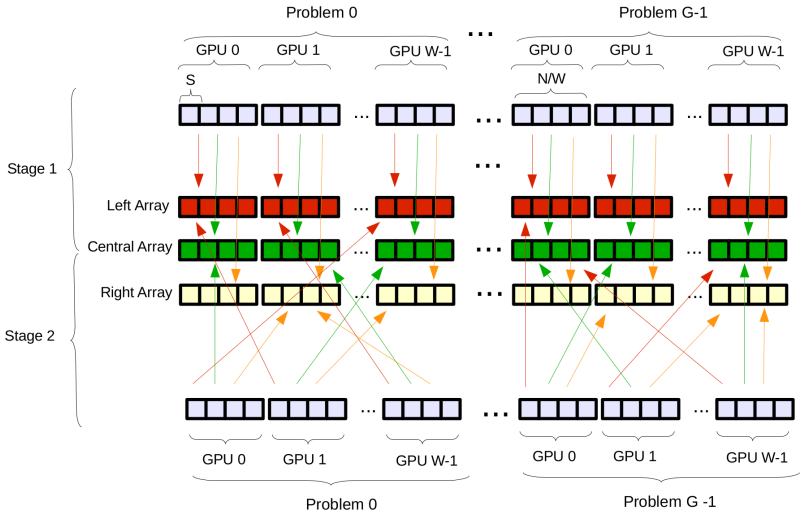


Figure 6.18: MPS approach for G problems and W GPUs

a local array with the *cudaMemcpyAsync* instruction, transfering data through host memory. Figure 6.20 depicts the partition of a 16-element problem between $W = 2$ GPUs, each executing $B = 2$ threadblocks.

In this case, the mapping vectors for the Multi-GPU environment is expressed as follows:

$$[\underbrace{t_{batch+n} \cdots t_{n+1}}_{b_y} \quad (6.6)$$

$$\overbrace{\underbrace{t_n \cdots t_{n-w+1}}_{w} \underbrace{t_{b_x+l_y+l_x+p} \cdots t_{l_y+l_x+p+1}}_{b_x} \underbrace{t_{l_y+l_x+p} \cdots t_{l_x+p+1}}_{l_y} \underbrace{t_{l_x+p} \cdots t_{p+1}}_{l_x} \underbrace{t_p \cdots t_1}_{p}}^{n}]$$

In the Multi-Node environment, the mechanism for performing communication between nodes is the use of MPI routines among them. This MPI communication introduces a huge overhead in the execution, even when using MPI CUDA-aware routines, as we have demonstrated in [35] for the scan primitive. For tridiagonal systems, the global penalty would be worse, as much more data transactions would be transferred through MPI. Thus, we only consider the Multi-GPU environment

```
1  //Allocate buffers in each GPU
2  //Copy data to buffers
3  //Create one stream for each GPU
4
5  dim3 blocks(..);
6  dim3 threads(..);
7
8  const int strideData=..;
9  const int strideTriads=..;
10
11 //Other parameters initialization
12
13 for each i in GPUs
14 {
15   cudaSetDevice(i);
16   WM_Stage1<N1,S1,P1><<<blocks,threads,0,stream[i]>>>(d_data[i], d_data[
         i]+strideData,..,triads[i],triads[i]+strideTriads,..);
17 }
18 ...
19 for each i in GPUs
20 {
21   cudaSetDevice(i);
22   cudaStreamSynchronize(stream[i]);
23 }
24 for each i in GPUs
25 {
26   cudaSetDevice(i);
27   WM_Stage2<N2,S2,P2><<<blocks2,threads2,0,stream[i]>>>(triads[i],
         triads[i]+strideTriads,..,d_data[i]+3*strideData,..);
28 }
29 for each i in GPUs
30 {
31   cudaSetDevice(i);
32   cudaStreamSynchronize(stream[i]);
33 }
34 //Copy solution to host
35 //Destroy stream
36 //Release GPU buffers
```

Figure 6.19: Pseudocode for the MPS approach where all GPUs belong to the same PCI-e

for solving the MPS approach in several GPUs.

6.5.3. Multi-GPU Problem with Prioritized Communications (MP-PC)

This approach basically takes advantage of the PCI-e networks within the computing node, computing the *Problem Parallelism* case. When the P2P API is not enabled, memory transfers are done through host memory, losing a good deal of performance. In order to minimize this loss, V GPUs of each PCI-e network work

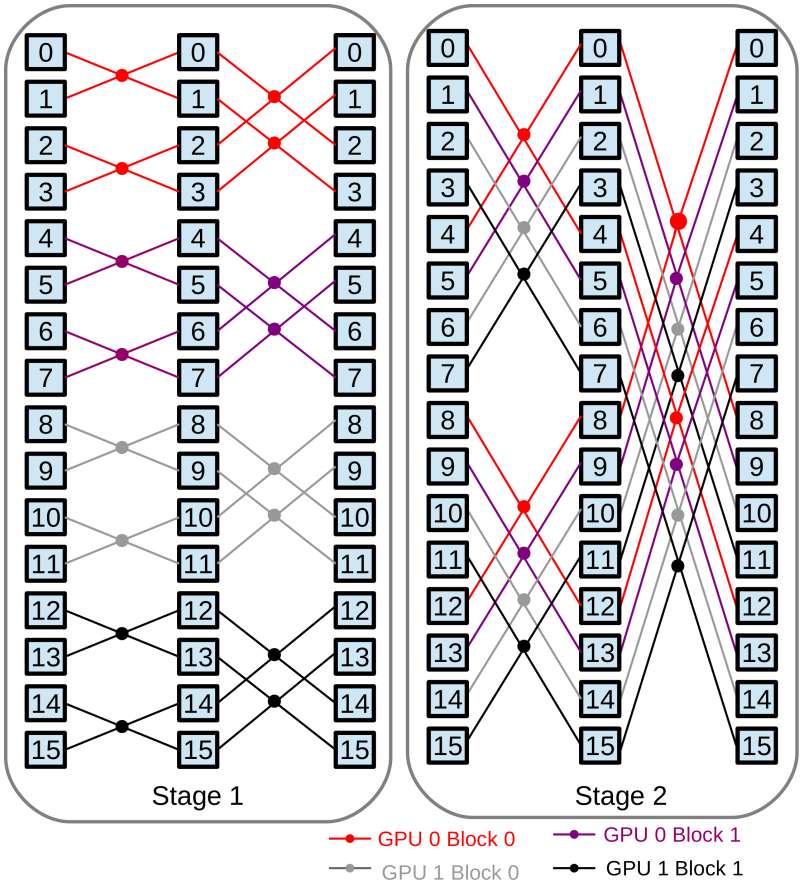


Figure 6.20: An example for the MPS approach with $N = 16$, $G = 1$, $W = 2$ and $B = 2$.

on the same G/Y problems, partitioning each problem into V chunks of size N/V. There is only communication among the V GPUs of the same PCI-e network node, whereas other PCI-e GPUs work on their problems, as it can be seen in Figure 6.6 with $V = 2$, $W = 8$, $Y = 4$ and $G = 12$. Thus, this approach uses the same code as the one shown in Figure 6.19 but, in terms of performance, this strategy improves the *MPS* approach by avoiding memory copies through host, and still scaling each problem along different GPUs.

As the only communication among GPUs is performed across the PCI-e bus, it makes no sense implement an MPI version for a Multi-Node environment.

In this approach, the *mapping vector* slightly varies taking the approach data distribution into consideration. In the case of a Multi-GPU environment:

$$[\underbrace{t_{batch+n} \cdots t_{b_y+n+1}}_{y} \underbrace{t_{b_y+n} \cdots t_{n+1}}_{b_y} \tag{6.7}$$
$$\overbrace{\underbrace{t_n \cdots t_{n-w+1}}_{v} \underbrace{t_{b_x+l_y+l_x+p} \cdots t_{l_y+l_x+p+1}}_{b_x} \underbrace{t_{l_y+l_x+p} \cdots t_1}_{s}}^{n}]$$

6.5.4. Performance Maximization of the Tridiagonal System Approaches

The value of the (s, p, l) performance parameters remains constant from the multi-stage proposal seen in Chapter 5, independently of the *Batch Parallelism*, the *Problem Parallelism* or the *Distributed Problem Parallelism* approach. It should be observed that the total number of chunks into which a problem is divided is equal to $\frac{N}{S}$, as each threadblock processes one chunk of size S that can store in its shared memory, and each GPU computes a set of this chunks. As explained in Chapter 5, the S_1 value must be as large as possible, as first kernel makes a better use of shared memory. Therefore, having as many steps as possible in first kernel increases global performance. Similarly, it must also be noted the fact that the number of chunks generated by spliting N into portions of S elements must be equal or greater than the number of GPUs employed in the case of *Problem Parallelism* and *Distributed Problem Parallelism* ($\frac{N}{S} \geq W$ and $\frac{N}{S} \geq V$, respectively), ensuring each GPU processes at least one chunk. This should not be a problem, as the use of several GPUs is justified for large datasets with many chunks.

Following the Premise 4 for dealing with the Multi-GPU configuration, W must be as high as possible, since this problem scale well when the number of GPUs is raised and the communication minimized. Accordingly, the number of communications among GPUs must also be minimized. Whenever possible, the *Multi-GPU Batch Parallelism* approach must be employed in order to avoid that communication. Otherwise, if all GPUs belong to the same PCI-e network, *Problem Parallelism* approach can be used. In the case of having GPUs distributed among different PCI-e networks, the *Distributed Problem Parallelism* approach must be considered, performing communication only between GPUs directly connected by the PCI-e bus.

6.6. Experimental Results for the Tridiagonal System Solver with Extremely-Large Problem Sizes

In this section, our Wang&Mou proposals are compared with state-of-the-art libraries, *CUSPARSE* [95]. The performance results for the Batch Parallelism case were taken in the test platform described in Table 6.4, a Multi-GPU platform with Kepler GPUs, and a Multi-Node platform also with Kepler GPUs; whereas the performance of the Problem Parallelism case is taken in the test platform of the Table 6.2. Regarding the number of problems and their size, we have established $N \leq 524,288$, and the number of simulatenous batches depends on the analysis. Therefore, all data were in GPUs memory before starting the execution, ignoring the time spent on these memory transactions from host in our measurements. In our tests, all libraries were initialized with the same diagonally dominant systems, a way of ensuring a good numerical stability. We would also like to point out the fact of growing the problem size implies an unavoidable accuracy lost. In the same way, *CUDPP* does not execute values larger than $N = 1024$. There is also a parallel library from NVIDIA, called *NCCL*, which is optimized for multiGPU communications, although it does not have any implementation for tridiagonal systems.

6.6.1. Batch Parallelism

The performance results for the *Multi-GPU Batch Parallelism* (MBP) on the Multi-GPU Platform are considered in Figure 6.21, when $G = 8$ problems are simultaneously solved. In this approach, there is no communication among GPUs, so performance scales very well. From $N = 1024$, the Multi-GPU WM approach outperforms our single-GPU multi-stage proposal, being up to $2.2x$ times faster. Thus, the implementation scales proportionally with the number of GPUs for this configuration. In the case of *CUSPARSE*, our Multi-GPU proposal always surpasses it, being $5.51x$ times faster on average. Figure 6.22 depicts the same analysis for $G = 64$, being up to $2.6x$ faster than the WM single-GPU implementation. While the WM single-GPU implementation is limited by the memory bandwidth for this amount of data, the execution over 2 GPUs distributes the workload better. How-

	Multi-GPU Platform	*Multi-Node Platform*
Description	Multi-GPU system: 1 node	Multi-Node system: 4 nodes
CPU	Intel Xeon E5-2660 2.2 GHz	Intel Xeon E5-2660 2.2 GHz
Memory	64 GB DDR3 1600	64 GB DDR3 1600
OS	CentOS 6.4	CentOS 6.4
GPU	*2x Nvidia Tesla K20*	*1x Nvidia Tesla K20*
Driver	367.57, SDK 7.5	367.57, SDK 7.5

Table 6.4: Description of the test platforms

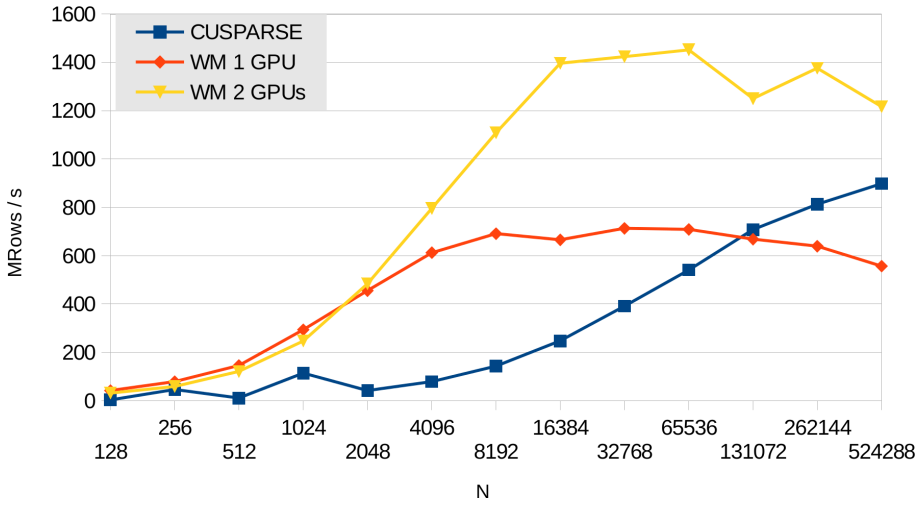


Figure 6.21: Multi-GPU approach for $G = 8$.

ever, performance also begins to decrease due to bandwidth saturation. This condition could be fixed by using more GPUs to distribute the workload and reduces the bandwidth consumption per GPU. For all problem sizes, the Multi-GPU implementation outperforms *CUSPARSE*, being $4.5x$ faster on average. Finally, Figure 6.23 shows the case of $G = 256$ batches. It should be noted that problem sizes greater than $N = 131072$ for this batch size are not allowed in single-GPU implementations due to memory limitations, it being essential to use our multi-GPU proposal. Again, performance drops due to the huge bandwidth consumption of the Wang and Mou implementation, but still being up to $2.65x$ faster than the WM single-GPU implementation, and $4.78x$ faster than *CUSPARSE* on average.

Additionally, Figure 6.24 depicts a previous analysis on the Multi-Node Platform, when $G = 8$. In the case of $M = 2$ nodes, our WM Multi-Node proposal runs

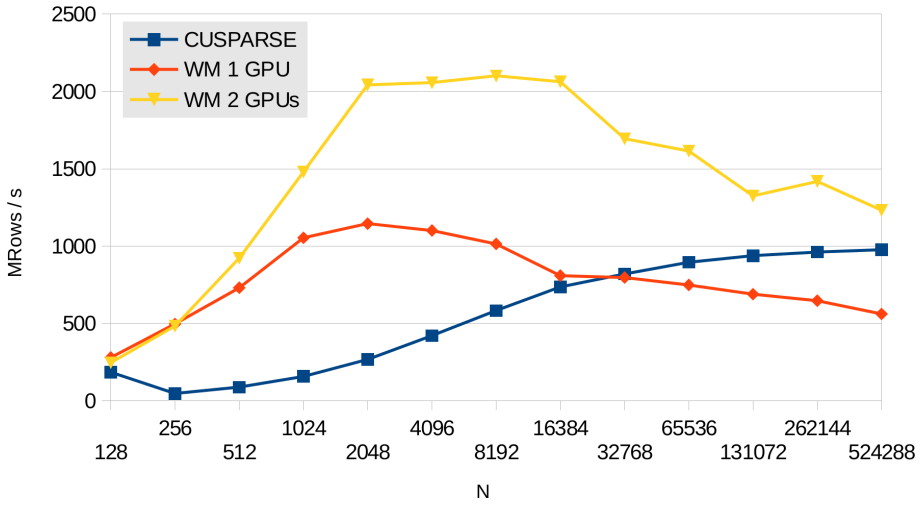


Figure 6.22: Multi-GPU approach for $G = 64$.

