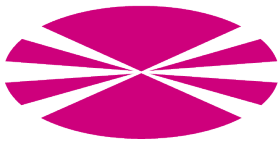
PhD Thesis

Towards Efficient Exploitation of GPUs: A Methodology for Mapping Index-Digit Algorithms

Jacobo Lobeiras Blanco

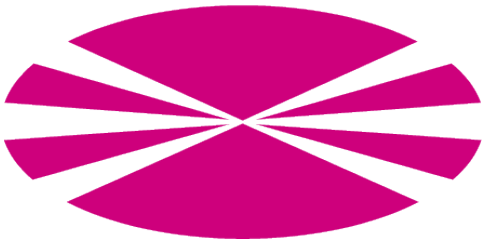
2014



Departamento de Electrónica y Sistemas

Universidade da Coruña

Departamento de Electrónica y Sistemas
Universidade da Coruña



PhD Thesis

Towards Efficient Exploitation of GPUs: A Methodology for Mapping Index-Digit Algorithms

Jacobo Lobeiras Blanco

June 2014

PhD Advisors:
Margarita Amor López
Ramón Doallo Biempica

Dr. Margarita Amor López
Titular de Universidad
Dpto. de Electrónica y Sistemas
Universidade da Coruña

Dr. Ramón Doallo Biempica
Catedrático de Universidad
Dpto. de Electrónica y Sistemas
Universidade da Coruña

CERTIFICAN

Que la memoria titulada "*Towards Efficient Exploitation of GPUs: A Methodology for Mapping Index-Digit Algorithms*" ha sido realizada por D. Jacobo Lobeiras Blanco bajo nuestra dirección en el Departamento de Electrónica y Sistemas de la Universidade da Coruña y concluye la Tesis Doctoral que presenta para optar al grado de Doctor en Ingeniería Informática.

En A Coruña, a de de 2014.

Fdo.: Margarita Amor López
Directora de la Tesis Doctoral

Fdo.: Ramón Doallo Biempica
Director de la Tesis Doctoral

VºBº: Carlos José Escudero Cascón
Director del Dpto. de Electrónica y Sistemas

The Dissertation Committee for Jacobo Lobeiras Blanco certifies that this is the approved version of the following dissertation:

Towards Efficient Exploitation of GPUs: A Methodology for Mapping Index-Digit Algorithms

Committe

_____

President,

_____

Member,

_____

Member,

_____

Member,

_____

Secretary

A mis padres.

Acknowledgements

It has been a long way since I began computer science engineering in the University of A Coruña. I have met a lot of interesting people and learned a lot, not only academically speaking but also in personal experiences. I am grateful to the Electronics and Systems Department for the opportunity to continue my studies as a PhD student. Since my initial steps in parallel computing, I found it very entertaining and developed great interest in the subject. When the first general purpose programming languages for *GPUs* emerged my department offered me the chance to work on a related project, providing all the required resources and tools. Margarita Amor, Manuel Arenaz and Basilio B. Fraguela offered guidance and encouragement during this first stages of my work. The collaboration with Jose A. García from the applied mathematics department was a great opportunity to work in a joint project, parallelizing a shallow water simulation on *CPU* and *GPU* architectures with very successful results. As I advanced in the thesis, the work became more focused on signal processing algorithms for *GPU* architectures, which is my main research topic. I would like to specially thank my PhD advisors Margarita Amor and Ramón Doallo, for their kind support and patience during these last years, because the work has been possible thanks to their guidance and commitment.

During all these years I had a comfortable and stimulating work place, mainly thanks to the nice company and cheerful members of the *GAC* (Computer Architecture Group), specially people at *Lab 0.2*. Many of my colleagues are no longer related to the university or are working abroad, but these words of appreciation are also dedicated to them. Additionally, I would like to thank my friends for their company and entertainment during weekends and holidays, which provided many memorable moments and served to disconnect from work once in a while. Of course I am also in debt with my parents, because I received great and unconditional support every single day and without their help I probably would have not managed to reach so far.

Finally, I would also like to thank the institutions and projects that have funded my work during these years, as well as the Computer Architecture Group and the Electronics and Systems Department for their support because they always did whatever was in their hands when something was needed. This research has been

financially supported by the Ministry of Education and Science and the former Ministry of Science and Innovation of Spain under the projects TIN2007-67537-C03-02 and TIN2010-16735, the Galician Government (Xunta de Galicia) under the Consolidation Program of Competitive Reference Groups Ref. 2010/6, 08TIC001206PR and INCITE08PXIB105161PR, cofunded by FEDER funds. We also would like to thank the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC) and the G-HPC network (ref. 2010/53) for promoting interdisciplinary collaborations between groups of the network.

Jacobo Lobeiras Blanco

Resumen

La computación de propósito general en GPUs supuso un gran paso, llevando la computación de alto rendimiento a los equipos domésticos. Lenguajes de programación de alto nivel como OpenCL y CUDA redujeron en gran medida la complejidad de programación. Sin embargo, para poder explotar totalmente el poder computacional de las GPUs, se requieren algoritmos paralelos especializados. La complejidad en la jerarquía de memoria y su arquitectura masivamente paralela hace que la programación de GPUs sea una tarea compleja incluso para programadores experimentados. Debido a la novedad, las librerías de propósito general son escasas y las versiones paralelas de los algoritmos no siempre están disponibles.

En lugar de centrarnos en la paralelización de algoritmos concretos, en esta tesis proponemos una metodología general aplicable a la mayoría de los problemas de tipo divide y vencerás con una estructura de mariposa que puedan formularse a través de la representación Índice-Dígito. En primer lugar, se analizan los diferentes factores que afectan al rendimiento de la arquitectura de las GPUs. A continuación, estudiamos varias técnicas de optimización y diseñamos una serie de bloques constructivos modulares y reutilizables, que se emplean para crear los diferentes algoritmos. Por último, estudiamos el equilibrio óptimo de los recursos, y usando vectores de mapeo y operadores algebraicos ajustamos los algoritmos para las configuraciones deseadas. A pesar del enfoque centrado en la flexibilidad y la facilidad de programación, las implementaciones resultantes ofrecen un rendimiento muy competitivo, que llega a superar conocidas librerías recientes.

Resumo

A computación de propósito xeral en GPUs supuxo un gran paso, levando a computación de alto rendemento aos equipos domésticos. Linguaxes de programación de alto nivel como OpenCL e CUDA reduciron en boa medida a complexidade da programación. Con todo, para poder aproveitar totalmente o poder computacional das GPUs, requírense algoritmos paralelos especializados. A complexidade na xerarquía de memoria e a súa arquitectura masivamente paralela fai que a programación de GPUs sexa unha tarefa complexa mesmo para programadores experimentados. Debido á novidade, as librarías de propósito xeral son escasas e as versións paralelas dos algoritmos non sempre están dispoñibles.

En lugar de centrarnos na paralelización de algoritmos concretos, nesta tese propoñemos unha metodoloxía xeral aplicable á maioría dos problemas de tipo divide e vencerás cunha estrutura de bolboreta que poidan formularse a través da representación Índice-Díxito. En primeiro lugar, analízanse os diferentes factores que afectan ao rendemento da arquitectura das GPUs. A continuación, estudamos varias técnicas de optimización e deseñamos unha serie de bloques construtivos modulares e reutilizables, que se empregan para crear os diferentes algoritmos. Por último, estudamos o equilibrio óptimo dos recursos, e usando vectores de mapeo e operadores alxébricos axustamos os algoritmos para as configuracións desexadas. A pesar do enfoque centrado na flexibilidade e a facilidade de programación, as implementacións resultantes ofrecen un rendemento moi competitivo, que chega a superar coñecidas librarías recentes.

Abstract

GPU computing supposed a major step forward, bringing high performance computing to commodity hardware. Feature-rich parallel languages like CUDA and OpenCL reduced the programming complexity. However, to fully take advantage of their computing power, specialized parallel algorithms are required. Moreover, the complex GPU memory hierarchy and highly threaded architecture makes programming a difficult task even for experienced programmers. Due to the novelty of GPU programming, common general purpose libraries are scarce and parallel versions of the algorithms are not always readily available.

Instead of focusing in the parallelization of particular algorithms, in this thesis we propose a general methodology applicable to most divide-and-conquer problems with a butterfly structure which can be formulated through the Index-Digit representation. First, we analyze the different performance factors of the GPU architecture. Next, we study several optimization techniques and design a series of modular and reusable building blocks, which will be used to create the different algorithms. Finally, we study the optimal resource balance, and through a mapping vector representation and operator algebra, we tune the algorithms for the desired configurations. Despite the focus on programmability and flexibility, the resulting implementations offer very competitive performance, being able to surpass other well-known state of the art libraries.

Resumen de la Tesis

El hardware especializado de las *GPUs* modernas (*Graphics Processing Units*) es capaz de ofrecer un rendimiento muy superior al de las *CPUs* (*Central Processing Units*) convencionales en muchas aplicaciones paralelas. Las *GPUs* son poderosos procesadores paralelos optimizados para hacer grandes cantidades de operaciones aritméticas, ofreciendo un desempeño especialmente bueno en algoritmos con estructura regular y código con pocos saltos o divergencia durante la ejecución. En general, las *GPUs* tienen un gran número de núcleos de procesado en comparación con las *CPUs*, pudiendo además asignar un cierto número de hilos a cada núcleo, con lo que consiguen reducir los ciclos de procesamiento ociosos a través de técnicas de *multi-threading* para aprovechar más eficientemente su enorme poder computacional.

Desde el punto de vista de la programabilidad, las *CPUs* tienen muchas ventajas sobre las *GPUs*, ya que son mucho más rápidas en algoritmos secuenciales, pueden ser programadas usando lenguajes estándar como *C++* o *Java*, cuentan con potentes herramientas de desarrollo y depuración, y existen *APIs* (Application Programming Interface) para su programación paralelas, tales como *OpenMP* [10] o librerías de programación paralela como *MPI* [106]. La mayoría de los lenguajes de programación para *GPU* normalmente exponen ciertas características y limitaciones del hardware, lo que puede restringir en cierta medida la flexibilidad de los programas en la *GPU* y forzar al programador a adquirir cierto conocimiento sobre la arquitectura para poder aprovechar eficientemente los recursos disponibles.

Puesto que los lenguajes de programación de alto nivel para *GPUs* son relativamente recientes, las herramientas de programación o librerías especializadas todavía no son muy abundantes. Esto puede suponer un problema durante el desarrollo de nuevas aplicaciones, puesto que con frecuencia es requerido implementar todos los módulos auxiliares desde cero. También es necesario considerar que normalmente no es posible portar las librerías existentes directamente a las *GPUs*, ya que a causa de su arquitectura especializada y su modelo más complejo de jerarquía de memoria, normalmente requieren el uso de algoritmos especiales o modificados, además de ajustar el código para evitar ciertos problemas potenciales que afectarían negativamente al rendimiento. Incluso, a causa de la rápida evolución que están experimentado las *GPUs* durante los últimos años, los parámetros de ejecución o los

algoritmo más adecuados para la arquitectura podrían variar de una generación a otra. En resumen, a pesar de los avances actuales en los lenguajes y herramientas para *GPU*, ser capaz de explotar eficientemente su arquitectura paralela es bastante más complejo que la programación de procesadores multi-núcleo convencionales. El desarrollo de algoritmos eficientes puede resultar un desafío incluso para programadores experimentados.

El objetivo de esta tesis es proponer una metodología que pueda ser usada en el desarrollo de aplicaciones paralelas para *GPU*. Más concretamente, nuestro trabajo se ha enfocado en el desarrollo de algoritmos Índice-Dígito para *GPUs*. La notación Índice-Dígito propuesta permite una representación compacta de la asignación de los datos a la jerarquía de memoria de la *GPU*, describiendo las operaciones de reordenamiento de la información en función de una serie de permutaciones comunes en los dígitos que determinan la posición relativa de los elementos. Usando la metodología propuesta es posible diseñar algoritmos tales como la *FFT* (*Fast Fourier Transform*) o algoritmos para la resolución de sistemas tridiagonales, demostrando que nuestro trabajo es capaz de sobrepasar el rendimiento ofrecido por otras librerías eficientes de la bibliografía.

Con el fin de alcanzar nuestro objetivo final, la investigación ha evolucionado a través de cuatro etapas progresivas. Inicialmente estudiamos la paralelización manual de dos aplicaciones: la transformada de *Fourier* y la simulación de aguas superficiales. Estos trabajos fueron usados para probar diferentes estrategias de paralelización y técnicas de optimización. En segundo lugar, analizamos el impacto de rendimiento de distintos factores en relación a la jerarquía de memoria. Usando el conocimiento adquirido, en la tercera etapa desarrollamos una librería para algoritmos representables mediante la notación índice-dígito, basándonos en un conjunto de sencillos bloques constructivos, gracias a los cuales es posible una implementación compacta y flexible. Finalmente, desarrollamos una metodología para el diseño de algoritmos basada en dos etapas, que incluye análisis de los recursos de la *GPU* y manipulación de las cadenas de operadores que definen los algoritmos para ajustarlos a la arquitectura deseada.

La tesis ha sido estructurada en seis capítulos. El primero de los capítulos consiste en una pequeña introducción a la programación de las *GPUs*, describiendo las arquitecturas usadas y los lenguajes más importantes. También se presentan

brevemente los distintos algoritmos que serán usados a lo largo de la tesis:

- La transformada rápida de Fourier [70] o *FFT*, es una operación vital en muchas aplicaciones, como el procesado de imágenes, el filtrado de datos, distintas técnicas de compresión con pérdidas, la resolución de ecuaciones diferenciales parciales o la manipulación de grandes números, entre otros ejemplos.

- La transformada de Fourier real [26], que se trata de una variante especializada de la *FFT* diseñada para trabajar exclusivamente sobre datos reales. Esto es útil en muchos campos como el procesado del audio digital, donde es conocido que la señal de entrada será puramente real.

- La transformada de Hartley [112] es especialmente interesante para el procesado en tiempo real de señales en dispositivos de baja potencia o recursos limitados, como podría ser el caso de los sistemas empotrados o los DSPs (Procesadores Digitales de Señales). Entre sus aplicaciones comunes [6] puede ser usada para realizar convoluciones rápidas, correlaciones de datos o interpolación de señales.

- La DCT [95] o transformada discreta del coseno es un algoritmo ampliamente usado en el ámbito de procesado multimedia y distintos algoritmos de compresión. También trabaja sobre señales reales, y gracias a su capacidad de compactar la energía (parte predominante de las señales para su reconstrucción) es extensamente usada en algoritmos multimedia de compresión con pérdidas [5, 130, 138].

- La resolución de sistemas de ecuaciones tridiagonales [119]. Este tipo de sistemas aparece un muchos problemas científicos como la simulación de fluidos [139], modelos oceánicos [39], o el método *ADI* [14, 51] (*Alternating Direction Implicit*) para la conducción y difusión del calor.

En el segundo capítulo nos centramos en el desarrollo y optimización manual para *GPU* de algoritmos conocidos. A pesar del esfuerzo que requiere diseñar de forma totalmente manual los algoritmos, se trata de una de las formas más comunes al portar algoritmos existentes de *CPU*, especialmente cuando el rendimiento es un

aspecto crítico. Por ejemplo, el procesado de señales normalmente requiere implementaciones muy eficientes, sobre todo si se desea el procesamiento en tiempo real, donde la latencia de respuesta es un factor crítico. Por ello, a causa del coste computacional y el amplio ámbito de aplicación de los algoritmos de procesado de señales, como por ejemplo la transformada de *Fourier*, han motivado que se investigue el desarrollo de implementaciones eficientes para las diversas arquitecturas.

En concreto, en el primer capítulo estudiamos el diseño de una versión optimizada manualmente [55] de la *FFT* para las *GPUs Radeon* de *AMD* usando el lenguaje de programación *Brook+* [7]. La única implementación conocida existente para este tipo de lenguajes de programación *streaming* pertenece al proyecto original BrookGPU [46] de la universidad de Stanford (California), sin embargo su rendimiento en la *GPU* era ligeramente inferior al obtenido por algoritmos recientes de *CPU*. En la segunda parte del capítulo estudiamos la implementación de un simulador de aguas superficiales con contaminante [61,86,87] para *GPU* usando el lenguaje de programación *CUDA* para tarjetas *NVIDIA*. La simulación de este tipo de problemas requiere muchísima capacidad de cómputo, de hecho, la simulación detallada de períodos largos de tiempo sobre grandes superficies podría necesitar varios días o incluso semanas para completarse. En el capítulo se analizan en profundidad varias estrategias de implementación, explicando sus ventajes, desventajas y comparando su eficiencia relativa. Los resultados obtenidos son comparados con una versión paralela desarrollada con *OpenMP* optimizada para procesadores multinúcleo.

El tercer capítulo se centra en el análisis de la arquitectura de las *GPUs* con el fin de proporcionar conocimiento que ayude a comprender y estimar con más precisión distintos factores que afectan al rendimiento [56, 57]. El análisis se centra principalmente en la jerarquía de memoria, estudiando el rendimiento de la memoria global, la memoria de texturas y la memoria compartida en diferentes escenarios. Cuando se procesan grandes problemas en la *GPU*, muchos algoritmos están principalmente limitados por el ancho de banda de la memoria, por lo que en dichos casos es especialmente importante hacer un uso eficiente, optimizando el empleo de la memoria caché en la medida de lo posible. La inclusión de la caché de propósito general $L2$ en la arquitectura de las *GPUs* fue un importante avance, aliviando una serie de restricciones que afectaban al acceso eficiente de los datos. No obstante, incluso esta mejora todavía no es suficiente para ocultar totalmente los requisitos especiales de

localidad espacial por parte de los hilos de las *GPUs* para evitar malgastar ancho de banda durante los accesos. En este capítulo hemos usado la *FFT* como algoritmo para las pruebas, ya que a causa de sus requisitos de ancho de banda, flexibilidad en los patrones de acceso y distribución de los datos, supone una herramienta adecuada para el análisis de rendimiento y el estudio de las distintas estrategias de implementación. El conocimiento adquirido será usado en capítulos posteriores para ajustar el código a las características de la arquitectura y diseñar implementaciones más eficientes.

El cuarto capítulo tiene como objetivo principal el desarrollo de una librería basada en una serie de funciones generales que serán los bloques constructivos de los distintos algoritmos [58], lo que reduce en gran medida la complejidad del código y el tiempo de desarrollo. Más específicamente, el diseño de estos bloques constructivos se basa en el uso de *templates* de *C++* [24]. Esta estrategia de implementación basada en *templates* ha demostrado ser bastante útil en el desarrollo de librerías para *GPU*, por ejemplo en el caso de *Thrust* [96] o *Bolt* [11], otorgando a los programadores herramientas poderosas y flexibles para la implementación de métodos genéricos aplicables en múltiples contextos y trabajando con distintos tipos de dato. Varias técnicas son usadas para generar código estático siempre que sea posible, que puede ser optimizado de forma más eficiente por los compiladores de *GPU*. Adicionalmente, las funciones de la librería propuesta pueden ser ajustadas en función de una serie de factores, como la cantidad de registros, el tamaño de la memoria compartida o el nivel de paralelismo deseado. La librería desarrollada en este capítulo soporta varios algoritmos conocidos y ampliamente usados, como son la *FFT* (tanto en su versión real como compleja), la transformada de *Hartley*, la transformada del coseno o *DCT*, e incluso la resolución de sistemas de ecuaciones tridiagonales. Aunque el enfoque principal de nuestra implementación es la modularidad y la flexibilidad de los algoritmos resultantes, ofrece rendimiento competitivo en comparación con otras librerías de *GPU* reciente y ampliamente extendidas. El diseño de los algoritmos basado en *templates* y la estrategia de optimización aplicada son bastante generales, por lo que puede ser reutilizados en otros trabajos y contextos relacionados que puedan ser representados usando un patrón de comunicación tipo mariposa.

En el quinto capítulo usamos todo el conocimiento previamente adquirido para presentar una metodología basada en dos etapas de desarrollo, que es aplicable para

aquellos algoritmos que puedan ser expresados mediante permutaciones en la representación índice-dígito. En la primera etapa se hace un análisis de recursos para obtienen una serie de factores que caracterizan el comportamiento de la *GPU* con respecto al rendimiento. En la segunda etapa, se usa la manipulación algebraica de cadenas de operadores [30] combinada con una representación de datos basada en un vector de mapeo, que permite el ajuste de las distribuciones de los datos en la jerarquía de memoria de la *GPU*, así como la distribución de los recursos conforme al análisis de recursos realizado en la primera etapa. Las cadenas de operadores permiten representar de forma compacta las secuencias ejecutadas por los algoritmos, y gracias a su manipulación algebraica conforme a una serie de propiedades descritas, es posible diseñar códigos altamente modulares para *GPU*, ajustados para la arquitectura concreta, lo que permite obtener una gran eficiencia en el uso de los recursos hardware disponibles. Nuevamente, nuestro diseño se basa en la programación usando *templates* de *C++* [104], por lo que muchas operaciones pueden ser realizadas en tiempo de compilación, reduciendo el coste de computación asociado a la vez que se minimiza la replicación de código. En concreto, nuestra metodología fue aplicada para el desarrollo de dos algoritmos Índice-Dígito para tarjetas con soporte *CUDA*; un algoritmo para el cálculo de la transformada de *Fourier* (llamado ID-FFT) y un resolutor de sistemas de ecuaciones tridiagonales (llamado ID-TS). Los resultados obtenidos demuestran que con esta metodología es posible sobrepasar el rendimiento de las librerías *CUFFT* [100] y *CUSPARSE* [101] de *NVIDIA*, así como otras propuestas de la bibliografía conocidas por su eficiencia.

Finalmente, en el sexto capítulo se presentan las conclusiones de nuestro trabajo, detallando las contribuciones y logros de mayor interés. También se comentan posibles líneas de investigación y trabajos futuros relacionados con la tesis. Por último, incluimos un listado de las publicaciones de revista y artículos presentados en congresos derivados del trabajo presentado en esta tesis.

Contents

1. Introduction to general purpose GPU programming **33**

1.1. GPU parallel programming 35

1.1.1. General-purpose computing on Radeon GPUs 35

1.1.2. General-purpose computing on GeForce GPUs 37

1.1.3. GPU programming languages 44

1.2. Signal processing algorithms on GPU architectures 49

1.2.1. Fourier Transform 50

1.2.2. Real Fourier Transform 54

1.2.3. Hartley Transform 55

1.2.4. Discrete Cosine Transform 57

1.3. Tridiagonal equation system resolution on GPUs 58

1.3.1. The Wang and Mou tridiagonal algorithm 60

1.4. Work structure summary 63

2. Efficient hand-tuned implementations on a GPU **67**

2.1. FFT processing on a streaming architecture using Brook+ 68

2.1.1. FFT implementation using Brook+ 68

2.1.1.1. Blending/fusion of FFT stages 68

2.1.1.2. FFT mapping techniques 70
2.1.1.3. Scheduling layouts 72
2.1.1.4. Generic performance optimizations 73
2.1.2. Experimental results 75
2.1.2.1. Optimal streaming strategy 77
2.1.2.2. Stage fusion impact 78
2.1.3. Performance comparison with previous work 79
2.2. Shallow Water Simulation 81
2.2.1. Coupled model: 2D shallow water equations with pollutant transport 83
2.2.2. Finite volume numerical scheme. 86
2.2.2.1. Wet-dry fronts 90
2.2.3. Naive GPU solution 92
2.2.4. Optimized GPU solution 95
2.2.4.1. Ghost cell decoupling solution 95
2.2.4.2. Two-phase reduction 99
2.2.4.3. Usage of the texture memory 100
2.2.5. Experimental results 100
2.2.5.1. Simulator accuracy: Comparison with a reference CPU implementation 101
2.2.5.2. Simulator behavior: synthetic test on Ría de Arousa (Spain) 103
2.2.5.3. Isolated impact of the improvements applied 106

3. Influence of memory access patterns to small-scale FFT 109

3.1. FFT benchmarks using CUDA 109

3.1.1. Storage type for thread data 114
3.1.2. Storage type for input data 115
3.1.3. Memory access pattern 115
3.2. Experimental results 116
3.2.1. Cache and ECC configuration 117
3.2.2. Registers vs Shared memory 118
3.2.3. Global memory vs Texture memory 120
3.2.4. Coalescent memory access vs Non-coalescent 122
3.2.5. Comparison with other state-of-the-art implementations . . . 125

4. BPLG: A tuned butterfly processing library for GPU 127
4.1. BPLG basic functions 128
4.1.1. Reordering blocks 129
4.1.2. Computing blocks 131
4.2. Algorithm design based on BPLG 134
4.2.1. Signal processing transforms 135
4.2.2. Tridiagonal system algorithm 137
4.3. Obtaining optimal parallelism 140
4.3.1. Streaming Multiprocessor (SM) parallelism 141
4.3.2. Batch execution in order to increase parallelism 142
4.3.3. Simultaneous block processing optimization 142
4.4. Experimental results 143
4.4.1. Orthogonal signal transforms performance 144
4.4.1.1. Balancing warp and block parallelism 145
4.4.1.2. Complex FFT performance 146

4.4.1.3. Real FFT performance 147
4.4.1.4. Discrete Cosine Transform performance 148
4.4.1.5. Hartley Transform performance 149
4.4.2. Tridiagonal Equations System performance analysis 150
4.4.2.1. Balancing warp and block parallelism 151
4.4.2.2. Tridiagonal system resolution performance 152

5. Efficient Index-Digit Algorithms Design for GPU Architectures 155

5.1. A 2-stage methodology for efficient index-digit algorithms design . . . 156
5.1.1. Applying Mapping Vector Techniques to GPUs 157
5.1.2. Index-digit permutations 159
5.1.3. Operator string algebraic properties 161
5.1.4. Optimized algorithm mapping using operator strings 163
5.2. GPU resources utilization analysis stage 166
5.2.1. Resource utilization analysis for FFT 169
5.2.2. Resource utilization analysis for tridiagonal systems 171
5.3. Operators string manipulation stage 173
5.3.1. The FFT Case 173
5.3.2. The Tridiagonal System case 177
5.4. Algorithm implementation strategies and optimizations 179
5.4.1. Implementation of operators 179
5.4.2. GPU memory access optimization examples 180
5.4.3. Obtaining the code from the operator strings 183
5.5. Experimental results 186
5.5.1. Complex FFT 186

5.5.2. Tridiagonal Equation System 190

6. Conclusions and Future Work **195**

6.1. Future work 198

6.2. Publications from the Thesis 201

References **204**

List of Tables

1.1. Fermi GPUs used in the thesis 42

2.1. L^1 norm at time $T = 1$ s for several meshes. The reference solution is *CPU* sequential 103

2.2. Execution times (in seconds) and speedups 106

2.3. Execution times (in seconds) and speedups after applying each improvement separately 107

3.1. Compiler information for the FFT kernel (Fermi CUDA cap. 2.0) . . 113

4.1. Parameter configuration for the complex FFT algorithm 143

4.2. Description of the test platforms 145

4.3. Impact of the task number for the FFT using Radix-2, Radix-4 and Radix-8 (Platform 1) 146

4.4. Impact of the task number for the FFT using Radix-2, Radix-4 and Radix-8 (Platform 2) 147

4.5. Impact of the task number for tridiagonal systems using Radix-2 and Radix-4 for BPLG-TS (Platform 2) 152

5.1. Resource factors table depending on warps/block 169

5.2. Description of the test platforms 187

5.3. Complex FFT kernel performance and profiler analysis for the different versions (Platform 1) 188

5.4. Tridiagonal system performance (Platform 2) 191

List of Figures

1.1. Evergreen architecture used in the Radeon 5870 38
1.2. Tesla architecture used in the GeForce 280 40
1.3. Fermi architecture used in the GeForce 480 41
1.4. Kepler architecture used in the GeForce Titan 43
1.5. Memory hierarchy used by the CUDA programming model 48
1.6. Examples of FFT algorithms for radix-2 and N=16 53
1.7. Detail of the butterfly operator for the Wang and Mou algorithm . . 61
1.8. Example of the Wang and Mou algorithm for a system with 4 equations 62

2.1. Butterfly blending/fusion in Brook+, N = 16 70
2.2. Brook+ mapping techniques. 71
2.3. Optimal scheduling layout 74
2.4. FFT batch data storage in 2D texture 74
2.5. FFT optimized code example 76
2.6. Butterfly fusion strategy analysis 78
2.7. Butterfly fusion performance 79
2.8. Comparison with CUFFT and SPIRAL 80
2.9. Sketch: pollutant transport. 85

2.10. Finite volume: structured mesh. 87
2.11. Naive algorithm 93
2.12. Recomputation-based solution on a multithreading system 94
2.13. Optimized GPU solution 96
2.14. The two phases of the Ghost cell decoupling solution 98
2.15. Diagram of the academic 2D test used for verification. 102
2.16. Evolution of the academic 2D test used for verification. 103
2.17. Evolution of the Ría de Arousa simulation. 104

3.1. FFT kernel for N=8 111
3.2. General kernel template 112
3.3. Test configuration 114
3.4. Memory access patterns 116
3.5. 48L1 vs 16L1 cache configuration and ECC performance (T2050) . . 118
3.6. RGC vs SGC performance (GF480 & S2050) 119
3.7. RGC vs SGC performance (GF480 & GF280) 119
3.8. RGC vs RTC performance (GF480 & S2050) 121
3.9. RGC vs RTC performance (GF480 & GF280) 121
3.10. RGC vs RGN performance (GF480 & S2050) 123
3.11. RGC vs RGN performance (GF480 & GF280) 123
3.12. SGN vs STN performance (GF480 & S2050) 124
3.13. SGN vs STN performance (GF480 & GF280) 124
3.14. Comparison with other solutions 126

4.1. Classification and module dependences of the building blocks involved in the library. 129

4.2. Template code for the reordering building blocks used by the butterfly transform. 130

4.3. Template code for computing building blocks used by the signal transform algorithms. 132

4.4. Specialized template code for the tridiagonal solver algorithm. 134

4.5. Kernel code for the DCT algorithm 136

4.6. Kernel code for the tridiagonal algorithm 138

4.7. Algorithm performance for the complex FFT algorithm 148

4.8. Algorithm performance for the real FFT algorithm 149

4.9. Algorithm performance for the Discrete Cosine Transform algorithm . 150

4.10. Algorithm performance for the Hartley Transform algorithm 151

4.11. Algorithm performance for tridiagonal equation system resolution . . 153

5.1. Input data mapping on the GPU resources where $r = 2$, $n = 4$, $s = 9$ and $p = 4$ 159

5.2. Simultaneous hardware blocks per SM in Fermi architecture depending on registers per thread 167

5.3. Simultaneous hardware blocks per SM in Fermi architecture depending on the shared memory bytes per thread 168

5.4. Data exchange, from shared memory directly to registers 181

5.5. Problematic global and shared memory access pattern for $n = l = p = r = 2$, $s = 4$ 183

5.6. Coalescent and bank conflict-free access pattern for $n = l = p = r = 2$, $s = 4$ 184

5.7. Example of pseudocode used for the 3D version of the algorithm . . . 185

5.8. Performance comparison of Complex ID-FFT proposal 189

5.9. Comparison of performance of *ID-TS* proposal 192

6.1. Performance comparison: BPLG-cFFT vs ID-FFT 199

6.2. Performance comparison: BPLG-TS vs ID-TS 200

Chapter 1

Introduction to general purpose GPU programming

The specialized hardware design of modern *GPUs* (*Graphics Processing Units*) can perform much faster than normal *CPUs* (*Central Processing Units*) in many general purpose parallel applications. They are powerful parallel processors optimized for intensive arithmetic operations, performing specially well in regular algorithms with reduced flow control. In general, *GPUs* feature a large number of cores in comparison with *CPUs*, moreover, they can map a certain number of threads to each core, reducing idle cycles through multi-threading and exploiting even more effectively their huge computational power.

From a programmability standpoint *CPUs* have many advantages over *GPUs*, as they are much faster in serial algorithms, they can be programmed using standard languages like *C++* or *Java*, there are very powerful tools for software development and debugging, and there are well known parallel programming *APIs* like *OpenMP* [10] or parallel programming libraries like *MPI* [106]. Most *GPU* programming languages often expose hardware features or limitations, which may restrict the flexibility of *GPU* programs and force the programmer to have some knowledge about the hardware to efficiently exploit the *GPU* resources.

As high level *GPU* languages are quite recent, specialized programming tools and libraries are still quite scarce. This poses a problem when developing new applications because it is required to write all necessary auxiliary modules from scratch.

Moreover, it is not possible to port the existing libraries directly to *GPUs*, because their special architecture and more complex memory hierarchy usually requires special or modified algorithms, or at least to tweak the code to avoid some potential performance issues. Due to the fast *GPU* evolution, the execution parameters or the most suited algorithm may even vary from one hardware generation to another. In summary, despite the current advances in *GPU* languages and tools, taking advantage of their parallel architecture is still far more complex than programming standard multi-core *CPUs*. Developing efficient algorithms may be a challenge even for experienced programmers.

The aim of this thesis is to propose a tuning methodology that can be used in the development of *GPU* parallel applications. Specifically, our work has followed the path towards the development of efficient Index-Digit algorithms for *CUDA GPUs*. The Index-Digit notation allows a compact representation of the data mapping, describing the reordering operations according to common permutations in the digits of each element's index, which symbolize the relative position in the data arrays. Using the proposed methodology is possible to design algorithms such as the *FFT* (*Fast Fourier Transform*) or a tridiagonal solver algorithm, showing that our work is able to surpass the performance of *NVIDIA's* libraries and other efficient proposals from the bibliography. To accomplish our objective, the research work has passed four distinct progressive stages. Initially, we studied the manual implementation of two applications, the *Fourier* transform and the parallelization of a shallow water simulator with pollutant transport. This was used to test different parallelization strategies and optimization techniques. Second, we analyzed the performance impact of different factors related to the memory hierarchy. Using the gathered knowledge, in the third stage we developed a library for index-digit algorithms based on a set of simple building blocks that enable a compact and flexible implementation. Finally, a tuning methodology based on 2-stages, resource analysis and operator string manipulation, was developed.

The remaining of this chapter is structured as follows: In Section 1.1 we will make a small introduction to general purpose computing on *GPUs*, explaining the programming languages and the *GPU* architectures used in this thesis. Next, Section 1.2 will do a brief description of the different signal transforms and in Section 1.3 we will explain some basic concepts about the resolution of tridiagonal systems. The

described algorithms will be used several times during the work, therefore it is important to provide some notions about their properties and the associated terminology. Finally, in Section 1.4 we will describe the structure of the thesis, summarizing the different chapters and their objectives.

1.1. GPU parallel programming

Speed of computer processors has quickly increased over time thanks to the reduction of transistor size and evolution of manufacturing technology. Unfortunately, due to thermal constraints and signal integrity issues among other issues, these speed gains have stagnated. To overcome this limit and further improve performance, the focus has changed from serial to parallel processing. Thanks to the interest on computer graphics, many rendering tasks were offloaded to a dedicated processor called the *GPU*.

GPUs are optimized for parallel processing of a function or *kernel* over a domain of elements, being able to efficiently execute a large number of threads in a transparent way, and can hide their own instruction latency by using time multiplexing techniques. Furthermore, *GPUs* hide memory access latencies changing the active threads each time a memory stall occurs.

Despite the advantage in computing power and bandwidth, not all kind of tasks are suited for the *GPU*. For instance, when working with inherently sequential algorithms, programs with very irregular memory access patterns, dynamic algorithms, complex data structures or software that require adaptive refinement, *CPUs* are generally more efficient, as they have comparatively high working frequencies and are capable of extracting a certain instruction parallelism at runtime.

Following we will describe the architecture of the *GPUs* used in this work.

1.1.1. General-purpose computing on Radeon GPUs

In 2007 *AMD* (back then still *ATI*) introduced their first discrete *GPUs* with unified shader architecture. The basic processing units are pipelined *ALUs*, each five of them are grouped in a *SP* (*stream processor*) to handle *VLIW* (*Very Long Instruction Word*) instructions. One of those units is more complex and can handle double precision and transcendental operations.

The *VLIW* design allowed the *GPU* to pack lots of these simple processing units in a small space, easily surpassing the *TFLOP* barrier. Each sixteen *SPs* form a *SIMD* module: these modules are the basic processing blocks of the *GPU* and support multi-threading to hide latencies due to memory accesses and stream operations. All the *SPs* in a *SIMD* module share a common register file (relatively large compared to normal *CPUs*) and must execute the same code path, thus the execution of conditional code or loops is serialized. Each *SIMD* has associated a texture unit and small texture cache. Later, a small memory called *LDS* (local data share) was added to each *SIMD* module, allowing its threads to efficiently exchange data. Regarding the memory, with the exception of some integrated models which rely only on system memory, the *GPU* board usually has several high speed but high latency *GDDR* memory chips connected to a wide memory bus. This is the main memory of the *GPU*, where textures, models or normal data will be stored. It is also possible to access the system memory through the *PCI-Express* bus, however, it can be up to an order of magnitude slower.