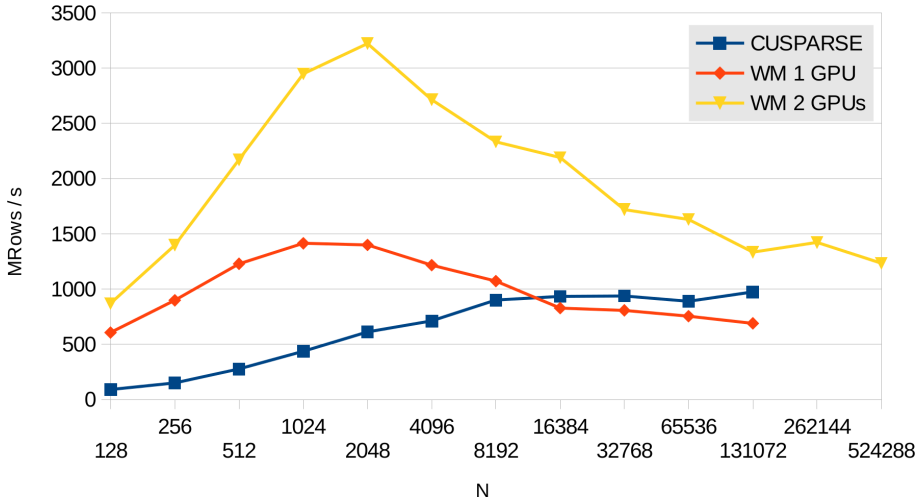


Figure 6.23: Multi-GPU approach for $G = 256$.

slower than the WM single-GPU implementation. This behavior can be explained as the result of synchronizing the nodes with MPI routines in the timing window, since these routines introduce a huge overhead. In the case of $M = 4$ nodes, our proposal is up to $3.85x$ faster than the WM single-GPU implementation, up to $3.74x$ faster than the WM Multi-Node implementation with $M = 2$ and $6.33x$ faster on average with respect to *CUSPARSE*. Similar results are obtained in Figure 6.25 for $G = 64$, achieving a speed-up of $5.7x$ against *CUSPARSE*. Figure 6.26 shows the case of $G = 256$, where only our Multi-Node proposal is capable of solving problem

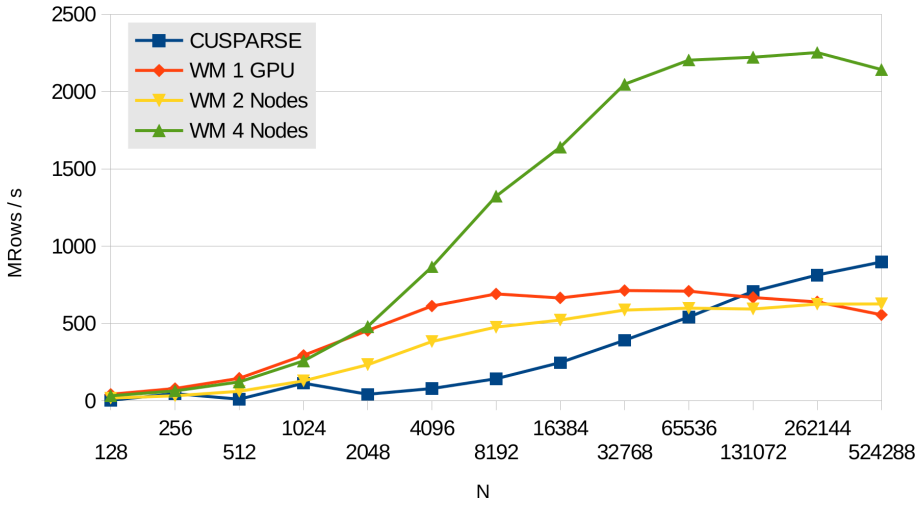


Figure 6.24: Multi-Node approach for $G = 8$.

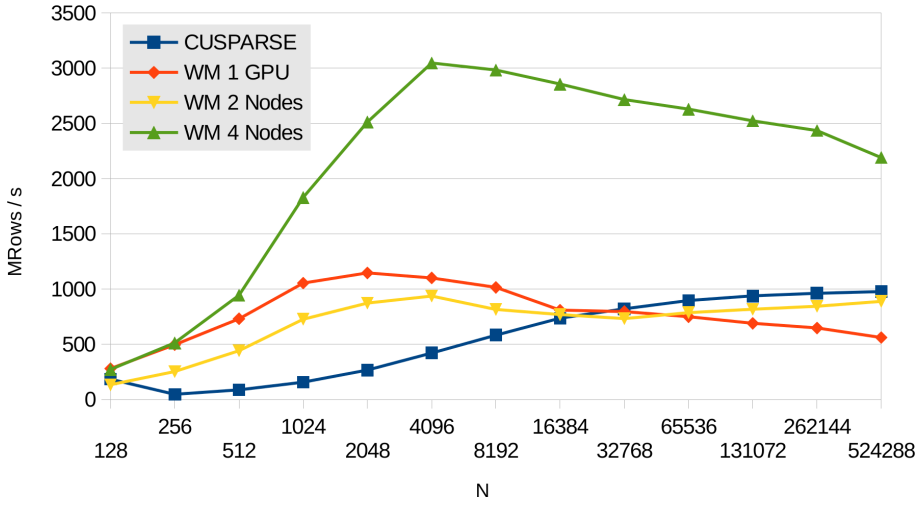


Figure 6.25: Multi-Node approach for $G = 64$.

sizes greater than $N = 131072$ for this batch, due to memory consumption. For this batch size, the WM implementation with $M = 4$ outperforms *CUSPARSE* obtaining a speed-up of $6.25x$, on average. The marked drop in performance from $N = 4096$ can be fixed by adding more nodes to computation (although there is no availability in our test platforms). Finally, Figure 6.27 depicts the analysis of $G = 512$ batches, where only our WM Multi-Node proposal with $M = 4$ is capable of solving $N = 524288$, due to the memory limitations in the other configurations. It achieves a speed-up of $6.11x$ with respect to *CUSPARSE*.

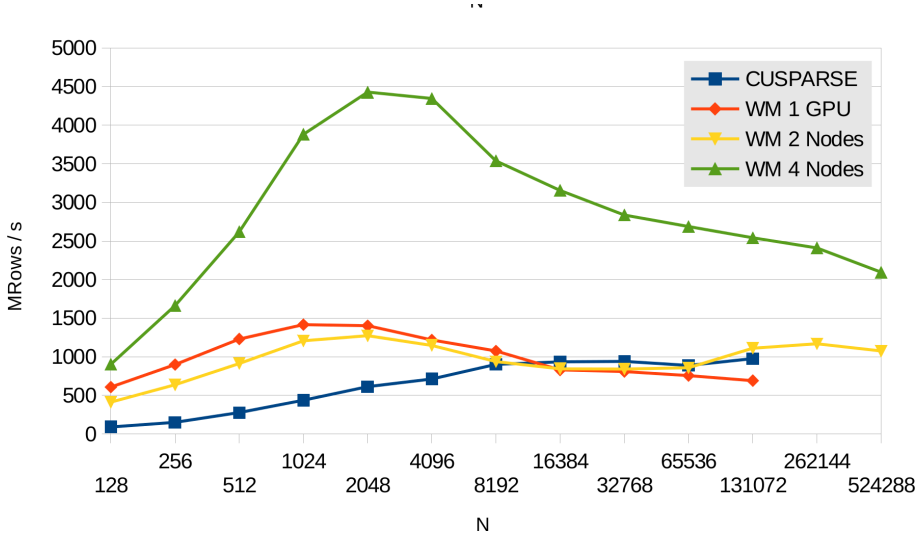


Figure 6.26: Multi-Node approach for $G = 256$.

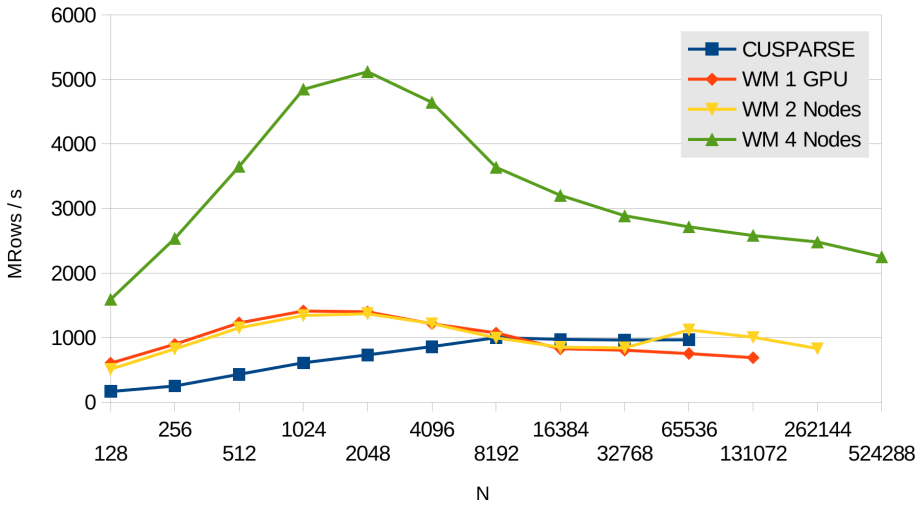


Figure 6.27: Multi-Node approach for $G = 512$.

6.6.2. Problem Parallelism

Here, the results for the Problem Parallelism case are explained. In contrast to the Batch Parallelism case, the performance here was tested on a different Platform (Table 6.2), as they were analyzed in different studies, but it uses the same GPU architecture, Kepler. Figure 6.28 shows the analysis for the *Multi-GPU Problem Scattering* approach (MPS) with $G = 8$. Please observe that there are 2 PCI-e networks, each one with 4 GPUs ($V = 4$) in this platform, thus the P2P API is

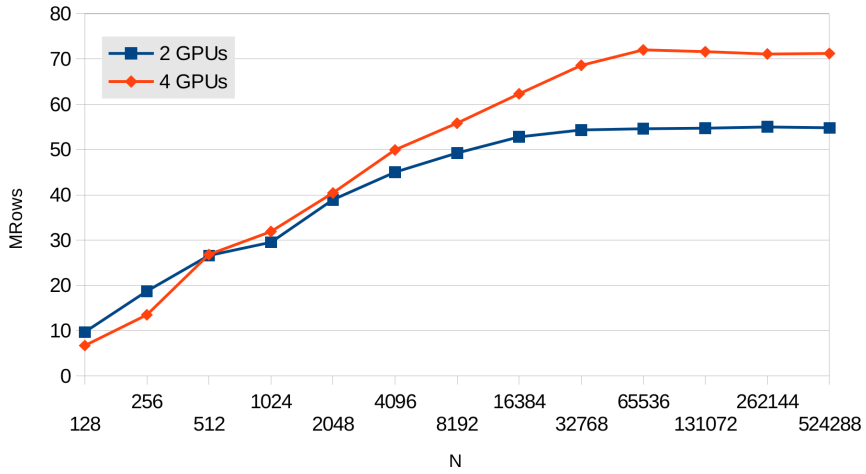


Figure 6.28: Performance analysis for the Multi-GPU Problem Scattering (MPS) approach with $G = 8$.

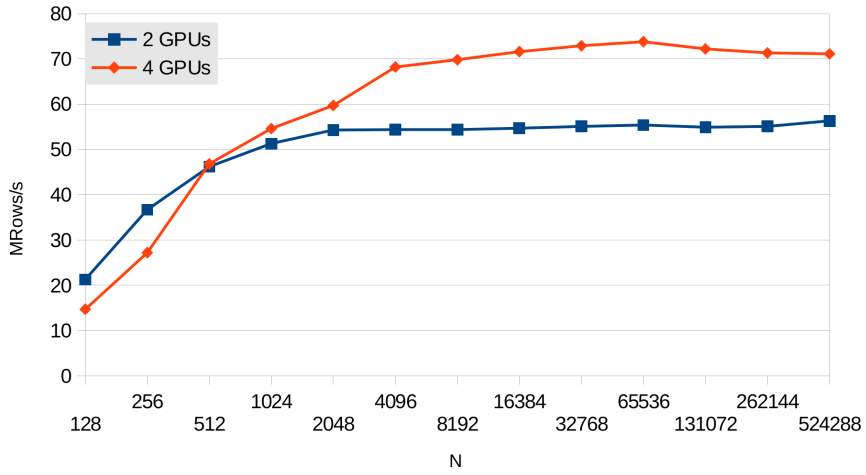


Figure 6.29: Performance analysis for the Multi-GPU Problem Scattering (MPS) approach with $G = 64$.

enabled. As the node configuration only enables 4 GPUs connected to the same PCI-e network, if $W > 4$, it would use host memory transactions reducing performance so markedly. However, although transactions do not pass through host memory when $W \leq 4$, performance is very poor compared with the Batch Parallelism case. The huge amount of data to be transferred saturates PCI-e bandwidth. Figure 6.29 shows the performance analysis for this approach when $G = 64$.

Figure 6.30 depicts the case of *Multi-GPU Problem with Prioritized Communi-*

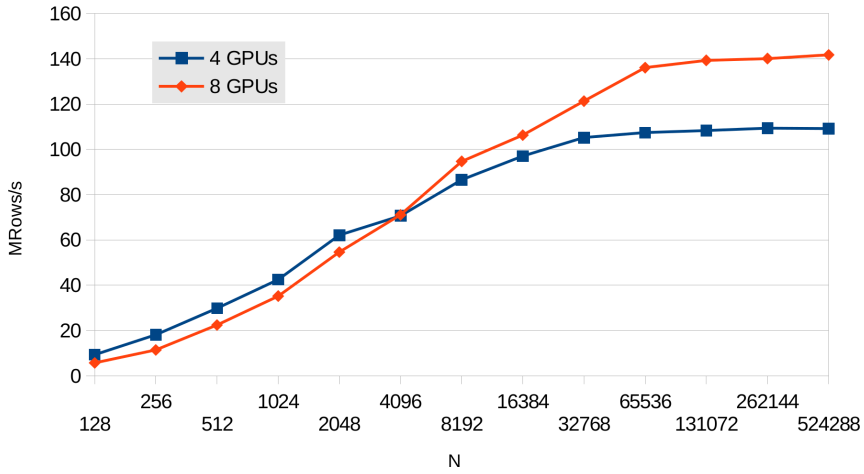


Figure 6.30: Performance analysis for the Multi-GPU Problem with Prioritized Communications (MP-PC) approach with $G = 8$

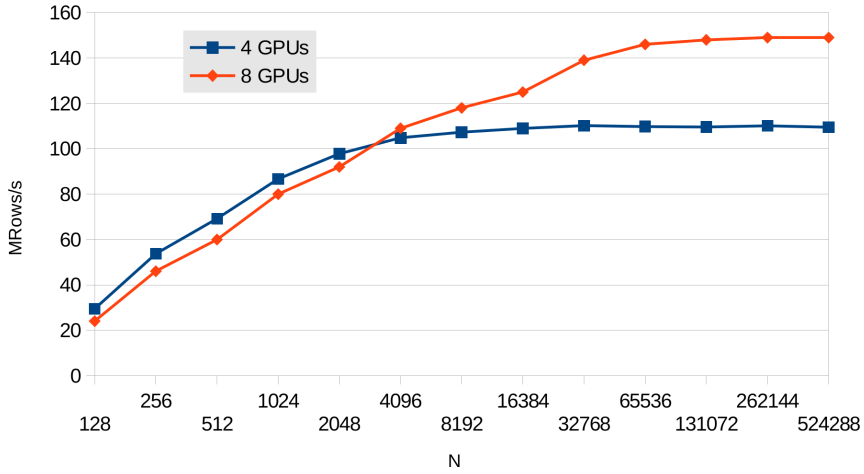


Figure 6.31: Performance analysis for the Multi-GPU Problem with Prioritized Communications (MP-PC) approach with $G = 64$

cations (MP-PC) approach for $G = 64$. It is slightly better than the MPS approach; there are communications between GPUs but these are still performed with P2P API. This approach can only be used when N is stored into V GPUs of the same PCI-e network. Please, note that this approach makes no sense with only 2 GPUs. In the case of *W=4*, we have considered $W = 4$, $Y = 2$ and $V = 2$ for this test, whereas *W=8* represents $W = 8$, $Y = 2$ and $V = 4$. As each problem is solved by V GPUs, if the number of problems, G, would be less than the number of PCI-e networks, Y, the number of GPUs being used has to be reduced. The same problem

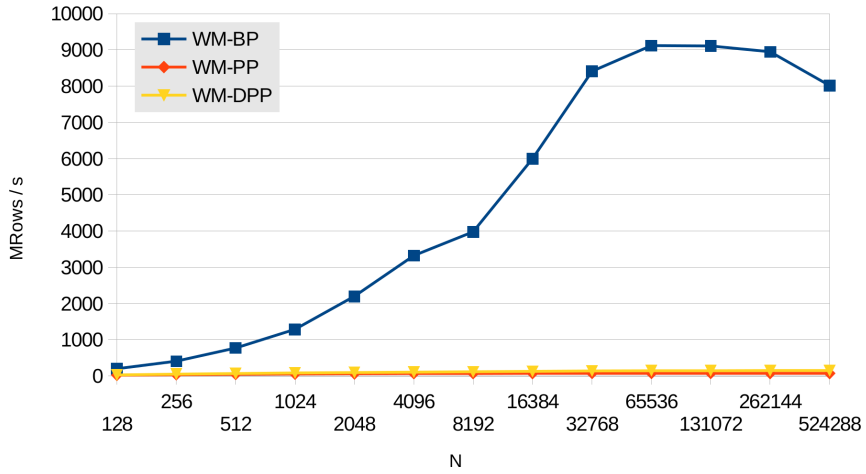


Figure 6.32: Performance evaluation for all Multi-GPU approaches with $G = 64$

for MPS arises here, the communication among GPUs via PCI-e introduces a huge latency. Figure 6.31 performs the same analysis for $G = 64$.

In order to show the performance difference among MPB, MPS and MP-PC, Figure 6.32 compares the three approaches in the test platform described in Table 6.2. For the MPS and MP-PC approaches, the huge amount of data transferred in the Wang&Mou algorithm (three equations per element, each with 4 coefficients) has a highly negative affect on the global performance. The Problem Parallelism approaches have to be discarded as alternatives for computing large problem sizes due to the low performance derived from the GPU communications. However, the MPB approach scales very well, as there is no communication among devices and the huge amount of data is distributed among GPUs.

6.7. Conclusions of the Chapter

In this chapter, our tuning methodology is extended to solve extremely large problem sizes, thanks to the use of multiple GPUs in the computation. In the case of general parallel prefix algorithm, this methodology is composed of three phases; whereas it is composed of 2 phases for ID-algorithms.