Depending on the model, each *GPU* may be composed by one or more *SIMD* modules. In some cases a few modules may be disabled to improve chip yields, specially in less expensive products. The explained architecture remained quite similar until the *GCN* (*Graphics Core Next*) generation presented in 2012. *GCN* drops the *VLIW* configuration in favor of a *RISC SIMD* architecture coupled with a scalar processor. The new architecture is more oriented for general purpose computing and has better resource utilization, specially in those cases where the compiler was not able to obtain independent instructions to fill the *VLIW* slots.

From a programming perspective, threads are executed in groups called *wavefronts* (currently each wavefront has 64 threads). Due to the internal design of the *SIMD* modules, the threads in each wavefront share the same program counter. This greatly reduces branch control hardware, but divergent code execution is serialized, which may lead to inefficient execution where some of the threads in the wavefront

remain idle. The wavefronts are grouped into blocks and each block can be composed of up to 4 wavefronts (256 threads). The execution of a block cannot finalize until the slower thread has finished. The main memory of the *GPU* can be accessed directly or through the texture unit, which provides some additional functions and improved speed when 2D locality is exploited.

In the tests programmed using the *Brook+* language we have used a *Radeon 5870 GPU* as the hardware platform. Figure 1.1 presents a basic diagram of the *Radeon 5870*, which is based on the *Evergreen* architecture. Specifically, *Radeon 5870* has 20 *SIMD* modules, each module has 16 *SPs* and each *SP* is composed by 5 processing elements. In total, each chip has 1600 scalar processing elements providing a maximum theoretical computational power of 2.7 *TFLOPS* in single precision or 0.54 *TFLOPS* in double precision. Each *SIMD* module has associated a 256 *KB* register file, 8 *KB* of *L1* texture cache and 32 *KB* of locally shared memory called *LDS* to enable collaborative work among its threads. The whole chip has another 512 *KB* of *L2* texture cache and 64 *KB* of globally shared memory called *GDS*. The *SIMD* modules do not access the memory directly, memory accesses are managed by dedicated hardware units instead. The *Radeon 5870* has a 256-bit wide memory bus (4×64 bit memory controllers) using *GDDR5* memory which is able to provide up to 153 *GB/s* of bandwidth.

1.1.2. General-purpose computing on GeForce GPUs

At the end of 2006 *NVIDIA* presented their first *GPUs* based on a unified shader architecture. Previously the *GPUs* used separate resources for the pixel and vertex shaders. The specialization of the resources was more efficient and required less area, but in case of load imbalance when rendering the scene, some of the resources will remain idle. With the unified shader architecture the *GPU* just has a single type of generic processors called *SPs* (Streaming Processors), which are dynamically assigned to the required workload. In contrast to the *Radeon architecture* and previous *NVIDIA GPUs* which used vector units, the *SPs* are small scalar processors.

The *SPs* are grouped in *SMs* (Streaming Multiprocessors), which share a control

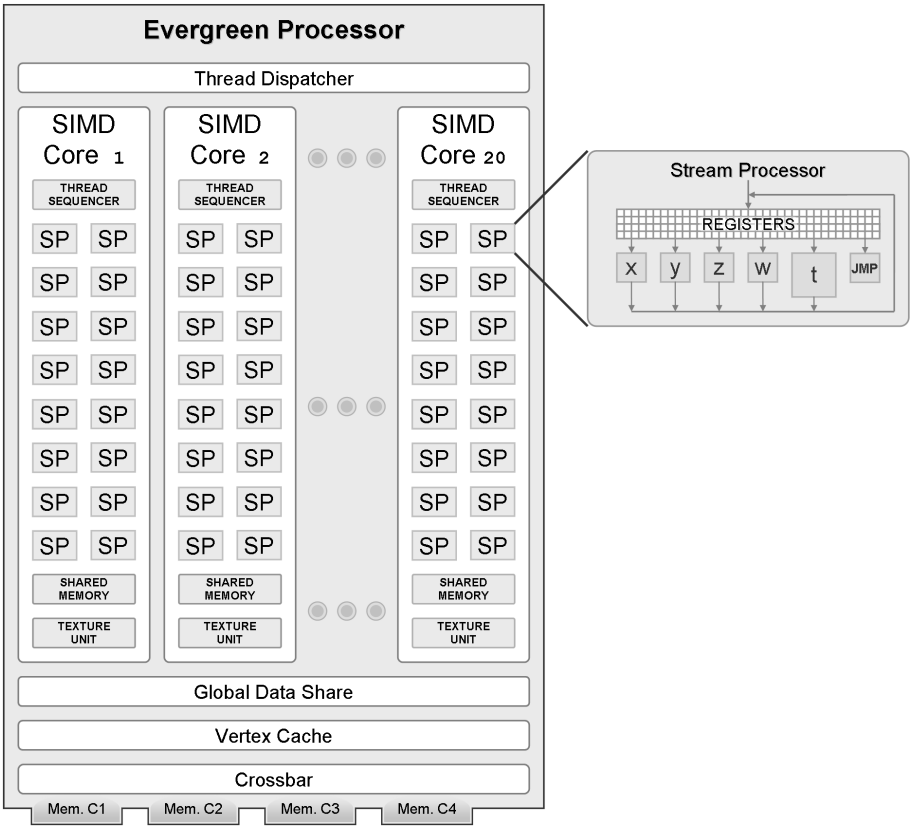


Figure 1.1: Evergreen architecture used in the Radeon 5870

unit and execute instructions in a *SIMD* (*Single Instruction Multiple Data*) fashion. Each *SM* also has a certain number of *SFUs* (*Special Function Units*) to handle more complex operations like transcendental mathematical functions. The *SMs* are grouped in processing clusters, and the number of *SMs* and *SPs* per cluster can vary for each specific model. The total amount of clusters also depends on the particular *GPU*.

Regarding the memory subsystem, each *GPU* has a certain amount of memory mounted in the same board, called the global memory. High-end *GPUs* usually have a very wide memory bus, which combined with fast memory provides high bandwidth. Nonetheless, this memory has an elevated latency that has to be compensated by using a large number of threads, so the *GPU* can switch the active group of threads when a memory stall occurs. *GPUs* commonly store the texture data in the global memory and access it using an spatially coherent access pattern.

To render smooth graphics they sample textures using data spatial interpolation and decompression on the flight. For this purpose, *GPUs* have dedicated hardware called texture units with a small texture cache to improve performance. *GPUs* also have a small constant cache, which is a fast on-die memory that can be read by all the *SPs*, however it cannot be modified by the *GPU* kernels, only by the *CPU*. This constant memory is normally used to store small arrays like filter data, thus saving global memory bandwidth. Inside each *SM* there is a set of common registers and a small amount of shared memory. Both are dynamically allocated based on the kernel requirements, and depending on the amount of resources required by each block, the *SM* may be able to execute several blocks simultaneously. The registers are private to each thread and offer the highest effective bandwidth. The shared memory can be accessed by all the threads within the same block, thus it can be used to communicate data among them.

The *GPU* is connected to the system *CPU* through the *PCI-Express* bus, which does not provide much bandwidth. Thus, memory transfers are expensive and should be minimized. Although the *GPU* is able to directly access *CPU* memory from kernels doing so would heavily reduce the maximum bandwidth, which for many applications already is a performance bottleneck. A more detailed description of *NVIDIA's GPU* architecture can be found in [25].

Tesla architecture

The first tests using the *CUDA SDK* in this thesis were performed in a *GeForce 280 GPU*. This model was launched in 2008 and although the basic architecture remained unchanged from the previous *Tesla* generation, it added some features like double precision and better atomic instruction support. Memory coalescence rules were somewhat relaxed with respect to the initial *Tesla* generation. Now it was easier to obtain higher bandwidth in unaligned or permuted memory access without having to perform data rearrangements in shared memory.

Figure 1.2 presents a diagram of the *Tesla* architecture used by *NVIDIA* in the *GeForce 280 GPU*. This particular model has 10 processing clusters, each one with 3 *SMs* (30 *SMs* in total) and 24 *KB* of texture cache. Inside each *SM* there are 8 *SPs* and 2 *SFUs*. Each *SM* also has a 64 *KB* register file (16384 32-bit registers), 8 *KB* of

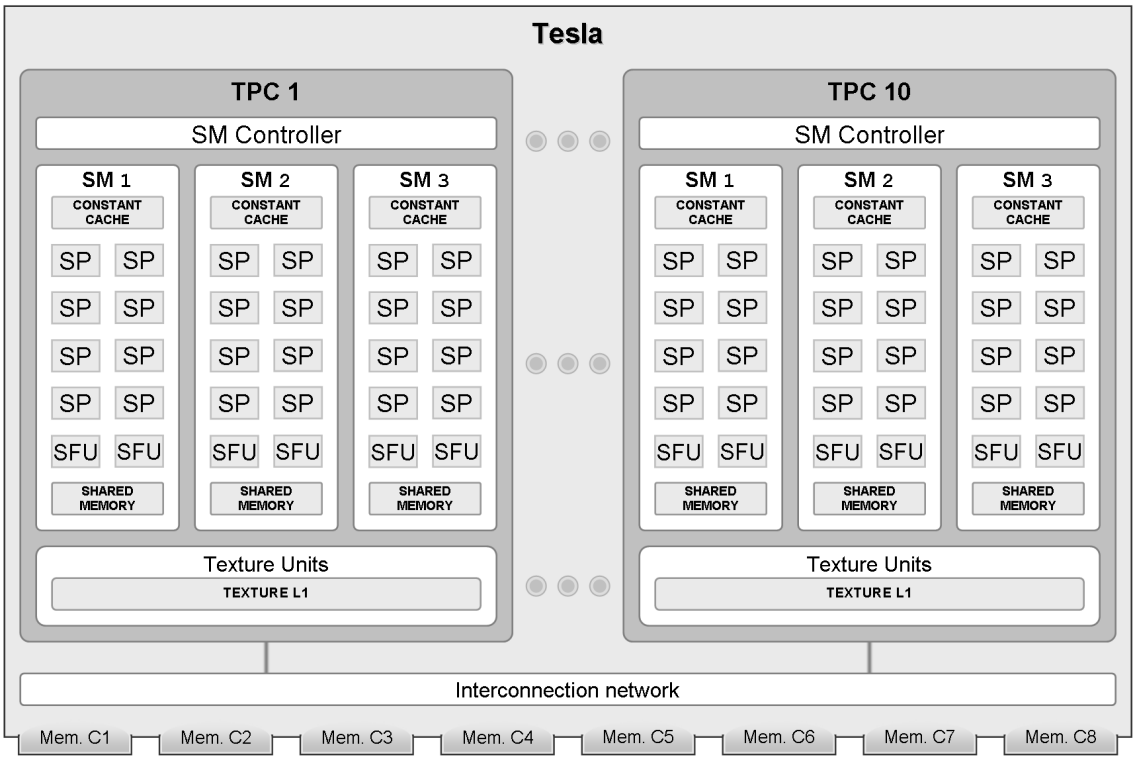


Figure 1.2: Tesla architecture used in the GeForce 280

constant cache and 16 *KB* of shared memory (distributed in 16 banks). The shared memory is a user-managed cache that can be used to store or exchange data within each block, specially useful to distribute work among different threads in computing applications. The whole chip has 256 *KB* of *L2* texture cache distributed among eight memory controllers. The *GeForce 280* has 1 *GB* of *GDDR3* memory on a 512-bit wide memory bus, which provides a maximum bandwidth of 141.7 *GB/s*. Its theoretical computing power is rated by *NVIDIA* at 933 *GFlops* in single precision or 78 *GFlops* in double precision.

Fermi architecture

The *Fermi* architecture was launched in 2010 and included many improvements that supposed a big step forward in performance (like larger shared memory or the addition of global memory cache) and programmability (like printf inside kernels, unified virtual address space for simpler pointers, or stack support for dynamic

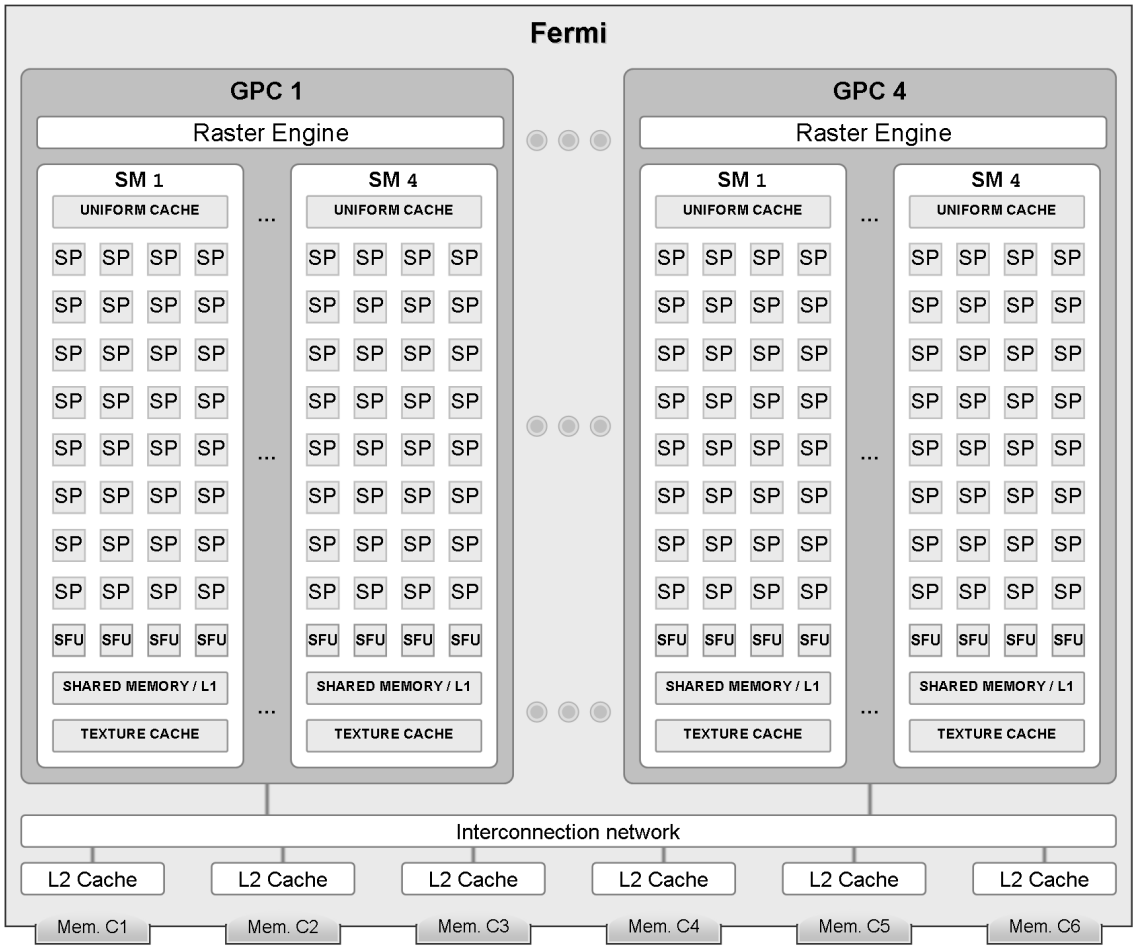


Figure 1.3: Fermi architecture used in the GeForce 480

memory allocation and recursion). In this thesis we will use three *GPUs* based on the *Fermi* architecture, the *GeForce 480* and the *GeForce 580* oriented for the consumer market, and the *Tesla T2050* oriented for the professional market. The architecture is basically the same, however they differ in the memory bandwidth, amount of disabled multiprocessors and special capabilities, like *ECC* support and double precision performance in the professional line.

Figure 1.3 presents a diagram of the *Fermi* architecture used in the three named *GPUs*. It is quite similar to the previous *Tesla* architecture, but the amount of resources was significantly increased. The *GPU* has 4 processing clusters or *GPCs*, with 4 *SMs* per cluster and 32 *SPs* per *SM*. Thus, the *GeForce 580* has $4 \times 4 \times 32 = 512$ *SPs* in total, however only 480 (15 *SMs*) are enabled in the case of the *GeForce*

Table 1.1: Fermi GPUs used in the thesis

	Tesla S2050	GeForce 480	GeForce 580
Enabled SMs	14	15	16
GFLOPs (single)	1030	1345	1581
GFLOPs (double)	515	168	198
Bandwidth (MB/s)	148	177	192

480 and 448 (14 *SMs*) in the case of the *Tesla S2050*. Each *SM* also has 4 *SFUs*, a 128 *KB* register file (32768 registers of 32-bit), 64 *KB* of uniform cache and 64 *KB* of additional cache, which is distributed in 32 banks and can be partially configured as *L1* cache or shared memory. The *L2* texture cache is increased to 768 *KB* and distributed in six *GDDR5* memory controllers (384-bit memory bus). Table 1.1 summarizes the theoretical computing power and maximum memory bandwidth of these three cards.

Compared to the previous *Tesla* architecture used by *NVIDIA*, *Fermi* has a more complex memory hierarchy with the addition of a small *L1* cache to each *SM* and a *L2* cache to each memory controller. In contrast to the texture cache used in the *Tesla* architecture, the global memory cache is now enabled by default for computing tasks. This further reduces coalescence issues and greatly improves effective bandwidth in many scenarios, where data is read from cache without wasting global memory bandwidth. Local memory is also cached, which improves performance in cases like register spilling or dynamic indexing.

Kepler architecture

The *Kepler* architecture was launched in 2012. It has many performance and power efficiency improvements, however it does not introduce as many new features as the *Fermi* architecture. The most significant additions are the shuffle instructions (to exchange data in a warp without using shared memory) and dynamic parallelism (to launch new kernels from a kernel). There are also other features reserved for the professional product line, like *Hyper-Q* (which enables multiple independent work queues) or *GPUDirect* (which allows to perform direct data transfers with other devices with minimal impact on the system *CPU*).

Regarding the architecture it remains quite similar compared to *Fermi*, but with

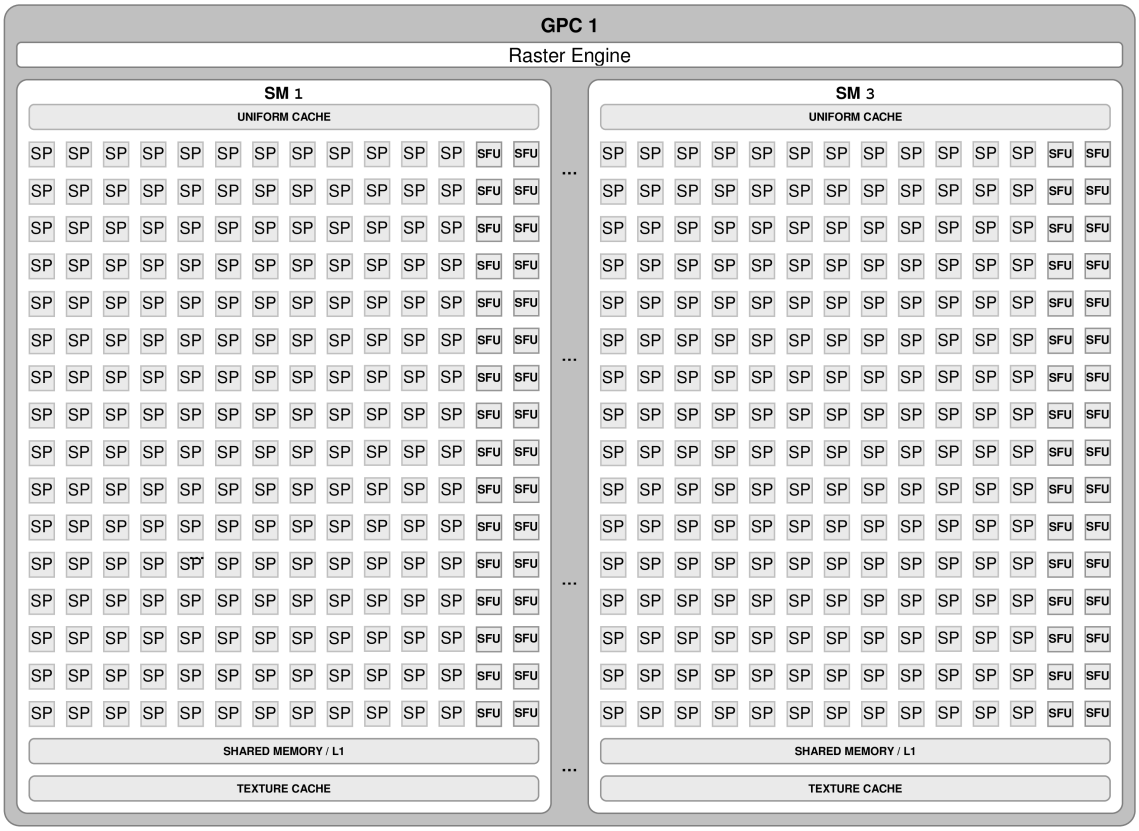


Figure 1.4: Kepler architecture used in the GeForce Titan

much more execution resources. In fact, most changes are at the *SM* level. Figure 1.4 presents a diagram of the new processing clusters used on the *GeForce Titan*, which is the *Kepler GPU* that we will use in some experiments of this thesis. As can be observed in the figure, the *GPU* now has 192 *SPs* and 32 *SFUs* in each *SM*. This does not directly translate in a 6× benefit as one may expect, because the instruction scheduler was simplified and the *SPs* in *Fermi* operated at twice the frequency. However, the *GeForce Titan* has 14 of this *SMs*, $192 \times 14 = 2688$ *SPs* in total, which is more than five times the amount of the *GeForce 580* and can provide up to 4500 *GFLOPs* in single precision or 1270 *GFLOPs* in double precision. The *L2* global cache was increased to 1536 *KB*, and the *GDDR5* memory speed was upgraded, providing 288 *GB/s* over a 384-bit memory bus.

Many supercomputers of the *TOP500* [128] integrate *NVIDIA K20x GPUs* based on the *Kepler* architecture, like the *Titan* [102] (Oak Ridge National Laboratory),

the *Piz Daint* [122] (Swiss National Supercomputing Centre) or the *TSUBAME 2.5* [40] (Tokyo Institute of Technology).

1.1.3. GPU programming languages

The first general programming languages that appeared for *GPUs*, for instance *Cg* [34], were graphics oriented *APIs* that required low level code, exposing graphics pipeline structure and limitations directly to the programmer. Despite the advances in the *GPU* architecture for general purpose computing and the evolution of the compilers, *GPU* programming is still complex, as it requires special languages and dedicated algorithms to exploit the high degree of fine-grained and coarse-grained parallelism.

Recent efforts to standardize the programming of modern *GPUs* have led to the creation of *NVIDIA's CUDA* [98], *DirectCompute* [13] for *DirectX* applications, *C++ AMP* [67], an open specification from *Microsoft*, and *OpenCL* [69], a standard programming language for heterogeneous computing. There are also other interesting proposals for high-level *GPU* programming, like *OpenACC* [103], which is mainly based on code annotated with directives and can target multiple devices, *HMPP* [109], which also supports several devices and provides a superset of *OpenACC* with some additional features or *HPL* [88], a high level *C++* framework that generates *OpenCL* code. Other interesting approaches include *ATI's Brook+* [7], based on a stream programming model, *hiCUDA* [123], an *OpenMP*-like extension of *CUDA* based on compiler directives, or *BSGP* [107], based on the bulk synchronous parallel paradigm that makes extensive use of parallel regions and synchronization constructs. Other related works include *rCUDA* [52], an interesting framework for remote *CUDA* execution and *UPC* (Unified Parallel C) for *GPU* clusters [71].

The programs executed by the *GPU* are usually called kernels (in general purpose computing) or shaders (in graphics). The execution of the each kernel is configured by the programmer and can be distributed among several blocks. However, one hardware limitation is that these blocks cannot directly communicate or synchronize during the kernel execution. Even if more than one block is allowed to write the same memory address, doing so will result in undefined behavior unless the kernel is finalized so the memory is consistent for the next kernel launch. Atomic

instructions may be supported by the hardware, but due to the threading model they cannot be used to synchronize different blocks and its abuse will result in very poor performance. Nonetheless, threads within each block can exchange data using a small shared memory, enabling the collaboration in the same task.

Following we will explain with more detail *Brook+*, *OpenCL* and *CUDA*. They are high-level *GPU* programming languages designed to address a wide range of problems and reduce the development effort, but with different features and philosophy.

Brook+

Brook+ is a *C* language extension for *AMD GPUs* that exposes a stream programming model. In this paradigm the same function (called the *streaming kernel*) is applied to a set of inputs (*input streams*) in parallel, producing another set of outputs (*output streams*). In particular, a thread is created for each output element. The streaming kernel is allowed to read several locations of the input streams but it can only write to one location of each output stream. Thus, the programmer is responsible for writing streaming kernels that are free of race conditions (there should be no data dependencies between the inputs and the outputs of a given *kernel*). There are also random access streams, but this kind of access reduces performance, specially in random write streams. *Brook+* natively supports reduction operations, for example to obtain the maximum or the sum of a vector. The language also supports short *SIMD* vectors, like *float4*, used to operate on several elements of the same data type at once. *Brook+* uses texture memory in order to access input data through *GPU* texture units, a dedicated hardware which provides cached memory access, good *2D* locality or memory access clamping. Although *Brook+* latest version (*v1.4*) permits the utilization of shared memory, it is a beta feature and in our tests it resulted in poor performance or even incorrect results.

Although *AMD* initially promoted *Brook+*, several years later after its introduction *AMD* switched to promote *OpenCL* as their primary programming language for *Radeon GPUs*. *OpenCL* more advanced features and better suitability for *AMD APU* processors that combine a *CPU* and a *GPU* were the reason for *Brook+* abandonment.

OpenCL

OpenCL is a standard language for heterogeneous computing backed by many important manufacturers. One important advantage is that it can be used in other devices than *GPUs*, for instance, in conventional *CPUs* (where it can take advantage of multi-core *CPUs* and *SIMD* instructions) or even in other accelerator devices like *FPGAs*. Nonetheless, even if thanks to the portability of *OpenCL* the same code can run on several devices, it is highly recommended to tune the kernels for the target hardware platform. Some devices offer custom extensions to expose additional functionality supported by the hardware.

AMD was one of the early *OpenCL* adopters, enabling programmers to take advantage of *GPUs* and multi-core *CPUs* to accelerate multimedia and computing applications. One interesting project by *AMD* is the development of *APUs* (*Accelerated Processing Units*), which are the unification of the *CPU* and the *GPU* in a single product. This is an important milestone because the impact of memory transfers is eliminated, therefore enabling the acceleration of many applications that otherwise would not result in any performance benefit. Moreover, *AMD's* commitment to their *OpenCL* implementation brought some features still not supported in the current standard, like *C++* template programming and static object support.

NVIDIA also was one of the early *OpenCL* adopters, which is supported in all *GPUs* based on the *Tesla* architecture or newer. Unfortunately, many features of the recent *GPU* generations are not exposed in *OpenCL*, for instance printf support, shuffle instructions or dynamic parallelism. Moreover, currently only *OpenCL* 1.1 is supported. Profiling and debugging tools are also far behind *CUDA*, which is the main language that *NVIDIA* is trying to push for high performance computing applications in their *GPUs*.

CUDA

Although current *NVIDIA GPUs* support *OpenCL*, *CUDA* is usually the preferred language because it has more advanced features and it is updated regularly by *NVIDIA*. Moreover, *CUDA* runtime management code is usually simpler and implementations typically outperform *OpenCL*. Despite the fact that the language is

only supported by *NVIDIA GPUs*, it has been very successful for high performance computing applications. One of the most interesting features from the programmability standpoint is that *CUDA* offers full *C++* support. Template programming is specially interesting and will be used extensively in this work to obtain easily configurable algorithms with very compact code.

More recently, *NVIDIA* also added support for advanced features like shuffle instructions, dynamic parallelism and unified memory. Shuffle instructions enable the efficient exchange of data among threads from the same *warp* without having to resort to shared memory. Dynamic parallelism enables *GPU* threads to spawn new kernels and results specially interesting for algorithms that require dynamic exploration or refinement. Unified memory facilitates programming by avoiding explicit memory transfers, which can be useful when porting existing code or in the case of complex data structures.

Regarding the *CUDA* programming model, Figure 1.5 presents a diagram that summarizes the work distribution and the memory hierarchy. The programs executed by the *GPU* are called *kernels*. The execution of these *kernels* is assigned by the programmer to one or more computing *blocks* (for instance, in the figure there are two blocks, $[0, 0]$ and $[1, 0]$). The blocks are distributed by the hardware among the available *SMs*, and depending on the amount of required resources, each *SM* may be able to simultaneously execute several blocks thanks to multithreading. Each block is composed by a certain amount of threads, for instance in the figure there are just two threads per block; thread $(0, 0)$ and thread $(1, 0)$. Threads are executed by the hardware in small groups called *warps* (since the first *CUDA* capable *GPUs*, each *warp* is composed by 32 threads). All the threads within a given *warp* share the same program counter, therefore in case of conditional code or loop divergence, the execution is serialized.

As can be seen in Figure 1.5, each thread has associated a certain amount of private registers and local memory. The local memory is also private to each thread and it is used to store dynamically addressed register arrays or data that does not fit in registers. Despite its name, it physically resides in global memory, thus being relatively slow. The shared memory is common to each block and can be used to exchange thread data. Internally, the shared memory is divided in banks and when several threads from the same *warp* try to simultaneously access different shared

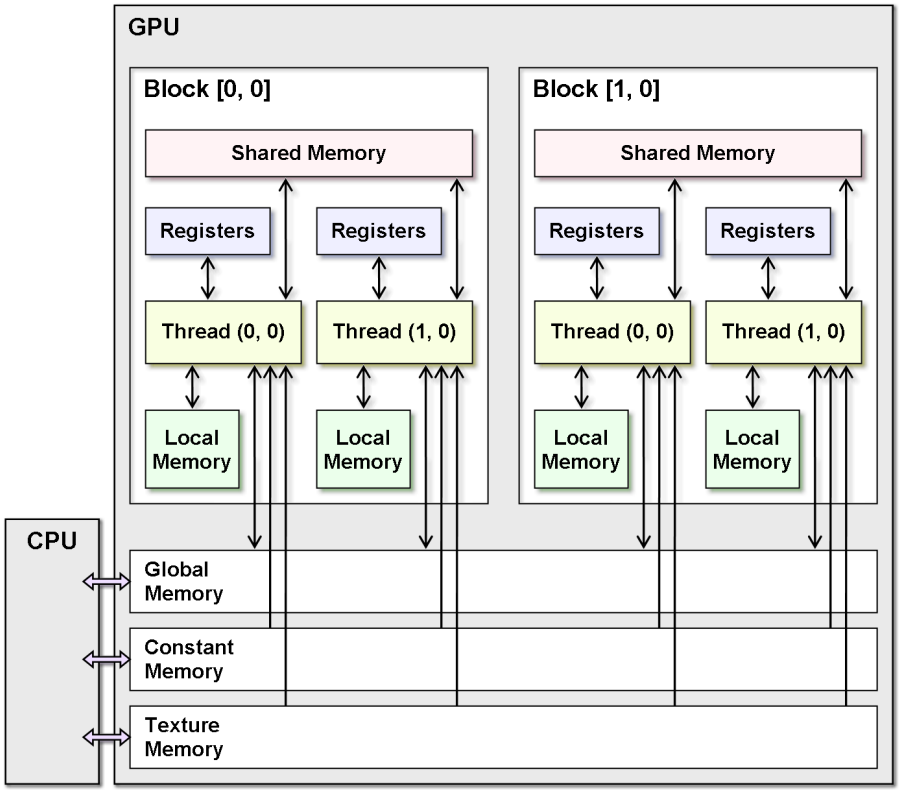


Figure 1.5: Memory hierarchy used by the CUDA programming model

memory locations stored in the same bank, a bank conflict occurs and the access is serialized. In the last level of the hierarchy, threads can only read from constant and texture memory, but they have read/write access to global memory. It is also possible to exchange thread data in global memory, but only at a fraction of the speed compared to shared memory, specially if atomic instructions are used. Execution can be synchronized within each block through barriers, however the only way to perform global synchronization among all the blocks is to finish the kernel execution and launch a new one. To increase efficiency, global memory access is performed in small segments instead of at the word level. For optimal performance, threads should fulfill a series of coalescence rules (that depend on the hardware capabilities) to avoid generating multiple memory requests, thus lowering the maximum attainable bandwidth.

To achieve good efficiency in *CUDA* there are some features that affect performance and need to be considered when designing applications. The main important features influencing *GPU* performance are:

- *Synchronization barrier.* Thread synchronization instructions reduce parallelism and their cost depends on the number of warps within each *CUDA* block. In particular, when using a single warp no synchronization instructions are required, however, for two or more warps the overhead of synchronizing the threads within the block increases. Therefore, the number of synchronization points should be minimized.

- *Available threading. GPUs* largely rely on multi-threading to hide memory access and instruction latency. Hardware resources are shared among threads within the same *SM*: registers and shared memory among others. Thus, there is a trade-off between available threading and shared resources. In the next subsection it will be seen that this can be described by means of what we call resource factors.

- *Coalescence issues in global memory access.* When the threads within a *CUDA* warp access memory locations in different segments several memory requests are generated. Depending on the algorithm and the cache hit rate, the effective memory bandwidth may be greatly reduced.

- *Shared memory bank conflicts.* When several threads within the same *CUDA warp* operate on different memory locations from the same shared memory bank, a bank conflict occurs and the accesses are serialized, thus the total available shared memory bandwidth is proportionally reduced.

1.2. Signal processing algorithms on GPU architectures

Signal processing algorithms are highly versatile and can be used in many areas, for example multimedia processing, data compression, pattern recognition or artificial vision. Most signal processing algorithms can be expressed using a divide and conquer strategy, which enables their parallel execution. Thanks to the advances in *GPU* computing power it is possible to efficiently process large amounts of data,

even enabling realtime processing in many areas were previously was impossible or resulted computationally too expensive. In the next subsection we will describe the *FFT* (*Fast Fourier Transform*), the *Hartley* transform and the *DCT* (*Discrete Cosine Transform*), which will be used in order to check our proposals.

1.2.1. Fourier Transform

The Discrete Fourier Transform [70] (*DFT*) is a very important operation for many applications, such as image and digital signal processing, filtering and compression, partial differential equation resolution or large number manipulation among others.

The equation used to calculate the *DFT* is:

$$y_k = \sum_{i=0}^{N-1} x_i W_N^{ik},\ 0 \leq k < N \tag{1.1}$$

where x is the input signal, y is the output, and $W_N = e^{-j\frac{2\pi}{N}}$ are called twiddle factors and are constants for a given signal of size N. Another common alternative formulation after applying Euler's formula $e^{jx} = cos(x) + j\ sin(x)$ to W_N is:

$$y_k = \sum_{i=0}^{N-1} x_i \left[cos\left(\frac{2\pi}{N} ik\right) - j\ sin\left(\frac{2\pi}{N} ik\right)\right] \tag{1.2}$$

This operation can be easily reversed to obtain the original time domain signal. The equation used to calculate the inverse *DFT* is:

$$x_i = \frac{1}{N} \sum_{k=0}^{N-1} y_k W_N^{-ik},\ 0 \leq i < N \tag{1.3}$$

This simple approach requires a lot of computational power, because the number of operations is proportional to the square of the signal size. The *FFT* reorganizes the computation of the *DFT* such that the transform can be performed in $log_R N$

stages, each one of them calculating N coefficients. A divide-and-conquer strategy is applied to the *DFT* of size $N = R^n$ by dividing the initial data sequence into R subsequences of length N/R (R depends on the used *radix-R* algorithm). Each subsequence is subdivided again into R subsequences repeating the scheme n times until the minimum sequence of size R is obtained. The *DFT* calculation for these small pieces of the signal is simple, and then they can be successively recombined into larger *DFTs* in n steps. Thus, the computational complexity is reduced from $O(N^2)$ to $O(Nlog_R N)$. The *FFT* performance is usually expressed in *GFlops*. The *GFlop* rate of the complex *FFT* can be easily obtained through the commonly used expression [66]:

$$5N \cdot log_2(N) \cdot batch \cdot 10^{-9}/t \ , \tag{1.4}$$

where *batch* is the total amount of signals processed and t is the time in seconds. A similar expression can be used for the real *FFT*:

$$2.5N \cdot log_2(N) \cdot batch \cdot 10^{-9}/t \ . \tag{1.5}$$

Many algorithms are based on a similar divide and conquer strategy as described for the *FFT*, where the main problem is recursively subdivided until reaching the base case or a point that can be easily managed by the threads.