In the case of general parallel prefix algorithms, our methodology is tested over the scan operator. Our scan proposal is compared against *CUDPP*, *Thrust*, *Mod-*

ernGPU, *LightScan* and *CUB* libraries, achieving a speed-up of up to $1.21x$, $7.8x$, $1.31x$, $1.31x$ and $1.04x$, respectively when computing a single batch ($G = 1$); and up to $9.48x$, $49.81x$, $33.77x$, $58.44x$ and $8.92x$ when solving G problems simultaneously. Thanks to our multi-stage design of the scan operator, the amount of data to be transferred between GPUs is low, and good speed-ups are achieved. Three different approaches are proposed to deal with multiple GPUs: the *Multi-GPU Batch Parallelism* (MPB), with no communication among devices; and the *Multi-GPU Problem Scattering* (MPS) and *Multi-GPU Problem with Prioritized Communication* (MP-PC) approaches, when there is partial communication among GPUs.

In the case of ID-algorithms, our methodology is applied to solve tridiagonal systems. Specifically, the Wang&Mou algorithm is tested in a multiple-GPU environment, also providing the MPB, MPS, MP-PC approaches. When there is no communication among GPUs, MPB approach, our solver is up to $6.11x$ faster than the state-of-the-art, the *CUSPARSE* library. However, the huge amount of data transferred in the approaches with communication, MPS and MP-PC, penalizes the global performance too much to be competitive.

Chapter 7

Using Accelerated Parallel Prefix Operations on Real Applications

In this chapter, our tuning methodology for designing efficient Parallel Prefix operations on GPUs is tested on computing applications that require high performance calculations. Specifically, the multiplication of large integers is a common operation in many real-world applications. Especially, many cryptography algorithms require operating on very large subsets of integer numbers, such as the public-key cryptography, which employ arithmetic with hundreds of digits [108]. Additionally, this multiplication is also frequently used to render fractal images at high magnification, such as those found in Malderbrot set [13]. In both cases, the multiplication can be performed involving a FFT operation.

In the following sections, two different approaches, which are based on our proposals, are presented to compute the multiplication of large integers, demonstrating the efficiency of our methodology. The work introduced in this chapter was originally presented in [29].

7.1. Introduction to High-Precision Integers

The multiplication of integers with a large number of digits is an operation commonly used in computer science, being especially important in cryptography.

Public-key cryptography is widely used in secure communication protocols, such as SSL (*Secure Sockets Layer*) and TLS (*Transport Layer Security*). These protocols rely on secure key exchange and signature algorithms such as ECC (*Elliptic-Curve Cryptography*), RSA (*Rivest-Shamir-Adleman*) or DSA (*Digital Signature Algorithm*). Unfortunately, public-key algorithms are not nearly as computationally cheap as symmetric encryption algorithms; they are much more time consuming. For example, according to the study [140], over 90% of the time in cryptographic operations was spent on the RSA key exchange. In the meantime, the RSA key length required for internet domains has increased, as Moore's law continues its scaling. The RSA-1024 has been overwhelmingly used to secure internet communications. Nevertheless, the US-based National Institute of Standards and Technology (NIST) recommended that as early as the year 2010, systems achieving a security level of 80 bits should be deprecated [20]; from 2010 until 2030 a security level of at least 112 bits should be enforced, and a minimum of 128-bit level was recommended from the year 2030 and beyond. These security levels can be instrumented via RSA using keys with bit-lengths of at least 1024, 2048 and 3072 bits, respectively. Thus, it is necessary to perform an urgent migration to higher levels of security, and this requires achieving highly-optimized implementations of the RSA system for larger key sizes. The main operation on an RSA cryptosystem is the modular multiplication for large integers used to compute their modular exponentiation. This is just one example of the importance of solving the multiplication of large integers efficiently.

The classical vector multiplication has $O(N^2)$ complexity, where N is the number of digits. By using the Strassen FFT multiplication algorithm [111], which has $O(Nlog_2N)$ complexity, the time is significantly reduced. This algorithm is derived from the fact that any integer multiplication can be expressed as a polynomial product, called vector convolution, followed by a carry normalization.

Previous studies on high-precision integer multiplication were designed for outdated GPU architectures, using GPU libraries which are no longer the most efficient and are mostly focused on the Strassen FFT algorithm. It is crucial to know the polynomial size at which the FFT Strassen algorithm starts to run faster than others, and this size threshold varies from one architecture to others, but it is also interesting to discover alternatives for different sizes. Additionally, most of the previous works do not examine the parallel implementation of the carry normalization.

There are a number of serial libraries and frameworks which perform large integer multiplication. For example, Microsoft introduced the BigInteger type in .NET 4.0 to compute large integers [86], which has no upper or lower bounds. Additionally, IntX [19] is a large precision integer library with fast multiplication based on the Hartley Transform [11]. The GNU MP Library [49] also includes fast calculation algorithms for arbitrary precision arithmetic. There are also several software packages for computing symbolically with polynomials and matrices, such as Linbox [124], MAGMA [10] and NTL [125], although most of these are devoted to serial implementation in the case of polynomials. In [42], the FFT multiplication is implemented and compared with the normal multiplication.

In the case of GPUs, there are several CUDA implementations of large integer multiplication. All of them focus solely on the Strassen FFT approach, ignoring any other approximation to work around the problem. Additionally, both the architectures and libraries employed in these works are completely outdated. The work presented in [43] employed the CUDA Fermi architecture, whereas the implementation of [139] is prior even to Fermi. In these works, many decisions were taken based on the latency and efficiency of some operations, which have been completely overhauled in current architectures. Other GPU implementations can be found in [71] and [84], although they were also tested on outdated CUDA SDKs and architectures. In [6], a fast integer multiplication based on the cuFFT library [94] is implemented, surpassing all other previous framework implementations.

7.2. The Strassen FFT Multiplication Algorithm

The Strassen FFT multiplication algorithm [111] is based on the polynomial multiplication. Any number in base x can be decomposed into a polynomial coefficient vector. A polynomial p is described by its coefficient vector $a = [a_0, a_1, \cdots, a_{N-1}]$ as follows:

$$p(x) = \sum_{i=0}^{N-1} a_i x^i \tag{7.1}$$

where x is the base of the polynomial, $p(x)$ is the evaluation of the polynomial for base x and N the number of digits of the number; i.e., the size of the polynomial. For example, considering the integer 54321, its polynomial form using $x = 10$ would be $a = [1, 2, 3, 4, 5]$ or $p(x) = 1 + 2x + 3x^2 + 4x^3 + 5x^4$. Multiplying two polynomials results in a third polynomial of size $2 \cdot N$, and this process is called vector convolution.

According to the convolution theorem, if c is the convolution of two input vectors a and b, $c = a \cdot b$, then the Discrete Fourier Transform (DFT) of c is equal to the pairwise multiplication of the DFT transform of each input vector, $DFT(c) = DFT(a)DFT(b)$, where *pairwise multiplication* means multiplying the vectors in pairs, element by element. Thus, the c vector can be also obtained as the Inverse Discrete Fourier Transform (IDFT) of this pairwise multiplication:

$$c = IDFT(DFT(a)DFT(b)) \tag{7.2}$$

Given two input values, a and b, each one with N and M digits, respectively, the Strassen FFT algorithm performs as follows. Firstly, the integers are represented as polynomials in their coefficient-form representation, $a = a_0, a_1, \cdots, a_{N-1}$ and $b = b_0, b_1, \cdots, b_{M-1}$. If the input vectors do not have the same length, $M < N$, the shortest one is filled with zeros until $M = N$. Once the integers are represented as polynomials, the convolution theorem is applied. In order to easily compute the DFT of each vector, the Fast Fourier Transform is performed for each vector, after which, the pairwise multiplication is applied as well as the inverse FFT. Finally, the coefficients have to be normalized to the same base as the one in which the integer is represented (looping to propagate the carry).

7.3. The CUDA FFT-based Multiplication Approach

At the beginning of Chapter 5 we mentioned that we had presented a tuned FFT proposal in [36] that follows the methodology developed in this Thesis. This FFT proposal (MS-ID-FFT) was focused on medium and large problem sizes, using a multi-stage strategy. In this approach, we have employed that *MS-ID-FFT*

approach to compute the FFT operation of the Strassen algorithm for multiplying large integers, in a similar way as the authors of [6] did with the cuFFT library.

When using the Strassen algorithm, most of the existing implementations use the finite field $\mathbb{Z}/p\mathbb{Z}$, with prime p, instead of the complex field $\mathbb{C}$, since the error analysis is easier. Nevertheless, there are several important restrictions with the finite field. Where x is the base and k the size of the FFT in the finite field:

- The field $\mathbb{Z}/p\mathbb{Z}$ requires a k^{th} root of unity.
- The length of the product a times b must be less than k.
- The maximum value must fit in the field; i.e., $k/2(x-1)^2 < p$.
- Multiplying in $\mathbb{Z}/p\mathbb{Z}$ must be modulo p, thus the existence of a fast modulo p operator is desirable (such as Montgomery reduction algorithm).

Taking previous restrictions into account, our FFT approach has been designed to work with the complex field $\mathbb{C}$. Additionally, this work uses the MS-ID-FFT implementation, which only supports the $\mathbb{C}$ complex numbers in base $x = 10$. However, this floating design forces us to attend to the numerical accuracy. Specifically, two different proposals were developed following this approach: the *Complex-ID* proposal and the *Real-ID* proposal.

7.3.1. The Complex-ID Proposal

This proposal is tagged as *Complex-ID* in Section 7.6. The steps explained in Section 7.2 are performed here for the vectors a and b of size N, using the MS-ID-FFT implementation for complex numbers (*Complex-ID* function). The resulting polynomial of multiplying a and b, c, will have a degree two times greater than the highest degree of a and b; thus, the size of c is $2 \cdot N$. Before performing the forward FFT, a and b are extended up to $2 \cdot N$, padding with zeros. The imaginary part of each element is set to zero, and the coefficients are assigned to the real part. Once the FFT is applied for each input vector, both signals are pairwise multiplied:

$$c.x = a.x * b.x - a.y * b.y; \tag{7.3}$$

$$c.y = a.x * b.y + a.y * b.x; \tag{7.4}$$

At this point, the inverse FFT is performed for the resulting vector from the pairwise multiplication. It should be noted that a pairwise multiplication kernel has also been developed to multiply the elements, which are already in the GPU memory.

7.3.2. The Real-ID Proposal

The previous proposal wastes half of the memory bandwidth carrying zeros in the imaginary part of each number. In this proposal, tagged as *Real-ID* in Section 7.6, the FFT operation of the MS-ID-FFT library is extended with real-number support. Two new functions are developed: a Real-to-Complex (R2C) function for the Forward FFT, and a Complex-to-Real (C2R) function for the Inverse FFT. To do this, the real signal is packed into a vector with half of the size (reading each two consecutive real values as a single complex number), and then the *Complex-ID* function performs the transform of this half-size signal. After this, a post-processing stage is used to combine the output and unpack the data, consuming the half of the memory bandwidth with respect to the previous proposal. This can be achieved thanks to the complex conjugate property, where half of the information in the transformed signal is redundant. It should be observed that the computation of the post-processing stage is performed in an additional kernel after (before) the forward (inverse) FFT; thus, in addition to the kernels given by the MS-ID-FFT approach, a kernel which computes the post-processing stage has had to be developed.

7.4. The CUDA Tiling Multiplication Approach

In this section, a new approach for computing an efficient multiplication of two large integers on a GPU is proposed. This new approach is based on the classical

vector convolution algorithm and avoids working with the Discrete Fourier Transform.

7.4.1. The vector convolution algorithm

Although the classical algorithm of polynomial multiplication seems sequential, as Figure 7.1 shows, it is possible to apply a divide-and-conquer strategy to compute the multiplication in parallel. This approach, tagged as *Tiling-based* in Section 7.6, divides the computation of the c reduction (line 5 in pseudo-code) through several data blocks, where each data block works with tiles of size T. Specifically, each data block computes $2 \times T - 1$ elements of c, taking T elements from vector a and T elements from vector b as inputs. Then, each data block has to integrate its partial result with the others, in a sequential reduction, in order to obtain the overall result; whereas the number of data blocks is given by $(N/T) \times (N/T)$. From a computer architecture perspective, each data block is computed by one computing unit of the target architecture and the optimal value of the tile size, T, also depends on the given architecture.

Figure 7.2 depicts an example of this approach with $N = 4$ and $T = 2$. There are $2 \times 2 = 4$ computing units, where each computing unit computes 3 elements of the solution, reading 2 elements from a, and 2 from b.

7.4.2. CUDA implementation

When this approach is implemented on a GPU, each computing unit corresponds with a threadblock. Each threadblock works with T elements from a and T from b. The whole computation is performed in a single kernel invocation and the overall result is calculated by integrating the partial results with atomic instructions in global memory.

Figure 7.3 shows the work performed by each threadblock. Each pair of T elements from input vectors is assigned to one threadblock. Then, each threadblock divides the computation of the multiplication among its threads. To do this computation, this approach does not use shared memory, since all exchanges are performed

```
1 for( k from 0 to 2*N-1)
2         c[k]=0
3 for (i from 0 to N-1)
4         for (j from 0 to N-1)
5                 c[i+j]+= a[i]*b[j]
```

Figure 7.1: Pseudocode of the vector convolution operation

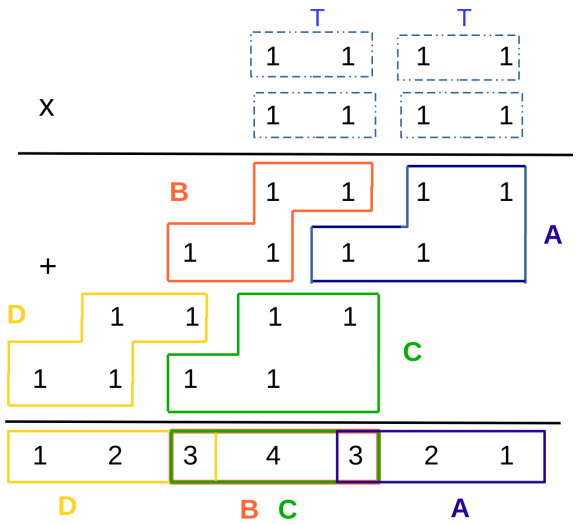


Figure 7.2: Classical multiplication operation in tiles

by shuffle instructions. The partial result of each threadblock is stored in private registers of its threads, and is carried to its positions in the result array, performing the corresponding reduction with other threadblocks in global memory.

Nevertheless, these operations constitute a significant bottleneck for large-size inputs. It should be noted that each memory location is atomically accessed as many times as the number of tiles. Thus, large problem sizes will suffer memory contention due to atomics, despite the new improvements on these operations in new architectures. Since higher number of tiles implies higher contention, and the number of tiles is equal to the number of threadblocks employed, each block must be executed with the greatest number of threads possible and each thread must be in charge of the maximum number of elements possible in order to reduce the number of threadblocks.

In order to find the suitable T value, an exhaustive search is empirically computed for each supported architecture, ascertaining its optimal value. The optimal T

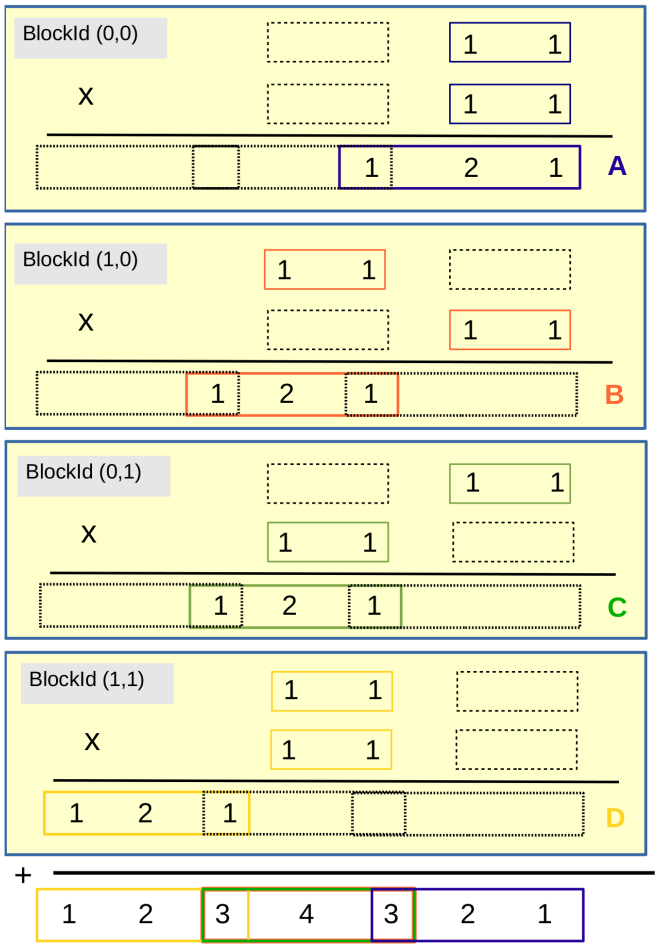


Figure 7.3: GPU implementation of the tiled multiplication where $N = 4$ and $T = 2$.

value is affected by the GPU global memory bandwidth, its SM parallelism, the performance of the global memory atomics in that GPU and the size N of the input vectors, using our tuning methodology to obtain the corresponding optimal values.

7.5. The Carry Normalization

After multiplying the vector inputs, each element in the output vector is the result of the corresponding product, and may be composed of several digits. In order to obtain the final result, each element must perform the module operation and propagate the carry to the next more significant elements. This implies that

each element receives a carry-in from the less significant elements, performs the module operation and propagates its carry-out to the more significant elements. This process is called *Carry Normalization* or *Carry Propagation* and its pseudo-code is illustrated in Figure 7.4 for the base $x = 10$.

In contrast with traditional adders, where the carry flag is a single digit used to indicate when a carry-out has been generated and is propagated to the immediately adjacent more significant position, the carry accumulator here (line 8) may be composed of several digits; i.e., the carry must be propagated to several more-significant elements. This fact limits the parallelization of the algorithm using a carry look-ahead scheme, since this scheme is designed to propagate a single-digit carry, not a multiple-digit one, as Figure 7.5 shows for an example, which is the result of multiplying two polynomials, and whose polynomial form is $a = [579, 23, 2, 0]$. In order to deal with this, the computation is broken into two different phases.

In the first phase, each element will be normalized to a two-digit number in base 10. To do this, considering integer type codification, each element may be composed of up to 10 digits; keeping the first digit as the element's value and the remaining 9 digits are assigned as the element's carry-out. Therefore, each digit of the element's carry-out has to be propagated to the corresponding adjacent more-significant elements, nine at most. In other words, each element receives a single-digit carry-in (a number in $[0, 9]$) from 9 elements at most. After adding the single-digit carry-in from its adjacent less-significant elements to itself, each element will be composed of two digits at most: considering the extreme scenario where the element's value is 9 and the nine carry-ins received are also 9, the total addition would be 90, two digits. The implementation of this idea is depicted in Figure 7.6 (a). Firstly, every element performs the module operation at a time, obtaining each element's value, $[9, 3, 2, 0]$ in the example. After this, the generated carry-outs, $[57, 2, 0, 0]$, have to be propagated to the next elements. Thus, each element propagates its generated carry-out to its adjacent elements, using shared memory, where each digit of the carry-out is sent to the corresponding adjacent element. In the example, the first element sends 7 to the second element and 5 to the third element; whereas the second element sends 2 to the third one and the third element sends 0 to the forth one.

In the second phase, the vector is decomposed into two vectors: the one with the

```
1  CarryPropagation(srcVector, dstVector)
2  {
3  carry:=0
4  for ( i from 0 to srcVector.length)
5          sum:= carry+srcVector[i]
6          mod:= sum \% 10
7          dstVector.Add( mod)
8          carry:= sum/10
9  while(carry>0)
10         if(carry.lenght >1)
11                 index:=carry.lenght-1
12         else
13                 index:= 0
14         dstVector.Add(carry[index])
15         carry:= carry /10
16 }
```

Figure 7.4: Pseudocode of the carry-propagation operation for large integer multiplication.

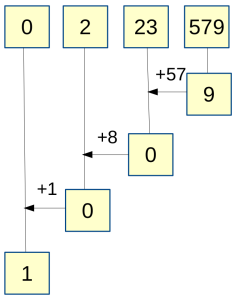
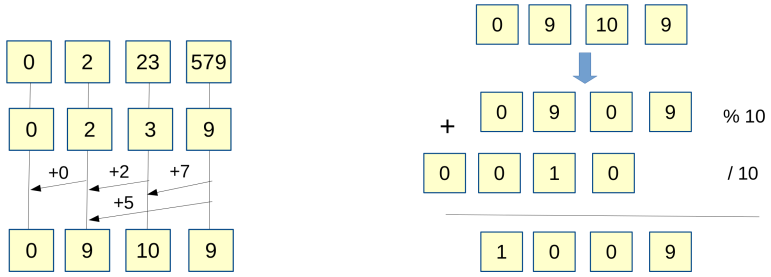


Figure 7.5: Carry propagation: Serial implementation

result of applying the module operation, and another one with the corresponding single-digit carry-out generated. The final result is built with their addition. As these two vectors are composed of single-digit elements, a carry look-ahead scheme can be applied now for their addition. Figure 7.6 (b) shows the second phase of the implementation. Specifically, in order to compute the carry look-ahead schema of the second phase, let us define $critical[i]$ as a boolean array where the i^{th} bit is set if the i^{th} element is critical; i.e., sensitive to produce a carry-out if and only if there is a carry-in (i.e., i^{th} element is the digit 9). Also, let us define $c[i]$ as a boolean array where the i^{th} bit is set if the i^{th} element generates a carry-out. Then, the carry look-ahead function is as follows:



(a) First phase (b) Second phase

Figure 7.6: Parallel carry propagation design

$$carry[i] = (c[i]) \quad \textbf{or} \quad (critical[i] \quad and \quad c[i-1]) \quad \textbf{or} \quad (critical[i] \quad and \quad critical[i-1] \quad and \quad c[i-2]) \quad \textbf{or} \quad \cdots \tag{7.5}$$

Although this expression seems very slow to evaluate, it can be replaced by integers instead of boolean arrays, getting the following expression that can be evaluated in a single step:

$$carry = ((c << 1) + carry - in + critical) \quad \textbf{xor} \quad critical \tag{7.6}$$

where carry-in is a single carry bit from the previous block of elements. Previous numerical expression can be evaluated at different levels: thread registers, warp and threadblock until reaching the final result, as explained in [43].

7.6. Experimental Results for the High-Precision Multiplication

In this section, an analysis of the results is presented. This analysis is split into two studies: a numerical study for the FFT-based approach, which uses floating

	Pascal Platform	*Volta Platform*
CPU	Xeon E5-2630	Xeon E5-2698
Memory	256 GB	512 GB
GPU	NVIDIA Pascal P100	NVIDIA Volta V100
Driver	375.51, SDK 8.0	384.81, SDK 9.0

Table 7.1: Description of the computing platforms employed

point precision, and a performance study for the two approaches.

7.6.1. Numerical analysis

While the classical algorithm and the finite-field FFT-based approaches work with exact computations, the FFT-based implementations on the Complex field can show some numerical inaccuracy due to the use of floating-point operations. Most of this numerical inexactness can be solved executing a round function after the calculation.

In order to analyze the numerical accuracy of our FFT proposals without any round, Table 7.2 shows the error obtained for floating-point 32-bit operation and floating-point 64-bit operations. In order to measure the relative error obtained with the magnitude employed, the following formula is used

$$Err(x_1, x_2) = \frac{\sqrt{\sum_{i=0}^{N-1}(x_1[i] - x_2[i])^2}}{\sqrt{\sum_{i=0}^{N-1} x_2[i]^2}}$$

where x_1 is composed of the FFT results and x_2 contains the theoretical integer results.

As can be observed in table, the numerical inaccuracy is highly acceptable in the case of working with 32-bit floating point, and extremely low when working with 64-bit floating point. In any case, this inaccuracy is almost non-existent if using a round function. It should be observed that the *Tiling-based* approach already works with integers, thus there is no numerical inaccuracy in its execution.

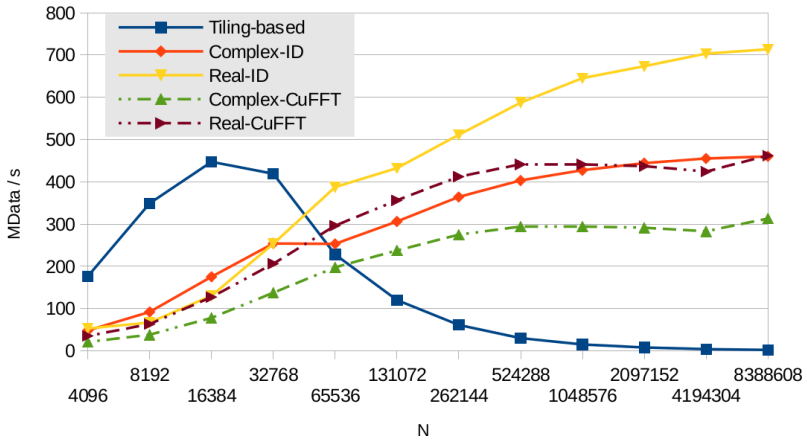
N	*FP32 Error*	*FP64 Error*
4096	1.5443149e-07	1.7168e-16
8192	2.1481627e-07	1.9978e-16
16384	1.7213162e-07	2.6860e-16
32768	1.8384862e-07	1.9637e-16
65536	2.3961446e-07	3.5997e-16
131072	3.2599550e-07	2.1811e-16
262144	3.2059985e-07	2.2378e-16
524288	2.9493298e-07	3.0587e-16
1048576	3.5447441e-07	2.5003e-16
2097152	3.4503765e-07	3.5733e-16
4194304	3.4624879e-07	3.8990e-16
8388608	3.2564203e-07	4.1682e-16

Table 7.2: Numerical analysis for our FFT proposals.

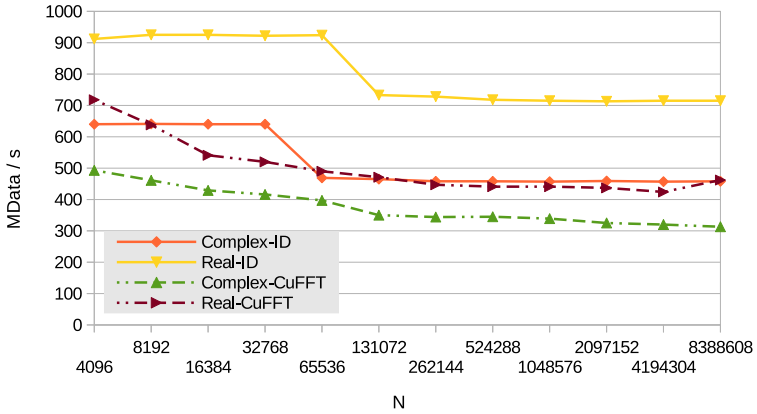
7.6.2. Performance analysis

The following results were taken in the computing platforms shown in Table 7.1. All data elements were in the GPU memory prior to the GPU execution, thus CPU-GPU memory transfers are not included in the metrics. Specifically, the $MData/s$ metric gives the number of digits (in millions) calculated by second for the resulting polynomial.

For each kernel launched in each proposal and architecture, the optimal performance parameters that maximize the GPU parallelism have been found empirically following our tuning methodology. It should be noted that the most important performance factor is to obtain the maximum memory bandwidth in the case of the FFT proposals, and to minimize the number of atomic operations in the case of the Tiling approach. In order to compare our approaches with other implementations, we have also implemented the FFT-based approach using the CuFFT library to perform the FFTs, since authors of [6] claimed that it surpasses any other state-of-the-art implementation.



(a) Single-batch execution



(b) Multi-batch execution

Figure 7.7: Performance comparison for our FP32 approaches in the Kepler Architecture

The Kepler Platform

Figure 7.7 shows the performance of both the FFT-based proposals and the Tiling approach for single precision in the Kepler Platform. As the main kernels of the Strassen algorithm are the FFT operations, the overall performance is limited by the FFT performance. Figure 7.7 (a) shows the performance when executing a single-batch problem. In the graphic, both the *Real-ID* and *Complex-ID* proposals are represented. The *Real-ID* proposal runs faster owing to its reduced memory bandwidth consumption, as expected. The performance drops are due to the launching of an additional kernel in the calculation due the problem size limits. Addition-

ally, the dashed lines represent the performance of the same FFT-based proposals but using the CuFFT library. The performance drops of the CuFFT-based proposals are also due to the launching of an additional kernel for computation. Our *Real-ID* proposal surpasses the *Real-CuFFT* one, obtaining a speed-up of up to $1.65x$, and the *Complex-ID* is up to $2.42x$ faster than the *Complex-CuFFT* proposal. The Tiling approach, tagged as *Tiling-based*, is compared against the single-precision FFT-based approaches for each architecture. The *Tiling-based* proposal outperforms the FFT-based ones while $N \leq 32768$. In the case of $N = 8192$, it is $5.2x$ faster than the *Real-ID* proposal and $5.53x$ with respect to the *Real-CuFFT* proposal. This approach only launches one kernel and performs much fewer operations than the FFT-based approaches. However, large problem sizes imply partitioning the input vectors through many threadblocks, performing many atomic operations over the same memory locations and generating a huge latency overhead.

However, the ID-FFT library has been designed to solve several batch problems simultaneously. Figure 7.7 (b) shows a multi-batch execution. Specifically, in order to perform a multi-batch execution, 2^{24} digits are allocated for the resulting polynomials where the number of batches, G, is equal to $G = \frac{2^{24}}{2 \cdot N}$, and N is the size of each input vector. In this case, the Tiling-approach is not shown for the sake of appearance, since it has a hugely higher performance. Our *Real-ID* proposal is up to $1.88x$ faster than the *Real-CuFFT* proposal, it being on average $1.61x$ faster. In the case of our *Complex-ID* proposal, it obtains a speed-up of up to $1.49x$ with respect to the *Complex-CuFFT* proposal, achieving $1.37x$ on average. The performance of the Tiling approach solving G problems simultaneously is shown in Figure 7.8 and it is compared against the FP-32 FFT-based proposals. As expected, it runs faster with small problem sizes, owing to the small number of atomic operations. Specifically in this case, problem sizes shorter than $N = 16384$ should use the Tiling-approach to compute the multiplication, achieving a speed-up of up to $5.53x$ against the FFT-based proposals; otherwise, for larger sizes, the *Real-ID* must be employed.

Figure 7.9 shows the case of double precision in this platform. It should be observed that this datatype only affects the FFT-based proposals, since the Tiling approach works with integer types; therefore, the Tiling-approach performance re-

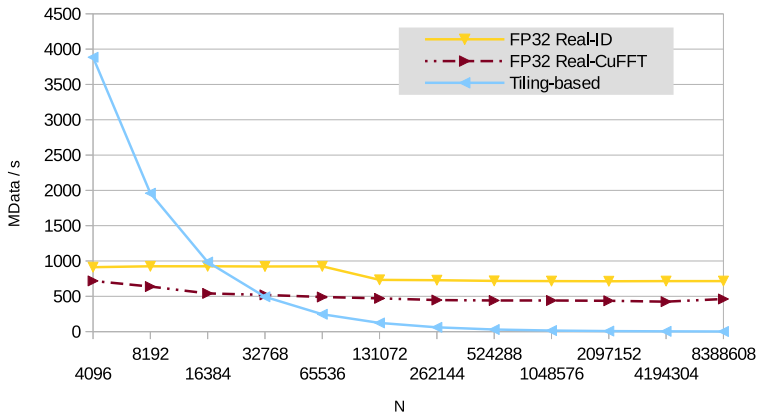


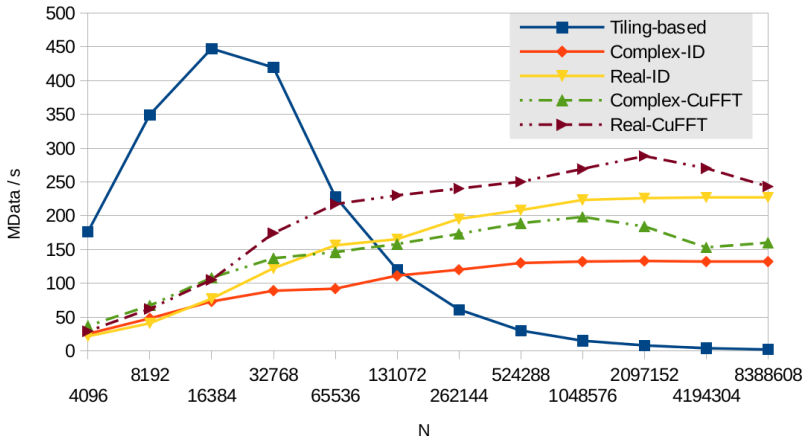
Figure 7.8: Performance comparison for the GPU-tiling proposal, when solving G problems, with respect to FP32 FFT-based proposals on the Kepler Architecture

mains constant. In the case of double precision, the shared memory consumption of the ID-FFT proposals is huge, reducing the SM occupancy excessively. Figure 7.9 (a) shows the FP64 execution for a single problem. For this case, the ID-FFT implementations are surpassed by the CuFFT-based implementations, since the ID-FFT proposals use too much shared memory; although this behavior changes for larger sizes. It can be observed in Figure 7.9 (b), which shows the FP64 execution for G problems, that our proposals surpass the CuFFT-based ones owing to having a huge amount of data being executed.

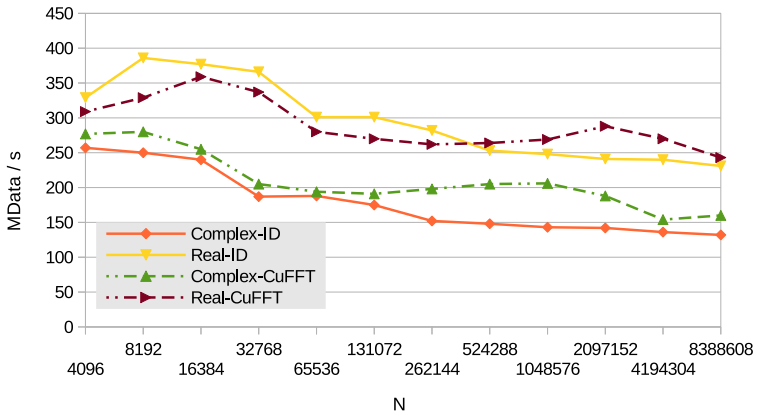
The Pascal Platform

Figure 7.10 (a) shows the performance of our approaches in the Pascal Platform, using FP32 for the FFT-based proposals, when executing a single-batch problem. Our *Real-ID* proposal is up to $2.08x$ faster compared with the real implementation of the *Real-CuFFT* proposal; whereas the *Complex-ID* proposal is up to $2.74x$ faster than the *Complex-CuFFT* proposal. The *Tiling-based* proposal outperforms the *Real-ID* proposal while $N \leq 65536$. In the case of $N = 8192$, the *Tiling-based* is $4.34x$ faster than the *Real-ID* proposal and $6.23x$ with respect to the *Real-CuFFT* proposal.