The *FFT* algorithm has two distinct parts, computation and data management, and depending on how the computation and the data flow are organized a large number of algorithms have been designed, for instance, *Cooley-Tukey* [66], *Stockham* [127], *Pease* [89]. There are also other definitions like the Good-Thomas algorithm [47, 74] (based on factorizations), *Bruun*'s [37] algorithm (based on polynomials), *Bluestein*'s [75] and *Rader*'s [17] algorithms (based on convolutions) or Winograd [118] *FFT* (an extension of the *Rader's* algorithm). There are very flexible *CPU* implementations like *SPIRAL* [32,33] that support several algorithms, but the complex or irregular structure of some of them makes their implementation in *GPU* more difficult of inefficient.

The *Cooley-Tukey* algorithm [66] is widely used for *FFT* processing. It recursively re-expresses the *DFT* of an arbitrary composite size $N = N_1 N_2$ in terms of smaller *DFTs* of sizes N_1 and N_2, one over the even-numbered indices $2i$ and the

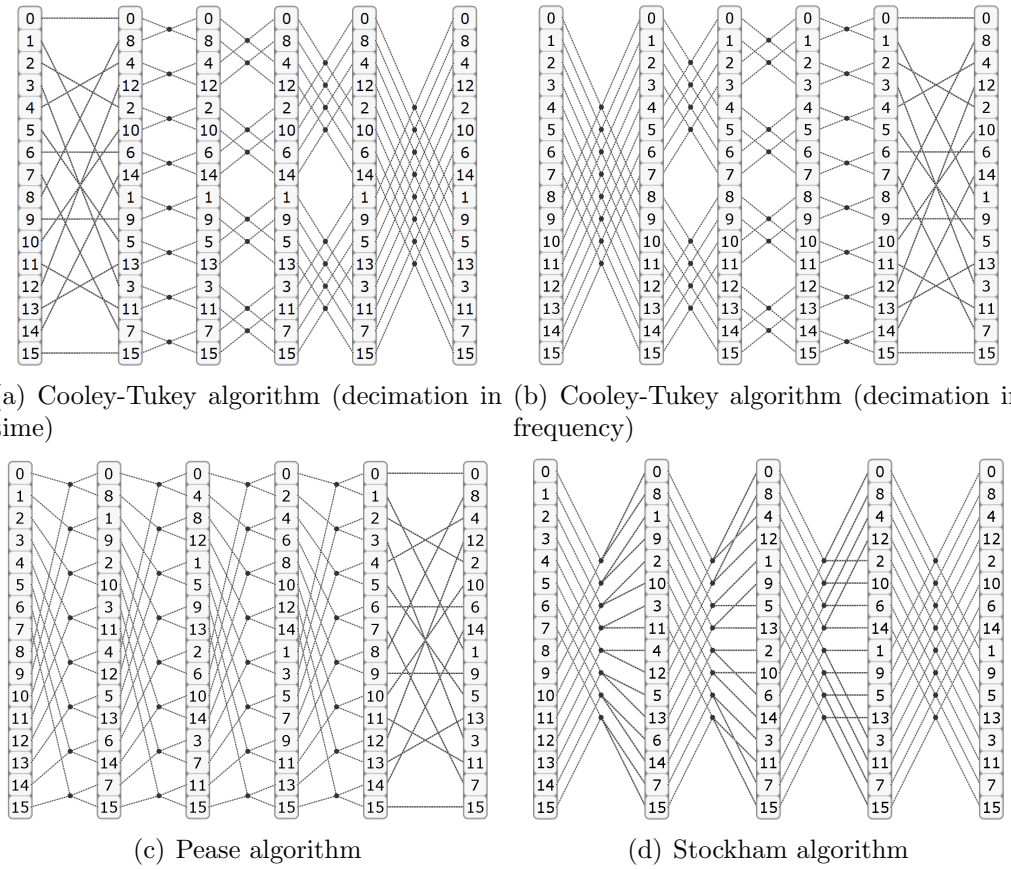
other over the odd-numbered indices $2i + 1$:

$$y_k = \sum_{i=0}^{N/2-1} x_{2i} W_N^{(2i)k} + \sum_{i=0}^{N/2-1} x_{2i+1} W_N^{(2i+1)k}, \ 0 \leq k < N, \tag{1.6}$$

Figure 1.6(a) presents an example of decimation in time (*DIT*) *FFT* with $N = 16$, while Figure 1.6(b) presents an example of decimation in frequency (*DIF*) for the same problem size. Both examples are based on the *Cooley-Tukey* algorithm, that requires an explicit bit-reversal reordering (a kind of permutation where the digits in the binary representation of the sequence index are reversed). If the bit-reversal stage is applied at the beginning this is called decimation in time, whereas if it is applied at the end of the process, it is called decimation in frequency. Due to the bit-reversal stage it may not be the most efficient algorithm, however, it is well-known and easy to implement.

The computations performed by the butterfly stages are represented by small circles, and as we are using the basic radix-2 algorithm, two values are read, operated and written each time. Higher radix versions operate on more values at once, thus reducing the number of stages. For instance, for $N = 16$ a radix-4 algorithm would require just 2 butterfly stages instead of 4. Observe how in Figure 1.6(a) (*DIT*) the bit reversal is followed by 4 stages, while in Figure 1.6(b) (*DIF*) the 4 butterfly stages are executed first followed by the bit reversal exchange. Nonetheless, regardless we use decimation in time or decimation in frequency, the order in which the signals elements are operated is preserved. That means one butterfly (in this case the first) is combining elements 0 and 8, and in the next stage one of the results will be operated with element 4 while the other will be operated with element 12, and so on.

The *Pease* [89] algorithm (see Figure 1.6(c)) also requires an explicit bit-reversal operation like *Cooley-Tukey*, but has the advantage of presenting a constant geometry in the other data reordering stages, therefore it may be more adequate in some architectures like *FPGAs* or *ASICs*. Self-sorting algorithms like *Stockham* [127] provide an output sequence that is digit reversed with respect to the input sequence, so a specific bit reversal stage is not required. Figure 1.6(d) presents an example of the *Stockham* algorithm for $N = 16$. Observe the uniformity of the read stride



(a) Cooley-Tukey algorithm (decimation in time)

(b) Cooley-Tukey algorithm (decimation in frequency)

(c) Pease algorithm

(d) Stockham algorithm

Figure 1.6: Examples of FFT algorithms for radix-2 and N=16

across the stages, which also coincides with the stride of the last write operation. The memory access pattern of the *Stockham* algorithm is usually more efficient on *GPU* architectures.

In addition to the different memory access patterns, there are many *FFT* variants, for instance, implementations optimized for Digital Signal Processors (*DSPs*) and low power embedded systems that use fixed-point arithmetic [36] or integer data [117]. The wide application range of the *FFT* and other related signal processing algorithms has motivated the investigation of efficient implementations that could exploit different hardware features to improve performance, like vector instructions, multi-core processing or hardware coprocessors.

Currently there are several efficient *FFT* proposals for multi-core *CPUs*, for example the implementations included in *Intel*'s *MKL* [48], *IPP* library [49], the

FFTW (*Fastest Fourier Transform in the West*) [78], or the *FFT* from the *Spiral* project [85]. The use of *GPUs* for general purpose computation or *GPU* computing is becoming more interesting due to their great computational power, scalability and low cost, so *GPU FFT* implementations have appeared, like [55], designed for the *Brook+* language, or [133] optimized for *CUDA*. Maybe the most used and well-known *GPU* implementations are *NVIDIA's CUFFT* [100] and *AMD's clAmdFft* [8] included in the *APPML* library. Their main advantage is that they are regularly updated and official support is provided by the hardware manufacturers. However, none of these *GPU* libraries directly supports other transforms like the *DCT* or the *Hartley* transform that are also considered in this thesis.

1.2.2. Real Fourier Transform

There is a specialized variant of the *FFT* algorithm designed to work on real data [26]. This is useful in many fields like audio processing where it is known that the input signal only takes real values. Despite the real input, the required arithmetic operations and the output signal are still complex. Nonetheless, the advantage of using a dedicate real *FFT* algorithm is that the memory usage is reduced in a half, while the processing speed can be nearly doubled.

There are many approaches to perform the real *FFT* efficiently [45]. For instance, it is possible to use the same complex algorithm, but storing another signal in the unused imaginary part. Thus, instead of processing a single signal, we use the same amount of operations to compute the transform of two signals. An additional pass is required to separate both signals. The main disadvantage of this method is that if the signals have very different magnitudes, there may be some numerical instability issues. Another approach, which is the one used in this thesis, is to pack the signal in a vector with half of the size (basically reading each two consecutive real values as a single complex number), and then use a post-processing stage to combine the output and obtain the final result.

The proposed optimization is based on the fact that given a signal of real data

x and its transform y (both of length N) the following symmetry is verified:

$$y_k = \overline{y_{N-k}},\ 1 \leq k \leq N/2 \tag{1.7}$$

Where $\overline{y_{N-k}}$ is the complex conjugate, such that given a complex number $\overline{a + b\,j} = a - b\,j$. The values in the range $\left[y_1 \dots y_{N/2-1}\right]$ can have an imaginary component, however, when N is even, both y_0 and $y_{N/2}$ are pure real numbers (in fact, for convenience in some implementations $y_{N/2}$ is stored in the imaginary part of y_0). Notice that in consequence of this property, half of the information in the transformed signal is redundant. Hereby, if the real signal x is processed as a complex signal of half the length such that:

$$x'_k = x_k + x_{2k+1}\,j \tag{1.8}$$

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

$$y_k = \frac{1}{2}\left(z_k + \overline{z_{N/2-k}}\right) - \frac{j}{2}e^{\frac{-2\pi}{N}k}\left(z_k - \overline{z_{N/2-k}}\right) \tag{1.9}$$

being z the complex transform of the signal x'_k. Finally, due to the periodicity of z the special case $y_{N/2}$ can be computed as follows:

$$y_{N/2} = \frac{1}{2}\left(z_0 + \overline{z_0}\right) + \frac{j}{2}\left(z_0 - \overline{z_0}\right) = Re(z_0) + j\ Im(z_0) \tag{1.10}$$

The remaining values in the range $\left[y_{N/2+1} \dots y_{N-1}\right]$ can be easily obtained using the symmetry enunciated in Expression 1.7.

1.2.3. Hartley Transform

The *Hartley Transform* [112] also operates on real data, but in contrast to the real *Fourier* transform which produces an output with complex data, the result will also be real. Not only half the memory is required, but also some implementations can completely avoid complex arithmetic operations. It is specially interesting for real-time signal processing in low power or resource limited embedded processors and *DSPs*. Among its common applications [6], it can be used to perform fast con-

volution, correlation or signal interpolation. It is also useful in acoustics and speech processing, spectrum analysis, image reconstruction, pattern matching or feature extraction. Other applications include artificial neural networks and biomedical imagery applications.

The Discrete Hartley Transform (DHT) of a real signal x of size N is defined as:

$$h_k = \sum_{i=0}^{N-1} x_i \left[cos\left(\frac{2\pi}{N} ik\right) + sin\left(\frac{2\pi}{N} ik\right)\right] = \sum_{i=0}^{N-1} x_i \left[\sqrt{2}\ cos\left(\frac{2\pi}{N} ik - \frac{\pi}{4}\right)\right] \quad (1.11)$$

The *Hartley* transform is its own inverse, therefore if we apply the DHT formula twice we will obtain the original data (scaled by a constant factor proportional to the signal size). Notice the similarity between Expression 1.11 and Expression 1.2, where the sin term is multiplied by $-j$.

Observe that if computed by the definition given in Expression 1.11, this transform would have $O(N^2)$ complexity, however the computation can be factorized reducing the complexity to $O(N\ log\ N)$. As in the case of the real FFT many fast algorithms have been designed for the *Hartley* transform [44, 68], nonetheless, as far as we know, up to the current date only one work [41] was published about its acceleration on the *GPU*. The implementation approach used in this thesis is to take advantage of the existing relation between the two transforms and compute the DHT using the FFT:

$$y_k = \frac{1}{2}\left(h_k + h_{N-k}\right) - \frac{j}{2}\left(h_k - h_{N-k}\right) \quad (1.12)$$

Where y_k is the real *Fourier* transform and h_k is the result of applying the *Hartley* transform to the real signal x_k. When $N/2 \leq k \leq N-1$ the output signal is defined taking advantage of the symmetry $y_k = \overline{y_{N-k}}$. The explained process can be easily reversed by applying:

$$h_k = Re(y_k) - Im(y_k) \quad (1.13)$$

Remember that the original signal x_k is real, therefore due to the property formulated in Expression 1.7 the transformed signal y_k is symmetric and only half the data needs to be computed. Despite the fact that we would be using complex arith-

metic, the *Hartley* algorithm is computationally light and mainly bandwidth bound in the *GPU*, as it only requires half the data compared to a complex *FFT*.

1.2.4. Discrete Cosine Transform

The Discrete Cosine Transform [95] (*DCT*) is a widely used algorithm for multimedia processing and compression which also works on real data. Thanks to its energy compaction properties it is extensively used in lossy compression algorithms, such as image compression [5, 80] like the *JPEG* image format, audio compression [64, 83, 138] like the *MP3* audio files or video compression [63, 130], such as the different *MPEG* video formats.

Like in the case of the *Hartley* transform, the *DCT* also has a real signal as input and a real signal as output. The *DCT-II* is the most common form used to compute the forward transform. For a real signal x of size N it is defined as follows:

$$y_k = \sum_{i=0}^{N-1} x_i \; cos \left[\frac{\pi}{N} \left(i + \frac{1}{2} \right) k \right] \tag{1.14}$$

The *DCT-III* is commonly used to compute the inverse *DCT*, it is defined as:

$$y_k = \frac{1}{2} x_0 \sum_{i=0}^{N-1} x_i \; cos \left[\frac{\pi}{N} \left(k + \frac{1}{2} \right) i \right] \tag{1.15}$$

Many optimized algorithms were defined to compute the *DCT* [70] reducing the complexity from $O(N^2)$ to $O(N \; log \; N)$. These optimized versions are commonly referred as Fast Cosine Transform (*FCT*) algorithms. As far as we now, only a few implementations were proposed for *CUDA* [84, 105] and *OpenCL* [12, 15]. In our case, due to its simplicity and performance, we decided to use a similar approach to the real *FFT* and the *Hartley* transforms. To compute the *DCT* based on the

complex *FFT* first we generate a sequence x'_k of the form:

$$x'_k = x_{2k}, \quad 0 \leq k < N/2 \tag{1.16}$$

$$x'_k = x_{2(N-i)-1}, \quad N/2 \leq k < N \tag{1.17}$$

Now taking the sequence x'_k as the new input, the *DCT* can be computed as:

$$y_k = Re\left(e^{\frac{-j\pi}{2N}k} z_k\right) \tag{1.18}$$

being z_k the complex transform of the signal x'_k. Notice that as x'_k is real, z_k is symmetric and only half the length needs to be computed due to the property explained in Expression 1.7. This process can be reversed by doing:

$$z_k = Re\left(e^{\frac{-j\pi}{2N}k} y_k\right), \tag{1.19}$$

next, x'_k is recovered doing the inverse *Fourier* transform of z_k and finally:

$$x_{2k} = x'_k, \quad 0 \leq k < N/2 \tag{1.20}$$

$$x_{2k+1} = x'_{N-1-m}, \quad 0 \leq k \leq N/2 \tag{1.21}$$

1.3. Tridiagonal equation system resolution on GPUs

The resolution of tridiagonal equation systems is another interesting problem that also poses a great challenge for efficient execution on GPU architectures. This kind of equations appear in many scientific and engineering problems, like fluid simulation [139], spectral Poisson solvers [113], numerical ocean models [39], preconditioners for iterative linear solvers [2], or the *Alternating Direction Implicit* method [14, 51] for heat conduction and diffusion equations.

There are many sequential algorithms for solving tridiagonal systems, such as Gaussian elimination [73], LU factorization [76] or cyclic reduction [134]. Parallel algorithms have also been developed, such as PARACR [114], recursive doubling [27] or substitution schemes applied to continued fractions [35]. There are other parallel

algorithms based on a divide-and-conquer strategy, like successive doubling [135] or recursive decoupling [38]. There is also a group called hybrid algorithms [16, 27, 115, 120] where the system is partitioned into blocks of equations, using a local algorithm to reduce the subsystem in each block and a global algorithm to solve the reduced system.

Well-known *CPU* parallel implementations of tridiagonal equation solvers include *MKL* [48] and *SPIKE* [28]. There were several efforts to port tridiagonal parallel solvers to *GPUs*, like [97], designed to simulate 3D viscid incompressible fluid with the *ADI* (Alternating Direction Implicit) method, cyclic reduction implementations [19,21], a derived version of the *SPIKE* algorithm [72], and other hybrid algorithms like [42, 54] or [142] (which later was used in the *CUDPP* [18] library). *NVIDIA* also proposed its own tridiagonal solver, included in the *CUSPARSE* [101] library.

Tridiagonal systems are a kind of linear equation systems where only the main diagonal, the diagonal above and the diagonal below contain non zero values. This property allows a more compact data representation and specific algorithms, more efficient than the standard methods used for general linear systems. Tridiagonal systems are composed by a set of N linear equations with N unknowns

$$Au = d, \tag{1.22}$$

where A is a tridiagonal matrix $N \times N$ of the form

$$A = \begin{pmatrix} b_0 & c_0 & & & \\ a_1 & b_1 & c_1 & & \\ & a_2 & b_2 & c_2 & \\ & & \cdots & & \\ & & & \cdots & \\ & & a_{N-2} & b_{N-2} & c_{N-2} \\ & & & a_{N-1} & b_{N-1} \end{pmatrix} \tag{1.23}$$

The i-th initial equation is of the type

$$E_i^0 = \{a_i^0 x_{i-1} + b_i^0 x_i + c_i^0 x_{i+1} = d_i^0\} \tag{1.24}$$

As far as we now, there is no standard formula to measure the performance of the tridiagonal solvers. In this thesis the performance will be measured in million rows per second [72], using the formula:

$$MRows/s = N \cdot batch \cdot 10^{-6}/t \ , \tag{1.25}$$

where N is the number of single-precision equations per tridiagonal system, *batch* is the total amount of problems processed and t is the time in seconds.

1.3.1. The Wang and Mou tridiagonal algorithm

In this thesis we will describe a tridiagonal solver based on the Wang and Mou algorithm [135]. This algorithm offers excellent performance due to its suitability for *GPU* architectures, thanks to the regular structure based on a successive doubling method. The algorithm also offers good numerical stability for diagonal dominant matrices or when no pivoting is needed. Instead of operating directly on signal data the algorithm operates on triads of equations, labeled **L**eft, **C**enter and **R**ight, and each row is represented by a triad:

$$[i]^{t-1} = [\ \underbrace{E^{t-1}_{q\cdot 2^{t-1}}}_{L_i}, \underbrace{E^{t-1}_{i}}_{C_i}, \underbrace{E^{t-1}_{(q+1)2^{t-1}-1}}_{R_i}\ ] \tag{1.26}$$

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

$$E^{t-1}_i = \{a^{t-1}_i x_{q2^{t-1}-1} + b^{t-1}_i x_i + c^{t-1}_i x_{(q+1)2^{t-1}} = d^{t-1}_i\} \tag{1.27}$$

The computation is divided into $log_R N$ stages, operating each butterfly on R elements and following a pattern similar to the decimation in time *Cooley-Tukey*, but excluding the initial bit-reversal stage. Figure 1.7 show details of how each pair of triads ($[i]^{t-1}$ and $[j]^{t-1}$) are combined by the Wang and Mou algorithm. Each circle represents a reduction operation, where one equation is used to exchange one of the unknowns in another equation. First, the last term of equation R_i is used to reduce the first term in the three equations of $[j]$. Next, the middle term of the

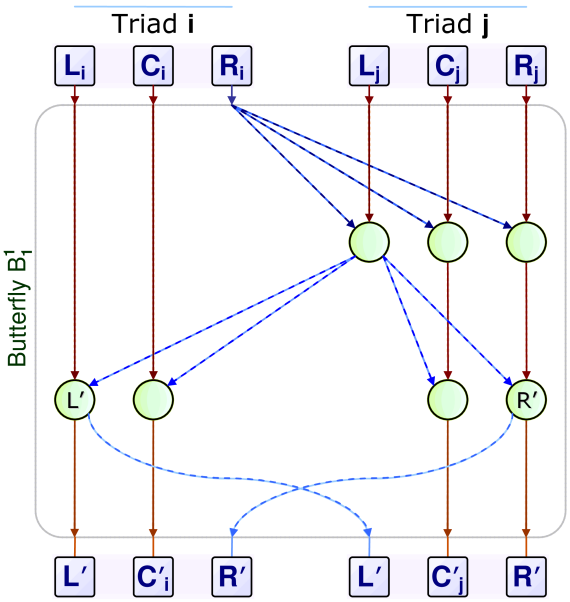


Figure 1.7: Detail of the butterfly operator for the Wang and Mou algorithm

new equation in L_j is used to reduce all the other equations. At the end of the sequence both left equations will be identical (see L'), the same happens with both right equations (see R'). This is the basic computation stage in the case of the tridiagonal solver, but higher radix versions can be used.

Figure 1.8 displays how the algorithm would handle the four equations to obtain the solution. Each box is one triad and the numbers inside the rows represent non-zero coefficients. Only the corresponding subindexs are displayed in the figure. For example, the first triad of Stage 0 is:

$$[1]^0 = [E_1^0, E_1^0, E_1^0] \tag{1.28}$$

being

$$E_1^0 = a_1^0 x_0 + b_1^0 x_1 + c_1^0 x_2 = d_1^0 \tag{1.29}$$

which is represented as [0 1 2] in Figure 1.8). Observe that initially the three members of each triad are initialized with the same value, which is given by the corresponding equation E_i^0, that is $L_i^0 = C_i^0 = R_i^0 = E_i^0$. Following the triads are

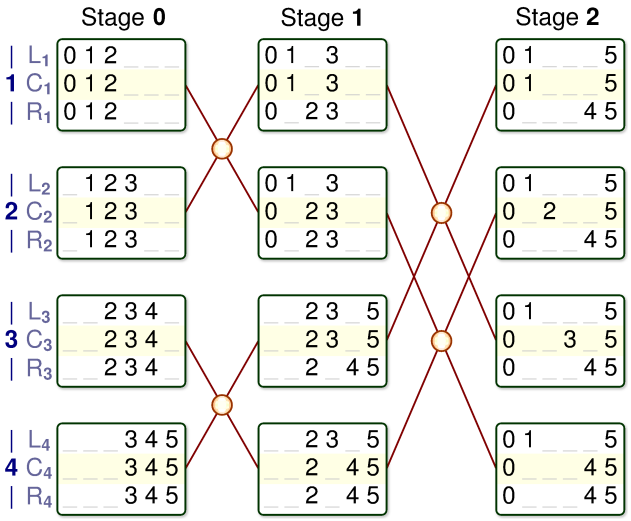


Figure 1.8: Example of the Wang and Mou algorithm for a system with 4 equations

operated pairwise until at the last stage, when $t = log_R N$, we have

$$\begin{aligned} L_i &= \{a_0^t, b_1^t, c_{n+1}^t, d_0^t\} \\ C_i &= \{a_0^t, b_i^t, c_{n+1}^t, d_i^t\} \\ R_i &= \{a_0^t, b_n^t, c_{n+1}^t, d_n^t\} \end{aligned} \tag{1.30}$$

At this point the solution is computed as $x_i = b_i^t / d_i^t$, with $1 \leq i \leq N$.

An interesting property of the algorithm is that, for each stage, the left and right equations are equal to two of the center equations. More specifically, in stage j the left and right equations of row i can be obtained as follows:

$$L_i = C_a \quad \rightarrow a = 2^j \times \lfloor i/2^j \rfloor \tag{1.31}$$

$$R_i = C_b \quad \rightarrow b = 2^j \times (1 + \lfloor i/2^j \rfloor) - 1 \tag{1.32}$$

This can be observed in Figure 1.8, for instance, in Stage 2 all the left equations L_i are equal to C_1 and all the right equations R_i are equal to C_4.

1.4. Work structure summary

The structure of this thesis has been divided into six chapters. The objective of this first chapter has been to provide a general overview of the state of the art in *GPU* programming, describing the most significant languages.

In the second chapter we focus on developing hand-tuned versions of known algorithms. Despite the required effort, this is one of the most common approaches for porting *CPU* applications to *GPU* when performance is critical. For instance, signal processing requires very efficient implementations, specially if real time applications are used, where response time is a decisive factor. Thus, the computational cost and wide application range of signal processing algorithms like the *Fourier* transforms has motivated the research of efficient implementations. In the first part of the chapter we study the design of a hand-tuned implementation of the *FFT* for *AMD Radeon GPUs* using the *Brook+* [7] programming language [55]. Regarding *FFT* implementations on *Brook+*, the only previous work found is an implementation from the original BrookGPU [46] project, but the performance of the *GPU* algorithm was slightly less than the one obtained for recent *CPU* algorithms. In the second part of the chapter we study the implementation of a shallow water simulator [61, 86, 87] on the *GPU* using *CUDA*. Shallow water algorithms requires huge amounts of computing power, and detailed simulations using large areas can take several days or even weeks to complete. Several parallelization techniques are thoroughly explored, explaining their advantages, disadvantages and comparing their efficiency. The results are also compared to a parallel *CPU* implementation using *OpenMP*.

The third chapter is focused on analyzing the *GPU* architecture [56, 57] to provide a better understanding of the performance impact of several hardware features and techniques. The analysis is mainly centered on the memory hierarchy, studying the performance of global memory, texture memory and shared memory in different scenarios. On the *GPU*, when processing large problems, many algorithms become bandwidth bound and the efficient exploitation of the memory is a key factor. The inclusion of general purpose *L2* cache in the *GPU* was an important step forward, but not enough to conceal the special locality requirements for efficient memory

access. The *FFT* algorithm was used in this chapter because it has a fair computational cost, as well as notable bandwidth requirements with good flexibility in the memory access pattern and data distribution, which makes it an adequate tool for performance analysis and study the most appropriate implementation strategies. The obtained knowledge will be used in the following chapters to tune the code and design more efficient implementations.

The fourth chapter addresses the development of a library [58] based on a set of functions that serve as the building blocks for the construction of different algorithms, allowing us to reduce code complexity and development time. Specifically, our design makes extensive usage of template metaprogramming [24]. Template programming for *GPU* libraries have proved to be very useful in other works like *Thrust* [96] or *Bolt* [11], giving a powerful set of tools to the programmer. Several techniques are used to generate static code whenever possible, which can be more efficiently optimized by the compiler. This allowed us to design a flexible implementation while minimizing code replication. Furthermore, the functions of this library can be tuned depending on various factors like the amount of registers, the shared memory size or the desired parallelism level. The proposed library supports many algorithms, like the complex and real *FFT*, the *Hartley* transform, the *Discrete Cosine Transform* or the resolution of tridiagonal equation systems. Although the implementation is focused on modularity and flexibility, it offers competitive performance compared to other state-of-the art alternatives. The template based design and the tuning approach can be reused in many other works and scenarios. Furthermore, the implementation strategy used here is general and can be extended to other algorithms that can be represented using a butterfly communication pattern.

The fifth chapter uses all the previously gathered knowledge to present an advanced methodology based on a two-stage methodology, which is suitable for algorithms that can be expressed as index-digit permutations. In the first stage a set of factors, that characterize the behavior of GPU in terms of performance, is obtained from a resource analysis. In the second stage, operator string manipulation [30] combined with tuning mapping vector is used to describe and adjust the data distribution in the *GPU* resources according to the resource analysis made at the first stage. Furthermore, the operator string manipulation enables the design of modular yet efficient kernels tuned for the *GPU* architecture, with a compact no-

tation to represent the operations carried by the algorithm. Once again our design makes extensive usage of template metaprogramming [104], so many operations are performed at compile-time reducing any performance penalty, with the advantage of designing a flexible implementation while minimizing code replication. Specifically, our methodology has been applied to develop flexible Index-Digit algorithms for *CUDA GPUs* such as the *FFT* (*Fast Fourier Transform*) algorithm (ID-FFT) and a tridiagonal solver algorithm (ID-TS), showing that our work is able to surpass the performance of *NVIDIA's* libraries and other efficient proposals from the bibliography.

In the sixth chapter the conclusions are presented, commenting the most interesting contributions and achievements of each chapter. Some possible future work and research topics are also proposed. Last, we include a list of the conference and journal publications derived from this thesis.

Chapter 2

Efficient hand-tuned implementations on a GPU

In this chapter we analyze and propose a set of efficient implementation techniques for *AMD* and *NVIDIA GPUs* as an example of the first step of our path towards a methodology for efficient algorithm design for *GPUs*. The implementations in this chapter have been straightforwardly developed as any experienced software developer would do after receiving some concepts about *GPU* architectures and programming. We describe several features and implementation strategies, analyzing the scalability and performance compared to other well-known existing solutions. In the first part of the chapter (Section 2.1.1) we study the *FFT* implementation on a *Radeon GPU* using the *Brook+* language. This work was previously presented in [55]. In the second part (Section 2.2), we study the parallelization of a shallow water simulator on a *GeForce GPU* using *CUDA*. The *CUDA* implementation described in this chapter is based on [86, 87]. Some of the explained parallelization techniques were previously analyzed for a *Radeon* architecture using *Brook+* in [61]. Both implementations were hand-tuned for efficient *GPU* usage using several optimization techniques as described. Some of these optimizations are quite common in *GPU* programming, resulting general enough to be also applied in other areas or algorithms.

2.1. FFT processing on a streaming architecture using Brook+

This section is included in the thesis due to historical reasons. At the beginning of this research work *Brook+* was still a state-of-the-art *GPU* programming language, nevertheless *CUDA* and *OpenCL* have progressively prevailed as better options. Here we present a *FFT* implementation that is able to exploit the power of current *GPUs*, outperforming existing *CPU* implementations like the *Spiral* library and being competitive with *NVIDIA's CUFFT*. Different optimization techniques are explained, like the use of stage fusion to reduce the number of kernels, scheduling layouts to execute the optimal sequences, or recomputation to take advantage of *GPU* arithmetic resources.

2.1.1. FFT implementation using Brook+

The *FFT* is an essential operation in a wide range of areas like digital audio or image processing, hence the importance of the availability of efficient libraries implementing it in the different platforms (for more details about the *FFT* and its related terminology see Section 1.2.1). Regarding *FFT* implementations on *Brook+*, the only previous work found in the literature is an implementation from the original BrookGPU [46] project, but the performance of the *GPU* algorithm was slightly less than the one obtained for the *CPU* algorithm. Following we discuss the main features and optimizations to efficiently implement the *FFT* operation using the *Brook+* streaming model (for more information about the *Brook+* language and its programming model see Section 1.1.3).

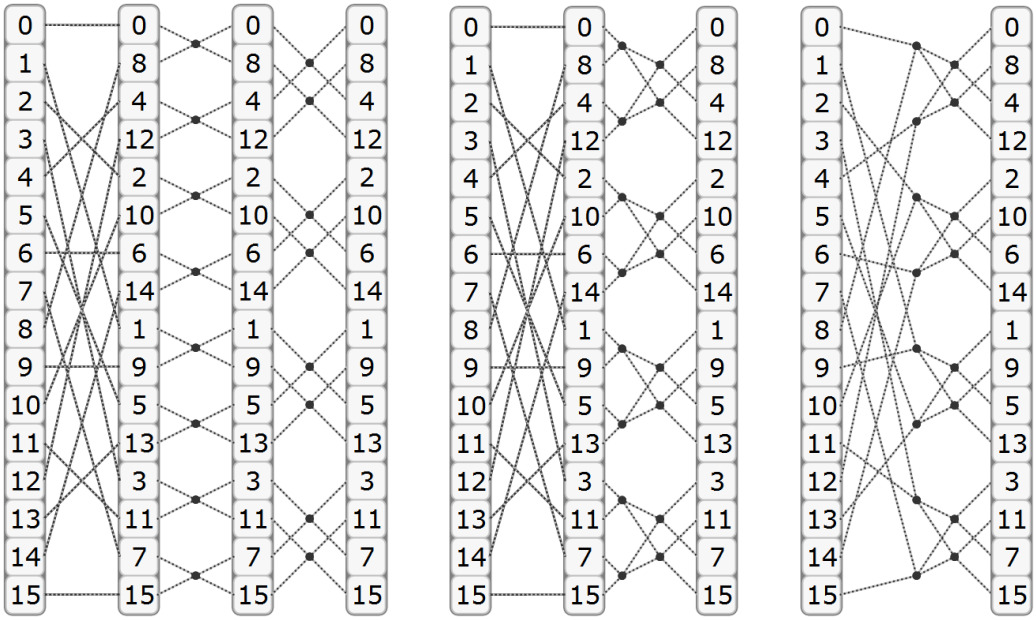
2.1.1.1. Blending/fusion of FFT stages

In order to efficiently exploit the computing power and hide latencies, *GPUs* need to execute a large number of threads. Thus, most algorithms require some way of

coordination or collaboration among the threads. *Brook+* latest release (v1.4) supports local memory (see Section 1.1.1 for a description of the *Radeon GPU* memory hierarchy and information about the architecture). This special memory can be shared among a group of threads resulting very useful to speed up computations when a group can collaborate in the same problem. However, in our tests due to language/driver issues this feature did not work as expected, leading to very poor performance or even incorrect results. Due to this constraint, we rely on global memory to perform inter-core communication, requiring a separate kernel call for the computation stages; however, the main problem is that kernel calls have quite high initialization cost. Thus, it is important to reduce the number of kernel calls while increasing the number of operations per thread. A good strategy is to unify several radix-2 stages into a single kernel, for example blending the bit reversal with the first butterfly stage. This blending reduces the usage of global memory bandwidth and minimizes the kernel call overhead when multiple stages are required.

Another problem resides on computing a single butterfly stage per kernel, so each thread will perform few arithmetic operations comparatively to the memory read/write operations. In this case, *GPU* latency hiding techniques will not work as well as expected. Also, *ATI Radeon's* architecture has a *VLIW* (*Very Long Instruction Word*) design, so in order to make an efficient usage of the processing resources, the code should have enough independent instructions to fill slots of the *VLIW* instruction packing. In a similar fashion to bit reversal and butterfly blending, we can merge two or more butterfly operations in a single kernel execution (from now on, butterfly fusion).

Figure 2.1(b) shows an example using the first 3 kernel calls for a $N = 16$ signal. In Figure 2.1(a) a separate kernel call is used to explicitly perform each compute and reorder operation. In Figure 2.1(b) the fusion of the two butterfly stages saves one kernel call and increases the number of operations per kernel. Finally, if we modify the memory access pattern of the resulting kernel, we can change the data locations that it reads according to the previous reordering, so the three stages are unified in a single kernel call as shown in Figure 2.1(c). We will denote the number of fused butterfly stages per kernel as q, thus, in this example we have $q = 2$.



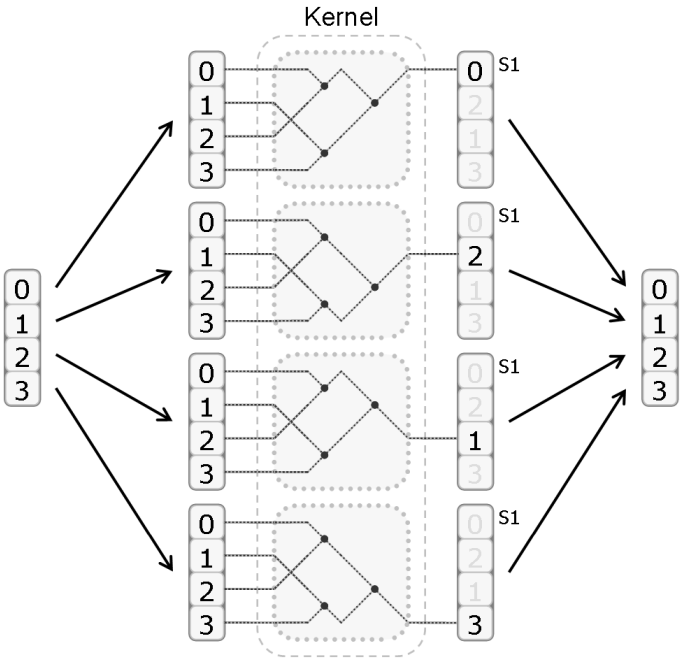
(a) Explicit operations (b) Butterfly fusion (c) Blending of operations

Figure 2.1: Butterfly blending/fusion in Brook+, N = 16

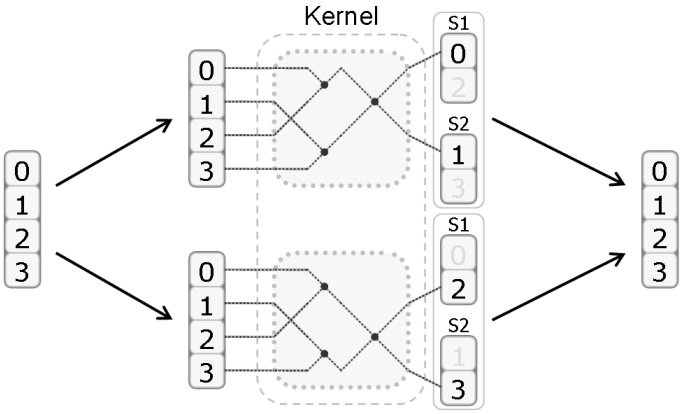
2.1.1.2. FFT mapping techniques

In principle, each thread of the stream processors can only write to a single location of the output, which is fixed and depends on the thread identifier. In *Brook+*, writing to arbitrary locations of a stream is possible, however it uses global buffers and performs uncached scatter memory writes, so it is slow and should be avoided if possible. Notice that both the bit reversal and the computing stages of the algorithm have diverse read and write patterns. For example, depending on the number of butterfly stage i, each computation writes at least two values (that is, x_j and $x_{j+2^{i-1}}$), or even more if we fuse several butterfly operation together as in Figure 2.1(b). Thus, to fuse the butterfly operations and avoid scatter writes there are several possibilities, and to decide which solution works best we will implement them and analyze their advantages and disadvantages:

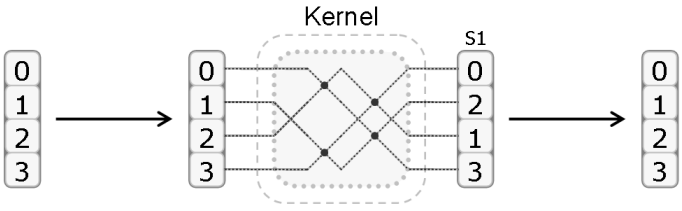
a) *Recomputation (R)*: In this implementation we only write one output value per thread. Each thread will perform the computations to obtain only one complex value of the output (see Figure 2.2(a), where only one element is written by each



(a) Recomputation (and single stream)



(b) Several Streams



(c) Multiple Outputs using floatN (and single stream)

Figure 2.2: Brook+ mapping techniques.

thread, although it will perform redundant operations with other threads). As we do not use shared memory, even though several threads work with the same input set, they can not directly collaborate by sharing their partial results, so as we try to increase the number of steps per kernel more work is wasted. However, for small q values, each thread can recompute the necessary data with minor performance penalty.

b) *Several Streams (SS)*: In this solution we use two or more output streams, so that several values can be written at once and less or no data must be recomputed (see Figure 2.2(b), where four outputs are computed by two threads). This looks quite efficient, but the more outputs we use, the more streams are needed, and the stream recombination and conditional read operations can become expensive so the performance is degraded. A related strategy to process large problems was previously proposed in [110].

c) *Multiple Outputs (MO)*: For problems with batches of small *FFTs* it is possible to use a pure streaming model, where each thread receives a set of data inputs and computes the solution for that set, writing the result in another stream. This strategy unrolls the whole *FFT* expression directly into a single *Brook+* kernel, so very efficient code can be generated. Each thread will work on a different input and produce its own output, so there is no data recomputation (see Figure 2.2(c)). It behaves like having two big *floatN SIMD* input and output vectors, but in *Brook+* this strategy is limited by the maximum size of kernel output, which is at most 128 bytes; if more are used a slower multiple pass shader will be generated, which would severely degrade performance.

2.1.1.3. Scheduling layouts

As stated in Section 2.1.1.1, stage fusion can be applied on *FFTs* to perform several steps in a single kernel. However, for large inputs, the number of stages q to use for optimal performance has proved to be variable, so for the sake of flexibility we created a set of tables with execution schedules to determine the sequence of kernels to use for each case. For example, the optimal sequence using *float4* as the base data type is shown in Figure 2.3. Observe that to obtain the best performance, there is a specific sequence depending on the problem size. The optimal kernel

combination to use may vary depending on the input size, data locality and stride, so the best sequence may be difficult to predict, thus it was obtained empirically testing every combination. For example, normally for a signal size like $N = 512$, in addition to the first bit reversal operation, we would require nine butterfly stages, but using the variable q approach we can perform several stages in a single kernel executing it in just three *kernel* calls. In this case, instead of using a fixed q size, the initial step blends an implicit reordering with the three first butterfly stages, the following step performs four fused butterfly stages, and lastly the remaining two butterfly are also computed in a single step. This sequence is represented in Figure 2.3 as $512 \rightarrow [\mathit{3}, 4, 2]$, observe that the first value is highlighted in cursive because it has blended a bit reversal operation together with the three butterfly stages, thus requiring a special kernel.

2.1.1.4. Generic performance optimizations

In *Brook+* many optimization strategies are similar to the ones used in computer graphics. For example, we can use *SIMD* short vector data types (like *float4*) to read and write two complex values at once reducing the number of memory fetch operations.

Brook+ uses texture memory to store the data, which is cached in a small texture cache optimized for 2D spatial locality, so threads in the same module that read texture addresses that are close together in 2D will achieve the best performance. Depending on the memory access pattern of the kernels many read operations could hit the cache, thus reducing the cost of the additional data that is read, specially in the recomputation approach (R in Section 2.1.1.2). This makes easy and efficient to implement batch execution using one of the texture dimensions to store an array of A input signals, processing all the signals at once (see Figure 2.4(a), where the vertical dimension of the texture is used to store A problems. We can also implement batch execution in the horizontal dimension (see Figure 2.4(b)) using bit masking of the thread identifiers within the kernels. Mixing both approaches is possible and provides the best performance, thus it was the batch mode used in our tests.

Code vectorization and proper *VLIW* slots usage is also important, but in the case of *Brook+*, *AMD's* compiler does a good role in vectorization and expression

$$
\begin{array}{lll}
SL = \{ & 1 \rightarrow [0] & 128 \rightarrow [3,4] \\
& 2 \rightarrow [1] & 256 \rightarrow [3,3,2] \\
& 4 \rightarrow [2] & 512 \rightarrow [3,4,2] \\
& 8 \rightarrow [3] & 1024 \rightarrow [3,3,2,2] \\
& 16 \rightarrow [4] & 2048 \rightarrow [3,4,2,2] \\
& 32 \rightarrow [5] & 4096 \rightarrow [3,4,3,2] \\
& 64 \rightarrow [3,3] & 8192 \rightarrow [3,3,3,2,2] \quad \}
\end{array}
$$

Figure 2.3: Optimal scheduling layout

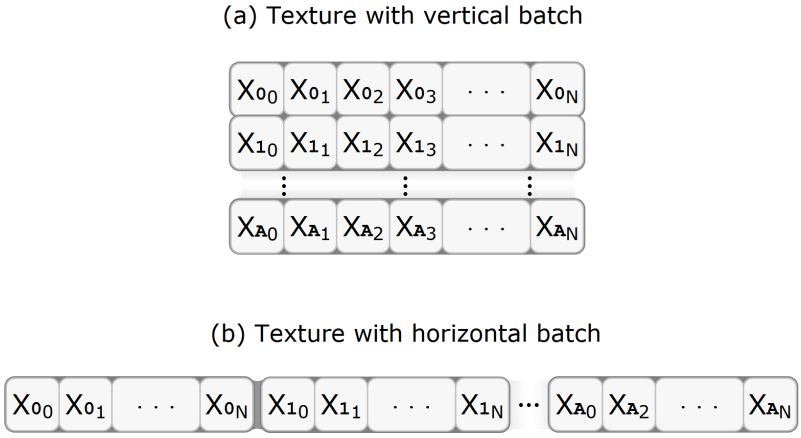


Figure 2.4: FFT batch data storage in 2D texture

optimization. We still can help writing more compiler friendly code, for example using floating point arrays indexes (texture coordinates), or modifying the code to take advantage of *MADD* (fused multiply-add) hardware capabilities.

In comparison to standard *CPUs*, current *GPUs* have a large number of registers, so small *FFTs* can be completely loaded into them and carried out efficiently using just registers. Also, taking advantage of predication or turning conditional statements into arithmetical expressions generally benefits *GPU* architectures, where branch cost is usually high. To further improve performance and make a more efficient usage of the computational resources, there are a few general techniques that can be applied. For example, small loops can be written as unrolled expressions, and because *Brook+* does not support private array variables we have to simulate them using sets of registers.

In order to exemplify some of the optimizations introduced in the previous paragraphs, Figure 2.5 shows a *Brook+* implementation of Figure 2.1(c) using the re-

computation strategy (see (R) in Section 2.1.1.2), where a single kernel performs an implicit reordering and two butterfly stages at once, and each thread (identifier *pos.xy*) performs all the computations involved to obtain only its output value (*sOut*) independently of other threads. We can see how the array indirections use floating point index values in lines 10 to 13 to load all the required data in registers at the beginning of the thread execution (variables $x1$ to $x4$). Also note that many expressions of the form $s = a \times b + c$ were used (for example in lines 12 and 18), these sentences will be optimized into *MADD* instructions by the compiler. Sign dependent operations on lines 15 and 16 have a simple statement that can be predicated to conditionally change the sign in the expressions 18 and 19 using a single *MADD* instruction. Notice how the twiddle factors where also simplified for this kernel, using simple scalar multiplications instead of complex number operations (see *mul1* and *mul2* in lines 18, 19 and 25). The omitted code between lines 7 and 8 obtains the first input location to read (*offset*) applying a bit reversal. The operation will be masked to split the address into *current_pos* and *current_grp*, enabling batch execution in both dimensions.

2.1.2. Experimental results

Several experiments to study and evaluate the performance of our *FFT* implementations have been conducted. All the tests were run in single precision and using power of two input signals in the range $N = 4, \ldots, 8192$, using batch execution to perform several *FFTs* each time. The size of the batch depends on the input size and is given by the expression $batch = 2^{24}/N$, so as the input signal increases the number of batch executions decreases. Test data is already on the *GPU*, thus there are no data transfers during the benchmarks. The performance of the experiments is measured in *GFLOPS*, computed the commonly used formula from Equation (1.4).

FFT libraries are usually optimized for power of two signal sizes because other inputs could be processed padding the data with zeros. Some implementations even use pruned algorithms [31] to take advantage of the zero elements to save some operations. Batch execution is specially adequate for processing small signals on *GPUs* due to their massive parallel architecture, otherwise the *GPU* would not be

```
1  kernel void
2  FFT_Init2(Complex sIn[][],          // Input signal
3            out Complex sOut<>,       // Output signal
4            float dist,               // Element distance
5            int size)                 // Size of signal
6  {
7      int2 pos = instance().xy;
       ...
8      float offset = (float)(current_pos | current_grp);
9      // Read data, x1..x4
10     Complex x1 = sIn[pos.y][offset           ];
11     Complex x3 = sIn[pos.y][offset + dist    ];
12     Complex x2 = sIn[pos.y][offset + dist * 2.0f];
13     Complex x4 = sIn[pos.y][offset + dist * 3.0f];

14     // Multiplier 1 calculation
15     Complex mul1 = pos.x & 0x01 ?
16         Complex(1.0f, 1.0f) : Complex(-1.0f, -1.0f);

17     // Compute, step 1
18     Complex a1 = x1 + x2 * mul1;
19     Complex a2 = x3 + x4 * mul1;

20     // Multiplier 2 calculation
21     if(pos.x & 0x01) a2 = Complex(-a2.y, a2.x);
22     Complex mul2 = pos.x & 0x02 ?
23         Complex(1.0f, 1.0f) : Complex(-1.0f, -1.0f);

24     // Compute step 2
25     Complex b1 = a1 + a2 * mul2;

26     // Write output
27     sOut = b1;
28 }
```

Figure 2.5: FFT optimized code example

able to properly exploit its computational resources and the cost of device memory transfers and kernel launch would outweigh any benefits of using the *GPU*.

Our test platform is composed by a *Core i7 920* processor running at 2.66 *GHz*, 6 *GB DDR3 1866 CL9* memory, *X58* chipset based motherboard and a *Radeon 5870*

GPU. The software setup is *Windows XP x64* operating system, using *Microsoft Visual C++ 2008* compiler (x64, release profile) and *Brook+ 1.4* with *Catalyst 10.3* driver.

2.1.2.1. Optimal streaming strategy

First, we will study the most adequate strategy to use for butterfly fusion from the three proposals (R, SS and MO) introduced in Section 2.1.1.2, trying to find the optimal q value. This test is restricted to small q values, up to $q = 6$ (thus 64 complex values at most); there are many applications like image filters that only need to process the input in small blocks, so they can benefit from special algorithms developed for a particular problem size.

As we can see in Figure 2.6 the three implementations offer very good performance, however the best performance up to $N = 16$ is obtained using the multiple output strategy (MO). This implementation is very efficient, reaching 170 *GFlops* for 16 point *FFT* ($q = 4$), but it needs the input and output data to be placed in special streams and can not scale well beyond that size because a multiple pass shader would be automatically generated by the compiler.

The use of several streams (SS) tends to have a similar performance to MO, albeit a little lower. The major problem of this implementation is the additional cost if the application needs an explicit step to recombine the streams, and that only 8 output streams can be used at most to avoid multiple pass shaders, thus restricting its applicability.

The strategy of recomputation (R) performs quite well on the *GPU*, reaching 180 *GFlops* for 32 point *FFT* ($q = 5$), however above that size, too much computing power would be wasted. This strategy also requires reading several times the same inputs, however due to temporal and spatial texture cache locality, the memory fetch is usually performed only the first time the data is requested.

We also conducted a test using scatter writes, but it offered the worst performance, always under 2 *GFlops*. On the one hand it tends to be a very slow operation in *Brook+*, and on the other hand, using small problems with little arithmetic intensity makes hard to compensate for its slower speed using computations.

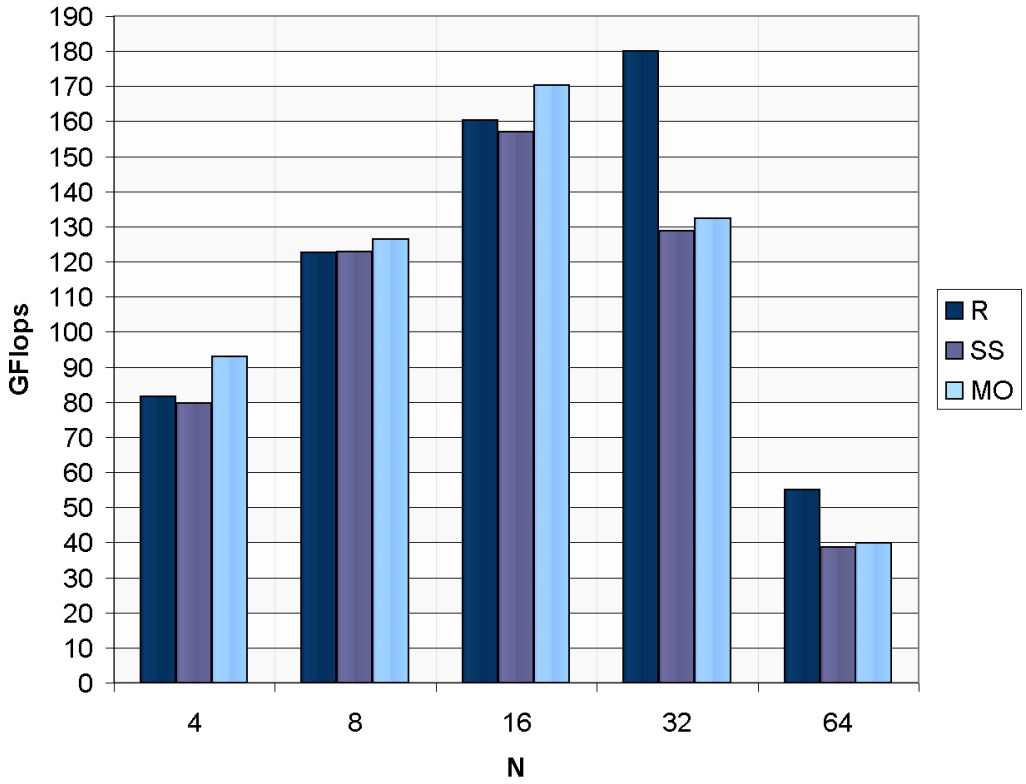


Figure 2.6: Butterfly fusion strategy analysis

2.1.2.2. Stage fusion impact

The most scalable streaming implementation was obtained using the recalculation approach. It is possible to combine recalculation stages to implement larger *FFTs* while offering good performance, so we will study the performance of the implementation using this strategy.

In order to examine the performance impact of stage fusion we can modify the value of q. In Figure 2.7 we can see the performance results when varying q between 1 and 4; it also displays the result of using the optimal scheduling layout (qA). The best performance is obtained with this adaptive solution, but it is quite close to the implementation using $q = 3$. Both methods are able to double the performance of the basic version ($q = 1$) for sizes up to 1024 elements. Using $q = 2$ or $q = 4$ the performance is lower, and for the $q = 4$ case we can even see some degradation for certain problem sizes. From 1024 elements onwards the data set is quite big and the

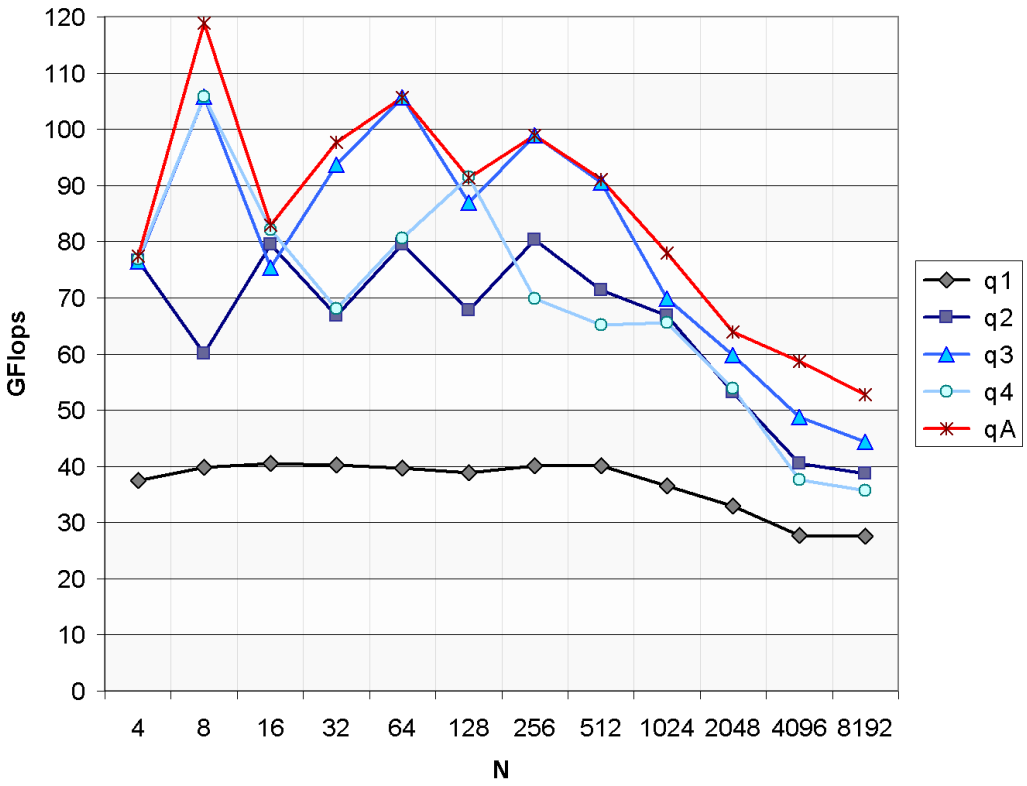


Figure 2.7: Butterfly fusion performance

memory spatial locality is not so good, starting the performance to degrade for all versions.

2.1.3. Performance comparison with previous work

In order to have a better overview of the performance offered by our proposal, we present in Figure 2.8 a global comparison with two representative *FFT* libraries, *CUFFT 3.0* and *SPIRAL 6.0*. The measures were obtained using the same environment and conditions as described in Section 2.1.2, except for the *CUFFT*, where we used a *GeForce GTX 280*. The *SPIRAL* library was compiled in x64 mode using the *SSE* version and *OpenMP* to launch 8 simultaneous threads. Our *Brook+* version uses the adaptive q approach and *float4* base data type. This provides very good performance at the expense of code complexity, now easily reaching over 100 *GFlops* for signals up to 512 elements, with a peak of 180 *GFlops* for $N = 32$. After

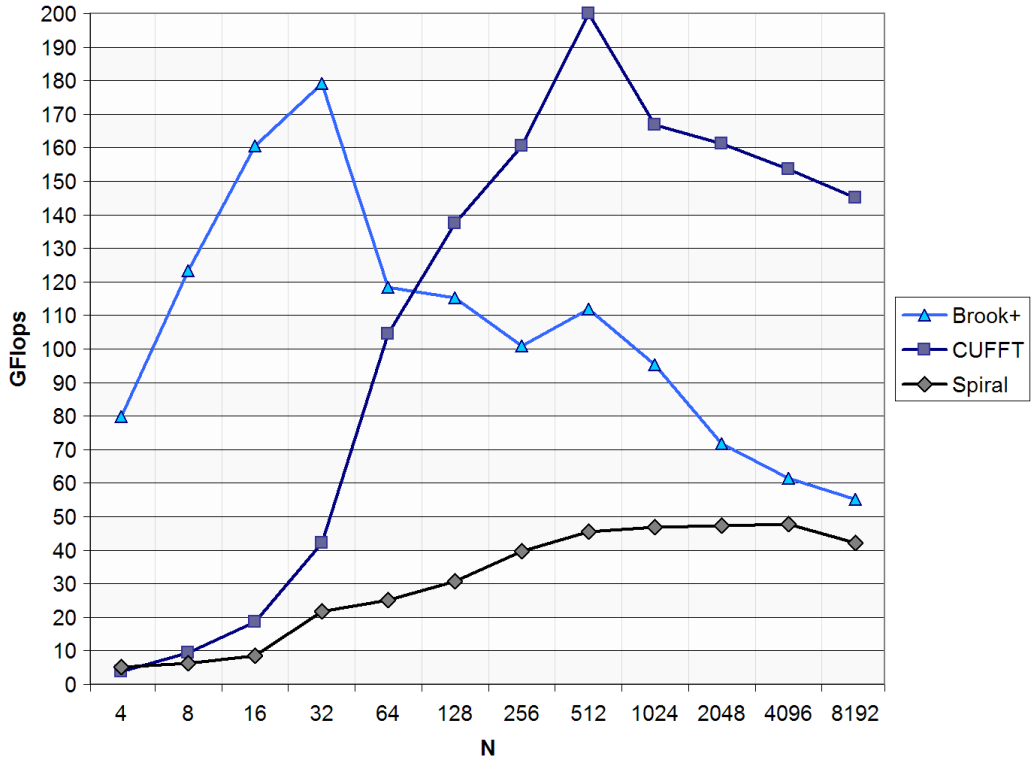


Figure 2.8: Comparison with CUFFT and SPIRAL

$N = 512$, due to the worse memory locality, the performance starts to degrade, but always surpasses 50 *GFlops*.

As we can see, for large batch groups of small transforms (up to $N = 64$) the performance of our solution is excellent, the *Radeon GPU* has an impressive raw power that can be very efficiently exploited using our strategy.

Starting at $N = 128$ the *CUFFT* library takes over the lead, thanks to an efficient usage of the shared memory. Recall that shared memory could not be used in *Brook+*, and as thread data is limited to register space, several passes are required for larger *FFTs* (see Figure 2.3), thus increasing the number of kernel invocations and bandwidth requirements (each pass needs to read and write N complex values). Large *FFTs* tend to have poorer memory locality as well, and in some stages the *GPU* texture cache may not be efficiently used. For large signals our solution offers quite less performance than the *CUFFT*, but always above the *SPIRAL SSE/OpenMP* version, providing at least double the performance between

$N = 128$ and $N = 1024$. As it will be shown in the next chapters, more efficient FFT implementations will be designed achieving very competitive results with respect to state-of-the-art libraries.

2.2. Shallow Water Simulation

This section presents several cost-effective parallel implementations of a finite volume numerical scheme for solving pollutant transport problems in bidimensional domains. The fluid is modelled by 2D shallow water equations, while the transport of pollutant is modelled by a transport equation. The 2D domain is discretized using a first order Roe finite volume scheme. Specifically, this section presents both a solution that exploits recomputation on the GPU, and an optimized solution that is based on a ghost cell decoupling approach.

Shallow water systems describe the evolution of an incompressible fluid in response to gravitational accelerations, where the vertical flow is small compared to the horizontal flow. These systems have many applications, enabling the simulation of rivers, canals, coastal hydrodynamics or dam-break problems, among others. In particular, the transport of pollutant in a fluid, that is modelled by a transport equation, has particular relevance in many ecological and environmental studies. Our solution uses a mathematical model that consists in the coupling of a shallow water system and a transport equation. These coupled equations constitute a hyperbolic system of conservation laws with source terms, that can be discretized using finite volume schemes [29, 111].

Finite volume schemes solve the integral form of the shallow water equations in computational cells of a geometrical mesh that describes the computational domain. Some main benefits of using explicit finite volume schemes are observed. First, mass and momentum are conserved in each cell, even in the presence of flow discontinuities. A good approximation of fast waves as moving shocks or wet-dry fronts appears in fluid or coastal hydraulics. Furthermore, much reduced memory overheads are involved, as complex iterative matrix solvers are not required. Finally, explicit finite volume schemes are easy to implement in multi-core and many-core

systems as the most intensive computational part of the algorithm consists of a set of operations that can be performed independently (and thus asynchronously) at each edge of the mesh. This set of operations can be identified with a lightweight computational kernel, which is invoked a large number of times for big meshes, and thus the algorithm fits perfectly a stream programming model.

The simulations of these problems have very large computing requirements which grow with the size of the space and time dimensions of the domain. For example, in the simulation of marine systems, the spatial domain can have many kilometers and the time integration of the problem can last several weeks or even months. Precise simulations over large detailed terrains require big meshes that usually result in prohibitive execution times.

Thus, due to the interest of this kind of problems and its high computational demands together with the fact that explicit FV solvers fit well with the streaming programming model, several parallel implementations have been proposed on a wide variety of platforms, such as computer clusters using *MPI* [91], a version combining *MPI* and *SSE* (*Streaming SIMD Extensions*) instructions [92] and other generic multi-platform implementations like [23]. Despite these efforts, the increasing computing power required by the most complex simulations motivated the development of *GPU* (*Graphics Processing Unit*) solvers [82, 125, 129] based on the first generation of *GPU* programming languages like *Cg* or *GLSL*. The rapidly increasing computational power and low cost of *GPUs* and the advances in *GPU* high-level programming languages motivated the development of new parallel versions for modern *GPUs*. Examples of *CUDA* implementations are a one-layer simulator [22, 79], a multi-*GPU* version [94] or high order implementations [9, 62]. The parallel implementations mentioned above do not handle pollutant transport problems. Even if single species transport does not introduce any mathematical difficulties, we have decided to consider SWE together with pollutant transport equation as this system is the basis of more complicated models as turbidity current system presented in [124]. Turbidity currents are of great interest as those have a big impact on the morphology of the continental shelves and ocean basins. Thus, the scheme presented in here can be easily adapted to solve 2D turbidity currents following the aforementioned work. A direct implementation for pollutant transport simulation on *CUDA GPUs* was presented in [86].

Here we propose a parallel shallow water simulator that solves a broad variety of problems, even with pollutant transport and the presence of wet-dry fronts in emerging bottom situations. We propose two different approaches. First we propose a naive solution that exploits recomputation on the *GPU*; and second, an optimized solution that is based on ghost cell decoupling, on the efficient use of the *GPU* shared memory to minimize global memory accesses, and on the use of the texture memory to optimize uncoalesced global memory accesses. The resulting implementations achieve excellent performance on *CUDA*-enabled *GPUs*, which makes feasible the execution of really large simulations even when dealing with pollutant transport problems and wet-dry zones on very complex terrains. Overall, this work shows that shallow water problems are well suited for exploiting the power of *GPUs*.

2.2.1. Coupled model: 2D shallow water equations with pollutant transport

A pollutant transport model consists in the coupling of a fluid model and a transport equation. In order to model the fluid dynamics we consider the bidimensional shallow water equations, which describe the evolution of a fluid over a bottom, where the thickness and the vertical flow is small compared to the horizontal flow:

$$\begin{cases} \dfrac{\partial h}{\partial t} + \dfrac{\partial q_x}{\partial x} + \dfrac{\partial q_y}{\partial x} = 0, \\ \dfrac{\partial q_x}{\partial t} + \dfrac{\partial}{\partial x}\left(\dfrac{q_x^2}{h} + \dfrac{1}{2}gh^2\right) + \dfrac{\partial}{\partial y}\left(\dfrac{q_x q_y}{h}\right) = gh\dfrac{\partial H}{\partial x} + ghS_{f,x}, \\ \dfrac{\partial q_y}{\partial t} + \dfrac{\partial}{\partial x}\left(\dfrac{q_x q_y}{h}\right) + \dfrac{\partial}{\partial y}\left(\dfrac{q_y^2}{h} + \dfrac{1}{2}gh^2\right) = gh\dfrac{\partial H}{\partial y} + ghS_{f,y}. \end{cases} \tag{2.1}$$

The unknowns of the problem are the vertically averaged height of the water column $h(\boldsymbol{x}, t)$ and the flux $\boldsymbol{q}(\boldsymbol{x}, t) = (q_x(\boldsymbol{x}, t), q_y(\boldsymbol{x}, t)) = h(\boldsymbol{x}, t) \cdot \boldsymbol{u}(\boldsymbol{x}, t)$, where $\boldsymbol{u}(\boldsymbol{x}, t) = (u_x(\boldsymbol{x}, t), u_y(\boldsymbol{x}, t))$ is the vertical averaged velocity of the fluid, at each point $\boldsymbol{x} = (x, y)$ of the computational domain and at time t. $H(\boldsymbol{x})$ is the function that describes the bottom bathymetry, measured from a fixed reference level (see Figure 2.9), and g is the gravitational constant.

The friction forces are given by a Manning Law:

$$S_{f,x} = n^2 \frac{u_x \|\boldsymbol{u}\|}{h^{1/3}}, \quad S_{f,y} = n^2 \frac{u_y \|\boldsymbol{u}\|}{h^{1/3}}, \tag{2.2}$$

where n is the bed roughness coefficient.

The pollutant transport equation is given by:

$$\frac{\partial (hC)}{\partial t} + \frac{\partial (q_x C)}{\partial x} + \frac{\partial (q_y C)}{\partial y} = 0, \tag{2.3}$$

where $C(\boldsymbol{x}, t)$ is the pollutant concentration.

The system given by Equations (2.1) and (2.3) can be written as a *system of conservation laws with source terms*:

$$\frac{\partial W}{\partial t} + \frac{\partial}{\partial x} F_1(W) + \frac{\partial}{\partial y} F_2(W) = S_1(W) \frac{\partial}{\partial x} H(\boldsymbol{x}) + S_2(W) \frac{\partial}{\partial y} H(\boldsymbol{y}) + \boldsymbol{S_f}, \tag{2.4}$$

where W is the vector of unknowns:

$$W = \begin{bmatrix} h \\ q_x \\ q_y \\ hC \end{bmatrix}, \tag{2.5}$$

where $h(\boldsymbol{x}, t)C(\boldsymbol{x}, t)$ is the amount of pollutant dissolved in the fluid, and

$$F_1(W) = \begin{bmatrix} q_x \\ \frac{q_x^2}{h} + \frac{1}{2} g h^2 \\ \frac{q_x q_y}{h} \\ q_x C \end{bmatrix}, \; F_2(W) = \begin{bmatrix} q_y \\ \frac{q_x q_y}{h} \\ \frac{q_y^2}{h} + \frac{1}{2} g h^2 \\ q_y C \end{bmatrix},$$

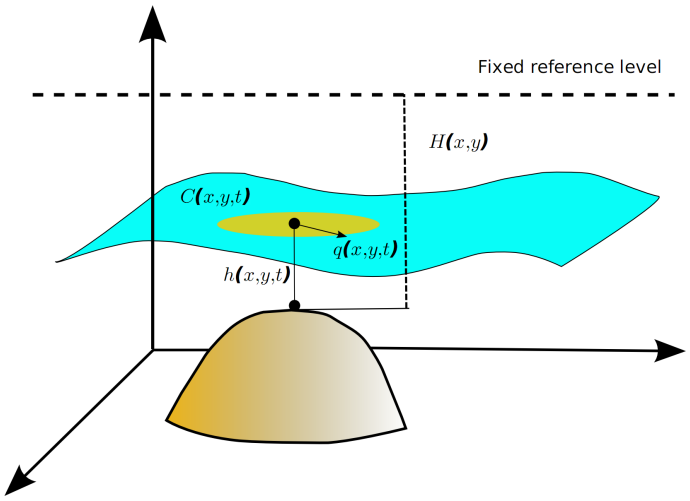


Figure 2.9: Sketch: pollutant transport.

$$S_1(W) = \begin{bmatrix} 0 \\ gh \\ 0 \\ 0 \end{bmatrix}, \quad S_2(W) = \begin{bmatrix} 0 \\ 0 \\ gh \\ 0 \end{bmatrix}, \tag{2.6}$$

and

$$\boldsymbol{S_f} = \begin{bmatrix} 0 \\ ghS_{f,x} \\ ghS_{f,y} \\ 0 \end{bmatrix}. \tag{2.7}$$

System (2.4) can be written in a more compact form:

$$\frac{\partial W}{\partial t}(\boldsymbol{x}, t) + \mathrm{div}\boldsymbol{F}(W) = \boldsymbol{S}(W) \cdot \nabla H(\boldsymbol{x}) + \boldsymbol{S_f}, \tag{2.8}$$

where $\boldsymbol{F} = (F_1, F_2)$ is the flux function and $\boldsymbol{S}(W) = (S_1(W), S_2(W))$.

Given an unitary vector $\boldsymbol{\eta} = (\eta_x, \eta_y)$, we define the matrix

$$A(W, \boldsymbol{\eta}) = A_1(W)\eta_x + A_2(W)\eta_y, \tag{2.9}$$

where

$$A_1(W) = \frac{\partial}{\partial W} F_1(W), \quad A_2(W) = \frac{\partial}{\partial W} F_2(W) \tag{2.10}$$

are the jacobian matrices of $F_1(W)$ and $F_2(W)$, respectively.

System (2.8) is hyperbolic if $h(\boldsymbol{x}, t) > 0$. Effectively, $A(W, \boldsymbol{\eta})$ is diagonalizable and the eigenvalues of $A(W, \boldsymbol{\eta})$ are

$$\lambda_1 = \boldsymbol{u} \cdot \boldsymbol{\eta}, \ \lambda_2 = \boldsymbol{u} \cdot \boldsymbol{\eta} - \sqrt{gh}, \ \lambda_3 = \boldsymbol{u} \cdot \boldsymbol{\eta} + \sqrt{gh}, \ \lambda_4 = \boldsymbol{u} \cdot \boldsymbol{\eta}. \tag{2.11}$$

2.2.2. Finite volume numerical scheme.

In this section we briefly describe the finite volume scheme that we use to discretize the Equation (2.8). More details can be found in [90, 91, 93].

Let us remark that the term $\boldsymbol{S_f}$ is discretized in a semi-implicit way as detailed in [93], thus in what follows we focus on the discretization of Equation (2.8) where $\boldsymbol{S_f}$ is supposed to be zero.

To discretize the Equation (2.8), we split the computational domain in cells or control volumes, $V_i \subset \mathbb{R}^2, \ i = 1, \ldots, L$. In our case we will consider a structured mesh given by squares. We will use the following notation: given a finite volume V_i, $\boldsymbol{x}_i$ is its center, $|V_i|$ its area, $\mathcal{N}_i$ is the set of indexes j such that V_j is the neighbor of V_i, E_{ij} is the edge shared by two neighbor cells V_i and V_j and $|E_{ij}|$ is its length, and $\boldsymbol{\eta}_{ij} = (\eta_{ij,x}, \eta_{ij,y})$ is the unitary vectorial normal to edge E_{ij} and that points towards the cell V_j (see Figure 2.10). Finally, we call V_{ij} the triangular subcell with one edge given by E_{ij} and the opposite vertex given by $\boldsymbol{x}_i$ (see Figure 2.10).

In finite volume schemes, constant approximations of the solution at each cell are computed. More precisely, if $W(\boldsymbol{x}, t)$ is the exact solution at point $\boldsymbol{x}$ and at time t, we will denote by ${W_i}^n$ an approximation of the average of the solution on the volume V_i at time t^n,

$${W_i}^n \simeq \frac{1}{|V_i|} \int_{V_i} W(\boldsymbol{x}, t^n) d\boldsymbol{x}. \tag{2.12}$$

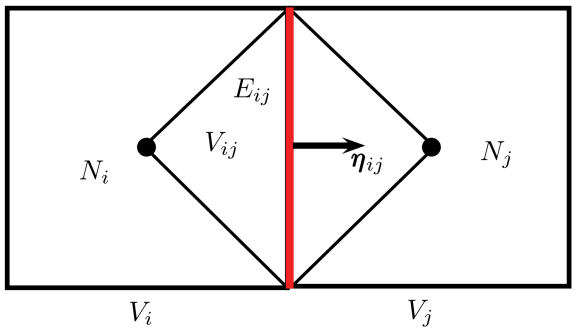


Figure 2.10: Finite volume: structured mesh.