However, the ID-FFT library has been designed to solve several batch problems simultaneously, with $G = \frac{2^{24}}{2 \cdot N}$. This multi-batch case is shown in Figure 7.10



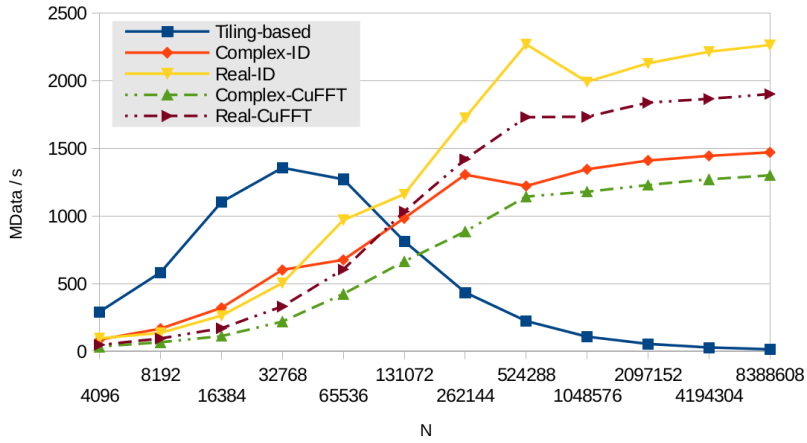
(a) Single-batch execution



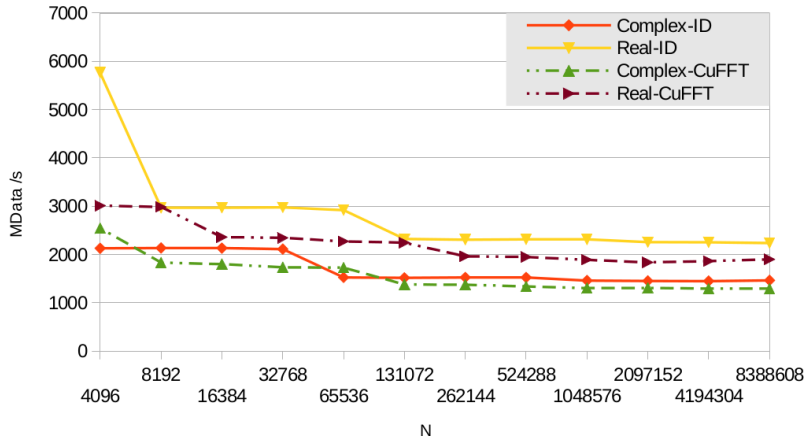
(b) Multi-batch execution

Figure 7.9: Performance comparison for our FP64 approaches in the Kepler Architecture

(b), where the Tiling-approach is not shown again for the sake of appearance, due to the range of the axis and the magnitude of the approach values. Our *Real-ID* proposal is up to $1.91x$ faster compared with the real implementation of the *Real-CuFFT* proposal; whereas the *Complex-ID* proposal is up to $1.21x$ faster than the *Complex-CuFFT* proposal. The performance of the Tiling approach solving G problems is shown in Figure 7.11 and compared against the single-precision FFT-based approaches for this architecture. It outperforms the *Real-ID* proposal when $N \leq 32768$. In the case of $N = 4096$, the *Tiling-based* is $3.92x$ faster than the *Real-ID* proposal and $7.49x$ with respect to the *Real-CuFFT* proposal.



(a) Single-batch execution



(b) Multi-batch execution

Figure 7.10: Performance comparison for our FP32 approaches in the Pascal Architecture

Figure 7.12 performs the same comparative for the double precision execution. As the FFT function is a memory-bound operation, the achieved performance is the half of the FP32-proposals performance. In the case of a single batch, the FP64 *Real-ID* proposal is up to $1.51x$ faster than the *Real-CuFFT* one; whereas up to $11x$ is obtained for the complex case, depending on the data point. In a multi-batch execution, the FP64 *Real-ID* proposal is up to $1.51x$ faster than the *Real-CuFFT* one; whereas $1.50x$ is obtained for the complex case.

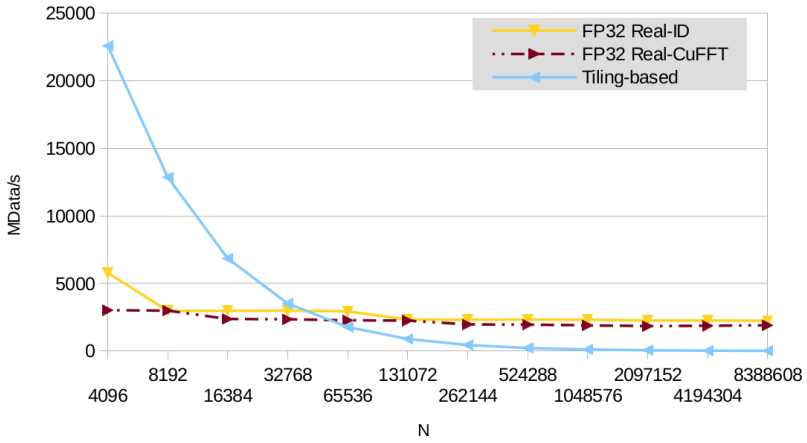
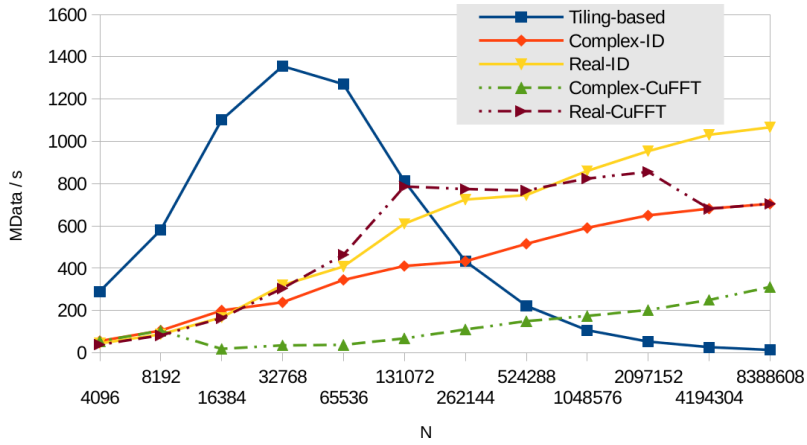


Figure 7.11: Performance comparison for the GPU-tiling proposal, when solving G problems, with respect to FP32 FFT-based proposals on the Pascal Architecture

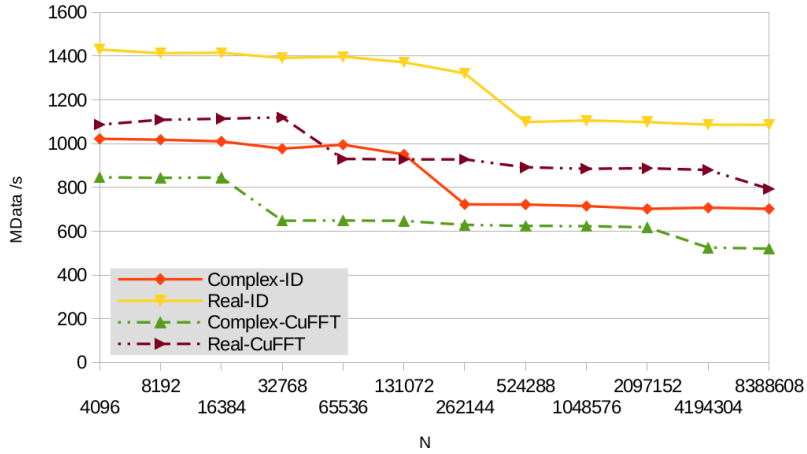
The Volta Platform

Figure 7.13 shows the performance analysis of our proposals in the Volta Platform, where the FFT-based proposals use FP32 datatypes. It should be observed that the Volta execution is $1.5x$ faster, on average, than the Pascal one for a single-batch execution. Again, this is because of the memory-bound nature of the problem. The new generation of memory controllers in Volta provides $1.5x$ delivered memory bandwidth with respect to the Pascal GP100. In this platform, the MS-ID-FFT proposals continue surpassing the CuFFT-based ones, achieving, on average, $1.44x$ and $1.42x$ speedups for the case of real and complex numbers, respectively. In general terms, this architecture performance is the double than the Pascal one, as can be observed. Regarding the Tiling approach, the atomic operations still limit the performance for larger problem sizes in this architecture, although this approach is up to $5.2x$ faster than the *Real-ID* proposal and up to $6.33x$ with respect to the *Real-CuFFT* proposal. In the case of a multi-batch execution, the MS-ID-FFT proposals continue surpassing the CuFFT-based ones, achieving, on average, $1.21x$ and $1.12x$ speedups for the case of real and complex numbers, respectively. Figure 7.14 shows the *Tiling-based* proposal in this platform, which is is up to $3.91x$ faster than the *Real-ID* proposal and up to $7.48x$ with respect to the *Real-CuFFT* proposal.

Figure 7.15 shows the results for double precision in this architecture, when executing a single-batch. In this case, the MS-ID-FFT proposals are competitive



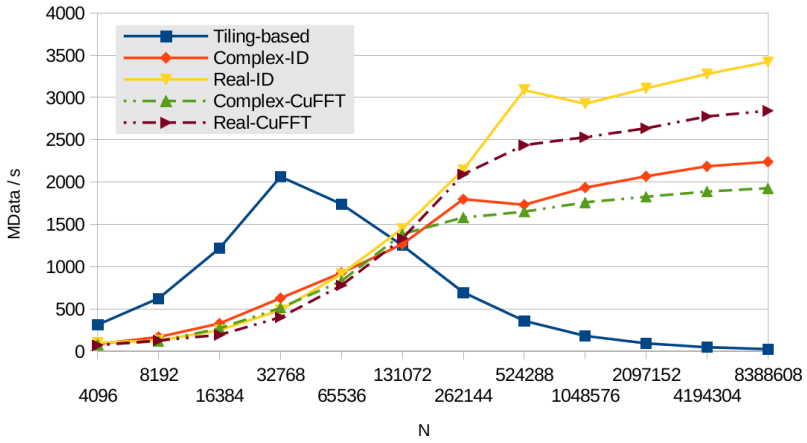
(a) Single-batch execution



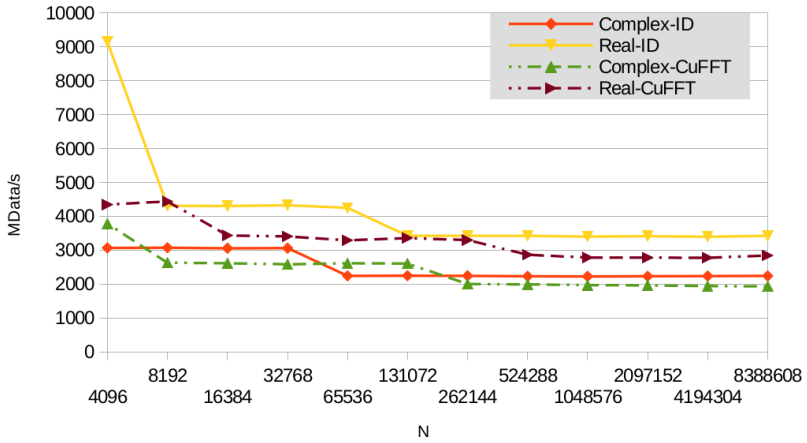
(b) Multi-batch execution

Figure 7.12: Performance comparison for our FP64 approaches in the Pascal Architecture

when $N > 1048576$, whereas the CuFFT-based proposals run faster for smaller problem sizes. In the case of executing several batches, we have obtained inconsistent results for all the versions when compiling with compute capabilities 7.0 and CUDA 9.0. Specifically, extremely low $MData/s$ is achieved when several bidimensional threadblocks write double values in global memory. This issue has been reported, since the performance increases, until reaching expected values, when compiling with other compute capabilities for this architecture.



(a) Single-batch execution



(b) Multi-batch execution

Figure 7.13: Performance comparison for our FP32 approaches in the Volta Architecture

7.6.3. Results Discussion

On the one hand, the FFT-based approaches have some performance weaknesses, in addition to working with precision inaccuracy. Firstly, each invocation of the FFT operation launches several kernels, as well as the kernel invocation for the pairwise multiplication, with the corresponding performance loss. Additionally, the MS-ID-FFT approach consumes a huge amount of shared memory and shuffle instruction optimization is not possible [36], affecting performance massively when working on double precision. Although new GPU architectures have higher theoretical performance and higher memory bandwidth, these two factors significantly limit the actual

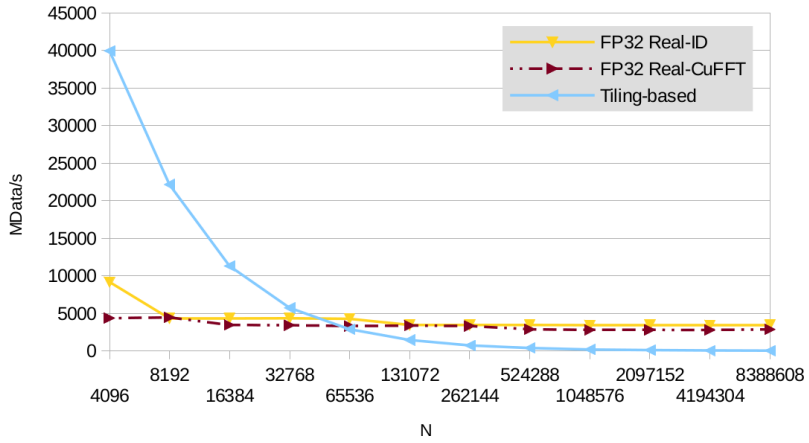


Figure 7.14: Performance comparison for the GPU-tiling proposal, when solving G problems, with respect to FP32 FFT-based proposals on the Volta Architecture

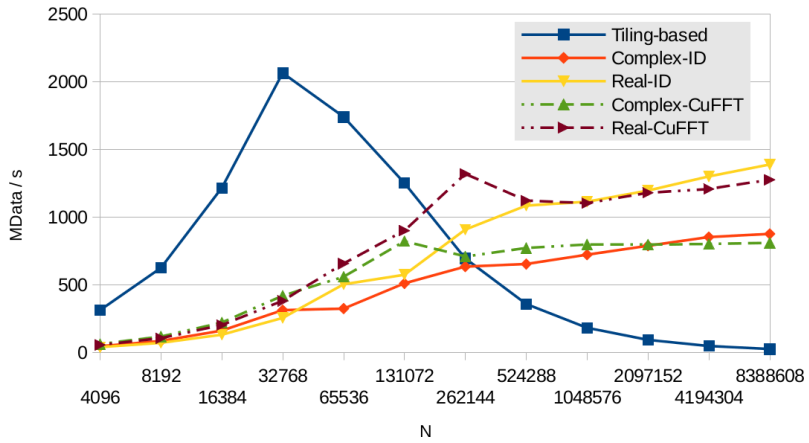


Figure 7.15: Performance comparison for the FP64 FFT-based proposals executing a single-batch in the Volta Architecture

performance that can be achieved. Despite of these aspects, they have been shown to be the most efficient implementation for the large problem sizes.

On the other hand, the Tiling approach performance is excessively dependent on the atomic implementation efficiency, and although the atomic operations can be replaced by a reduce-pattern, this would imply the use of additional kernels and global memory. Specifically, the reduction of each element in a reduce-pattern would need an additional vector in memory with as many elements as the number of threadblocks invoked by the *Tiling-based* kernel. This amount of memory exceeds the memory availability of a single-GPU when solving large problem sizes. In

CUDA 9, it is possible to perform this reduce operator in a single kernel and with no additional memory, using the *gridSynchronize* global barrier. However, the number of invoked threadblocks must be less than or equal to the number of resident active threadblocks, in the same manner as for persistent threads, but the *Tiling-based* kernel uses more than that number of threadblocks. Thus, the current implementation of the Tiling approach is the most efficient we have found.

Thus, analyzing the results obtained, we can conclude that the Tiling approach shows high performance when multiplying a small number of digits, and the FFT-based approaches are the most suitable ones for computing large problem sizes. Specifically for the single-batch execution, the Tiling approach should be used when $N \leq 65536$ for both Pascal and Volta architectures, whereas larger problem sizes should be solved with the FFT-based approaches. In the case of a multi-batch execution, the Tiling approach should be used when $N \leq 32768$. Although the FFT-based approaches suffer from some numerical inaccuracy, this work shows that this is very low and may be acceptable for most of the applications which use the large-integer multiplication. Additionally, this work demonstrates that the FFT-based approaches that use the MS-ID-FFT approach to compute the Fast Fourier Transform run faster than those which use the CuFFT library.

7.7. Conclusions of the Chapter

This chapter provides a performance analysis of the Parallel Prefix proposals developed with our tuning methodology, when they are embedded in real-world application codes; specifically, we test the multiplication of large integers, widely used in cryptography. Two approaches for computing an efficient multiplication of large integers on different GPU architectures are presented: the FFT-based approach, which uses the Strassen-FFT algorithm, and the Tiling approach, which is based on the classical algorithm, partitioning the data vectors in tiles. The FFT-based approach is focused on complex numbers, instead of on a finite-field, and uses a FFT proposal based on our methodology (MS-ID-FFT) to implement the Strassen algorithm. This approach outperforms other implementations which use the CuFFT library by $2.74x$ in Pascal and $1.44x$ in Volta architectures. Additionally, a numerical accuracy analysis is performed, since this FFT-based approach works with

complex numbers rather than finite fields. The results show that the inaccuracy is very low when using FP32 complex numbers and almost non-existent with FP64 complex numbers. The FFT-based approaches are highly suitable when dealing with extreme-large polynomials. Additionally, the flexibility of our methodology allowed to easily develop a Tiling proposal of the classic algorithm. The Tiling approach is extraordinarily efficient when working with small and medium polynomials. For each architecture and execution type(simple-batch or multi-batch), this work also provides the optimal algorithm for each data point. Finally, a parallel implementation for the carry propagation is also given.