Integrating the Equation (2.8) over each finite volume V_i

$$\frac{\partial}{\partial t}\int_{V_i} W(\boldsymbol{x},t)\,dV + \int_{V_i}(\mathrm{div}\mathbf{F}(W))\,dV = \int_{V_i}\boldsymbol{S}(W)\cdot\nabla H(\boldsymbol{x})\,dV. \tag{2.13}$$

Dividing by $|V_i|$ and applying the Divergence Theorem:

$$\begin{aligned}\frac{\partial}{\partial t}\left(\frac{1}{|V_i|}\int_{V_i} W(\boldsymbol{x},t)\,dV\right) = \\ -\frac{1}{|V_i|}\left(\sum_{j\in\mathcal{N}_i}\int_{E_{ij}}\mathbf{F}(W)\cdot\boldsymbol{\eta}_{ij}\,d\gamma - \int_{V_i}\boldsymbol{S}(W)\cdot\nabla H(\boldsymbol{x})\,dV\right).\end{aligned} \tag{2.14}$$

To discretize Equation (2.14) we will use the finite volume numerical scheme presented in [93]. Once the approximation of W_i is known at time t^n, W_i^n, the approximation at time t^{n+1} is given by:

$$W_i^{n+1} = W_i^n - \frac{\Delta t}{|V_i|}\sum_{j\in\mathcal{N}_i}|E_{ij}|\mathcal{F}_{ij}^-(W_i^n, W_j^n, \boldsymbol{\eta}_{ij}), \tag{2.15}$$

with

$$\begin{aligned}\mathcal{F}_{ij}^-(W_i^n, W_j^n, \boldsymbol{\eta}_{ij}) = \mathcal{P}_{ij}^n\Big(\boldsymbol{F}(W_j^n)\cdot\boldsymbol{\eta}_{ij} - \boldsymbol{F}(W_i^n)\cdot\boldsymbol{\eta}_{ij} - \boldsymbol{S}_{ij}^n\Big) \\ -\frac{\boldsymbol{F}(W_j^n)\cdot\boldsymbol{\eta}_{ij} - \boldsymbol{F}(W_i^n)\cdot\boldsymbol{\eta}_{ij}}{2} + \boldsymbol{F}_\alpha(W_i^n, W_j^n, \boldsymbol{\eta}_{ij}) - \boldsymbol{S}_{\alpha,ij}^n,\end{aligned} \tag{2.16}$$

where the projection matrix, $\mathcal{P}_{ij}$, is given by:

$$\mathcal{P}_{ij}^n = \frac{1}{2} K_{ij}^n \Big(I - \text{sgn}(D_{ij}^n) \Big) (K_{ij}^n)^{-1}, \tag{2.17}$$

being I the identity matrix and K_{ij}^n the matrix whose columns are the eigenvectors related to the Roe matrix A_{ij}^n given by

$$A_{ij}^n = A(W_{ij}^n, \boldsymbol{\eta}_{ij}) = A_1(W_{ij}^n)\eta_{ij,x} + A_2(W_{ij}^n)\eta_{ij,y}, \tag{2.18}$$

where

$$W_{ij}^n = \begin{bmatrix} h_{ij}^n \\ h_{ij}^n u_{ij,x}^n \\ h_{ij}^n u_{ij,y}^n \\ h_{ij}^n C_{ij}^n \end{bmatrix}, \tag{2.19}$$

is the intermediate Roe's state, which is the state that satisfies the equation

$$\boldsymbol{F}(W_j^n) \cdot \boldsymbol{\eta}_{ij} - \boldsymbol{F}(W_i^n) \cdot \boldsymbol{\eta}_{ij} = A_{ij}^n (W_j^n - W_i^n) \tag{2.20}$$

and it is given by:

$$h_{ij}^n = \frac{h_i^n + h_j^n}{2}, \tag{2.21}$$

$$u_{ij,l}^n = \frac{\sqrt{h_i^n} u_{i,l}^n + \sqrt{h_j^n} u_{j,l}^n}{\sqrt{h_i^n} + \sqrt{h_j^n}}, \; l = x, y, \tag{2.22}$$

$$C_{ij}^n = \frac{\sqrt{h_i^n} C_i^n + \sqrt{h_j^n} C_j^n}{\sqrt{h_i^n} + \sqrt{h_j^n}}. \tag{2.23}$$

D_{ij}^n the diagonal matrix whose elements are the eigenvalues of A_{ij}^n that are given by:

$$\begin{cases} \lambda_{ij,1} = \boldsymbol{u}_{ij}^n \cdot \boldsymbol{\eta}_{ij}, \\ \lambda_{ij,2} = \boldsymbol{u}_{ij}^n \cdot \boldsymbol{\eta}_{ij} - \sqrt{g h_{ij}^n}, \\ \lambda_{ij,3} = \boldsymbol{u}_{ij}^n \cdot \boldsymbol{\eta}_{ij} + \sqrt{g h_{ij}^n}, \\ \lambda_{ij,4} = \boldsymbol{u}_{ij}^n \cdot \boldsymbol{\eta}_{ij}, \end{cases} \tag{2.24}$$

and

$$\operatorname{sgn} D_{ij}^n = \begin{bmatrix} \operatorname{sgn} \lambda_{ij,1} & & & \\ & \operatorname{sgn} \lambda_{ij,2} & & \\ & & \operatorname{sgn} \lambda_{ij,3} & \\ & & & \operatorname{sgn} \lambda_{ij,4} \end{bmatrix}. \tag{2.25}$$

The term $\boldsymbol{S}_{ij}^n$ is given by

$$\boldsymbol{S}_{ij}^n = \begin{bmatrix} 0 \\ g h_{ij}^n (H_j - H_i) \eta_{ij,x} \\ g h_{ij}^n (H_j - H_i) \eta_{ij,y} \\ 0 \end{bmatrix}. \tag{2.26}$$

$$\boldsymbol{F}_\alpha(W_i^n, W_j^n, \boldsymbol{\eta}_{ij}) = \frac{\boldsymbol{F}(W_{(1-\alpha)i+\alpha j}) \cdot \boldsymbol{\eta}_{ij} + \boldsymbol{F}(W_{\alpha i+(1-\alpha)j}) \cdot \boldsymbol{\eta}_{ij}}{2}, \tag{2.27}$$

where we denote:

$$W_{(1-\alpha)i+\alpha_j} = \begin{bmatrix} h_{(1-\alpha)i+\alpha_j} \\ (q_x)_{(1-\alpha)i+\alpha_j} \\ (q_y)_{(1-\alpha)i+\alpha_j} \\ hC_{(1-\alpha)i+\alpha_j} \end{bmatrix} = (1-\alpha)W_i^n + \alpha W_j^n, \quad \alpha \in [0,1], \tag{2.28}$$

a convex combination of W_i^n and W_j^n, and finally,

$$\boldsymbol{S}_{\alpha,ij}^n = \begin{bmatrix} 0 \\ \frac{g}{2}\left(\frac{h_{(1-\alpha)i+\alpha j} + h_i^n}{2}(H_{(1-\alpha)i+\alpha j} - H_i) + \frac{h_{\alpha i+(1-\alpha)j} + h_j^n}{2}(H_{\alpha i+(1-\alpha)j} - H_j)\right)\eta_{ij,x} \\ \frac{g}{2}\left(\frac{h_{(1-\alpha)i+\alpha j} + h_i^n}{2}(H_{(1-\alpha)i+\alpha j} - H_i) + \frac{h_{\alpha i+(1-\alpha)j} + h_j^n}{2}(H_{\alpha i+(1-\alpha)j} - H_j)\right)\eta_{ij,y} \\ 0 \end{bmatrix}, \tag{2.29}$$

where

$$H_{\alpha i+(1-\alpha)j} = \alpha H_i + (1-\alpha)H_j, \tag{2.30}$$

is again a convex combination of H_i and H_j.

The Equations (2.27) and (2.29) are used to avoid entropy corrections needed by the Roe scheme in critical points (see [93]). The authors propose different values of the parameter α. In practice, the value $\alpha = 1/8$ gives good results (see [93]), so here we take $\alpha = 1/8$. Note that in the case $\alpha = 0$ we obtain the usual Roe Scheme (see [93]).

The previous numerical scheme is exactly well-balanced for the stationary solution corresponding to water at rest (see [93]) and linearly L^∞ under the usual CFL condition:

$$\Delta t = \min_{i=1,\dots,L} \left\{ \frac{\sum_{j \in \mathcal{N}_i} |E_{ij}| \, \|D_{ij}^n\|_\infty}{2\gamma |V_i|} \right\} \tag{2.31}$$

where γ, $0 < \gamma \leq 1$, is the CFL parameter and $\|D_{ij}^n\|_\infty$ is the infinite norm of the matrix D_{ij}^n, that is, the maximum eigenvalue of the matrix A_{ij}^n.

The resulting time step can be small, which gives rise to a large number of time steps for simulations that occur on large time scales, which is the case for many geophysical flow problems. Thus, from the computational point of view, the solution of the problem is reduced to a huge number of matrix operations and vectors of size 4×4.

Finally, let us recall that the finite volume scheme described in this section is of first order. High order schemes have been implemented in *CPU* [90] and in *GPUs* [9,62] and they provide very good results in academic examples. Nevertheless, the extension of those schemes to simulate real flows with real bathymetries is not a simple task and sometimes, they produce inaccurate results in wet-dry fronts. Let us also remark that this scheme is a generalization of Roe scheme, which gives very precise results and, moreover, it can approximate stationary regular solutions up to second order (see Theorem 10 in [90]).

2.2.2.1. Wet-dry fronts

One of the main difficulties that can appear in practical applications is the presence of wet-dry fronts. These fronts develop when, due to the initial conditions or as a consequence of the fluid motion, the thickness of the layer vanishes. These situations arise very frequently in practical applications such as flood waves, dam-breaks or coastal tidal currents. We handle this situation in two ways. First, we compute

the velocities and concentrations as follows [3]:

$$lu_{i,x} = \frac{\sqrt{2}h_i q_{i,x}}{\sqrt{h_i^4 + max(h_i, \varepsilon)^4}}, \tag{2.32}$$

$$u_{i,y} = \frac{\sqrt{2}h_i q_{i,y}}{\sqrt{h_i^4 + max(h_i, \varepsilon)^4}}, \tag{2.33}$$

$$C_i = \frac{\sqrt{2}h_i q_{i,C}}{\sqrt{h_i^4 + max(h_i, \varepsilon)^4}}, \tag{2.34}$$

where $\varepsilon = 10^{-6}$ is the single precision limit. In practical situations this value gives good results.

Second, if the thickness of the layer of fluid becomes tiny at both cells V_i and V_j, that is $h_i, h_j < h_{eps} = 10^{-4}$, then the fourth component of the numerical flux $\mathcal{F}_{ij}^{-}(W_i^n, W_j^n, \boldsymbol{\eta}_{ij})$ is defined as follows:

$$\mathcal{F}_{ij\,[4]}^{-} = \begin{cases} \mathcal{F}_{ij\,[1]}^{-} \cdot C_j & \text{if } \boldsymbol{u}_{ij} \cdot \boldsymbol{\eta}_{ij} < 0, \\ \mathcal{F}_{ij\,[1]}^{-} \cdot C_i & \text{if } \boldsymbol{u}_{ij} \cdot \boldsymbol{\eta}_{ij} > 0, \end{cases} \tag{2.35}$$

where $\mathcal{F}_{ij\,[l]}^{-}$, denotes the l-th component of the vector $\mathcal{F}_{ij}^{-}$. This value of h_{eps} has been chosen such as it gives the best results in the numerical experiments we have performed.

It must be remarked that the numerical scheme described in the previous section corresponds to the case where the fluid occupies the whole domain. If this numerical scheme is applied without any modification to a case with wet-dry fronts (situations with emerging bottom topography), the results obtained have spurious values. In those cases it is necessary to modify the scheme, as is proposed in [93]. This modification allows to balance the fluxes against the driving forces so that the non-physical pressure forces disappear in the case of bottom emerging topographies.

Finally, let us remark that in order to provide numerical simulations in real domains, friction terms are very important to reproduce the correct position of wet-dry fronts. Moreover, the semi-implicit way of discretizing the friction terms enforces the numerical stability of the scheme in areas where h is small (see [93] for more details).

2.2.3. Naive GPU solution

This section explains a naive *GPU* solution that uses a recomputation-based algorithm to take advantage of the computational power of the *GPU*. To test our application we will use an *NVIDIA GPU* and *CUDA* as the programming language (see Section 2.2.5 for a description of the execution platform and Section 1.1.2 for detailed information about the architecture). This initial naive version is developed using only the basic features of *CUDA* programming and avoiding hardware-dependent tuning techniques. In Section 2.2.4 the implementation will be refined using a more advanced strategy with platform specific parallelization techniques.

Figure 2.11 shows the algorithm corresponding to the numerical scheme of the coupled system given by Equation 2.4. The main loop performs the simulation through time. In each time step, the amount of flow that crosses through each edge is calculated in order to compute the flow of data for all finite volumes. This algorithm performs a huge number of small vector and matrix operations to solve the equations of the coupled system for each edge of the mesh. Each time iteration is divided into 3 stages:

Stage1. Computation of the flow of data ΔM and the time step Δt for each volume v applying a recomputation-based solution (see ① in Figure 2.11). For each volume, the recomputation-based algorithm calculates four flow contributions (up, down, left and right) that are associated to each of the four edges of a volume. This implies that each edge is processed twice, once for each neighbor volume. For each volume, $\Delta t[v]$ is computed in a similar way. Our naive *CUDA* implementation maps volumes to threads that run concurrently in a conflict-free manner. Although one half of the computations will be redundant, the great computational power of the *GPUs* allows to obtain a competitive performance. Figure 2.12 depicts an example for four threads. For example, the contribution that volume $V_{2,2}$ does to volume $V_{2,3}$ takes the same value (though opposite sign) as the contribution of volume $V_{2,3}$ does to volume $V_{2,2}$. However this contribution is recalculated when volume $V_{2,2}$ computes the contribution from its neighbors. This first kernel only performs accesses to global memory and thus, it saves arrays ΔM and Δt in global memory.

Stage2. Computation of the global time step $\Delta tGlobal$ as the minimum of the local

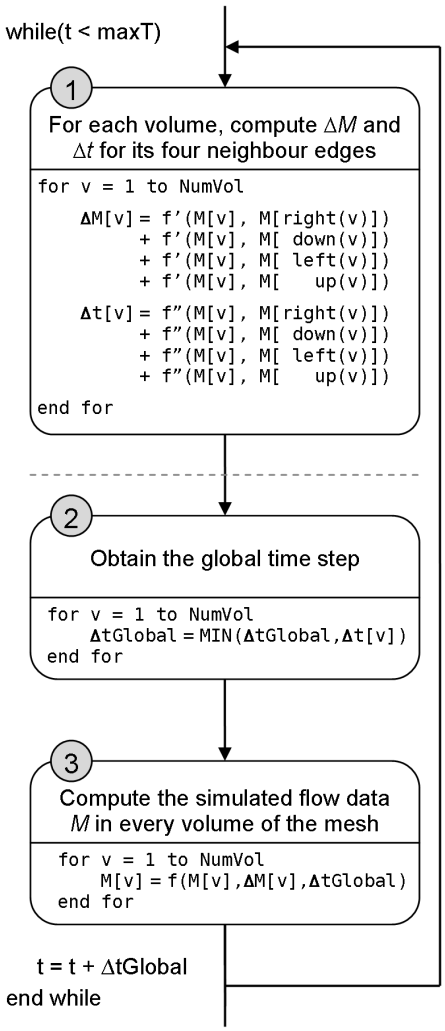


Figure 2.11: Naive algorithm

time steps Δt computed for each volume in *Stage1* (see ② in Figure 2.11). *CUDA* supports atomic operations on global memory, but we do not use them because their performance is very poor in practice. The implementation of *Stage2* is based on a reduction kernel that is launched many times in order to reduce the array Δt allocated in global memory. In each invocation of this reduction kernel, the set of loop iterations is partitioned among thread blocks so that read accesses to global memory are coalesced. Each thread block runs a tree-based parallel reduction that operates only on a buffer allocated in shared memory. The partial result is saved in a private copy of $\Delta tGlobal$ allocated in

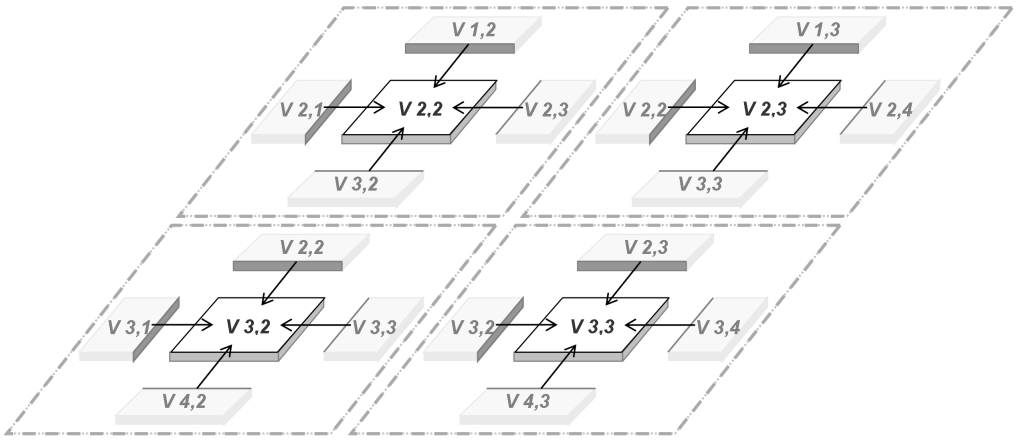


Figure 2.12: Recomputation-based solution on a multithreading system

shared memory. At the end of the kernel invocation, each thread block writes this partial result into a different element of array $\Delta tGlobal$ allocated in global memory. Finally, when the size of the array is twice the thread block size, no more reduction kernels are launched and this array is reduced by the *CPU*.

Stage3. Computation of the simulated flow data M for each volume (see ③ in Figure 2.11). This is achieved by updating in each volume the pollutant density and the fluid data using ΔM from *Stage1* and $\Delta tGlobal$ from *Stage2*. This stage computes a set of operations that do not depend on each other and that, therefore, will be executed in parallel in different threads.

The naive *GPU* implementation described above differs from the solution presented in [86], where a reduction kernel based on `reduce3` of the *CUDA SDK* [81] is used in the *Stage2*. Now we use a kernel based on the `reduce5` implementation of the *CUDA SDK*. This kernel is completely unrolled, and avoids divergence, shared memory bank conflicts and unnecessary synchronization points.

The first stage is the most computationally intensive part of the algorithm, being the huge number of small vector and matrix operations needed to solve the equations specially costly. This way, a profiling execution for an example mesh of 1000×1000 volumes shows that about 80% of the runtime is consumed by the computations done in this first stage.

2.2.4. Optimized GPU solution

In this section, an efficient *GPU* implementation based on the ghost cell decoupling technique is proposed. This implementation, whose structure is shown in Figure 2.13, contains three improvements with respect to the naive implementation presented in Section 2.2.3. The first improvement is the application of a ghost cell decoupling technique to *Stage1* in order to avoid most of the duplicated computations of the recomputation-based solution. Our ghost cell decoupling strategy uses shared memory to save the local time steps (see ①ⓐ in Figure 2.13) before storing them into global memory (see ①ⓑ in Figure 2.13). This leads naturally to the second improvement, which consists in splitting the reduction of *Stage2* into two phases: first, each thread block of the kernel of *Stage1* reduces its local time steps in shared memory and saves partial results in global memory (see ①ⓒ in Figure 2.13); and second, the kernel of *Stage2* reduces the partial results using the `reduce5` *CUDA* implementation (see ② in Figure 2.13). The third improvement is the usage of texture memory when uncoalesced memory accesses occur, provided that the arrays affected by those accesses do not change during the execution and the consistency of the texture memory can be guaranteed. This avoids the time penalties of uncoalesced accesses to global memory. The rest of this section describes these three improvements in more detail. Their impact on the execution time will be studied in Section 2.2.5.

2.2.4.1. Ghost cell decoupling solution

This improvement is aimed to reduce the large number of duplicated computations that arise in the recomputation-based solution used in *Stage1* (see ① in Figure 2.11). This improvement starts with a decomposition of the 2D domain using the ghost cell decoupling technique. This technique enables a memory conflict-free execution of the thread blocks (avoiding communications and synchronization between thread blocks). For this purpose, the 2D domain is splitted into 2D subdomains that include several ghost cells. The ghost cells represent flux contributions that are recomputed in two neighbor thread blocks. In our shallow water problem, these ghost cells are a row and a column of each 2D subdomain. This way, this memory region (*ghost region* from now on) is read by two thread blocks, although it is only

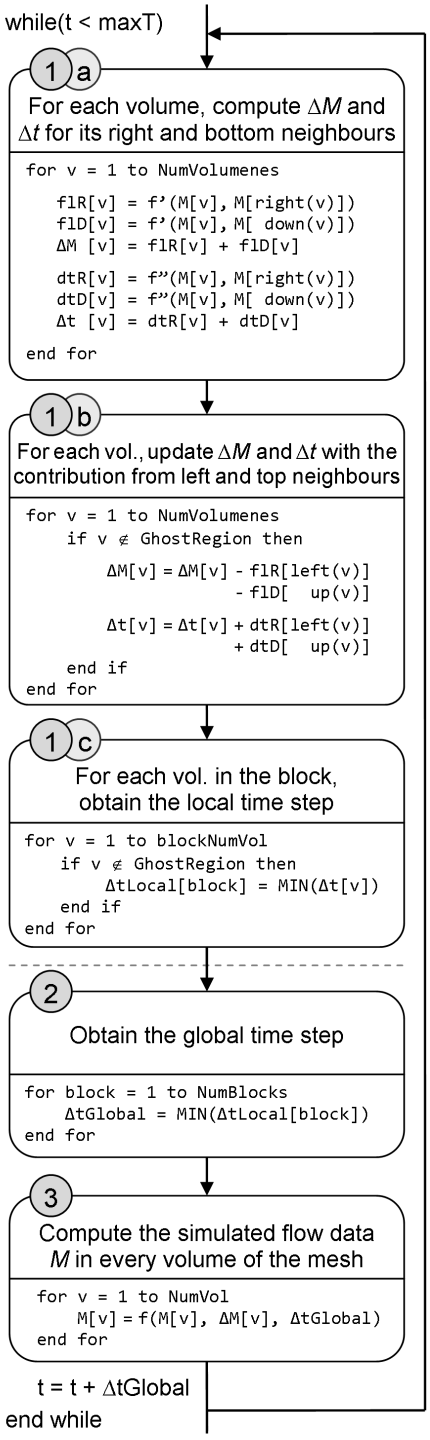


Figure 2.13: Optimized GPU solution

updated by one. The reason is that the ghost region, together with the rows and columns whose updating is responsibility of the thread block, provide the information the thread block needs to perform its computations, making therefore the block self-sufficient. Overall, the ghost cell decoupling technique removes the replicated computations for most of the cells, the exception being the ghost cells of each thread block.

The algorithm shown in Figure 2.13 shows the implementation details of the ghost cell decoupling technique. In stage ①ⓐ the thread responsible for volume v computes the flow from the neighbor volumes on the right (flR) and bottom (flD). Next, a partial flow $\Delta M[v]$ is calculated as $flR[v] + flD[v]$. The same procedure is followed to obtain the partial $\Delta t[v]$. These partial values $flR[v]$, $flD[v]$, $dtR[v]$ and $dtD[v]$ are stored in the shared memory, so that in a second phase (see ①ⓑ in Figure 2.13) the thread responsible for volume v only has to add to its partial flow the opposite contribution of its left and up neighbors. A synchronization barrier is needed between the first and the second phase because in the second phase ①ⓑ each thread reads the partial flows stored in shared memory by another thread in the first phase ①ⓐ. In *CUDA*, a synchronization barrier (`__syncthreads()`) stops all warps within a given thread block until all the warps have reached the synchronization barrier. This way, the synchronization barrier guarantees that all threads of the thread block have stored their partial flows and timesteps in shared memory before another thread makes use of them.

The first two phases of this approach are illustrated in Figure 2.14 using a thread block size of 3×3. In each volume there are two arrows that symbolize the storage of its right and down flux contributions in buffers allocated in shared memory. The ghost region of a thread block consists of the volumes located in the frontiers of the 3×3 thread block (see shaded boxes in Figure 2.14). Note that in order to achieve a conflict free concurrent execution of the thread blocks, the computations of some frontier volumes (like volumes $V_{3,3}$, $V_{3,4}$, $V_{3,5}$, $V_{4,3}$ and $V_{5,3}$) are replicated among the blocks. In the second phase, the volumes that do not belong to a ghost region update their flux by accumulating the left and up contributions saved in the shared memory buffers at the end of the first phase. Therefore, in the example only four (2×2) threads of each block do work in this second phase.

The improvement obtained with the implementation of the ghost cell decoupling

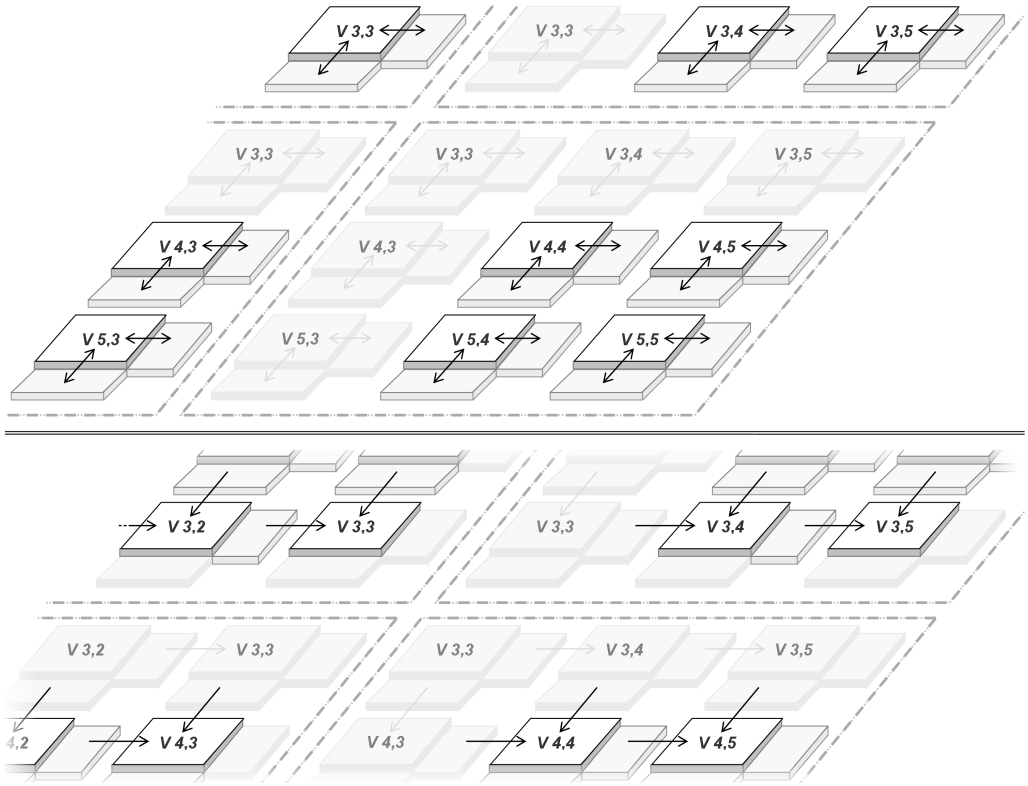


Figure 2.14: The two phases of the Ghost cell decoupling solution

technique grows with the thread block size because there are fewer threads that do not work in the second phase. For a block size `blockdimX`×`blockdimY`, the ratio of threads that perform the second stage is given by:

$$\% \; threads = \frac{(blockdimX - 1) \times (blockdimY - 1)}{blockdimX \times blockdimY} \times 100$$

For example, if the block size is 64 (8×8), then 49 (7×7) threads work in the second phase, which represents 76% of the threads in the block. If the block size is 16×16, this ratio increases to 88%. Thus, the percentage of threads by block that do not make the second phase is smaller for larger block sizes.

2.2.4.2. Two-phase reduction

This improvement consists in executing a part of the reduction of *Stage2* at the end of *Stage1*, thus reducing the amount of work of the reduction kernel of *Stage2*. Following this strategy, in the kernel of *Stage1* each thread block makes a local reduction of the $\Delta t[v]$ calculated by the threads belonging to this block (see $\Delta tLocal[v]$ in ①ⓒ of Figure 2.13) and that have already been stored in shared memory buffers (see ①ⓑ in Figure 2.13). Next, in *Stage2* the $\Delta tGlobal$ is computed reducing just the $\Delta tLocal$ obtained for each block in the previous stage.

For example, without this improvement, given a grid of 1000×1000 volumes, there would be 1000000 time steps (one per volume) to be reduced by the reduction kernel. Considering a thread block size of 8×8 in *Stage1*, the size of the vector Δt would be $1000000/49 \approx 20408$ elements. Note that the denominator is 49 because although the thread block size is 64 (8×8 threads), only 49 (7×7) of the threads are responsible for computing the time step ($\Delta tLocal$) in *Stage1* (see ①ⓒ in Figure 2.13). The remaining 15 threads process the volumes of the ghost regions and therefore do not compute time steps. Furthermore, let us realize that the thread block has the 49 values of $\Delta t[v]$ it has computed (see ①ⓑ in Figure 2.13) in shared memory, where the accesses needed to reduce them to a single value are much faster than in global memory. As a result, in this example only 20408 accesses to global memory will be required in the kernel of *Stage2*.

Another important performance consideration is that the thread block size must be well-balanced. On one hand, it must be large enough so that the percentage of threads that perform useful work in the second phase of the *Stage1* is high. On the other hand, it must be small enough to enable the parallel execution of enough thread blocks to keep busy the cores in the device. According to this, we have tried a set of thread block sizes and we have obtained the best performance for 8×8. Note that with size 8×8, each thread block needs 4 *KB* of shared memory. For a configuration of 48 *KB* for shared memory, it enables more than 8 simultaneous blocks in a single *SM*. Finally, this optimization requires another two changes with respect to the naive implementation: a buffer to perform the local reduction in the kernel of *Stage1*, and an adjustment of the grid size of the kernel of *Stage2*.

2.2.4.3. Usage of the texture memory

Despite the optimization described above, this algorithm still presents uncoalesced accesses to the *GPU* global memory because of the accesses in the y-direction of the grid of volumes. Specifically, the threads belonging to the same *halfwarp* access to different memory segments. *NVIDIA* advises in [99] to use texture memory for these cases, and this exploits its higher bandwidth if there is 2D locality in the texture fetches, avoiding this way uncoalesced loads. Although this recommendation is for devices with compute capability 1.X, in the case of the compute capability 2.X the performance is still better than the one obtained using global accesses and the *L1* cache [57], reason why we have applied this optimization.

It is important to mention that we use texture memory both for reads and writes. *NVIDIA* indicates [99] that if the global memory pointed by a texture is overwritten, the texture cache will stay in an inconsistent state and the following reads (within the same kernel) to these texture memory positions will return wrong values. Nevertheless, we have used the texture memory for arrays that are read and written by different kernels, therefore this problem does not exist in our case. The arrays that are accessed by the texture unit are: (1) the array of fluid of the previous iteration, which is stored as a 2D texture of `float4` elements and which changes in each iteration; and (2) the array of parameters, which is stored as a 2D texture of `float` elements and remains constant during the whole simulation. Let us emphasize that these data cannot be stored into constant memory because they require more than 64 *KB*, which is the maximum of constant memory size for devices of compute capability 2.X.

In [77] the texture memory is used instead of shared memory, meanwhile we store the parameters of the fluid in shared memory but we access to the global memory through the texture memory in order to avoid the uncoalesced accesses that would appear when the y-direction of the grid is followed.

2.2.5. Experimental results

Our evaluation has been performed in a *NVIDIA S2050* preconfigured cluster with 4 *M2050 GPUs*. The nodes have 12 *GB* of host *DDR3* memory and the general

purpose *CPU* is an *Intel Xeon X5650* at 2.67 *GHz*, with 6 cores and *hyperthreading* of 2 threads per core reaching a maximum memory bandwidth of 32 *GB/s*. Each *M2050 GPU* has 448 *streaming processors* and 3 *GB* of *GDDR5* memory. The software setup is *Debian GNU/Linux 6.0.1 (squeeze)* operating system using *g++ 4.3.5* and *nvcc* 4.0 compilers.

Following we will describe two different problems. The first one, presented in Section 2.2.5.1, is a simple academic test designed to measure the accuracy of the *GPU* simulator. The second, introduced in Section 2.2.5.2, is a more complex test designed to test the performance of the proposed implementations. It is based on a realistic world scenario that analyzes the evolution of a pollutant discharged in an estuary.

2.2.5.1. Simulator accuracy: Comparison with a reference CPU implementation

In this section a simple academic 2D test case is presented to verify the accuracy of the simulator. The test consists of a dam-break problem where a water column falls in a water tank creating a series of ripples that can be easily examined. We use a small $[-5,-5]\times[5,5]$ domain with the depth function defined as:

$$H(x,y) = 1 - 0.4e^{-x^2-y^2}, \tag{2.36}$$

and with the following initial condition:

$$W(x,y,0) = \begin{pmatrix} h(x,y,0) \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{2.37}$$

where

$$h(x,y,0) = \begin{cases} 4 & \text{when } x^2+y^2 \leq 0.36 \\ 2 & \text{otherwise} \end{cases} . \tag{2.38}$$

and the size of the side $\Delta x = \Delta y = 10$ / *number of volumes per side*.

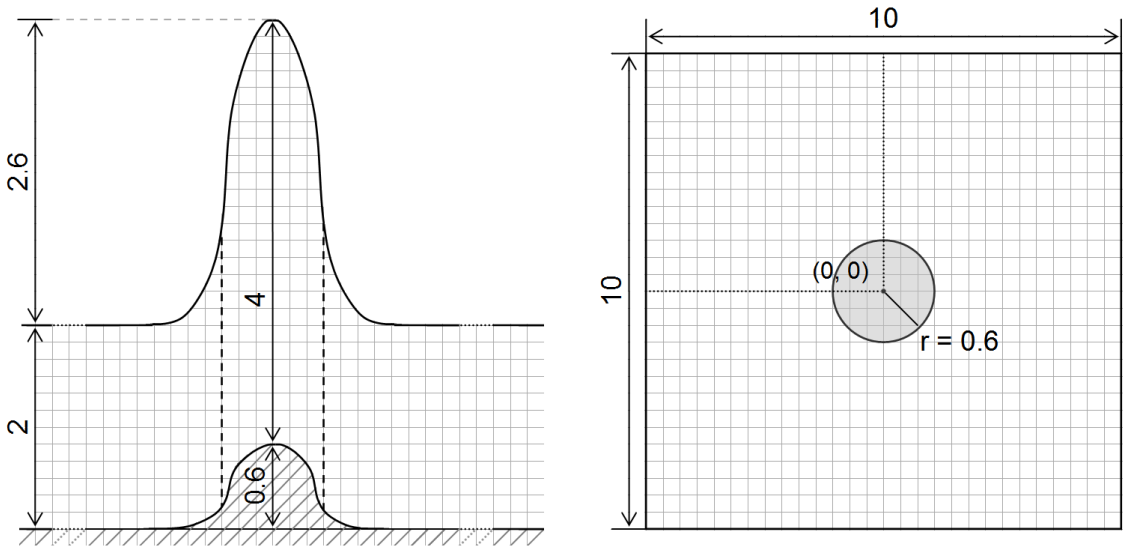
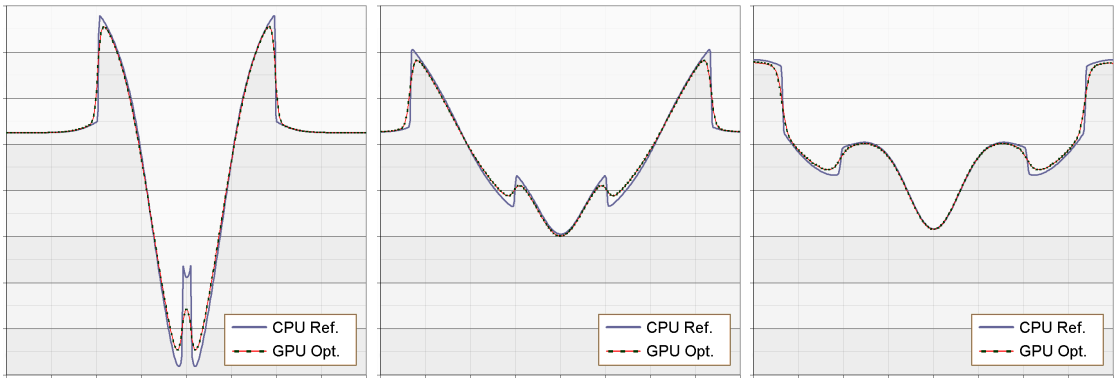


Figure 2.15: Diagram of the academic 2D test used for verification.

The simulations are executed in the time interval $[0, 1]$ for several mesh sizes using wall boundary conditions ($\boldsymbol{q} \cdot \boldsymbol{\eta} = 0$) and $CFL = 0.9$. Figure 2.15 shows a diagram of the initial setup. This test does not require wet-dry zone processing and serves to study the proper behavior of the forces and conservation of the fluid. Figure 2.16 represents the evolution of the test showing a bisection plane of the domain for 0.33, 0.66 and 1.00 seconds. In each figure there are two lines representing the water height: the *CPU* reference version (thick solid line), which is using a very fine mesh of 3200×3200 volumes and the initial waves were purposely quite high and sharp to be able to observe the behavior in the test; and *GPU* optimized version (thin dashed line), which is using a 400×400 mesh size and due to the lower resolution presents a slight difference with respect to the reference solution in the inflection points, where their contour tends to be more rounded.

Table 2.1 shows the value of the L^1 norm for $T = 1$ second for the meshes 100×100, 400×400, 1000×1000 using the *GPU* optimized version compared to the *CPU* reference version executed with the same resolution. The rows of the table show the error for each conformant parameter of the fluid. The measured numerical error for single precision data is negligible, so it does not affect the accuracy of the parallel shallow waters simulator.



(a) Time = 0.33 seconds (b) Time = 0.66 seconds (c) Time = 1.00 seconds

Figure 2.16: Evolution of the academic 2D test used for verification.

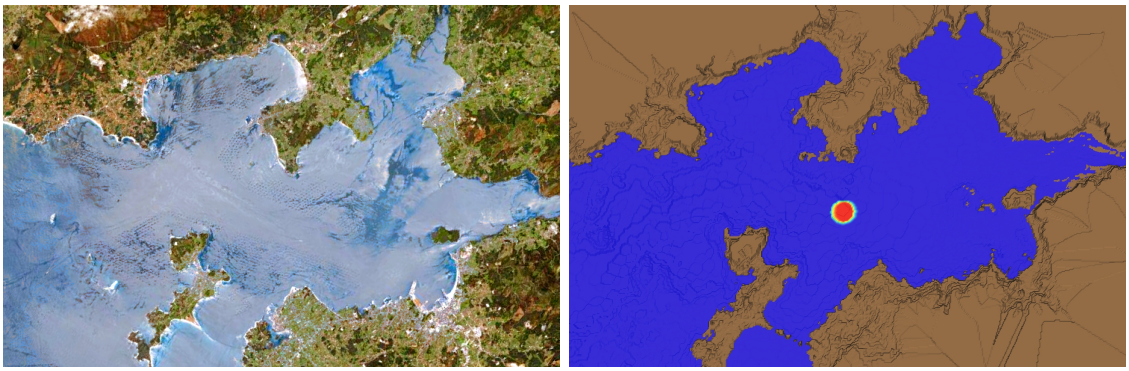
Table 2.1: L^1 norm at time $T = 1$ s for several meshes. The reference solution is *CPU* sequential

$L^1 error$	100×100	400×400	1000×1000
h	1,10e-7	8,75e-8	1,79e-7
q_x	1,40e-7	1,78e-7	5,79e-7
q_y	9,00e-8	1,28e-7	3,58e-7

2.2.5.2. Simulator behavior: synthetic test on Ría de Arousa (Spain)

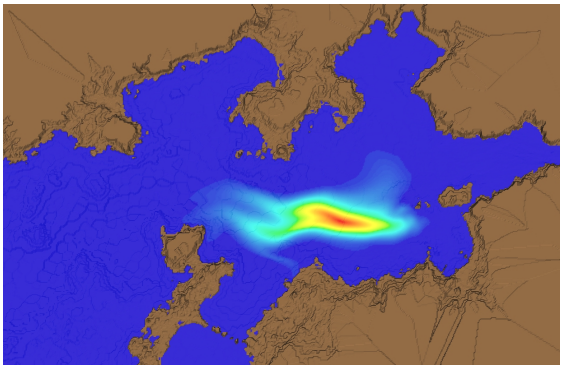
This second test uses a synthetic case in order to study the simulator behavior in a real world scenario. The simulation is based on an estuary in Northwest Spain called the *Ría de Arousa*, whose satellite image is displayed in Figure 2.17(a). This natural environment is simulated using the real terrain and bathymetry data in our test. While the north and east limits of the area involved in the simulation have free boundary conditions, the tides in the west and south borders are simulated using the main barotropic tidal components. Wet-dry fronts appear very often in this test in the coastal zones and emerging islands. The purpose of the simulation is to study the evolution of a pollutant that is discharged in this environment, determining its propagation and which are the most affected areas. The total simulated period is one week of real time.

The initial setup is represented in Figure 2.17(b). It corresponds to the moment when the pollutant is discharged in a circle with a radius of 400 m in the middle of the estuary. The normalized concentration of pollutant is given by the color scale

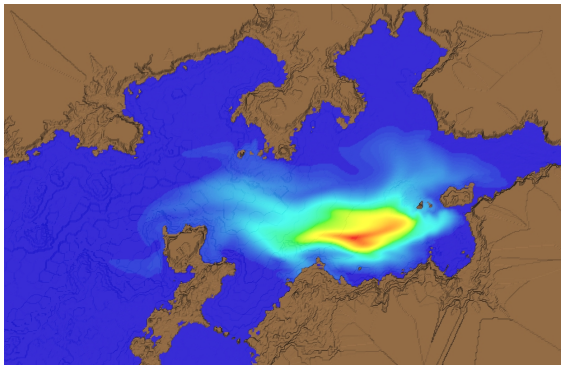


(a) Satellite image (GoogleMaps)

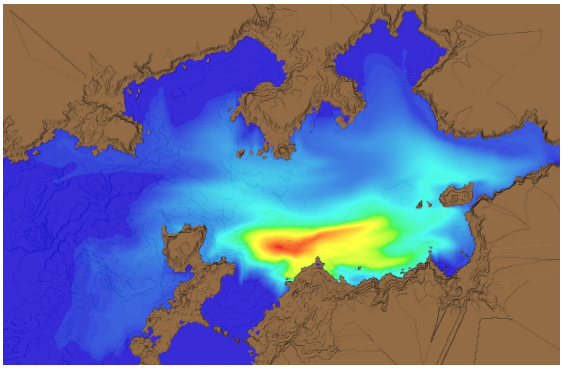
(b) Initial setup



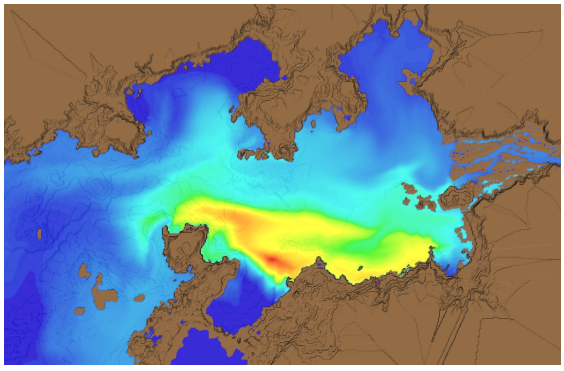
(c) Pollutant concentration after one day



(d) Pollutant concentration after two days



(e) Pollutant concentration after four days



(f) Pollutant concentration after eight days

Figure 2.17: Evolution of the Ría de Arousa simulation.

at the bottom of the figure. Figure 2.17(c) is a capture of our simulation after 24 hours. Here the sea currents have started to extend the pollutant along the estuary, but if containment measures and cleanup activities started at this moment, it would be possible to safely remove a large part of the contaminant. After another 24 hours of simulated time we reach the situation depicted in Figure 2.17(d), where the pollutant has spread further, increasing portions of it beginning to reach the seashore. Cleaning efforts could still be concentrated in a well defined zone and remove most of the contamination. In Figure 2.17(e), four days after the spill, the pollutant has spread over a large area, but a reasonable amount of waste material could still be drawn from the center of the stain. Cleaning activities can begin in some coastal zones too. After eight days (see Figure 2.17(f)) the damage is extensive and only a few areas remain relatively safe, such as the two north bays and the south one. Now most of the shore requires cleaning efforts, specially the south zone, but depending on the toxicity of the pollutant the process may have reached catastrophic dimensions. The test benchmark only simulates seven days, but here we used the eighth day to display an image during low tide, in which we can observe some small emerging islets. The model has provided a valuable simulation of the disaster evolution that makes possible to predict the most affected areas. Pollutant discharge may not only have a deep impact on the natural environmental, but also affect very negatively the economy of regions where seafood products or tourism are relevant industries.

Table 2.2 shows the execution time and the speedups for several mesh sizes. All our implementations use single precision data. The *CPU* times were taken on the system that was described above, using *OpenMP* [108] to take advantage of that multicore chip. Since the second thread provided by *hyperthreading* typically only provides 15% to 20% of the performance of a real core, we can see that our *OpenMP* implementation is very efficient. The speedups of the naive *GPU* implementation are calculated with respect to the *CPU* times. The speedups of the *GPU* optimized version have been obtained with respect to the *GPU* naive version. The simulation of the biggest mesh requires about 36 hours in a multithreaded *CPU* implementation and just 107 minutes for the *GPU* version based in recomputation. With the optimized *GPU* version the same simulation takes only 91 minutes.

Table 2.2: Execution times (in seconds) and speedups

Mesh size	CPU			*GPU*			
	sequential time	OpenMP		*naive*		*optimized*	
		time	speedup	time	speedup	time	speedup
100×100	932	157	5.92x	12.28	12.78x	11.6	1.06x
200×200	7440	1086	6.85x	64.96	16.71x	59.04	1.10x
300×300	23912	3443	6.95x	203.90	16.89x	169.77	1.20x
400×400	56256	8361	6.73x	447.19	19.71x	387.69	1.15x
500×500	109201	16527	6.61x	878.55	18.81x	730.09	1.20x
600×600	188125	28580	6.58x	1455.18	19.64x	1231.97	1.18x
700×700	297849	45344	6.57x	2343.87	19.35x	1928.40	1.22x
800×800	443247	67110	6.60x	3322.87	20.20x	2849.49	1.17x
900×900	629210	95461	6.59x	4848.76	19.69x	4020.39	1.21x
1000×1000	860457	130815	6.58x	6448.80	20.29x	5455.51	1.18x

2.2.5.3. Isolated impact of the improvements applied

Table 2.3 shows the evolution of the performance after applying step by step the improvements explained in Section 2.2.4 to the naive implementation. It is an incremental development so that each version includes all the improvements of the previous ones. The speedups of each version have been measured with respect to the times of the previous version. There are two intermediate versions: *evol-I* and *evol-II*. The *evol-I* version is equal to the *GPU naive* version after replacing its first recomputation-based kernel of *Stage1* with the ghost cell decoupling-based kernel (see details in Section 2.2.4.1). The *evol-II* version includes additionally the local reduction in the kernel of *Stage1* taking advantage of using shared memory buffers and the subsequent modifications of the size of the kernel of *Stage2* (see Section 2.2.4.2). Finally, the last version, *GPU optimized*, also contains the last improvement applied in our development, i.e., the usage of texture memory (see Section 2.2.4.3).

The *local reduction* improvement evaluated in the *evol-II* column represents a poor contribution to the overall speedup. This improvement is aimed at reducing the work and the number of accesses to global memory of the reduction kernel of *Stage2*. This kernel performs little work for the smaller meshes and applying this improvement has no impact on performance. As the work of the reduction kernel increases, the speedup provided by this optimization grows too. The best

Table 2.3: Execution times (in seconds) and speedups after applying each improvement separately

Mesh size	Num. Iter.	*GPU naive*	*evol-I*		*evol-II*		*GPU optimized*	
		time	time	speedup	time	speedup	time	speedup
100 × 100	164243	12.28	11.71	1.05x	11.95	0.98x	11.55	1.03x
200 × 200	335514	64.96	63.35	1.03x	64.84	0.98x	59.04	1.10x
300 × 300	503362	203.90	187.49	1.09x	189.06	0.99x	169.77	1.11x
400 × 400	671293	447.19	424.27	1.05x	426.26	1.00x	387.69	1.10x
500 × 500	839237	878.55	797.99	1.10x	799.51	1.00x	730.09	1.10x
600 × 600	1007255	1455.18	1337.99	1.09x	1334.99	1.00x	1231.97	1.08x
700 × 700	1175349	2343.87	2141.43	1.09x	2139.18	1.00x	1928.40	1.11x
800 × 800	1343453	3322.87	3196.86	1.04x	3128.72	1.02x	2849.49	1.10x
900 × 900	1511582	4848.76	4550.57	1.07x	4458.91	1.02x	4020.39	1.11x
1000 × 1000	1679708	6448.80	6146.36	1.05x	6025.03	1.02x	5455.51	1.10x

improvement percentage is achieved by the usage of the texture memory. The *GPU* used in this study (*Tesla S2050*) is a device with 2.0 compute capability, which has *L1* cache for the global memory. The usage of texture memory is more recommended for *GPUs* of lower compute capabilities because the use of texture cache has a bigger impact in devices that have no *L1* cache for their global memory. However, in our case the usage of texture memory means a noticeable 10% of improvement percentage the optimization of the uncoalesced memory accesses (see details in Section 2.2.4.3).

Chapter 3

Influence of memory access patterns to small-scale FFT performance

In this chapter we use the *FFT* (*Fast Fourier Transform*) as a benchmark tool to analyze different aspects of *GPU* architectures, like the influence of the memory access pattern or the impact of the register pressure. The *FFT* is a good tool for performance analysis because it is used in many digital signal processing applications and has a good balance between computational cost and memory bandwidth requirements. This chapter represents a second step towards our methodology for efficient algorithm design on *GPU* architectures where memory access patterns and register usage are analyzed. The work presented in this chapter was originally introduced in [56, 57].

3.1. FFT benchmarks using CUDA

To analyze the performance of the different *GPU* storage types a set of different *FFT* configurations were executed in *CUDA*. For information about the *NVIDIA GPU* architecture and a description of the memory hierarchy exposed by *CUDA* see Section 1.1.2. The *FFT* requires a significant amount of computation, and

using transforms of different sizes and distributions [66, 89] it is possible to study the influence of several parameters separately. Our implementation is based on the *Cooley-Tukey* algorithm [66], characterized by its regular structure and easy implementation. This algorithm performs a bit-reversal operation at the beginning of the process, and then it operates on the data in pairs while increasing the stride in each stage.

In our FFT benchmarks each thread calculates an independent FFT and each block is composed by L threads which operate on different input data in batch mode. Therefore, we have developed a set of kernels for each signal of size $N = \{4, 8, 16, 32\}$. The kernels are recursively subdivided into smaller problems until reaching the base case of $N = 2$. The tests are centered on architecture analysis, therefore, in order to exercise different configurations in analogous conditions, only small problems were used to simulate a computational workload. Twiddle factors w_N^{ik} are stored in constant memory. This memory can be used to store the result of precalculated formulas, thus avoiding redundant computations or expensive global memory requests. Constant memory is optimized for broadcast memory access, where many threads read the same location, otherwise the requests may be serialized.

Figure 3.1 shows an example of the kernel used to compute the FFT for a signal of $N = 8$ data. It presents a complex input signal which can be processed in either registers or shared memory, recursively performing an in-place FFT (functions in lines 2, 9 and 18). Global and texture memory can also be used directly as inputs for the FFT_time kernel (line 32), however in our tests data will reside locally in the *SPs* (*Streaming Processors*) before calling the *FFT*. FFT_time is part of a bigger function $FFTy$ (see Figure 3.2), which is the main kernel that manages the storage type and it will be called by the host. In line 5, $FFTy$ reads the stride that will be used to access the different input signals within the batch, and in line 6 this stride is used to obtain a pointer to the problem that will be processed by the current thread. Following, the memory to store the signal is reserved, either using shared memory (lines 9 and 10) or registers (line 12), and then the data is copied to the current thread from texture memory (line 16) or global memory (line 18). Next, the FFT is performed, calling a forward FFT function (line 23) or a reverse and scale function (lines 25 and 26) depending on the direction DIR, which is a compile-time parameter. Finally, in line 29 the data is copied back to global memory. Notice

```
1:  inline __host__ __device__ void
2:  FFT2(Complex &a0, Complex &a1) {
3:     COMPLEX c0 = a0;
4:     a0 = c0 + a1;
5:     a1 = c0 - a1;
6:  }
7:
8:  inline __host__ __device__ void
9:  FFT4(Complex &a0, Complex &a1, Complex &a2, Complex &a3) {
10:    FFT2(a0, a2);
11:    FFT2(a1, a3);
12:      a3 = make_COMPLEX(a3.y, -a3.x);
13:    FFT2(a0, a1);
14:    FFT2(a2, a3);
15: }
16:
17: inline __host__ __device__ void
18: FFT8(Complex &a0, Complex &a1, Complex &a2, Complex &a3,
19:       Complex &a4, Complex &a5, Complex &a6, Complex &a7) {
20:    FFT4(a0, a2, a4, a6);
21:    FFT4(a1, a3, a5, a7);
22:      a3 =       make_COMPLEX(       a3.y,     -a3.x);
23:      a5 = a5 * make_COMPLEX( ANG_4_8,-ANG_4_8);
24:      a7 = a7 * make_COMPLEX(-ANG_4_8,-ANG_4_8);
25:    FFT2(a0, a1);
26:    FFT2(a2, a3);
27:    FFT2(a4, a5);
28:    FFT2(a6, a7);
29: }
30:
31: template<> inline __host__ __device__ void
32: FFT_time<8>(Complex *a) {
33:      FFT8(a[0], a[4], a[2], a[6],
34:           a[1], a[5], a[3], a[7]);
35: }
```

Figure 3.1: FFT kernel for N=8

that data could also be directly operated using the *src* pointer without performing the explicit register copy operations (like in conventional *CPU* code), however this would result in only a quarter of the performance. In line 32 the template $FFTy$ (line 1) is instantiated with different parameters in a table of function pointers.

Table 3.1 depicts some information for each kernel compiled for *CUDA* 2.0 capabilities using the verbose flag *–ptxas-options=-v* and allowing the compiler to take as many registers as necessary, up to a maximum of 63 registers for the *Fermi* architecture. If the maximum number of registers is reached, the compiler will resort

```
template<int N, int DIR> __global__ void
FFTy(Complex* src) {
    int posX = get_global_id(0);
    int posY = get_global_id(1);
    int stride = get_global_size(0);     // Stride among elements
    src += posX + posY * N * stride;     // First pos in current batch

    #if SHM_MODE == 1                    // Registers or Shared Mem.
        __shared__ Complex _tmp[(N + 1) * 32];
        Complex* tmp = _tmp + (N + 1) * get_local_id(0);
    #else
        Complex tmp[N];
    #endif

    #if TEX_MODE == 1                    // Texture or Global Mem.
        copyY<N>(tmp, posX, posY * N);
    #else
        copy<N>(tmp, src, stride);
    #endif

    rev<N>(tmp);                         // Bit reversal
    if(DIR > 0) {                        // Compile-time condition
        FFT_time<N>(tmp);                // Forward FFT
    } else {
        IFFT_time<N>(tmp);               // Inverse FFT
        scale<N>(tmp);                   // Scaling
    }

    copy<N>(src, stride, tmp);           // Write back to Global Mem.
}

void(*fft_funptrs[2][4])(Complex*) = {
    { FFTy<4, 1>, FFTy<8, 1>, FFTy<16, 1>, FFTy<32, 1> },
    { FFTy<4,-1>, FFTy<8,-1>, FFTy<16,-1>, FFTy<32,-1> }
};
```

Figure 3.2: General kernel template

to local memory to supply enough space for the private thread data. For example, in the register based implementation for $N = 16$, 63 registers and 16 bytes of local memory will be required, while for $N = 32$, the private space requirements are doubled, however no more than 63 registers can be allocated. Thus, the compiler will reserve 336 bytes of local memory (42 single precision complex values), which will cause a performance degradation when accessing these data. The older *Tesla* architecture allows up to 128 registers per thread. If the same kernel is compiled for *GPUs* with *CUDA* 1.3 capabilities, 58 registers would be allocated for $N = 16$ and 123 registers would be allocated for $N = 32$, thus, preventing the local memory

Table 3.1: Compiler information for the FFT kernel (Fermi CUDA cap. 2.0)

	Register based (RGC)			Shared memory based (SGC)			
N	Registers	Local (bytes)	Const (bytes)	Registers	Local (bytes)	Const (bytes)	Shared (bytes)
4	18	0	0	20	0	0	1280
8	36	0	4	26	0	4	2304
16	63	16	12	44	0	12	4352
32	63	336	28	63	52	28	8448

spilling in both cases. In the shared memory based implementation the local memory spilling is reduced. For $N = 16$ the kernel uses 4352 bytes of shared memory, 44 registers and no local memory, while for $N = 32$ the kernel uses 8448 bytes of shared memory, 63 registers and only 28 local memory bytes are allocated. Notice that this amount of shared memory can reduce the *GPU* ability to handle block level parallelism. Depending on the cache configuration, only one block may be scheduled per *SM*, resulting in idle resources and thus, leading to a performance degradation. Regarding the *GPU* occupancy, according to the *NVIDIA Compute Visual Profiler* tool, it is only at 16.6% when using $L = 32$ and 8 concurrent blocks. In this case, lowering the occupancy provides more resources for each thread. Even at lower occupancies the *GPU* can offer very good performance if the code presents enough opportunities to hide both instruction and memory latency using *SM* parallelism (the *Fermi* architecture is able to simultaneously schedule up to 48 *warps* or up to 8 blocks per *SM*) and useful computations [131, 132]. Table 3.1 also displays information about the total constant memory reserved by the kernel (in bytes), for example used by twiddle factors.

Our tests will be executed with $L = 32$ threads per block. The utilization with just 32 threads may seem rather low but, according to our tests, in most cases there is no significant advantage using 64 threads per block, and with 128 a small performance drop is experienced. Also note that the amount of shared memory in some of the tests may be too tight, thus restricting the maximum block size, so a common size of 32 was used for all the executions. Using just 32 threads it is possible to take advantage of the maximum number of *GPU* registers and, whenever enough resources remain available, the *GPU* will be able to transparently execute up to 8 blocks per *SM*.

Following, three decisive parameters will be analyzed for an optimal *CUDA* im-

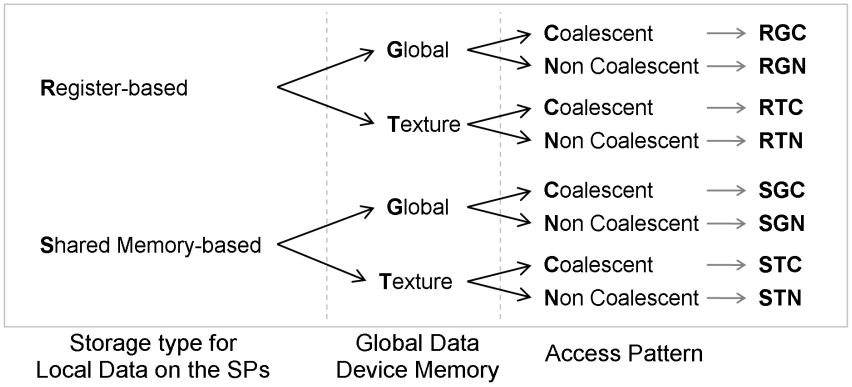


Figure 3.3: Test configuration

plementation: Storage type for Local Data on the *SPs*, storage type for input data, and access pattern of global data. The different configurations considered in our tests are shown in Figure 3.3. Each test will be assigned a three letter code according to its configuration parameters. For example, *STC* will mean that the test was performed using **S**hared memory, reading data from **T**extures with a **C**oalescent memory access pattern.

3.1.1. Storage type for thread data

With respect to the register-based solution (R), a key feature of this implementation is how the register pressure of an algorithm may affect performance. When more registers than the maximum allowed by the compiler configuration or available in the architecture are required, local memory is allocated. Local memory is private in the scope of each thread and does not offer very good performance because at the hardware level on the *Fermi* architecture is implemented using normal cached global memory. In the shared memory implementation (S), to avoid bank conflicts the data is stored in the shared memory using one element of padding between consecutive signals. This slightly increases the amount of shared memory required by the test from $N \times L$ to $(N + 1) \times L$ elements, but it is far more efficient.

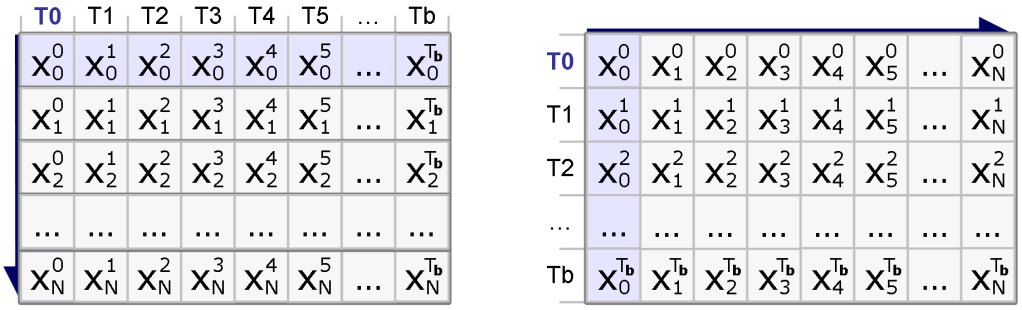
3.1.2. Storage type for input data

Storage type for input data is also analyzed. There are two different memory spaces accessible by all the *SPs*: Global (G) and Texture (T) memory. Texture memory can only be used for reading data, but the texture cache is specially efficient if there are redundant read operations or there is spatial coherency in the access pattern.

3.1.3. Memory access pattern

The impact of the memory access pattern and coalescence is studied using two different data distributions. Coalescent memory access is used to group several global memory requests in a single one, thus reducing effective bandwidth usage and also pressure in the memory controller, that will receive less requests. In a coalescent access pattern the tasks within a half-warp access data in the same memory segment, and in a non coalescent access two or more tasks access different segments, so they are not performed simultaneously.

Figure 3.4 shows the two signal distributions used in this work. The first data distribution (see Figure 3.4(a)) is a coalescent memory access pattern (C), where the data of the input signals is stored sequentially, so each thread L_i is assigned to read $\{x_0^i, x_1^i, \ldots, x_N^i\}$, therefore the data read by the corresponding warp in each iteration is located in the same segment. For example, the first read request of the first warp will be $\{x_0^i, x_0^{i+1}, \ldots, x_0^{i+32}\}$ (elements shaded in Figure 3.4(a)). The total amount of bytes required by the block will be $BSize = L \times N \times 8$ bytes per single precision complex value, and the total number of read operations is $BSize/128$ bytes per transaction for aligned data. The second data distribution is a non-coalescent pattern (N). Figure 3.4(b) displays this distribution in which each signal is stored sequentially, so the accesses in the same segment are only $segment\_size/N$. For example, in the case of Figure 3.4(b) (assuming $N \times batch_x > segment\_size$), the first read performed by the first warp will be composed by the shaded elements $\{x_0^0, x_0^1, \ldots, x_0^L\}$, which in principle will originate L read requests in different segments. Depending on the data alignment and the hardware *CUDA* capabilities, a non-coalescent access may generate up to $N \times L$ different memory requests per



(a) Coalescent (b) Non-coalescent

Figure 3.4: Memory access patterns

block. Both access patterns are important, as sometimes applications may require complex changes in order to prevent non-coalescent data access.

3.2. Experimental results

All the tests were run in single precision using complex input signals in the range $N = \{4, 8, 16, 32\}$ and batch execution to perform several *FFTs* each time. The memory requirements are kept constant at 2^{24} elements and the size of the batch depends on the input size and is given by the expression $batch = 2^{24}/N$, so as the input signal increases the number of batch executions decreases. To prevent interactions with the study of the memory hierarchy all the data resides on the *GPU* device memory at the beginning of each test. The performance of the experiments is measured in *GFLOPS* through the commonly used formula given by Equation (1.4).

One of our test platforms is a *Core 2 Duo E8400* processor with 2 *GB DDR3 1333* memory and two *GPUs*: a *GeForce 280*, based on the *Tesla* architecture, and a *GeForce 480*, based on the *Fermi* architecture. The software setup is *Windows XP x64* operating system, using *Microsoft Visual C++ 2008* compiler (x64, release profile) and *CUDA 4.0 SDK* with the *270.81 GPU* driver. The other test platform is a dedicated server composed by a *Xeon X5650* processor, 4 *GB DDR3 1333* memory and a *Tesla S2050 GPU* node with *ECC* disabled. The software setup is *Debian 6.0.5*, using *gcc 4.3.5* compiler (x64 mode, *-02* optimization level) and *CUDA 4.0 SDK* with the *270.41 GPU* driver.

3.2.1. Cache and ECC configuration

GPUs based on the *Fermi* architecture let the user choose the cache configuration between having either 48 *kB* of shared memory but only 16 *kB* of *L1* cache, or just 16 *kB* of shared memory and 48 *kB* of *L1*. The *Tesla S2050* also allows the user to enable *ECC* (Error-Correcting Code) memory capability. Figure 3.5 compares the efficiency of the two cache configurations and the *ECC* memory on the *Tesla S2050* (the *GeForce 480* does not support *ECC* and the cache configuration results are similar). Only the best cache configuration *ECC* results are presented in the figure.

Regarding the influence of the cache mode, for small problems the difference using *RGC* configuration is not significant (see *48L1 RGC S2050* and *16L1 RGC S2050*), but for $N = 16$ the bigger *L1* cache configuration offers a bit more performance (around 5%), while *SGC* configuration loses about a 12% for $N = 8$ and nearly a 50% for $N = 16$ (see *48L1 SGC S2050* and *16L1 SGC S2050*). For $N = 32$ both differences increase, using the $48L1$ cache configuration it is possible to improve the result in about a 33% in *RGC*, while losing 65% of the performance for *SGC*. The big difference between cache configurations in this case points to a limitation in the number of simultaneous blocks per *SM*: Each thread requires enough shared memory to fit a whole *FFT*, thus more than 8 *kB* of shared memory are reserved for $N = 32$ and only one block will be executed with 16 *kB* of shared memory, so latency hiding techniques will not work as expected. In the following experiments of this work, the $48L1$ configuration will be used for register based tests and the $16L1$ configuration for shared memory based tests.

Regarding the influence of enabling *ECC* it causes about a 10% performance degradation in the *SGC* test, and between 13% and 25% degradation in the *RGC* test because it is more sensitive to the reduction in the memory bandwidth. Based on the results, the *RGC* configuration is mostly memory bandwidth bound, as the performance drop is proportional to the 23.6% effective bandwidth reduction due to *ECC*. Additionally, the *Compute Visual Profiler* tool confirms that the non-*ECC* version of the *RGC* test achieves 128 *GB/s* for $N \leq 8$, which is an 86% usage of the available memory bandwidth. For $N = 16$ the computing part gains more weight, but the effective bandwidth only decreases to 127 *GB/s*, reinforcing the bandwidth bound hypothesis. For $N = 32$ only 118 *GB/s* are obtained.

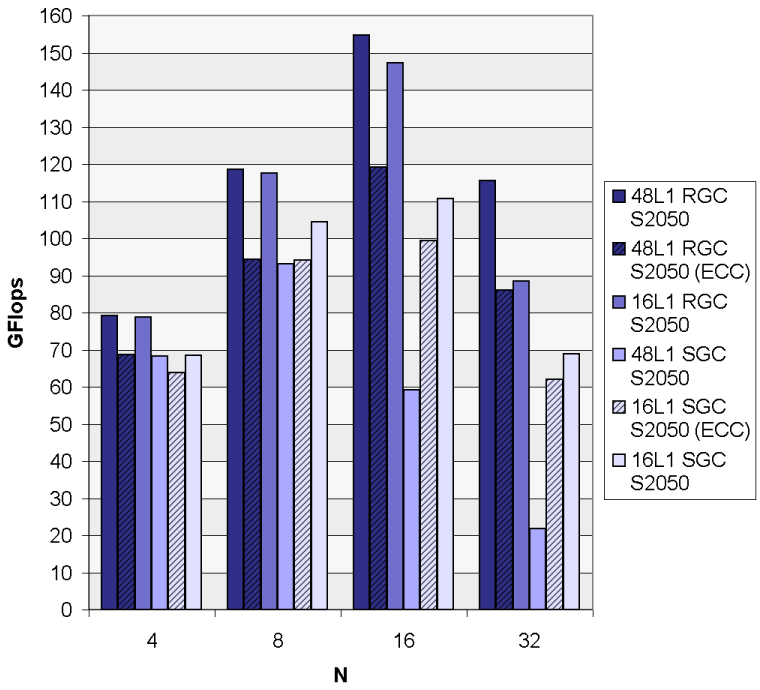


Figure 3.5: 48L1 vs 16L1 cache configuration and ECC performance (T2050)

3.2.2. Registers vs Shared memory

In Figure 3.6 the performance of the register implementation (*RGC*) is compared to the shared memory version (*SGC*) for both *Fermi GPUs*, using coalesced access to global memory. Observe that the performance of *RGC* configuration is always higher than *SGC* configuration. For example, for $N = 16$ *RGC* achieves 185 *GFLOPS* on the *GeForce 480* and 176 *GFLOPS* on the *Tesla 2050*, while *SGC* obtains just 144 *GFLOPS* on the *GeForce 480* and 111 *GFLOPS* on the *Tesla 2050*. The bandwidth of the shared memory is lower than the register bandwidth, therefore it results in reduced performance for the *SGC* test. The results are similar for small N, but if N increases the difference between *SGC* and *RGC* also increases. Observe how the performance improves for all cases until $N = 16$ but then decreases for $N = 32$. The number of operations per thread increases with the size of the problem, therefore the *GPU* can make better usage of the execution resources. However, if a thread has a big working set which does not completely fit in the registers, its content is spilled to the next level in the *GPU* memory hierarchy, the slower *local memory*, as seen in Table 3.1. Additionally, when kernels require too many registers, less blocks

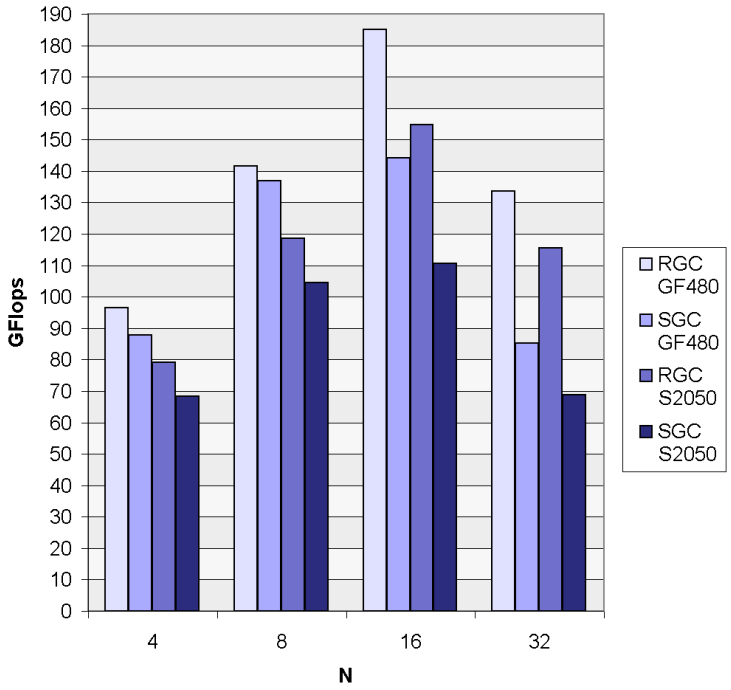


Figure 3.6: RGC vs SGC performance (GF480 & S2050)

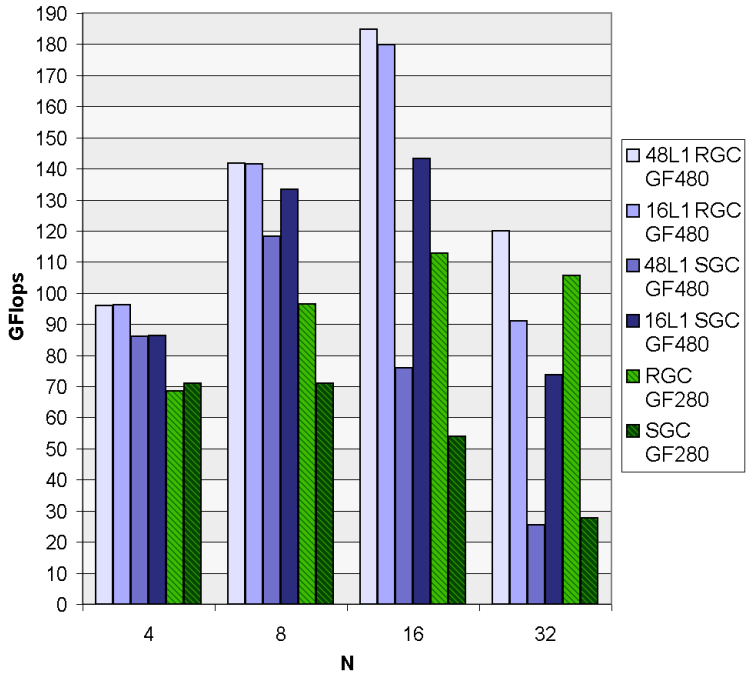


Figure 3.7: RGC vs SGC performance (GF480 & GF280)

may be simultaneously scheduled in the *GPU*. To avoid the performance degradation experienced for $N \geq 32$ due to the high register pressure, a different implementation where several threads cooperate in the same problem would be more suited.