Chapter 8

Conclusions and Future Work

This Thesis had two main goals: on the one hand, designing new parallel algorithms, which match well to any programming paradigm, to solve very common computer science operations; on the other hand, developing a general methodology for *NVIDIA GPUs* to solve efficiently many different parallel operations that may be formulated through a parallel prefix algorithm representation. The aim of this methodology is to provide optimized proposals with competitive performance, as well as to facilitate the programmer the development of additional parallel prefix algorithms following the methodology.

Parallel prefix algorithms are a kind of regular parallel algorithm whose communication pattern is static, and does not depend on the runtime. Furthermore, each element is the result of combining the previous result of other elements. Additionally, there is a subset of parallel prefix algorithm, called Index-Digit algorithms, which have special properties. Depending on the category of the algorithm, the proposed methodology is adapted. Specifically, this Thesis has tested the provided methodology on the scan primitive, sorting operators and tridiagonal system solvers.

In this Thesis, we have presented different new parallel prefix algorithms, from an algorithmic formulation, independently of the parallel programming paradigm to be implemented. These algorithms were novelty designed in this Thesis, to the best of our knowledge. Starting with tridiagonal system solvers, we firstly presented the *Redundant Reduction*, a new reduction operation which performs the reduction over a pair of equations, rather than using three equations, impliying fewer accesses

to memory; and solving the substitution phase of the algorithm in one single step. This new operator is combined with two different communication patterns, *Ladner-Fischer* and *Kogge-Stone*, to produce two new tridiagonal system solvers. In addition to these algorithms, we have also presented the *Tree-Partitioning Reduction* which focuses on solving systems of large size. The main problem of solving large problem sizes is to divide the problem into independent slices to be solved simultaneously, since there are dependencies between slices in most of the computing steps. The proposed algorithm solves the system in two phases with no dependencies between slices (just in one step of each phase to unify the problem). We also proposed a new sorting algorithm, the *Bitonic Merge Comb Sort*, an algorithmic variant of the Bitonic Merge Sort. Whereas the classic Bitonic Merge Sort has $log_2 N$ computing steps, where N is the size of the problem, and each computing step has another $log_2 N$ internal steps; the Bitonic Merge Comb Sort reduces the number of steps to have $log_2 N - 1$ computing steps, each with $log_4 N$ internal steps. Additionally, an efficient hand-tuned GPU implementation was given to the Redundant Reduction algorithms (up to $3.25x$ than *CUSPARSE*) and the Bitonic Merge Comb Sort ($10x$ with respect to *CUDPP* and $2.6x$ in contrast to *ModernGPU*); whereas the GPU implementation of the Tree-Partitioning Reduction is later provided under the methodology.

Depending on the size of the dataset to be solved, the methodology was incrementally extended from small datasets to extremely large datasets. Starting from small problem sizes (i.e., datasets that fit in the CUDA shared memory of a single GPU), the proposed methodology first identifies the GPU parameters which influence on performance and declares a set of performance premises to obtain the suitable values for these parameters. The CUDA *kernels* were later implemented with our CUDA *skeletons*, blocks of general, modular and reusable code, which enable the portability and extension of new kernels. Then, the optimal values of the GPU performance parameters were obtained according to the dataset size, the algorithm and the GPU architecture. The methodology for Index-Digit algorithm slightly varied due to the special properties of these algorithms. Although the methodology for ID-algorithms was already proposed for small problem sizes in [8], we extended it for larger sizes and for any parallel prefix algorithm. It should be observed that this methodology focused on solving several *batches* simultaneously. In the case of parallel prefix algorithms, we tested the methodology for three different tridiagonal system solvers (the Cyclic Reduction, the Parallel Cyclic Reduction, and our Redundant Reduction on

the Ladner-Fischer pattern), two scan operators (based on the Ladner-Fischer and Kogge-Stone patterns respectively) and a sorting algorithm (Bitonic Merge Comb Sort).

Regarding the tridiagonal system solvers with small problem sizes, each reduction of the Cyclic Reduction algorithm works with three equations, resulting in a huge use of memory bandwidth. Additionally, the existence of two phases with several computing steps to solve the problem slows down the execution time in comparison to other algorithms in Kepler architecture. However, the usage of our methodology, which optimizes the global memory access patterns as well as the communication between steps with *shuffle* instructions, provided competitive results. This is a memory-bound problem; thus, problem sizes larger than $N \geq 256$ decrease the performance owing to the enormous consumption of shared memory that reduces the number of resident threadblocks. The Parallel Cyclic Reduction, despite the fact that the substitution phase is performed in only one step, produced similar conclusions owing to a similar structure with Cyclic Reduction. In order to demonstrate the effectiveness of our methodology, our proposals were up to $13x$ faster than a direct implementation of these algorithms in CUDA, outperforming the state of the art. We also studied our Redundant Reduction algorithm with the Ladner-Fischer pattern. This algorithm offers a good trade-off between computation and memory bandwidth for small problem sizes, surpassing previous algorithms. However, the extensive use of shared memory of this algorithm reduces the threadblock occupancy excessively after $N > 128$, being less competitive than Cyclic Reduction and Parallel Cyclic Reduction. Nevertheless, this behavior is reversed with the Maxwell architecture, which increases the amount of shared memory per Streaming Multiprocessor, keeping almost constant the performance rate in the $64 \leq N \leq 1024$ interval, and greatly surpassing the state-of-the-art *CUDPP* and *CUSPARSE*. In order to check the effectiveness of the strategy, both the performance achieved following our methodology and the optimal performance obtained after an exhaustive empirical search were compared, demonstrating the success of our methodology criteria.

Respecting the scan primitive with small problem sizes, the effectiveness of Ladner-Fischer can be extrapolated here. In comparison to tridiagonal systems, here each element occupies much fewer bytes, thus the shared memory is not as limiting as in the tridiagonal system case. Both Ladner-Fischer and Kogge-Stone

patterns obtained a similar performance, despite the fact that Ladner-Fischer keeps the number of active threads constant along steps and produces fewer shared memory bank conflicts. This is easily explained thanks to our methodology and the communication pattern of these algorithms, which allow us to implement them entirely with shuffle instructions, obtaining both an excelent warp and threadblock occupancy, maximizing performance to theoretical peaks in a multi-batch execution for both Kepler and Maxwell. In a multi-batch execution, our proposals surpass all the competition, obtaining several magnitudes of speed-up; and they even outperform *Thrust*, *CUB* and *CUDPP* for a single-batch execution. As in the tridiagonal system case, a comparison between a direct implementation of the operation and the implementation under the methodology is provided. Additionally, the effectiveness of our proposal was tested, comparing the parameter values proposed by the methodology to the ones obtained empirically, demonstrating once again the success of the methodology. With respect to sorting, the Bitonic Merge Comb Sort is implemented under the methodology. As in previous cases, the methodology implementation is compared with both a direct implementation of the algorithm and the state-of-the-art in Kepler and Maxwell architectures, surpassing the state-of-the-art and resulting in highly satisfactory results in the multi-batch execution.

Next, the methodology was extended to medium and large problem sizes that cannot be stored in the shared memory, but which still fit in the global memory of a single GPU, distinguishing between parallel prefix algorithms and Index-Digit algorithms. In both cases, we used a multi-kernel strategy to split and synchronize the computation among threadblocks, which prove to be the most efficient strategy when well-implemented. It was necessary to consider the influence of this partition in the performance premises. In the case of ID-algorithms, we tested the methodology on the Wang&Mou tridiagonal system solver, obtaining very competitive results against *CUSPARSE* on Kepler K20, Kepler K40 and Maxwell architectures, for both single and multi-batch executions. The performance decreases proportionally with the problem size, owing to the features of this algorithm (related with the memory bandwidth and the shared memory consumption), reducing the threadblock occupancy. However, the speed-up achieved against *CUSPARSE* is up to $26.8x$ in Kepler; and $31.8x$ in Maxwell, where there is less penalty due to having more shared memory bytes per Streaming Multiprocessor. Regarding parallel prefix algorithms, the scan primitive with the Ladner-Fischer pattern and the Tree-Partitioning Re-

duction method for solving tridiagonal systems were tested. In the case of the scan, our proposal was not very impressive in the case of a single-GPU on Maxwell, since our methodology is focused on solving multiple batches simultaneously, and there are kernels where the GPU parallelism is underused. When solving several batches, our proposal outperforms several well-known libraries in most cases. With respect to tridiagonal systems, the Tree-Partitioning Reduction algorithm and our methodology design are especially suitable for computing large problem sizes, performing an efficient partition of the problem into slices. Although it has more computing steps and invokes more kernels than the previous Wang&Mou implementation, it improves the coalescence in the access to global memory and reduces the use of shared memory, being especially notable when solving large problem sizes, outperforming the state-of-the-art for both single and multiple batch execution (up to $13.28x$ with respect to *CUSPARSE*). Although the results given so far were tested with floating point simple precision, the flexibility of the kernels developed (thanks to the use of templates) and the established performance premises allow the efficient execution of other datatypes. Specifically, this method was also tested on floating point double precission, demonstrating its competitiveness (up to $7.48x$ with respect to *CUSPARSE*). Additionally, depending on the application in which the solver is executed, the numerical stability may be essential or a minor role. The proposal also allows the user to choose the rate between performance and numerical stability, as the slice size used to partition the problem determines the numerical accuracy of the method. Larger slice sizes imply more equations participating, thus increasing stability. In comparison to other solvers, even when choosing performance over stability, the numerical accuracy provided was quite acceptable.

We then analyzed the extension of the methodology to extremely-large datasets and Multiple-GPU systems, again distinguishing between ID-algorithms and parallel prefix algorithms. To do so, it was necessary to consider the penalty of transferring data between GPUs, and the hardware distribution of the GPUs on the network. We concluded that the fastest topology is the one where the GPUs are connected to the same PCI-e bus, since data move directly between devices without passing through host memory. This is possible thanks to the *Unified Virtual Addressing* (UVA) and the *Peer-to-Peer* API. However, when the system is more complex and GPUs are connected along different PCI-e buses, it is important to distribute the computation among the GPUs which belong to the same PCIe, and then to unify the result in

the minimum number of steps possible. Otherwise, there are two options: if the GPUs belong to the same Node, it is possible to use transfers through host memory; or, if they belong to different computing nodes, MPI must be employed. Although MPI CUDA-aware avoids copies through host memory, there is a large overhead associated to MPI initialization. However, when the problem size is sufficiently large, this option was empirically preferable rather than copies through host memory in the same computing node. As ID-algorithm, we used the Wang&Mou solver. As each element is an equation composed of 4 values, there is a huge overhead atached to transferring elements between GPUs. Thus, for this case, the best idea is to distribute batches among GPUs, where each GPU computes entire problems, scaling very well and achieving up to $6.11x$ with respect to *CUSPARSE*. As a parallel prefix algorithm, we chose the scan primitive with the Ladner-Fischer pattern to test the methodology. The pattern and features of scan allow us to reduce the number of transfers between GPUs, providing two interesting approaches: one is to distribute the batches between GPUs, as in the Wang&Mou case, and the other is to compute one problem between several GPUs. Our scan proposal was compared against *CUDPP, Thrust, ModernGPU, LightScan* and *CUB* library, surpassing all of them for single and multi-batch executions.

Finally, an accelerated library, which is built with the implementations proposed, was designed to efficiently solve the operations addressed in this work. In order to demonstrate the efficiency of the methodology, we used this library in a real-world application: the multiplication of high-precission integers, which is used in many applications, such as cryptography. There is an algorithm called *Strassen-FFT* which uses the FFT and a normalization to compute the multiplication; thus, we used our library to compute the FFT operations in Pascal and Volta architectures, surpassing other works which follow the same approach. Although the Strassen-FFT strategy is the most common approach when working with GPUs, the flexibility of our methodology allowed us to design, with little effort, an implementation of the classic multiplication algorithm, taking advantage of the new advances of Pascal and Volta, being a novelty on the field. This new approach outperformed any other GPU implementation for medium-large problem sizes.

To sum up, this Thesis presents a general methodology for efficiently developing any parallel prefix algorithm and Index-Digit algorithm for any GPU architecture

and datasize, as demonstrated. In addition to analyzing these algorithms, we have also studied the recent GPU architectures, its memory hierarchy and its new execution model. This work also provides new parallel prefix algorithms that can be implemented in any parallel programming paradigm, not only on a GPU. We also handled the computation and communication of extremely large datasets on Multiple-GPU systems, so important after the incursion of Big Data and the immense amount of data available nowadays. In addition to study these algorithms and the GPU architectures, the proposed methodology provides a set of modular, efficient and reusable blocks of CUDA code to build the implementation of new algorithms with little effort. The operations tested on this work under the proposed methodology have also been gathered in an optimized library, which provide users with an easy way to invoke tuned functions, independently of the problem size and the GPU architecture, as continuously demonstrated throughout the text.

Future Work

Considering the good results achieved, there are many interesting research lines that can benefit from the work conducted in this Thesis as future work.

First, any other parallel prefix algorithm or Index-Digit algorithm can be efficiently implemented under the proposed methodology with little effort, and easily integrated into the provided library. GPUs represent one of the most promising trends for HPC in the near future, as reflected in TOP500, thus many different parallel operations used in science and engineering must benefit from these devices. The methodology provides any programmer with an easy mechanism for developing new parallel operations for these devices. Additionally, the GPU industry is burgeoning, thus it is important to adapt the proposed performance parameter values to the future GPU generations. The modular design of the methodology and the use of templates in the code allow effortless updating.

Secondly, the performance achieved by the proposals were mostly measured in number of data processed by time unit. However, there is a growing trend toward prioritizing the number of data processed by each power unit, seeking power efficiency; thus, it may be interesting to adapt the performance premises to consider

the power efficiency in the calculation of the optimal values.

Next, although the GPUs employed on this Thesis are powerful devices designed for intensive computation, it may be also interesting test our methodology on GPUs attached to less powerful GPUs, such as the ones contained on the *NVIDIA Tegra* system-on-a-chip.

Additionally, the advent of Machine Learning and Artificial Intelligence in recent years also offers an interesting opportunity to ascertain the optimal performance parameter values for each GPU generation in the methodology, rather than using a set of performance premises, in order to compare the training time and the efficiency of the Artificial Intelligence search. Additionally, many Deep-Learning operations, related with the convolution operation, may be improved with our current methodology and CUDA skeletons.

Finally, the *blockchain* technology is a highly interesting candidate for using the results of this Thesis for efficiently solving the hashing algorithms, as demonstrated in the Chapter 7 of this text, and many works can be conducted on the field.

Bibliography

[1] C. Alberto and H. Sato. Linear Performance-Breakdown Model: A Framework for GPU kernel programs performance analysis. *International Journal of Networking and Computing*, 5(1):86–104, 2015.

[2] M. Amini, B. Creusillet, S. Even, and Keryell. Par4all: From convex array regions to heterogeneous computing. In *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC*, 2012.

[3] F. Argüello, D. Heras, M. Bóo, and J. Lamas-Rodríguez. The Split-and-Merge Method in General Purpose Computation on GPUs. *Parallel Computing*, 38(6–7):277 – 288, 2012.

[4] F. Argüello, M. Amor, and E. L. Zapata. Ffts on mesh connected computers. *Parallel Computing*, 22(1):19 – 38, 1996.

[5] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu. An Adaptive Performance Modeling Tool for GPU Architectures. *SIGPLAN Not.*, 45(5):105–114, 2010.

[6] H. Bantikyan. Big Integer Multiplication with CUDA FFT (cuFFT) Library. *International Journal of Innovative Research in Computer and Communication Engineering*, 2(11):6317–6325, 2014.

[7] K. E. Batcher. Sorting networks and their applications. In *Proc. of the 1968 Spring Joint Computer Conference (AFIPS'68)*, pages 307–314, 1968.

[8] J. L. Blanco. *Towards Efficient Exploitation of GPUs: A Methodology for Mapping Index-Digit Algorithms*. PhD thesis, Universidade da Coruña, 2014.

[9] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, 1990.

[10] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997.

[11] S. Boussakta and A. G. J. Holt. Fast multidimensional discrete hartley transform using fermat number transform. *IEEE Electronic Circuits and Systems journal*, 135(6):253–257, 1988.

[12] R. Brent and H. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, 31(3):260–264, 1982.

[13] R. Brooks and J. Matelski. The Dynamics of 2-generator Subgroups of PSL(2,C). In *Proc. of the 1978 Stony Brook Conference. Riemann Surfaces and Related Topics*, volume 97, pages 65–71, 1978.

[14] A. Burnetas, D. Solow, and R. Agarwal. An analysis and implementation of an efficient in-place bucket sort. *Acta Informatica*, 34(9):687–700, 1997.

[15] D. Cederman and P. Tsigas. GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors. *J. Exp. Algorithmics*, 14:4:1.4–4:1.24, Jan. 2010.

[16] L.-W. Chang and W.-W. Hwu. Mapping tridiagonal solvers to linear recurrences. *Technical Report, University of Illinois at Urbana-Champaign*, 2013.

[17] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. Wrox Press Ltd., Birmingham, UK, UK, 1st edition, 2014.

[18] E. Chu and A. George. *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*. Computational Mathematics. Taylor & Francis, 1999.