Figure 3.7 repeats the test, but comparing the *GeForce 480* using its two different cache configurations and the *GeForce 280*. The results are quite close for $N = 4$, but as N increases the difference becomes quite significant, specially for $N = 16$. The performance of the *RGC* configuration for the *GeForce 280* also improves until $N = 16$, slightly decreasing for $N = 32$. Interestingly, for $N = 32$ the difference between the two *GPUs* is quite small. The reason behind the small performance drop for the *GeForce 280* (just about a 6%, compared to more than a 40% for the *GeForce 480*) is that the architecture supports up to 128 registers per thread, therefore preventing local memory spilling. Nonetheless, allocating too many registers for a single thread reduces the parallelism leading to the observed 6% performance drop. Observe that in the *SGC* tests, the *GeForce 280* has a proportional scaling behavior to the *GeForce 480* when using the smaller shared memory configuration. For $N = 32$ *SGC* it barely reaches a 26% of the *RGC* performance, since allocating such big portions of shared memory for a block reduces too much the number of simultaneous blocks per *SM*. Moreover, the performance of *SGC* for $N = 32$ in the *GeForce 280* is less than half compared to $N = 16$, nonetheless it is slightly better than the *GeForce 480* when using just 16 *KB* of shared memory.

3.2.3. Global memory vs Texture memory

The second configuration parameter that will be studied is the impact of the choice between texture memory (*RTC* configuration) and global memory (*RGC* configuration). Only test results using register configuration (R) are shown, as it was the best performing option according to Section 3.2.2. As seen in Figure 3.8, both *Fermi GPUs* experience a similar degradation using texture memory. Under 10% for $N \leq 16$ and under 20% for $N = 32$. According to the *CUDA C Best Practices Guide*, the *L1* data cache of the *Fermi* architecture has higher bandwidth than the texture cache. In contrast, for the *GeForce 280* (see Figure 3.9) the texture memory provides up to a 32% improvement over the global memory version. This is the normal behavior, as on the *Tesla* architecture the global memory is uncached, so

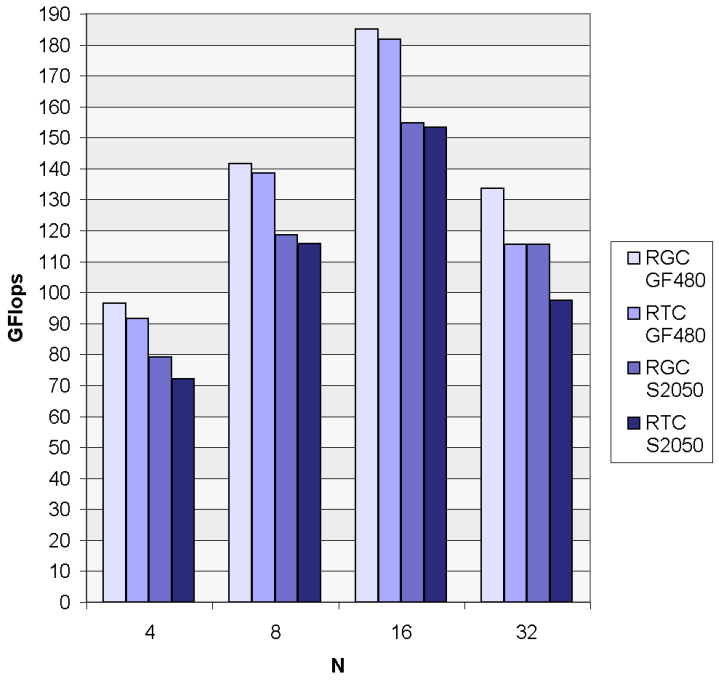


Figure 3.8: RGC vs RTC performance (GF480 & S2050)

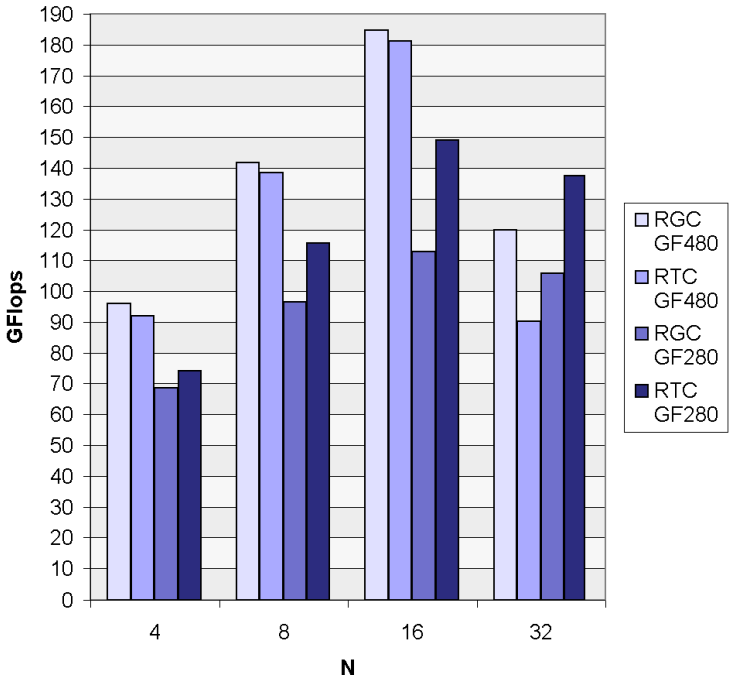


Figure 3.9: RGC vs RTC performance (GF480 & GF280)

the texture cache can be used to reduce the number of memory fetches. Furthermore, for the particular case of $N = 32$, *RTC* executed in the *GeForce 280* is able to outperform both *RGC* and *RTC* configurations on the *GeForce 480* thanks to the greater number of registers per thread available in the *Tesla* architecture and the usage of the texure cache.

3.2.4. Coalescent memory access vs Non-coalescent

The last parameter considered was the impact of the access pattern. In Figure 3.10 *RGC* and *RGN* are compared when executed in the *Fermi* architecture. Changing the global memory access pattern to force a non-coalescent access causes a performance drop in both *GPUs*, as it was expected. For instance, for $N = 8$ the *GeForce 480* loses around a 36% of the performance, while the *Tesla 2050* loses around a 42%. Nevertheless remark that, even with the advances in *Fermi* with the cached memory access, nearly half the performance may be lost for non-coalescent memory access patterns. In the case of the *GeForce 280* based on the *Tesla* architecture there is no global memory cache. Thus, it becomes around seven times slower for some problem sizes (see Figure 3.11). Observe that even with the advantage of the global memory cache, the *GeForce 480* using the *RGN* configuration offers lower performance than the *GeForce 280* in the *RGC* test, therefore this is a very important factor.

An interesting additional comparison is the impact of the coalescence when making heavy use of shared memory and trying to minimize the performance cost of the uncoalesced access through texture memory. In this sense, the texture cache can be exploited playing a similar role to the *L1* cache. Figure 3.12 compares the performance of *SGN* and *STN* configurations in the *Fermi* architecture. Notice how in this case (in contrast to Figure 3.8), texture cache memory improves performance in about a 20% for both *GPUs*, because additional cache memory can be used without decreasing the amount of cache assigned to shared memory. Comparing *RGN* and *RTN* configurations does not yield a similar improvement because the kernel is already using $L1$ for caching, which provides slightly better bandwidth than texture cache. Thus, in this case *RGN* performs about 15% better than *RTN*.

Figure 3.13 repeats the previous test comparing the *GeForce 480* and the *GeForce*

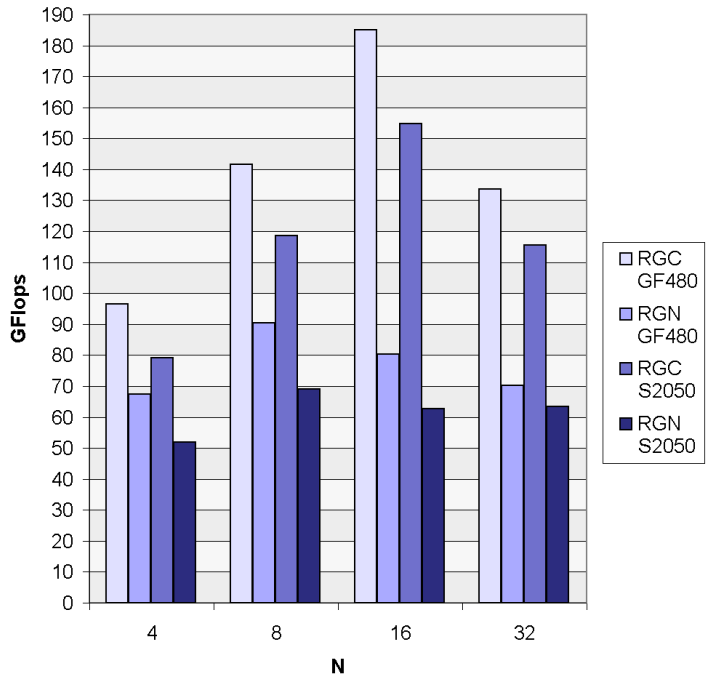


Figure 3.10: RGC vs RGN performance (GF480 & S2050)

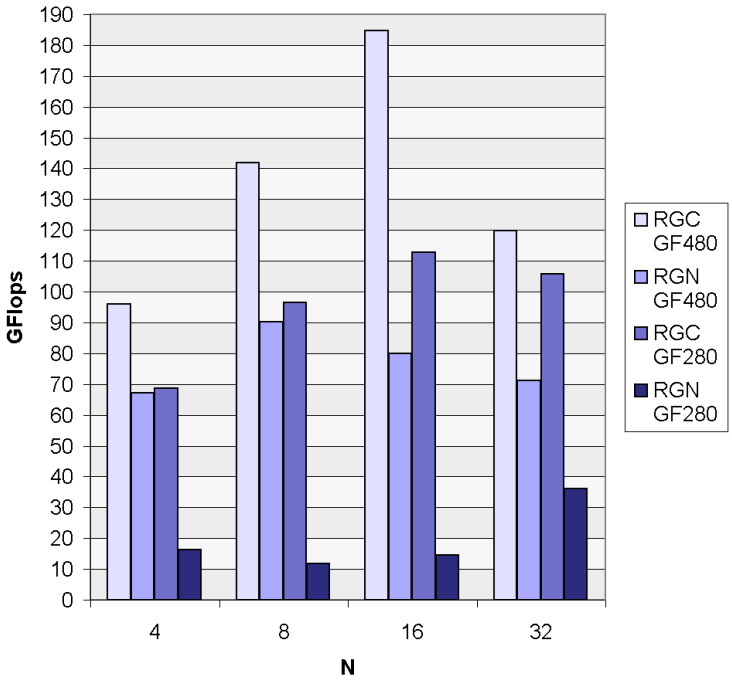


Figure 3.11: RGC vs RGN performance (GF480 & GF280)

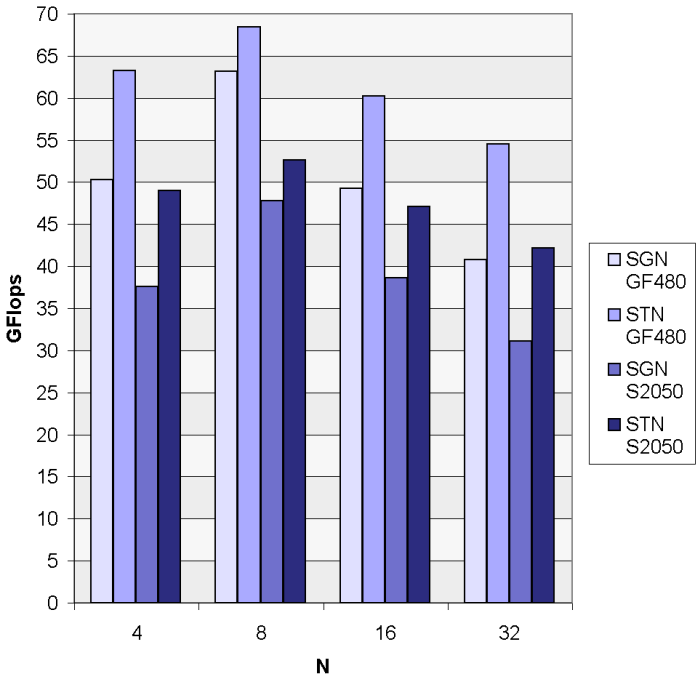


Figure 3.12: SGN vs STN performance (GF480 & S2050)

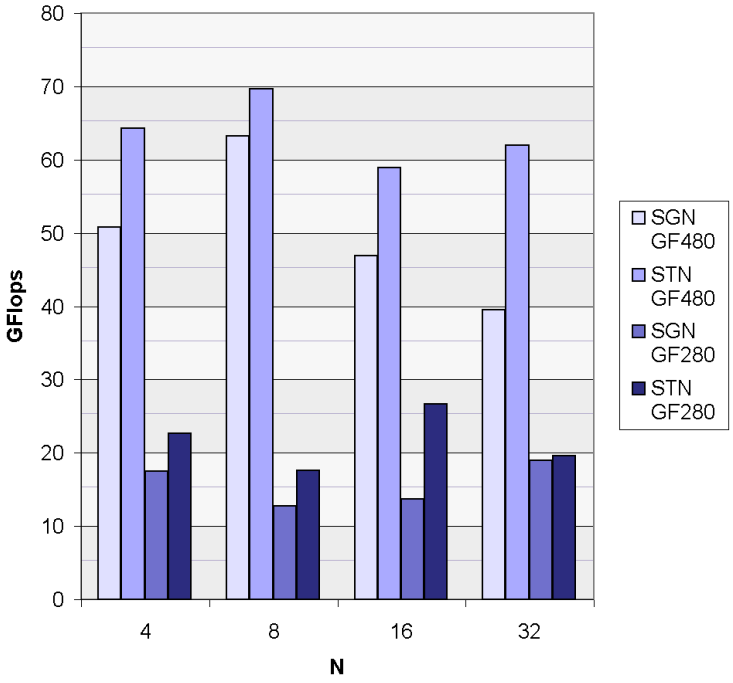


Figure 3.13: SGN vs STN performance (GF480 & GF280)

280. The *GeForce 280* experiences a similar improvement due to the texture cache except for $N = 32$, where the dispersion of data and pressure on the texture cache is too big. It is very interesting to observe that for $N = 16$ the texture cache configuration (STN) is able to double the performance of the standard global memory access (SGN). Even with the improvement of the texture cache, the coalescence is a critical factor in the *Tesla* architecture and the *GeForce 280* is too far form the *GeForce 480*, only offering around a third of the performance.

3.2.5. Comparison with other state-of-the-art implementations

Finally, although this chapter had the initial aim of developing a *FFT* implementation focusing on flexibility and programmability instead of performance, the results are very competitive for the addressed problem sizes.

Figure 3.14 compares *RGC* executed in the *GeForce 480* (which in general is between 15% and 30% faster than the *Tesla S2050*) with the *CUFFT 4.0* (also executed on the *GeForce 480*), the *RTC* version executed on the *GeForce 280*, the *Brook+ GPU* version presented in [55] (running on a Radeon 5870 *GPU*), and the *Spiral 6.0* library as a reference *CPU* implementation. With exception of the *Brook+* version, the rest of the tests were executed on the first platform described in Section 3.2. As can be observed, *GPU* based solutions offer a clear advantage over *CPU* (in this case *Spiral*, but other solutions like the *Intel IPP* library [49] offer similar performance), resulting at least twelve times faster. Comparing our two best implementations, *RGC* running on the *GeForce 480* is around 23% faster than *RTC* running on the *GeForce 280*, which nonetheless is a good result for the older generation *GPU*. Furthermore, for $N = 32$ due to the additional registers the *Tesla GPU* obtains a 14% advantage over *Fermi*, showing the importance of preventing local memory spilling. Last, observe that the proposed *FFT* is quite efficient, in fact when executed on the *Fermi* architecture it is slightly faster than the *CUFFT* for small signals up to 16 elements. The good scaling for these signal sizes also reveals that the main limiting factor is not the computing power but the effective memory bandwidth. Again, remark that next chapters will show new *FFT* implementations achieving competitive performance for larger signal sizes.

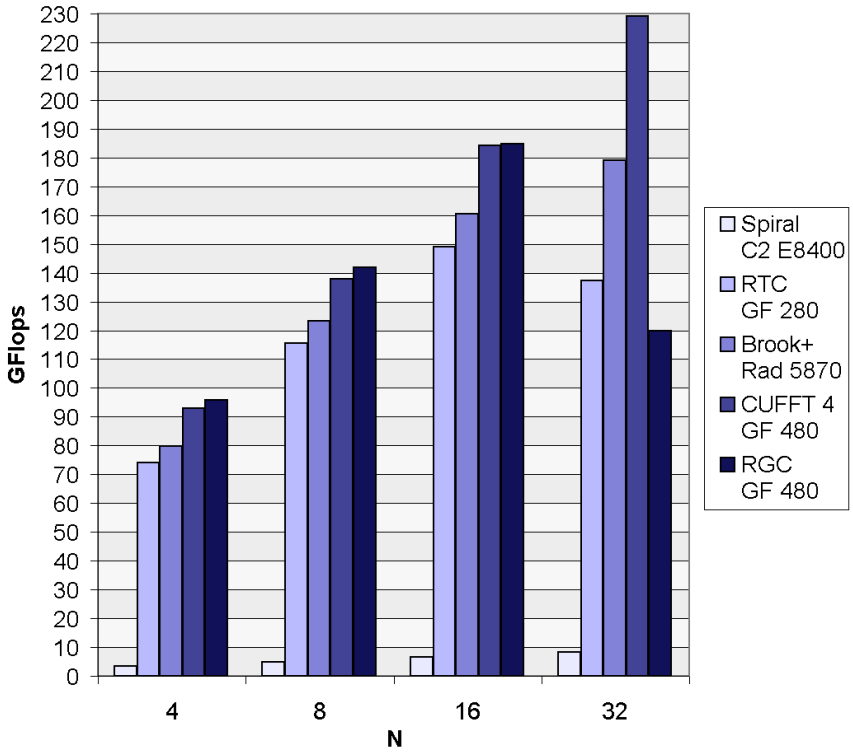


Figure 3.14: Comparison with other solutions

Chapter 4

BPLG: A tuned butterfly processing library for GPU architectures

In this chapter we present a library for butterfly algorithms, Butterfly Processing Library for *GPUs* (BPLG); more specifically, a set of orthogonal signal transforms and an algorithm to solve tridiagonal equation systems are implemented using this library. The methodology used in our library is based on a series of building blocks that enable us to easily design several well-known algorithms with little effort. The library was built paying special attention to flexibility and adaptability. In this chapter we focus on how a generic approach can be used to easily design these *GPU* algorithms while obtaining competitive performance on two recent *NVIDIA GPU* architectures, which results specially interesting from the productivity point of view.

Specifically, we make two main contributions. The first one is the *CUDA* implementation of a compact yet very efficient tuned library using a *CPU*-like methodology based on a set of functions used as building blocks; its careful design allows to greatly reduce the complexity of the code without compromising performance. The second one is the performance characterization of the algorithms on two recent *NVIDIA GPU* architectures, which allows us to obtain a set of parameters in order to achieve the optimal level of parallelism to be exploited on the architecture. The analysis takes advantage of the parametrization capabilities of the proposed library,

which allows to study the most adequate programming practices and task distribution for optimal parallelism. The work presented in this chapter was originally introduced in [58, 59].

4.1. BPLG basic functions

This section describes the basic functions that will be used to build our library. The building blocks of the algorithms are created in several layers, and each function only performs a small part of the work. Thanks to the behavior of templates many optimizations will take place at compile-time, like reducing code complexity or avoiding temporal registers for function calls. Furthermore, more efficient code is generated using the additional information that is provided to the compiler about things like the problem size, the thread configuration or the radix-sequence. In fact, when this information is known at compile-time, private thread data reordering is performed using register renaming and data are directly accessed in the original structures instead of navigating through pointers. Another advantage is that array data can be processed using loops without being a major efficiency concern because static loops will be fully unrolled. In particular this avoids dynamic addressing of register arrays, which in the current *GPU* generations and latest SDK produces local memory spilling (see *CUDA C Best Practices Guide* [99], Section 6.2.3).

Section 4.1.1 and Section 4.1.2 will present the function templates that will become the building blocks of our library for butterfly algorithms. All the described functions were designed to operate in any space of the *GPU* memory hierarchy, but it is recommended to use data in registers for computations because they offer the highest bandwidth. However, when making heavy use of registers it is important to check the compiler statistics, because if too many registers are used the compiler will generate local memory spilling with the consequent performance loss. Due to this space limitation, to handle the processing of a large problem the input data has to be split in smaller chunks among the threads collaborating in each *CUDA* block. Each thread usually operates over a subset of data stored in registers, therefore it should be aware of the relative position in the data input.

In summary, the proposed library is mainly composed by optimized high level

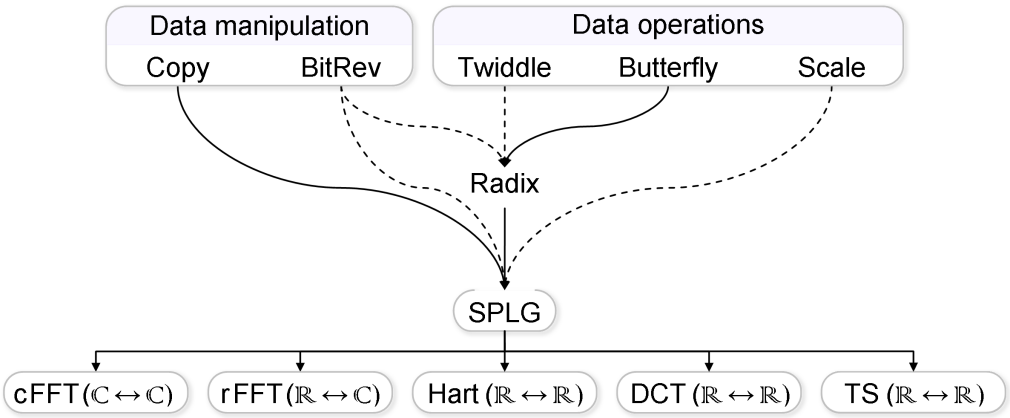


Figure 4.1: Classification and module dependences of the building blocks involved in the library.

blocks that implement the basic functions and one parametrized kernel for each algorithm that properly combines these blocks. The building blocks can be classified in computation (butterfly, twiddles or signal scaling) and reordering operations (like the bit-reversal or the family of strided copy operations). Figure 4.1 presents a scheme with the classification of the principal functions used to build the five algorithms: Complex *FFT*, Real *FFT*, Hartley Transform, Discrete Cosine Transform and Tridiagonal System Solver. The solid line indicates a dependency while the dashed line represents an optional component.

4.1.1. Reordering blocks

Data are passed among the different blocks as pointers, without worrying about the amount of modules cooperating. The blocks are combined in a final kernel with the appropriate code to manage the parallel work distribution, creating the sequences of the different algorithms.

Data is moved inside the *GPU* using the *Copy* function (see Figure 4.2(a)), which reads an array X of N elements with stride ss and writes the result to a different array Y of the same size N but with stride sd: $X[1:N:sd] := Y[1:N:ss]$. Both data buffers can be pointers to arrays in global memory, shared memory or registers; the input buffer can also point to read-only constant memory or texture memory. The source Y and destination X do not have to reside in the same memory

```
1: template<int N> inline __device__ void
2: Copy(TData* Y, int sd,
3:       const TData* X, int ss = 1) {
4:      #pragma unroll
5:      for(int i = 0; i < N; i++)
6:           Y[i * sd] = X[i * ss];
7: }
```

(a) Copy

```
1:  template<int N> inline __device...
2:  BitReverse(TData* X, int ss = 1);
3:
4:  template<> inline __device__ void
5:  BitReverse< 4>(TData* X, int ss) {
6:       Swap(X[ 1 * ss], X[ 2 * ss]);
7:  }
8:
9:  template<> inline __device__ void
10: BitReverse< 8>(TData* X, int ss) {
11:      Swap(X[ 1 * ss], X[ 4 * ss]);
12:      Swap(X[ 3 * ss], X[ 6 * ss]);
13: }
```

(b) BitReverse

Figure 4.2: Template code for the reordering building blocks used by the butterfly transform.

space, however it is recommended to minimize memory transfers, specially on the lower levels of the *GPU* memory hierarchy where bandwidth is a precious resource. The strides *sd* and *ss* are optional parameters and by default consecutive data is assumed. When the strides are known at compile time, the data offsets can also be precomputed. The caller function has the responsibility to use the adequate strides in order to minimize bank conflicts or to perform coalescent global memory access. Thanks to the *C++* function overloading mechanism it is possible to define source stride without explicitly specifying a value for the destination stride. Observe that the amount of elements is known at compile time, thus, the unroll directive in line 4 will instruct the compiler to unroll the loop in line 5 and optimize the stride expression whenever possible. It is very important to prevent runtime addressing of thread-private arrays declared in the kernel as local variables, because the compiler would use slow local memory to store the data instead of registers.

The bit-reversal operation is a kind of binary data permutation efficiently performed by the *BitReverse* function (see Figure 4.2(b)). It reads an array of N elements with stride *sd* and writes the result to the same array using the same stride: $X\,[1 : N : sd] := X\,[1 : N : sd]$. For example, the function prototype (line 1) and the $N = 4$ (lines 4 to 7) and $N = 8$ (lines 9 to 13) specializations are displayed in the figure. The definition of a series of specializations usually results more efficient than the same operation performed by a size-independent generalized algorithm. Furthermore, if *sd* is known at compile-time the function performs static

indexing and the source code will be optimized by the compiler into a simple register renaming.

4.1.2. Computing blocks

In contrast to the reordering blocks, the computing blocks modify their input data. The different algorithms will be defined properly combining the basic reordering and computing blocks. Although some parts like size-dependent specializations or function overloads will be omitted due to space constraints, most of the code will be shown in the corresponding figures. Note that small auxiliary functions may be required for some tasks, for instance, our *DCT* and *Hartley* implementations are derived from the complex *FFT*, therefore a pre-processing or post-processing filtering stage is required [68, 70]. The real *FFT* is also a specialized version of the complex *FFT* algorithm that treats the real input signal as a complex array of half the length, avoiding redundant computations [68]. Last, tridiagonal systems have a similar algorithm structure, but use another data exchange pattern and a radically different radix operator as will be seen in Figure 4.4.

If problem data scaling for inverse transform is desired the *Scale* operator can be applied to the data array (see Figure 4.3(a)). The input data (X) is multiplied by a scalar value, which is inversely proportional to the specified scaling factor F. In our case, the value of F is the complete problem size being processed, which is used in line 3 to calculate at compile-time the corresponding $factor$ value. Then, for each element of the array the scaling factor is applied (see line 6) taking into account the optional stride parameter ss. As the number of iterations is known at compile-time the loop in line 5 will be fully unrolled. Furthermore, if the stride is known at compile time, the data offsets can also be precomputed.

The $Butterfly$ function handles the computations associated to each radix stage, receiving a data vector X and a stride value ss. The function prototype is indicated in Figure 4.3(b) (lines 1 and 2), however the actual computations are performed by the corresponding specializations (in the example line 5 for $N = 2$ and line 12 for $N = 4$). Two sets of specializations are defined depending on the direction of the transform ($D = 1$ for forward transforms like in the figure, and $D = -1$ for inverse transforms). Observe that when $N > 2$ the operation is implemented splitting the

```
1: template<int N, int F> inline __devic...
2: Scale(TData* X, int ss = 1) {
3:     const float factor = 1.0f / (float)F;
4:     #pragma unroll
5:     for(int i = 0; i < N; i++)
6:         X[i * ss] = factor * X[i * ss];
7: }
```

(a) Scale

```
1: template<int N> inline __device__ void
2: Twiddle(TData* X, float ang, int ss = 1) {
3:     #pragma unroll
4:     for(int i = 1; i < N; i++) {
5:         TData factor;
6:         fastCosSin((float)i * ang, factor);
7:         X[i * ss] = X[i * ss] * factor;
8:     }
9: }
```

(b) Twiddle

```
1: template<int N, int D> inline __dev...
2: Butterfly(TData* X, int ss = 1) { }
3:
4: template<> inline __device__ void
5: Butterfly<2, 1>(TData* X, int ss) {
6:     TData t0 = X[0 * ss];
7:     X[ 0 * ss] = t0 + X[1 * ss];
8:     X[ 1 * ss] = t0 - X[1 * ss];
9: }
10:
11: template<> inline __device__ void
12: Butterfly<4, 1>(TData* X, int ss) {
14:     Butterfly<2, 1>(X + 0 * ss, 2 * ss);
15:     Butterfly<2, 1>(X + 1 * ss, 2 * ss);
16:
17:     TData t3 = X[3 * ss];
18:     X[3*ss] = make_TData(t3.y, -t3.x);
19:
20:     Butterfly<2, 1>(X + 0 * ss, 1 * ss);
21:     Butterfly<2, 1>(X + 2 * ss, 1 * ss);
22: }
```

(c) Butterfly

```
1: template<int N, int D> inline __devic...
2: Radix(TData* X, float ang, int ss = 1) {
3:     if(D != 0) Twiddle<N>(X, ang, ss);
4:     Butterfly<N, D>(X, ss);
5:     BitReverse<N>(X, ss);
6: }
```

(d) Radix

Figure 4.3: Template code for computing building blocks used by the signal transform algorithms.

input and calling the same operation again over each part (with a different data pointer X and stride ss) until reaching a two element butterfly. This is not a real recursion, because a different specialization is called. All the stride and pointer arithmetics involved in the $Butterfly$ function template will be resolved at compile-time, without any runtime performance penalty. In the case of tridiagonal systems, as detailed in Section 1.3.1, the butterfly operator is quite different. Figure 4.4 displays two examples of the corresponding code. $Butterfly\_step$ (Figure 4.4(b)) is the general case and $Butterfly\_init$ (Figure 4.4(a)) is a specialization optimized for the first stage of the algorithm, where the three equations of each triad are equal. These functions work over equations, progressively reducing the variables as defined

by the algorithm. As the equations are represented by a single *float4* no information is provided about the relative position of the variables, therefore the *reduce* function receives a parameter to properly align the equation data for the reduction operation.

The twiddle factors are computed and applied by the corresponding $Twiddle$ function template (see Figure 4.3(c)). It just multiplies the elements of the input array X accessed with a stride ss by a complex number. This value is derived from the location of the element being operated (the iteration index i, declared in line 4 and used in lines 6 and 7) and the twiddle angle ang that was specified when calling the function. As in other examples, the size parameter N is used to statically unroll the loop in line 4. In line 5 a complex variable is defined to hold the twiddle factor. The trigonometric computation is performed in line 6 using the iteration number, the value specified by ang and the $fastCosSin$ function. This will call the *CUDA* native $\_\_sincosf$ intrinsic, which normally executes faster than many separated requests on a relatively long precalculated table stored in constant memory (see [99], Section 6.2.5). The angle parameter will always be in the $[0 \dots \pi]$ range and there is no loss of precision.

Finally, the $Radix$ function (see Figure 4.3(d)) is used to perform the computations of each stage. The parameter N determines the radix size, which is used to select the basic $Twiddle$, $Butterfly$ and $BitReverse$ kernels to call inside the template. When the transform direction D is 0 operations are disabled, thus only the bit-reversal (line 5) and other data permutation take place. As in the previous cases, the data is stored in the array X and will be accessed using the stride specified by ss. The optional ang parameter is only used to specify the corresponding twiddle for line 3 in multi-stage *FFTs* after the first stage. A mixed-radix overload is defined for those cases when the input array X contains data from different signals, which have to be processed in batch mode without interactions among them. Figure 4.4 presents two examples of the $Radix$ function used by the tridiagonal algorithm. Tridiagonal systems do not use the bit-reversal operator nor the computation of the twiddle, however they still require the code to handle mixed-radix cases. This task is performed by the $MixR$ specialization (Figure 4.4(c)), which will always perform the first stage of the algorithm, thus instead of calling the general $Butterfly\_step$ function (like in Figure 4.4(d)) it will use $Butterfly\_init$.

```
1: template<> inline __device__ void
2: Butterfly_init<2>(float4* eL,
3: float4* eC, float4* eR, int s) {
4:       float4 L2 = reduce(eC[0], eC[s],  0);
5:       float4 R2 = reduce(eC[s], eC[0],10);
6:
7:       eL[0] = L1; eC[0] = L1; eR[0] = R2;
8:       eL[s] = L1; eC[s] = R2; eR[s] = R2;
9: }
```

(a) Butterfly init

```
1:  template<> inline __device__ void
2:  Butterfly_step<2>(float4* eL,
3:  float4* eC, float4* eR, int s) {
4:        float4 L2t = reduce(eR[0], eL[s], 2);
5:        float4 L1 =  reduce(eL[0],   L2t , 1);
6:        float4 C1 =  reduce(eC[0],   L2t , 1);
7:        ...
8:        eL[0] = L1; eC[0] = C1; eR[0] = R2;
9:        eL[s] = L1; eC[s] = C2; eR[s] = R2;
10: }
```

(b) Butterfly step

```
1: template<int R, int MixR> inline __d...
2: Radix(float4* eqL, flt4* eqC, flt4* eqR) {
3:       for(int i = 0; i < R; i += MixR)
4:             Butterfly_init<MixR>(
5:                   eqL + i, eqC + i, eqR + i);
6: }
```

(c) Radix MixR

```
1: template<int R> inline __device__ void
2: Radix(float4* eqL, flt4* eqC, flt4* eqR) {
3:       Butterfly_step<R>(eqL, eqC, eqR);
4: }
```

(d) Radix R

Figure 4.4: Specialized template code for the tridiagonal solver algorithm.

4.2. Algorithm design based on BPLG

Thanks to the functions that were introduced in Section 4.1 it is possible to generate the code of the different algorithms in an easy and compact way. Each algorithm is implemented by a single main kernel, which is a template with four parameters: the problem size N, the transform direction D used in the orthogonal transforms, the radix size R and the amount of shared memory S. These parameters are known at compile-time, therefore conditional execution and many function calls involving offsets and strides can be easily optimized by the compiler as stated before. When $S > N$ each *CUDA* block will be assigned to process S/N independent problems in batch mode, therefore increasing performance for large quantities of small problems. In the general structure of the algorithm, each task will be assigned a small part of the computation using a divide and conquer approach. The portion of the problem and the assigned batch depends on the thread and block identifiers. After performing the required computations, the threads collaborating in the same problem will exchange data before the next computing stage. Data exchanges among threads will be performed in shared memory, which is much faster and more energy efficient than global memory, however it is relatively small, thus restricting the maximum problem. Following we will describe two of the library

kernels in order to explain the general form of the algorithms with more detail.

4.2.1. Signal processing transforms

As already explained, the real *FFT* (see Section 1.2.2), the *DCT* (see Section 1.2.4) and the *Hartley* transform (see Section 1.2.3) will be derived from the complex *FFT* algorithm using a pre-processing and/or post-processing stage [68,70]. In Figure 4.5 the kernel code for the *DCT* algorithm is presented. This example will also allow us to display the usage of the functions that were introduced in Section 4.1. The kernel has four template parameters which are supplied at compile-time (see N, D, R and S in line 1). The signal transforms are performed in-place and the kernel only requires two runtime parameters (see line 2): a data pointer *src* (used as both input and output) and *size* (which is the actual signal size and can be used to add padding in order to process non power of two problems). Regarding the structure of the algorithm, it can be divided into six main sections:

1) Initialization section (represented by lines 3 and 4), where thread and group identifiers are used to obtain the global memory offsets. In this section the registers and shared memory resources are also statically allocated based on the kernel template parameters. Although they are reserved as two simple arrays (*shm* and *reg*), the code completely avoids dynamic indexing of registers because the compiler would move the structure to local memory.