[19] CodePlex Open Source Project. *CodePlex IntX Library*, 2015. Available at `https://intx.codeplex.com` (Last Access Sept 2018).

[20] N. Cruz-Cortés, E. Ochoa-Jiménez, L. Rivera-Zamarripa, and F. Rodríguez-Henríquez. A GPU Parallel Implementation of the RSA Private Operation. In *High Performance Computing: Third Latin American Conference (CARLA'16)*, pages 188–203, 2017.

[21] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[22] T. T. Dao, J. Kim, S. Seo, B. Egger, and J. Lee. A Performance Model for GPUs with Caches. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):1800–1813, 2015.

[23] A. Davidson and J. D. Owens. Register Packing for Cyclic Reduction: A Case Study. In *Proc. of the Fourth Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-4)*, pages 1–6, 2011.

[24] A. Davidson, Y. Zhang, and J. Owens. An Auto-tuned Method for Solving Large Tridiagonal Systems on the GPU. In *Proc. of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'11)*, pages 956–965, 2011.

[25] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008.

[26] A. P. Diéguez, M. Amor, and R. Doallo. Efficient scan operator methods on a GPU. In *Proc. of the 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'14)*, pages 190–197, 2014.

[27] A. P. Diéguez, M. Amor, and R. Doallo. BPLG-BMCS: GPU-sorting Algorithm Using a Tuning Skeleton Library. *Journal of Supercomputing*, 73(1):4–16, 2017.

[28] A. P. Diéguez, M. Amor, and R. Doallo. Parallel Prefix Operations on GPU: Tridiagonal System Solvers and Scan Operators. *Journal of Supercomputing. Accepted under second revision*, 2018.

[29] A. P. Diéguez, M. Amor, R. Doallo, A. Nukada, and S. Matsuoka. Efficient High-Precision Integer Multiplication on the GPU. *International Journal of High Performance Computing Applications. Submitted*, 2018.

[30] A. P. Diéguez, M. Amor, J. Lobeiras, and R. Doallo. Operator String Algebraic Properties and Usage. *Internal Report at University of A Coruña*, 2016. `http://gac.des.udc.es/~aperezdieguez/AnnexA5.pdf`.

[31] A. P. Diéguez, M. Amor, and R. Doallo. Bs-comb: An efficient sorting algorithm for GPUs. In *Proc. of the 15th International Conference Computational and Mathematical Methods in Science and Engineering (CMMSE'15)*, pages 461–473, 2015.

[32] A. P. Diéguez, M. Amor, and R. Doallo. New Tridiagonal Systems Solvers on GPU Architectures. In *Proc. of the 22nd International Conference on High Performance Computing (HiPC'15)*, pages 85–94, Dec 2015.

[33] A. P. Diéguez, M. Amor, and R. Doallo. A Tuning Strategy for Tridiagonal System Solvers on GPU. In *Proc. of the 18th International Conference Computational and Mathematical Methods in Science and Engineering (CMMSE'18)*, pages 461–473, 2018.

[34] A. P. Diéguez, M. Amor, and R. Doallo. Tree Partitioning Reduction: A New Parallel Partition Method for Solving Tridiagonal Systems. *ACM Transactions on Mathematical Software. Accepted under second revision*, 2018.

[35] A. P. Diéguez, M. Amor, R. Doallo, A. Nukada, and S. Matsuoka. Efficient Solving of Scan Primitive on Multi-GPU Systems. In *Proc. of the 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'18)*, pages 794–803, 2018.

[36] A. P. Diéguez, M. Amor, J. Lobeiras, and R. Doallo. Solving Large Problem Sizes of Index-Digit Algorithms on GPU: FFT and Tridiagonal System Solvers. *IEEE Transactions on Computers*, 67(1):86–101, 2018.

[37] A. P. Diéguez, M. A. López, and R. D. Biempica. Solving Multiple Tridiagonal Systems on a Multi-GPU Platform. In *Proc. of the 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP'18)*, pages 759–763, 2018.

[38] Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd, and N. K. Govindaraju. Auto-tuning of Fast Fourier Transform on Graphics Processors. In *Proc. of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*, pages 257–266, 2011.

[39] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli. Fast scan algorithms on graphics processors. In *Proc. of the 22n Annual International Conference on Supercomputing (ICS'08)*, pages 205–213. ACM, 2008.

[40] E. Dufrechou and P. Ezzatti. A New GPU Algorithm to Compute a Level Set-Based Analysis for the Parallel Solution of Sparse Triangular Systems. In *Proc. of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS'18)*, pages 920–929, 2018.

[41] Ö. Eğecioğlu, E. Gallopoulos, and Ç. Koç. *A Parallel Method for Fast and Practical High-order Newton Interpolation.* Center for Supercomputing Research and Development Urbana, Ill: CSRD report. University of Illinois at Urbana-Champaign, 1989.

[42] A. Emerencia. *Multiplying Huge Integers Using Fourier Transforms.* ICS Project. University of Groningen, 2007.

[43] N. Emmart and C. Weems. High precision integer addition, subtraction and multiplication with a graphics processing unit. *Parallel Processing Letters*, 20(4):293–306, 2010.

[44] J. Enmyren and C. W. Kessler. SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In *Proc. of the Fourth International Workshop on High-level Parallel Programming and Applications (HLPP'10)*, pages 5–14, 2010.

[45] D. Erguiz, E. Dufrechou, and P. Ezzatti. Assessing Sparse Triangular Linear System Solvers on GPUs. In *Proc. of the International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW'17)*, pages 37–42, 2017.

[46] F. Argüello, M. Amor and E.L. Zapata. FFTs on Mesh Connected Computers. *Parallel Computing*, 22(1):19–38, 1996.

[47] Forum, Message. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, USA, 1994. `http:`

//www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail&id=oai%3Ancstrlh%3Autk_cs%3Ancstrl.utk_cs%2F%2FUT-CS-94-230.

[48] M. Frigo and S. G. Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[49] GNU Open Source Project. *GNU Multiple Precision Arithmetic Library*, 2016. Available at `https://gmplib.org` (Last Access Sept 2018).

[50] G. Graefe. Implementing sorting in database systems. *ACM Comput. Surv.*, 38(3), 2006.

[51] K. Gupta, J. Stuart, and J. D. Owens. A study of persistent threads style GPU programming for GPGPU workloads. In *Proc. of Innovative Parallel Computing (InPar'12)*, pages 1–14, 2012.

[52] H.-S. Kim, S. Wu, L.-W. Chang, W.W. Hwu. A Scalable Tridiagonal Solver for GPU. In *Proc. of the Int. Conf. on Parallel Processing (ICPP'11)*, pages 444–453, 2011.

[53] S.-W. Ha and T.-D. Han. A Scalable Work-Efficient and Depth-Optimal Parallel Scan for the GPGPU Environment. *IEEE Transactions on Parallel and Distributed Systems*, 24(12):2324–2333, 2013.

[54] T. Han and D. Carlson. Fast Area-Efficient VLSI Adders. In *Proc. of the Eighth Ann. Sym. Computer Arithmetic*, pages 49–56, 1987.

[55] T. D. Han and T. S. Abdelrahman. hiCUDA: A High-level Directive-based Language for GPU Programming. In *Proc. of the 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2)*, pages 52–61, 2009.

[56] M. Harris, S. Sengupta, and J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3*. Addison Wesley, 2007.

[57] J. Hensley, T. Scheuermann, G. Coombe, M. Singh, and A. Lastra. Fast summed-area table generation and its applications. *Computer Graphics Forum*, 24(3):547–555, 2005.

[58] W. D. Hillis and J. Steele. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[59] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321–, July 1961.

[60] R. Hockney and C. Jesshope. *Parallel Computers 2: Architecture, Programming and Algorithms*. Taylor & Francis, 1988.

[61] R. W. Hockney. A Fast Direct Solution of Poisson's Equation Using Fourier Analysis. *J. ACM*, 12(1):95–113, 1965.

[62] S. Hong and H. Kim. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, 2009.

[63] D. Horn. *Stream Reduction Operations for GPGPU Applications in GPU Gems 2*. Addison-Wesley, 2005.

[64] INL, University of Tennessee. Matrix Algebra on GPU and Multicore architectures (MAGMA), 2015. `http://icl.cs.utk.edu/magma` (Last Access Sept 2018).

[65] Intel Corporation. Intel Math Kernel Library, v10.2. `http://software.intel.com/en-us/articles/intel-mkl/`, 2009. Last Access Sept 2018.

[66] Intel Corporation. Intel Integrated Performance Primitives for Intel Architecture, Reference Manual, 2012.

[67] J.W. Cooley and J.W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965.

[68] Khronos OpenCL Group. *The OpenCL Specification*, 2011.

[69] P. Kipfer and R. Westermann. Improved GPU Sorting. GPU Gems 2-Chapter 46. Adisson Wesley, 2005.

[70] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., 1st edition, 2010.

[71] K. Kitano and N. Fujimoto. Multiple precision integer multiplication on GPUs. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'14)*, pages 1–7, 2014.

[72] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Comput.*, 22(8):786–793, 1973.

[73] J. Kurzak, S. Tomov, and J. Dongarra. Autotuning GEMM Kernels for the Fermi GPU. *IEEE Trans. Parallel Distrib. Syst.*, 23(11):2045–2057, 2012.

[74] L.-W. Chang, J.A. Stratton, H.-S. Kim, W. W. Hwu. A Scalable, Numerically Stable, High-performance Tridiagonal Solver Using GPUs. In *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*, pages 27:1–27:11, 2012.

[75] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, 1980.

[76] S. Lakshmivarahan and K. Dhall. *Parallel Computing Using the Prefix Problem.* Oxford University Press, 1994.

[77] E. László, M. Giles, and J. Appleyard. Manycore algorithms for batch scalar and block tridiagonal solvers. *ACM Trans. Math. Softw.*, 42(4):31:1–31:36, June 2016.

[78] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter. A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves. In *Proc. of the Euro-Par 2016: Parallel Processing*, pages 617–630, 2016.

[79] Y. Liu and S. Aluru. Lightscan: Faster scan primitive on CUDA compatible manycore processors. *Computing Research Repository*, 2016.

[80] J. Lobeiras, M. Amor, and R. Doallo. FFT Implementation on a Streaming Architecture. In *Proc. of the 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'11)*, pages 119–126, 2011.

[81] J. Lobeiras, M. Amor, and R. Doallo. BPLG: A Tuned Butterfly Processing Library for GPU Architectures. *Int. J. Parallel Program.*, 43(6):1078–1102, 2015.

[82] J. Lobeiras, M. Amor, and R. Doallo. Designing Efficient Index-Digit Algorithms for CUDA GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1331–1343, 2016.

[83] E. Manca, A. Manconi, A. Orro, G. Armano, and L. Milanesi. CUDA-quicksort: An Improved GPU-based Implementation of Quicksort. *Concurr. Comput.: Pract. Exper.*, 28(1):21–43, 2016.

[84] M. M. Maza and W. Pan. Fast polynomial multiplication on a GPU. *Journal of Physics: Conference Series*, 256(1):009–012, 2010.

[85] D. Merrill and A. Grimshaw. Parallel scan for stream architectures. In *Technical report*. Dept. of Computer Science, Univ. of Virginia, 2009.

[86] Microsoft. *.NET BigInteger Library*, 2010. Available at `https://msdn.microsoft.com/en-us/library/system.numerics.biginteger(v=vs.110).aspx` (Last Access Sept 2018).

[87] D. Nehab, A. Maximo, R. S. Lima, and H. Hoppe. GPU-efficient recursive filtering and summed-area tables. *ACM Transactions on Graphics (Proceedings of the ACM SIGGRAPH Asia 2011)*, 30(6):176, 2011.

[88] C. Nugteren and H. Corporaal. Bones: An Automatic Skeleton-Based C-to-CUDA Compiler for GPUs. *ACM Trans. Archit. Code Optim.*, 11(4):35:1–35:25, 2014.

[89] A. Nukada and S. Matsuoka. Auto-tuning 3-D FFT Library for CUDA GPUs. In *Proc. of the Conf. on High Perf. Computing Networking, Storage and Analysis (SC'09)*, pages 1–10, 2009.

[90] A. Nukada, K. Sato, and S. Matsuoka. Scalable Multi-GPU 3-D FFT for TSUBAME 2.0 Supercomputer. In *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*, pages 44:1–44:10, 2012.

[91] Nvidia Comp. *Fermi Compute Architecture Whitepaper*, 2009.

[92] Nvidia Comp. NVIDIA CUDA C programming guide, 2010. Available at `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[93] Nvidia Comp. *CUDA Compute Unified Device Architecture*, 2011.

[94] Nvidia Comp. *CUDA CUFFT Library*, 2012. `https://developer.nvidia.com/cufft` (Last Access Sept 2018).

[95] Nvidia Comp. CUDA CUSPARSE Library, 2012. https://developer.nvidia.com/www.nvidia.com/getcuda (Last access Sept 2018).

[96] Nvidia Comp. *NVIDIA Kepler GK110 Architecture Whitepaper*, 2012.

[97] Nvidia Comp. Modern GPU library, 2013. `https://github.com/NVlabs/moderngpu` (Last access Sept 2018).

[98] Nvidia Comp. CUDPP: CUDA Data Parallel Primitives Library, 2014. `http://cudpp.github.io/` (Last access Sept 2018).

[99] Nvidia Comp. *NVIDIA GeForce GTX980 Whitepaper*, 2014.

[100] Nvidia Comp. Cub library, 2015. `http://nvlabs.github.io/cub/` (Last access Sept 2018).

[101] Nvidia Comp. Thrust Library, 2015. `https://github.com/thrust/thrust` (Last access Sept 2018).

[102] Nvidia Comp. *NVIDIA Tesla P100 Whitepaper*, 2016.

[103] Nvidia Comp. *NVIDIA Tesla V100 GPU Architecture Whitepaper*, 2017.

[104] J. Park, G. Bikshandi, K. Vaidyanathan, P. T. P. Tang, P. Dubey, and D. Kim. Tera-scale 1D FFT with Low-communication Algorithm and Intel&Reg Xeon Phi&Trade Coprocessors. In *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'13)*, pages 34:1–34:12, 2013.

[105] J. L. Pey. *Design and Evaluation of Tridiagonal Solvers for Vector and Parallel Computers*. PhD thesis, Universitat Politecnica de Catalunya, 1995.

[106] R. Pietersz and M. Regenmorte. Bridging brownian libor. *Wilmott Magazine*, 18:98–103, 2005.

[107] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[108] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[109] A. H. Sameh and D. J. Kuck. On stable parallel linear system solvers. *J. ACM*, 25(1):81–91, 1978.

[110] N. Satish, M. Harris, and M. Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. In *Proc. of the 2009 IEEE International Symposium on Parallel&Distributed Processing (IPDPS'09)*, pages 1–10, 2009.

[111] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7:281–292, 1971.

[112] S. Sengupta, M. Harris, and M. Garland. Efficient Parallel Scan Algorithms for GPUs. *Technical Report*, 2008.

[113] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan Primitives for GPU Computing. In *Proc. of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware (GH'07)*, pages 97–106, 2007.

[114] S. Sengupta, A. E. Lefohn, and J. D. Owens. A work-efficient step-efficient prefix sum algorithm. *Workshop on Edge Computing Using New Commodity Architectures*, pages 26–27, 2006.

[115] E. Sintorn and U. Assarsson. Fast Parallel GPU-sorting Using a Hybrid Algorithm. *J. Parallel Distrib. Comput.*, 68(10):1381–1388, 2008.

[116] J. Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic Computers*, EC-9(2):226–231, 1960.

[117] S. W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing.* California Technical Publishing, 1997.

[118] M. Steuwer and S. Gorlatch. SkelCL: a high-level extension of OpenCL for multi-GPU systems. *The Journal of Supercomputing*, 69, n.1(1):25–33, 2014.

[119] H. S. Stone. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *J. ACM*, 20(1):27–38, 1973.

[120] D. Takahashi. Implementation of Parallel 1-D FFT on GPU Clusters. In *Proc. of the IEEE 16th International Conference on Computational Science and Engineering (ICCS'13)*, pages 174–180, 2013.

[121] T.G. Stockham. High-speed Convolution and Correlation. In *Proc. of the April 26-28 Spring joint computer conference (AFIPS '66)*, pages 229–233, 1966.

[122] L. H. Thomas. Elliptic Problems in Linear Difference Equations over a Network. *Watson Sci. Comput. Lab. Rep., Columbia University*, 1949.

[123] I. Venetis, A. Kouris, A. Sobczyk, E. Gallopoulos, and A. Sameh. A direct tridiagonal solver based on givens rotations for GPU architectures. *Parallel Computing*, 49:101 – 116, 2015.

[124] B. Vialla and J.-G. Dumas. *LinBox - C++ library for exact, high-performance linear algebra*, 2017. Available at `https://github.com/linbox-team/linbox` (Last Access Sept 2018).

[125] Victor Shoup. *NTL: A Library for doing Number Theory*, 2015. Available at `http://www.shoup.net/ntl/`.

[126] V. Volkov. Better performance at lower occupancy. In *Proceedings of the GPU technology conference, GTC*, volume 10, page 16, 2010.

[127] V. Volkov. Use Registers and Multiple Outputs per Thread on GPU. In *Presentation at the 6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10)*, 2010.

[128] V. Volkov and B. Kazian. Fitting FFT onto the G80 architecture. *Technical Report University of California, Berkeley*, 2011.

[129] C. Wang, S. Chandrasekaran, and B. Chapman. cusFFT: A High-Performance Sparse Fast Fourier Transform Algorithm on GPUs. In *Proc. of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS'16)*, pages 963–972, 2016.

[130] H. S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[131] S. Wienke, P. Springer, C. Terboven, and D. an Mey. OpenACC: First Experiences with Real-world Applications. In *Proc. of the 18th International Conference on Parallel Processing (Euro-Par'12)*, pages 859–870, 2012.

[132] X. Wang and Z.G. Mou. A divide-and-conquer method of solving tridiagonal systems on hypercube massively parallel computers. In *Proc. of the Third IEEE Symposium on Parallel and Distributed Processing (IPDPS'91)*, pages 810–817, 1991.

[133] Y.-Ch. Lin and L.-L. Hung. Fast problem-size-independent parallel prefix circuits. *Journal Parallel Distributed Computing*, pages 382–388, 2009.

[134] Y. Dotsenko, S.S. Baghsorkhi, B. Lloyd and N.K. Govindaraju. Auto-Tuning of Fast Fourier Transform on Graphics Processors. In *Proc. of Principles and Practice of Parallel Programming (PPoPP'11)*, pages 257–266, 2011.

[135] Y. Yang and H. Zhou. CUDA-NP: realizing nested thread-level parallelism in GPGPU applications. In *Proc. of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'14)*, pages 93–106, 2014.

[136] Y. Zhang, J. Cohen, J.D. Owens. Fast Tridiagonal Solvers on the GPU. In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 127–136, 2010.

[137] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. In *Proc. of the Supercomputing '91*, pages 712–721, 1991.

[138] D. Zhao and J. Yu. Efficiently Solving Tri-diagonal System by Chunked Cyclic Reduction and single-GPU Shared Memory. *J. of Supercomputing*, 71(2):369–390, 2015.

[139] K. Zhao. Implementation of Multiple-precision Modular Multiplication on GPU. *Tech. Report*, 2010.

[140] L. Zhao, R. Iyer, S. Makineni, and L. Bhuyan. Anatomy and performance of ssl processing. In *Proc. of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*, pages 197–206, 2005.

Apéndice A

Resumo Estendido en Galego

Nos últimos anos, as tarxetas gráficas, tamén coñecidas como *GPUs* [17], veñen de experimentar un notable incremento no seu uso na computación de altas prestacións (HPC), pois poden operar moito máis rápido que as convecionais CPUs. A *computación paralela* é un tipo de computación no que se realizan moitas operacións simultáneamente, e ten dúas perspectivas, a *arquitectura de computadores* e a *programación paralela*. Dun lado, a arquitectura de computadores, enfocada no aspecto *hardware*, refírese a soportar o paralelismo ao nivel da arquitectura, mentras que a programación paralela, centrada no aspecto *software*, refírese ao uso dos recursos da arquitectura para obter o maior rendemento posíbel.

Desde unha perspectiva de arquitectura de computadores, as GPUs modernas poden executar até miles de tarefas físicas por dispositivo, o que as fai moi óptimas para a computación intensiva de operacións aritméticas, sendo especialmente óptima en algoritmos regulares onde o fluxo de control é reducido, así como para agachar as latencias de execución ao solapar a comunicación coa computación. Este solapamento é posíbel grazas á asignación dun certo número de tarefas lóxicas a cada núcleo.

Desde unha visión de programación, programar nunha CPU ten moitas máis vantaxes. Primero, hai moitas *APIs* que fan sinxela a adaptación dun código secuencial a un código paralelo, tales como *OpenMP* [21], e librerías de programación paralela como *MPI* [47]. Alén do anterior, tamén existe unha gran comunidade de soporte aos linguaxes centrados na CPU, tales como *C++*, *Python* ou *Java*, provendo de ferramentas doadas e potentes aos programadores. Porén, a maioría de linguaxes

GPU de alto nivel son aínda moi recentes, polo que as ferramentas especializadas, APIs ou librerías, son todavía escasas. Amais, a programación GPU tamén está limitada pola complexidade intrínseca do seu *hardware*, onde os programadores teñen que escoller entre algoritmos paralelos que se poidan adaptar á arquitectura, que tamén require de linguaxes especiais tales como *CUDA* [93] e *OpenCL* [68], así como coñecer con profundidade o seu modelo de execución para poder sacarlle rendemento ás execucións.

Aínda que existen moitas propostas para facer máis sinxela a programación GPU, todas elas teñen as súas desvantaxes. *Autotuning* [38] determina a mellor combinación de parámetros que maximizan a métrica de rendemento establecida. Sen embargo, esta técnica require escribir o código dun xeito parametrizado. Outra aproximación é o *uso de directivas* para paralelizar bloques de código, tales como *OpenACC* [131]. Non obstante, a maioría delas obrigan a ter coñecemento avanzado en GPU, o código xerado non é lido de xeito doado e ten certas limitacións, como non poder usar funcións intrínsicas. Os *compiladores automáticos* son outra opción interesante que xera automáticamente código para GPU, como *Par4all* [2]. Nembargantes, estas aproximacións confían no coñecemento do usuario, e moitas das traducións automáticas xeradas poden reducir o rendemento agardado. Finalmente, o uso de *librerías aceleradas*, como *SkePU* [44] ou *MAGMA* [64], permiten aos usuarios obter o máximo rendemento da operación executada para a arquitectura concreta subxacente para a que foi tuneada. Debido á rapida evolución do mercado GPU, cada nova versión varía moito da anterior, mudando o seu deseño considerablemente entre diferentes xeracións, o que fai que os parámetros de tuneado e rendemento tamén teñan que ser reaxustados.

Esta Tese básease principalmente no tuneado dunha librería acelerada, xa que proporciona una implementación óptima das operacións tratadas, independentemente da arquitectura subxacente, o que proporciona xeneralidade, portabilidade, usabilidade e sinxeleza, sendo transparente ao usuario. Exactamente, esta Tese ten dous obxectivos claros. Dunha parte, o deseño de novos algoritmos paralelos que poidan ser executados en calquera paradigma de programación paralela; e, doutra parte, desenvolver unha metodoloxía xeral que permita resolver diferentes operacións paralelas de xeito óptimo para diferentes arquitecturas GPU diferentes.

Concretamente os algoritmos paralelos empregados son os algoritmos de prefixo

paralelo [75], un tipo de algoritmo regular e paralelo cuxo patrón de comunicación é estático, é dicir, non depende da execución. Ademais, cada elemento é o resultado de combinar o resultado previo doutros elementos. Tamén existe un subconxunto destes algoritmos, chamados algoritmos Índice-Díxito (ID) [82] que teñen propiedades especiales, e nos que esta Tese tamén se centra. Dependendo do tipo de algoritmo, a metodoloxía proposta é adaptada. Concretamente, a metodoloxía proposta próbase para as operacións de *scan*, ordenación e resolución de sistemas tridiagonais.

Esta Tese proporciona tres novos algoritmos de prefixo paralelo, desde un punto de vista algorítmico, e independentes do paradigma de programación paralela onde vaian ser implementados. O primeiro de eles chámase *Reducción Redundante* (RR) [32], e é unha operación de reducción para a resolución de sistemas tridiagonais. Esta nova técnica opera sobre un par de elementos, en vez de tres coma outros algoritmos, o que aforra accesos a memoria, substituíndo as incógnitas nun só paso computacional. Esta nova operación combínase con dous patróns de comunicación paralelos diferentes, *Ladner-Fischer* (LF) e *Kogge-Stone* (KS), xerando así dous novos algoritmos de resolución de sistemas tridiagonais. A continuación, tamén presentamos o algoritmo *Reducción Particionada en Árbore* (TPR) [34], outro algoritmo que resolve sistemas tridiagonais. Este novo algoritmo céntrase en sistemas de gran tamaño, xa que o principal problema deles é a dificultade (dependenzas) de partilos en bloques e que cada bloque do problema sexa resolto independentemente. O algoritmo proposto permite resolver o sistema en dúas fases, sen que existan dependenzas entre os bloques (só no último paso computacional de cada fase para unificar o resultado). O derradeiro algoritmo deseñado é referente á ordeación, cuxo nome en inglés é *Bitonic Merge Comb Sort* (BMCS) [31], unha variante algorítmica do *Bitonic Merge Sort* (BMS). O algoritmo BMS ten $log_2 N$ pasos computacionais, onde N é o tamaño do problema, e cada un destes pasos leva asociado outros $log_2 N$ pasos internos. Sen embargo, a nosa versión BMCS resolve o problema en $log_2 N - 1$ pasos con $log_4 N$ etapas internas, reducindo o número total de pasos computacionais. Para o caso de RR e BMCS, proporciónanse unhas implementacións GPU tuneadas a man. Sendo no caso de RR até un $3{,}25x$ máis rápidas que a librería estado-do-arte *CUSPARSE* [95]; e no caso de BMCS até $10x$ con respecto a *CUDPP* [98] e $2{,}6x$ contra *ModernGPU* [97].

A metodoloxía proposta foise incrementando segundo o tamaño do problema a

ser resolto. Comezando por problemas pequenos, é dicir, aqueles que poden ser almacenados na memoria compartida dunha GPU, a metodoloxía proposta identifica os parámetros que inflúen no rendemento, e logo, en base a unhas premisas teóricas, obtéñense os valores axeitados para cada algoritmo, tamaño de problema e arquitectura GPU. A maiores, facilítanse unha serie de *kernels CUDA*, baseados en *CUDA skeletons*, que son bloques de código xeral, modular e reusable, facilitando a portabilidade e a creación de novas implementación a partir deles sen maior esforzo. A metodoloxía para os algoritmos de Índice-Díxito varía mínimamente á proposta para os algoritmos de prefixo paralelo, debido ás propiedades destes. Para estes tamaños de problema, a metodoloxía foi probada en tres algoritmos de resolución de sistemas tridiagonais [33], dous para resolver a primitiva *scan* [28] e outro de ordenación [27]. No caso de sistemas tridiagonais, implementáronse os algoritmos de *Redución Cíclica* (CR) [61], *Redución Cíclica Paralela* (PCR) [60] e o RR-LF, xa presentado. A maiores, tamén se presenta unha implementación directa, sen o uso da metodoloxía, para analizar o bo rendemento desta. Na arquitectura GPU Kepler, os mellores resultados acadados amósanse con RR-LF, aínda que o seu uso excesivo de memoria compartida, o factor limitante da implementación que decrementa o paralelismo, fai que PCR a mellore para grandes tamaños. Na arquitecuta GPU Maxwell, que proporciona máis memoria compartida por *Streaming Multiprocessor*, non se penaliza a implementación RR-LF, sendo a que mellor rendemento obtén. En ambas arquitecturas, a nosa proposta mellora considerablemente o rendemento do estado do arte *CUDPP* [98] e *CUSPARSE* [95]. Así mesmo, adxúntase unha táboa que demostra que os valores óptimos obtidos empíricamente tras unha ardua búsqueda, cadran cos propostos pola metodoloxía. No eido da primitiva *scan*, ambas as dúas propostas baseadas na nosa metodoloxía renden de xeito similar, tamén sobrepasando o estado do arte: *Thrust* [101], *CUDPP* [98] and *CUB* [100] para as arquitecturas GPU Kepler e Maxwell. Finalmente, a proposta de ordeación de BMCS adáptase perfectamente ás arquitecturas Kepler e Maxwell, conseguindo unha aceleración de até $11{,}7x$, $7{,}5x$ e $5{,}3x$ respectivamente para as librerías *CUDPP*, *CUB* e *ModernGPU*.

A continuación, a metodoloxía foi estendida para resolver tamaños de problema de media e larga lonxitude. Estes problemas non caben na memoria compartida dunha GPU pero si na súa memoria global. A metodoloxía distingue neste caso algoritmos de prefixo paralelo e Índice-Díxito, debido ás propiedades especiais destes

últimos. En ambos casos, a principal novidade introducida na implementación dos algoritmos é a necesidade de sincronizar o traballo de diferentes bloques de tarefas, e para iso faise necesario o uso de varios *kernels*. Como distribuir a carga computacional entre os diferentes *kernels* é clave para o rendemento global acadado, polo que introdúcese este suposto nas premisas de rendemento. No caso dos algoritmos Índice-Díxito, a metodoloxía é probada [36] co algoritmo de resolución de sistemas tridiagonais Wang&Mou (WM) [132], que obtén resultados moi competitivos contra a librería *CUSPARSE* nas arquitecturas Kepler K20, Kepler K40 e Maxwell, tanto resolvendo un único problema como varios simultáneamente. Analizando o algoritmo, que require dunha inxente cantidade de transferencia de datos para estes tamaños, o principal problema é o ancho de banda pero, sobre todo, a memoria compartida. Por iso, para tamaños grandes, o paralelismo decrece polo gran uso deste recurso. A aceleración lograda neste caso é de até 26,8x contra *CUSPARSE*. En canto a metodoloxía en algoritmos de prefixo paralelo, probouse sobre a primitiva *scan* co patrón LF e co algoritmo de Reducción Particionada en Árbore (TPR) para sistemas tridiagonais. No caso do *scan*, aínda que para a resolución dun único problema non é tan competitivo, xa que a metodoloxía está enfocada en resolver varios problemas simultáneamente e quedan *kernels* infrautilizados para este caso, si que o é para varios problemas, mellorando na maioría dos casos á competencia. En canto ao TPR e o seu deseño enfocado a minimizar ás dependenzas entre as particións do problema (e bloques de tarefas) consegue unhas aceleracións moi superiores a WM segundo o tamaño do problema vaise facendo maior, pois minimiza o uso de memoria. A flexibilidade na implementación dos nosos *kernels*, co uso de *templates*, permiten a execución de calquera tipo de dato. Os valores óptimos para os parámetros de rendemento, como é obvio, varían co tipo de dato, pero esta característica está soportada nas premisas da metodoloxía, polo que TPR tamén é probado para tipos de coma flotante en dobre precisión (non só en simple precisión, como até agora), obtendo unha aceleración positiva con respecto a *CUSPARSE*, de até o 7,48x. Finalmente, tamén se fixo un estudo sobre a estabilidade numérica do algoritmo, sendo satisfatoria para a maioría de tipos de sistema testados. Así mesmo, facilítaselle ao usuario a posibilidade de escoller o equilibrio entre rendemento e estabilidade numérica para o tipo de execución desexada.

O seguinte paso foi a estensión da metodoloxía para tamaños de problema moi grandes. Neste caso, úsanse sistemas compostos de varias GPUs para distribuir o

problema. O deseño da topoloxía de conexión e rede son considerados nas premisas da metodoloxía para este tipo de problemas. Se as GPUs están conectadas ao mesmo bus *PCI-e*, grazas á tecnoloxía *Unified Virtual Adressing* e a API *Peer-to-Peer*, a transferencia de datos é directa entre GPUs, sen pasar pola memoria do *host*, sendo esta a opción deseada, pois minimiza o sobrecusto de comunicación. Cando non pertencen ao mesmo bus *PCI-e*, se están no mesmo nodo computacional, a comunicación faise a través da memoria do *host*. Se pertencen a diferentes nodos computacionais, o uso da librería *MPI* é necesaria. Para os algortimos Índice-Díxito, probouse o algoritmo de Wang&Mou [37]. Sen embargo, a gran cantidade de datos a ser transferidos polo algoritmo, penaliza seriamente o uso das comunicacións entre GPUs, engadindo un gran sobrecusto ao tempo de execución. Tense que considerar que cada ecuación do algoritmo está formado por catro elementos, e que ca cada elemento necesita de tres ecuacións. Neste caso, para minimizar o número de comunicacións, decidiuse repartir o número de problemas a ser resolto entre as GPUs dispoñíbeis, onde cada problema é resolto por unha soa GPU. Sen embargo, para o caso de algoritmos de prefixo paralelo, usouse o *scan* con LF [35], cuxa estructura, patrón e implementación permiten mover moitos menos datos, resultando eficiente o uso de comunicacións entre GPUs. Neste caso, un mesmo problema pode ser resolto por varias GPUs sen case penalización, onde o rendemento escalou moi ben co número de GPUs participantes. A proposta foi probada tanto en entornos onde as GPUs estaban conectadas ao mesmo bus *PCI-e*, ao mesmo nodo pero con distintos *PCI-e* e incluso entre diferentes nodos utilizando a librería MPI. O resultado acadado foi o esperado, acelerando en varios ordes de magnitude ás librerías *CUDPP* [98], *Thrust* [101], *ModernGPU* [97], *LightScan* [79] e *CUB* [100].

Para rematar, as implementacións tuneadas das operacións tratadas na Tese baixo a metodoloxía proposta son recollidas nunha librería que permite a execución eficiente destas operacións para as arquitecturas referidas. Para demostrar a eficiencia da metodoloxía, probouse dita librería nunha aplicación real: a multiplicación de enteiros de gran precisión, que é moi empregada en diferentes aplicacións como a criptografía. Existe un algoritmo chamado *Strassen-FFT* [111] que usa a transformada rápida de Fourier (FFT), máis un proceso de normalización do resulto, para realizar o cálculo. Para tal, usamos unha implementación FFT da nosa librería nas últimas arquitecturas GPU CUDA, Pascal e Volta, sobrepasando o rendemento doutros traballos que seguiron o mesmo enfoque. Pero ademais, a flexibilidade pro-

porcionada polos bloques de código CUDA desenvolvidos, xunto cos avances destas novas arquitecturas, permitiunos implementar sen esforzo a operación seguindo o algoritmo clásico de multiplicación, unha aproximación totalmente novedosa no eido das GPUs. Dita implementación resultou tremendamente eficiente para tamaños de problema medios [29].