2) Pre-processing stage (lines 6 to 11), which loads data from global memory and may also perform some operations over it depending on the particular algorithm and the direction parameter. For instance, in the case of the forward *DCT* data has to be reordered before the execution of the radix stages (*packDCT* in line 10), while in the case of the *Complex-FFT* data is loaded directly to registers with no pre-processing.

3) First radix stage of the algorithm (lines 13 to 15). When $N \ mod \ R \neq 0$ a mixed-radix algorithm [26] is performed. There is an optional template argument $MixR$ (computed at compile-time) which just orders the *radix* function to perform several independent radix operators of smaller size with data in the registers. This section also includes the signal scaling operation for the inverse transform (*scale* in line 14).

```
1:  template<int N, int D, int R, int S> __global__ void
2:  SPLG_DCT(float* src, int size) {
3:      // Define some thread/group identifiers and their memory offsets
4:      // Register/ShMem static allocation based on template parameters
5:
6:      // DCT pre-processing, loads data from GlbMem into registers
7:      if(D >= 0) {
8:          copy<R>(shm + threadXY, S / R,  src + dctPos, S / R);
9:          __syncthreads();
10:         packDCT<N, R, D, S>(reg, shm, threadId, batchId);
11:     } else ...
12:
13:     // The first mixed-radix stage is always computed
14:     if(D < 0) scale<R, N/2>(reg); // Inverse transform scaling
15:     radix<R, MixR, D>(reg, R / MixR);
16:
17:     // This loop computes the remaining radix stages
18:     for(int accRad = MixR; accRad < N; accRad *= R) {
19:
20:         // Obtains strides and offsets from the iteration and thread id
21:         int readOffset  = ... , readStride = ... ;
22:         int writeOffset = ... , writeStride = ... ;
23:
24:         // Reordering stage in shared memory, Reg->Shm->Reg
25:         __syncthreads();
26:         copy<R>(shm + writeOffset, writeStride, reg);
27:         __syncthreads();
28:         copy<R>(reg, shm + readOffset, readStride);
29:
30:         // Computation with data in registers
31:         float ang = getAngle<D, R>(accRad, threadId >> cont);
32:         radix<R, D>(reg, ang);
33:     }
34:
35:     // DCT post-processing, stores the result from registers to ShMem
36:     __syncthreads();
37:     if(D >= 0) {
38:         radixDCT<N, R, D, S>(reg, threadId);
39:         copy<R>(shm + shmPos, N / R, reg);
40:     } else ...
41:
42:     // Stores the final result from ShMem to GlbMem
43:     __syncthreads();
44:     copy<R>(src + dctPos, S / R, shm + threadXY, S / R);
45: }
```

Figure 4.5: Kernel code for the DCT algorithm

4) Remaining radix stages (lines 17 to 33). They are computed using a loop and each iteration is composed by a reordering stage (lines 24 to 28), which uses shared memory to exchange information among the threads, and a computing stage (lines

30 to 32). The data exchange is easily performed by the *copy* function (lines 26 and 28) when called with different offsets and strides (obtained in lines 20 to 22). The computing stage is performed by the *radix* function (line 32), which only requires the angle for the current iteration and thread (obtained in 31).

5) Post-processing stage (lines 35 to 40), that depending on the algorithm may perform some computations or reorder the previous results before writing to global memory.

6) Results are written to global memory (lines 42 to 44). In the case of the *DCT* signal data will reside in shared memory, while in the case of the *Complex-FFT* no post-processing is required, therefore data can be directly written from registers after the final radix stage (either line 15 or line 32, depending on N).

4.2.2. Tridiagonal system algorithm

Figure 4.6 presents the code structure for tridiagonal systems, which shares many similarities with Figure 4.5. The algorithm will solve tridiagonal systems of N equations using radix R, and once again when $S > N$ each *CUDA* block will be assigned to process S/N independent problems in batch mode. Observe that in this case problem data is stored in a sparse format (see lines 2 and 3 with the function prototype), using four separate arrays like in *NVIDIA's CUSPARSE* library (*gtsvStridedBatch* function). There are three read-only buffers for the diagonals (*srcL* for lower, *srcC* for main and *srcR* for upper) plus another read/write buffer (*dstX*) for the right-hand-side term, that will be also used to store the solution at the end of the kernel. The read only buffers are declared as *const __restrict* pointers, which in the *Kepler* architecture enables to optimize the memory access using texture cache. In contrast to signal processing algorithms there are no pre-processing or post-processing stages and the code can be divided into five main sections:

1) Initialization section (lines 5 to 8). Thread and group identifiers are used to obtain global memory offsets. Register data (line 7) and shared memory space (line 8) are allocated for the equations. The equations are represented by the customized *Float4* data type, and each row requires three equations (*regL*, *regC* and *regR*)

```
1:  template<int N, int D, int R, int S> __global__ void
2:  SPLG_TRI(const float* __restrict srcL, const float* __restrict srcC,
3:               const float* __restrict srcR, float* dstX, int stride)
4:  {
5:      // Define some thread/group identifiers and their memory offsets
6:      // Register/ShMem static allocation based on template parameters
7:      Float4 regL[R], regC[R], regR[R];
8:      __shared__ Float4 shm[N > R ? S : 1];
9:
10:     // Load data from GlbMem into registers
11:     switch(R) {
12:         case 2:       // Loads data using float2
13:             regC[0] = ... ; regC[1] = ... ; break;
14:         case 4:       // Loads data using float4
15:             regC[0]=...; regC[1]=...; regC[2]=...; regC[3]=...; break;
16:         default: ...  // Otherwise loads data for radix > 4
17:     }
18:
19:     // The first mixed-radix stage is always computed
20:     radix<R, MixR>(regL + i, regC + i, regR + i);
21:
22:     // The loop computes the remaining radix stages
23:     for(int accRad = MixR; accRad < N; accRad *= R) {
24:
25:         // Obtaines strides and offsets from the iteration and thread id
26:         int readOffset = ... , readStride = ... , readLo = ... , readHi = ... ;
27:         int writeOffset = ... , writeStride = ... ;
28:
29:         // Reordering stage in shared memory, Reg->Shm->Reg
30:         if(accRad > MixR) __syncthreads();
31:         copy<R>(shm + writeOffset, writeStride, regC);
32:         __syncthreads();
33:         copy<R>(regC, shm + readOffset, readStride);
34:         copy<R>(regL, shm + readLo, readStride);
35:         copy<R>(regR, shm + readHi, readStride);
36:
37:         // Computation with data in registers
38:         radix<R>(regL, regC, regR);
39:     }
40:
41:     // Stores the final result from registers to GlbMem
42:     #pragma unroll
43:     for(int i = 0; i < R; i++)
44:         dstX[glbOffset + glbStride * i] = regC[i].w / regC[i].y;
45: }
```

Figure 4.6: Kernel code for the tridiagonal algorithm

to store the triads in the registers. However, to minimize shared memory usage, a single equation array *shm* will be allocated.

2) Load data from global memory (lines 10 to 17). The algorithm can easily perform coalescent read operations to load data at the beginning of the algorithm without any shared memory reordering stage. Instead of accessing a single data element per memory request, we will use 64 bit loads for radix-2 (lines 12 and 13) to fetch 2 consecutive elements from each array, or 128 bit loads for radix-4 (lines 14 and 15), which provides 4 consecutive elements. These elements can be directly used as the input arguments of the first radix stage. Remember that initially the three equation of the triad will be equal, therefore only $regC$ needs to be initialized.

3) First radix stage of the algorithm (lines 19 to 20). In a similar fashion to the signal processing algorithms, in order to reduce the number of processing stages (and consequently the amount of intra-block synchronizations) it is possible to define a radix-R algorithm. The first radix stage is separated from the rest because when $N \ mod \ R \neq 0$ an initial mixed-radix $MixR$ stage is performed. The higher radix operators can be defined either recursively or using a specialization for the desired size, which helps reducing the amount of private registers. For instance, line 20 uses an optimized version: To keep track of the state of the algorithm, each row needs to store 3 equations (left, center and right). However at the beginning of the process, the three equations will be identical, therefore the computations can be simplified.

4) Remaining radix stages (lines 22 to 39). Each iteration of the loop (line 23) reorganizes data (lines 29 to 35) and then performs a radix stage (line 38). In lines 26 and 27 the offset and strides for the data exchange are efficiently computed using bit masks, binary operators and displacements. Following, the data exchange is performed using shared memory.

An interesting optimization over the original algorithm proposed in [135] is to use only the center equation when performing the exchange. Each thread still requires to read $3 \times R$ equations (lines 33 to 35) to perform the radix-R stage, however it only needs to write R equations (line 31). This is based on a property of the algorithm, which relies on the fact that the left and right equations are equal to two of the center equations (see Section 1.3.1 for more details). This way, the shared memory bandwidth required by the algorithm can be reduced. Shared memory multiplexing techniques [140] can be used to further reduce the size of the buffer that exchanges the equations. Notice that nonetheless computations are performed in registers, and a radix-4 algorithm requires at least $4 \ rows \ \times 3 \ equations \ \times \ float4 = 48$ registers

just to store problem data, not including any temporal storage or intermediate variables generated by the compiler. Therefore, a radix-8 version would need 96 registers and consequently not suited for the current *GPU* architectures.

5) Results are written to global memory (lines 41 to 44). Due to the topology of the Cooley-Tukey algorithm when using decimation in time, coalescence will be good for equation systems where $N \geq 32 \times R$. In practice, even for smaller problems coalescence will not be an issue thanks to the cache hierarchy of the *GPU*.

4.3. Obtaining optimal parallelism

One of the main requirements for *GPU* performance is to explicitly expose sufficient parallelism. The parallelism is controlled through the *CUDA* grid configuration (block parallelism) and block size (thread parallelism). In order to tune the kernel and configure the execution for optimal parallelism it is recommended to determine the main performance limiting factor.

The most relevant factors are the amount of registers assigned per thread, the shared memory per block, the desired number of concurrent blocks per *SM* (up to 16 in *Kepler* and up to 8 in *Fermi*), and last, the block size (up to 1024 threads per block in both architectures). Blocks are not started until enough resources are available, and once started they lock their resources until completion. The programmer can tweak the balance between the number of simultaneous blocks and the amount of threads per block in order to offer the hardware enough independent instructions to accommodate multi-issue scheduling and enough tasks to take advantage of latency hiding techniques. Note that large block sizes may result in more expensive synchronizations, furthermore a high number of very light tasks may introduce some overhead. In fact, one of the most time consuming tasks in order to achieve high performance on the *GPU* is profiling and tuning the code to find the right balance among the resources.

4.3.1. Streaming Multiprocessor (SM) parallelism

Regarding the *SM* parallelism limiting factor, suppose that B_{SM} is the amount of blocks that can be processed simultaneously by each *SM*. It is given by the expression:

$$B_{SM} = Min(B^r_{SM}, B^s_{SM}, B^l_{SM}, B^{max}_{SM}) \tag{4.1}$$

Where the first term B^r_{SM} is the amount of blocks limited by the number of registers available in the *SM* and how many are allocated for each block, and is computed as:

$$B^r_{SM} = R^{max}_{SM}/R_B, \text{ with } R_B = R_t \times L. \tag{4.2}$$

with R^{max}_{SM} the total number of registers per *SM* (65536 in *Kepler* and 32768 in *Fermi* architectures) and R_B are the amount of registers used by each block, which is computed as the amount of registers per task R_t (obtained with the *–ptxas-options=-v nvcc* compiler flag) multiplied by the number of tasks per block L (can be adjusted independently for each problem size).

The second term of Expression 4.1 (B^s_{SM}) is the amount of blocks limited by shared memory, computed as:

$$B^s_{SM} = S^{max}_{SM}/S_B, \text{ with } S_B = sizeof(D_t) \times R \times L \tag{4.3}$$

where S^{max}_{SM} is the size of the shared memory (49152 bytes for both architectures) and S_B is the amount of shared memory reserved for each block, which is computed as the size of the data type (4 bytes for *float* data, 8 bytes for *complex* data and 16 bytes for *float4* used in tridiagonal equations) multiplied by the product of the amount of registers R used to store signal data in each thread (depends on the desired radix size) and the amount of tasks L created in each *CUDA* block.

The third term of Expression 4.1 (B^l_{SM}) is the maximum amount of blocks, limited by the number of warps in-flight of the architecture:

$$B^l_{SM} = 32 \times L^{max}_{SM}/L \tag{4.4}$$

where L_{SM}^{max} is the *SM* warp limit of the architecture (64 for *Kepler* and 48 for *Fermi*).

Finally, B_{SM}^{max} is the last term of Expression 4.1 and represents the *SM* block limit of the hardware, which is 16 for *Kepler* and 8 for *Fermi* architectures.

4.3.2. Batch execution in order to increase parallelism

In our algorithms each thread performs the computations associated to a R butterfly in each stage, with data being stored in private registers. Thus, without considering temporal data storage used by the compiler for any operations, our algorithm would require at least $sizeof(D_t) \times R$ bytes, where D_t is the data type used by the algorithm. Each problem is processed by $L_1 = N/R$ tasks and, in order to increase the thread parallelism, when L_1 is low the amount of tasks per block can be increased using batch execution L_2. Therefore, each block will be composed by $L = L_1 \times L_2$ tasks. Although computations are entirely performed in registers, data interchanges within each block rely on shared memory, which should be large enough to fit the data stored in registers during the exchanges.

Depending on the data type, the problem size and the desired batch execution, the required shared memory will be $S_B = sizeof(D_t) \times N \times L_2$ bytes. The maximum size for our algorithm using real signal data is $N = 8192$, for complex signals is $N = 4096$, and for tridiagonal systems is $N = 2048$. These limits are given by the maximum shared memory that can be allocated for a single block. Bigger problems would require a different approach, like a staggered data exchange or a multi-pass algorithm, which will be studied in a future work.

4.3.3. Simultaneous block processing optimization

Our main objective in the optimization of the algorithm is to maximize B_{SM}, adjusting R and $L2$ to tune the algorithm for each problem size on each architecture.

To illustrate our proposal, suppose that we are trying to optimize the processing of the complex FFT with $N = 128$ and $R = 4$ for the *Kepler* architecture. According to the compiler this kernel requires 28 registers, therefore $B_{SM}^{r} = 65536/(28 \times L)$.

Table 4.1: Parameter configuration for the complex FFT algorithm

N	n	R	R_t	L_1	L_2	S_B	B_{SM}
4	2	2	14	2	64	2048	16
8	3	2	18	4	32	2048	16
16	4	2	18	8	16	2048	16
32	5	2	18	16	8	2048	16
64	3	4	28	16	8	4096	12
128	3.5	4	28	32	4	4096	12
256	4	4	27	64	2	4096	12
512	4.5	4	28	128	1	4096	12
1024	5	4	27	256	1	8192	6
2048	$3.\overline{6}$	8	37	256	1	16384	3
4096	4	8	36	512	1	32768	1

On the other hand, $B^s_{SM} = 49152/(sizeof(D_t) \times 4 \times L)$ and $B^l_{SM} = 32 \times 64/L$. In order to maximize these expressions they can be rewritten assuming that $B^x_{SM} = 16$, which is the maximum allowed value given by B^{max}_{SM}. Consequently: $B^r_{SM} \rightarrow L = 65536/(28 \times 16) = 146.3$, $B^s_{SM} \rightarrow L = 49152/(8 \times 4 \times 16) = 96$, and $B^l_{SM} \rightarrow L = 32 \times 64/16 = 128$. Therefore, the recommended values for L are 96 or, rounding to the next power of two, 128 (in practice both configurations offer nearly identical results).

Regarding the amount of threads per block, $L = L1 \times L2$ and $L_1 = 128/4 = 32$, which is exactly one warp, therefore in this case $L2 = 128/32 = 4$, thus 4 warps will be working in batch mode processing different signals within each block and 12 blocks will be simultaneously processed by each *SM*. In the example, using $R = 4$ the algorithm will perform $n = log_4(128) = 3.5$ stages, which means one radix-2 stage followed by three radix-4 stages (the optimal R will be analyzed in Section 4.4).

For more information, the final configuration table for the complex FFT is presented in Table 4.1. Observe that for $N > 512$ B_{SM} keeps decreasing as more shared memory S_B is reserved for each block, however the performance impact is mitigated by the increase in thread parallelism L_1.

4.4. Experimental results

Table 4.2 describes the test platforms. All the tests were evaluated in single precision using problem sizes in the range $N = \{4, \ldots, 4096\}$, with data already on the *GPU* memory at the beginning of each test. Batch execution is used to process $2^{24}/N$ different problems, therefore as the input signal increases the number of batch executions decreases. In the case of the *FFT* the performance will be expressed in *GFlops* using Equation (1.4) for the complex *FFT* and Equation (1.5) for the real *FFT*. As far as we know there is no standardized expression for the *DCT* and the *Hartley* transform, therefore to offer a similar measurement we will use the same formula as the real *FFT*.

The *Titan GPU* is based on the *Kepler* architecture and allows the programmer to select the desired bank size configuration for the shared memory, which can be either 4 bytes (by default, offers finer grained access) or 8 bytes (improved bandwidth). After testing the influence of this parameter we saw little difference in our algorithms (around 0.1% better for 4 bytes), therefore the default configuration will be used. Regarding the configurable *L1* cache and *shared memory* size of the *GPUs*, the default configuration with 48 *KB* of shared memory and only 16 *KB* of *L1* cache was used. Our implementation makes extensive usage of shared memory for data exchange among the tasks, however it does not benefit from the increased $L1$ configurations. Therefore, the smaller shared memory configuration would result in lower performance due to the reduced amount of *SM* block parallelism.

4.4.1. Orthogonal signal transforms performance

In this section we will analyze the global performance of the complex *FFT*, real *FFT*, *DCT* and *Hartley* transforms in the two *GPU* architectures. Obviously the more powerful *Kepler GPU* will offer better performance, but it is also interesting to check the scalability of the algorithm. For comparison purposes, *NVIDIA's CUFFT 5.0* results will be included in the figures. As the *CUFFT* does not support the *Hartley* and the *DCT* transforms, they were implemented using a similar strategy to *BPLG*. However, as we do not have access to the source code, this requires to launch at least two separate kernels, thus limiting the maximum performance. As far as we know there are no other general *GPU* implementations for these algorithms. The *DCT* included in the *CUDA SDK* is an specialized version for small 2D blocks

Table 4.2: Description of the test platforms

	Platform 1	Platform 2
CPU	Core i7 2600	Core 2 Duo E8400
Memory	8 GB DDR3 1333	2 GB DDR3 1333
OS	Win7 x64 SP1	WinXP x64 SP2
Compiler	MSVC 2010 SP1	MSVC 2010 SP1
GPU	GeForce 580	GeForce Titan
GPU	v320.17, SDK 5.0	v320.17, SDK 5.0

with a fixed size of 8×8, and other publications like [84] do not offer comparable benchmarks. The *FFTW 3.3.3* library [78] will be also included as a reference point to represent the performance of a *SSE* (Streaming SIMD Extensions [50]) optimized implementation running on a recent multi-core *CPU* (only Platform 1 results are used for the *FFTW*).

4.4.1.1. Balancing warp and block parallelism

As mentioned in Section 4.3 it is very important to configure the kernel for optimal *SM* usage. In fact, finding the right balance between warp parallelism and block parallelism usually results a time-consuming task for programmers. To facilitate the study of this factor Table 4.3 presents the complex *FFT* performance on *Platform 1* (*GeForce 580*) for three different radix configurations depending on the signal size N and the amount of tasks per block L. Unavailable configurations are shaded in gray, while the best cases are marked in bold. The first group shows the results for the radix-2 algorithm. Observe that with some exceptions $L = 128$ tends to offer the best performance. The next group repeats the analysis for radix-4. Excluding the first three cases, the best configuration is again $L = 128$. Finally, the third group displays the results for radix-8. In this case the best performance is usually obtained for $L = 64$, however the results do not outperform the radix-4 version. The only exceptions are the two last cases, $N = 2048$ which requires 256 threads ($L = 256$) and $N = 4096$ which requires $L = 512$. Comparing the results for the different radix values it can be observed that for $N = \{4 \dots 64\}$ the best performance is obtained with the radix-2 version of the algorithm. In general, due to the small step size, radix-2 algorithms are only suitable for these smaller problems or to perform the mixed-radix stage. For $N = \{128 \dots 1024\}$ the situation changes and radix-4 performs better. Finally, for $N = \{2048, 4096\}$ the radix-8 version

Table 4.3: Impact of the task number for the FFT using Radix-2, Radix-4 and Radix-8 (Platform 1)

	Radix-2			Radix-4				Radix-8			
N	L64	L128	L256	L64	L128	L256	L512	L64	L128	L256	L512
4	103.5	***105.9***	105.5	***76.1***	75.7	75.5	71.0				
8	149.1	***156.3***	155.5	***117.2***	116.5	116.4	104.5	***77.4***	77.3	76.7	72.7
16	159.9	***192.6***	***192.6***	***156.4***	154.1	152.5	141.2	***81.6***	81.4	80.3	65.7
32	188.7	***251.1***	249.4	211.4	***213.0***	212.4	205.6	***157.3***	156.8	153.2	117.3
64	266.1	315.1	***316.4***	312.5	***315.9***	315.8	312.9	***261.6***	256.7	251.0	182.5
128	239.5	***317.5***	301.4	366.4	***368.2***	368.0	358.7	***316.7***	305.4	288.0	172.8
256		***323.9***	308.3	418.7	***421.1***	420.8	407.9	***407.1***	399.0	372.1	216.2
512			***317.5***		***473.6***	468.5	416.8	***453.0***	442.2	404.1	239.6
1024						***521.8***	461.2		***484.9***	441.8	254.1
2048							***455.0***			***486.0***	278.0
4096											***300.9***

comes ahead. The radix-16 configuration requires too many registers per thread, resulting in local memory spilling.

Table 4.4 repeats the same analysis for *Platform 2* (*GeForce Titan*). In the case of radix-2, once again $L = 128$ tends to offer the best performance. In the case of radix-4 the results are a bit different, and excluding those few cases where radix-2 performs better, the best configuration is now $L = 256$. Finally, for radix-8 the best performance is usually obtained for $L = 64$, but like in the previous platform, the results do not outperform the radix-4 version (except the two last cases). The optimal ranges for each radix are similar, for $N = \{4 \dots 32\}$ the best performance is obtained with the radix-2, for $N = \{64 \dots 1024\}$ radix-4 performs better and for $N = \{2048 \dots 4096\}$ the radix-8 version is faster. According to our tests, going to a radix-16 configuration would not increase the performance.

4.4.1.2. Complex FFT performance

Figure 4.7 shows the performance of the complex *FFT* on both platforms. As it can be observed, on Platform 2 our generic approach (*BPLG-cFFT*) offers very similar performance to the *CUFFT* for problem sizes up to $N = 1024$ with 701.9 *GFlops*, while the *CUFFT* only offers 680.2 *GFlops*. For bigger problems the shared memory becomes the main limiting factor of our algorithm. Surprisingly, although the reduced complexity of the proposed algorithm, the average advantage of the

Table 4.4: Impact of the task number for the FFT using Radix-2, Radix-4 and Radix-8 (Platform 2)

	Radix-2			Radix-4				Radix-8			
N	L64	L128	L256	L128	L256	L512	L1024	L64	L128	L256	L512
4	143.6	***143.7***	142.1	111.3	***111.8***	106.1	80.6				
8	211.6	***220.3***	220.2	171.3	***172.9***	155.4	117.0	***116.1***	115.9	114.9	108.0
16	222.8	266.6	***271.0***	***227.4***	224.4	207.4	170.3	***123.5***	123.1	121.0	98.1
32	273.9	***344.7***	343.0	302.5	***308.4***	299.4	203.2	***236.9***	235.7	228.5	175.9
64	369.2	***434.0***	432.4	438.7	***439.4***	432.9	265.4	***358.9***	353.1	339.0	266.5
128	316.5	***371.6***	370.4	511.9	***513.2***	495.1	283.8	***436.5***	418.7	395.8	248.1
256		***391.3***	391.1	584.2	***585.9***	562.0	325.7	***574.3***	560.1	519.3	309.9
512			***383.0***	***646.2***	628.5	576.1	324.9	***618.9***	596.0	562.8	343.5
1024					***701.9***	628.4	364.7		***663.6***	619.6	368.2
2048						***629.5***	360.5			***667.3***	401.3
4096							***396.0***				***436.1***

CUFFT is only 7.3%. Our *BPLG-cFFT* algorithm is able to adapt quite well to the *Fermi* architecture of Platform 1, however in this case the *CUFFT* is very optimized and has more advantage (around 14.2% on average). Nonetheless, in some cases our algorithm is able to overtake the *CUFFT*, for instance, for $N = 128$ we achieve 322.9 *GFlops*, while *NVIDIA's* implementation only offers 313.2 *GFlops*. The *FFTW* library is one of the most efficient *CPU* libraries, nonetheless, it only achieves around 50 *GFlops* for the best cases.

4.4.1.3. Real FFT performance

Next, Figure 4.8 analyzes the performance of the real *FFT*. Although the amount of transforms per second is higher than the complex *FFT*, each transform performs fewer arithmetic operations, thus penalizing the *GFlop* estimation. The optimal radix is similar to the complex *FFT*, but as half the data is required it is displaced one location. For instance, while for $N = 2048$ *BPLG-cFFT* performs better with radix-8, *BPLG-rFFT* would obtain the optimal behavior using radix-4. The performance scaling is not so proportional to the signal size, but is quite good, specially compared to the *CUFFT*. Excluding the outlier case for $N = 4$, the average improvement over the *CUFFT* on Platform 1 is on average 42.2%, while on Platform 2 reaches a remarkable 60.4%. Observe that our *BPLG-rFFT* algorithm executed on Platform 2 has very similar performance to the *CUFFT* executed on Platform 1, which is quite

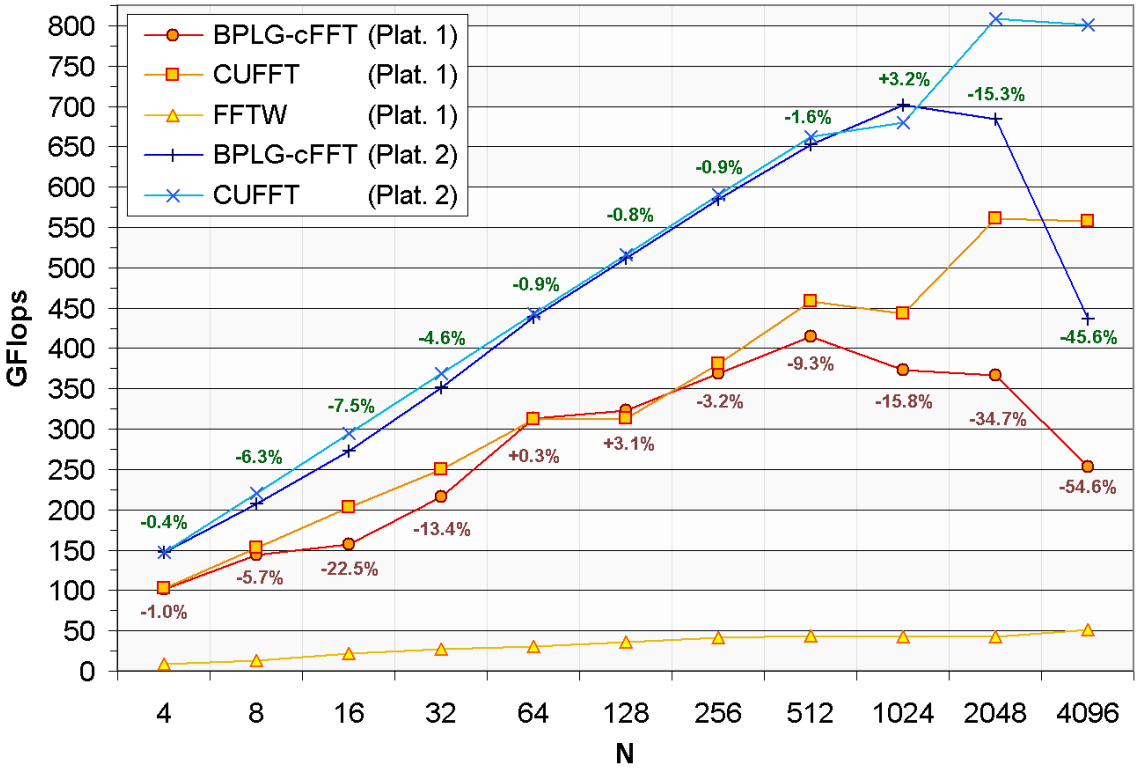


Figure 4.7: Algorithm performance for the complex FFT algorithm

more powerful. The reason behind this behavior is that according to *NVIDIA's* profiler the *CUFFT* is launching two separate kernels for each transform, therefore requiring twice the global memory bandwidth for the same signal size. The real *FFTW* offers around half the performance of the complex transform, at around 25 *GFlops* when $N > 256$.

4.4.1.4. Discrete Cosine Transform performance

Figure 4.9 displays the test results for *DCT* algorithm on both platforms. The *CUFFT* results are also displayed, but recall that *NVIDIA's* library does not directly support this transform, therefore it is computed with the aid of a second filtering kernel, doubling the global memory bandwidth requirements. As expected *BPLG-DCT* outperforms the *CUFFT* version, moreover, in many cases it is able to offer more than twice the computation rate. On average our library is around 87.1% faster on Platform 1 and 149.0% faster on Platform 2. Both *GPU* libraries clearly

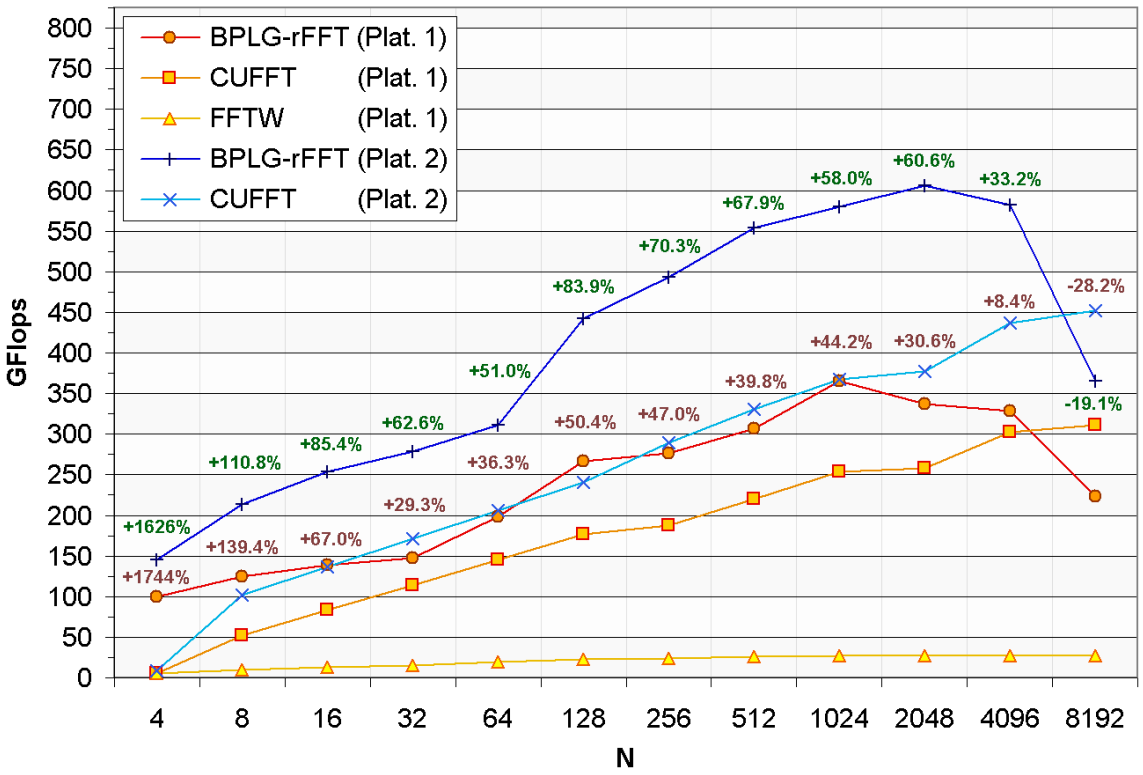


Figure 4.8: Algorithm performance for the real FFT algorithm

outperform the *FFTW*, which is always below 16 *GFlops*.

According to the profiler analysis, the pre-processing and post-processing stages introduce some overhead, and the shared memory access pattern generates more replays than the real *FFT*, which explains that lower performance. For instance, for $N = 1024$ the *BPLG-DCT* algorithm reaches 142.9 *GFlops* on Platform 2 and 346.3 *GFlops* on Platform 1, while in the real *FFT* transform *BPLG-rFFT* was able to obtain 365.9 *GFlops* and 580.2 *GFlops*, respectively.

4.4.1.5. Hartley Transform performance

Figure 4.10 presents the execution results of our library for the *Hartley* algorithm using both platforms. Once again the *CUFFT* version is computed using the *FFT* and filtering stage kernel, offering about half the performance. More specifically, on Platform 1 our library is on average 110% faster than the *CUFFT*, while on Platform

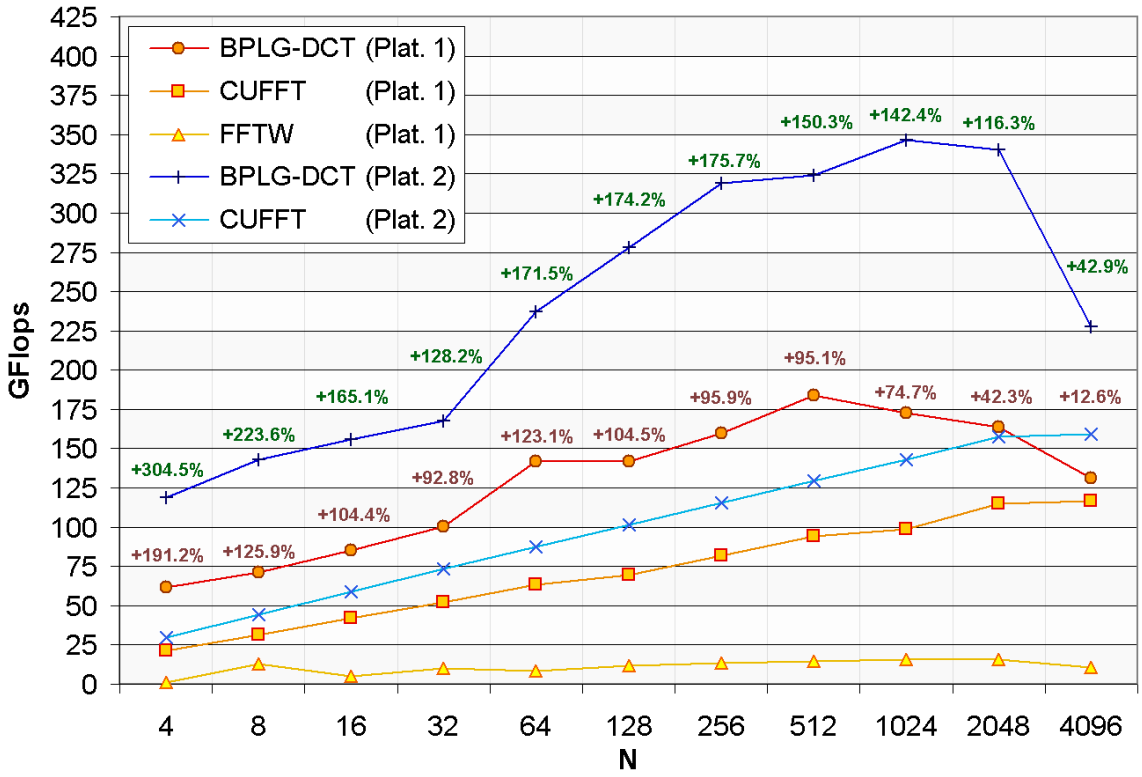


Figure 4.9: Algorithm performance for the Discrete Cosine Transform algorithm

2 our mean advantage is 118% (in both cases excluding the outlier observed for $N = 4$). Notice that although the overall graphic outline is very similar to the *DCT*, both *GPU* libraries obtain better results in the *Hartley* transform. For instance, for $N = 1024$ *BPLG-Hart* obtains 216.2 *GFlops* on Platform 1 and 408.1 *GFlops* on Platform 2. The reason behind this is the shared memory access pattern of the filtering stage, which is simpler in the *Hartley* transform. Observe that the *FFTW* is also a bit faster than in the case of the *DCT*, although always below 20 *GFlops*.

4.4.2. Tridiagonal Equations System performance analysis

The performance of the tridiagonal solvers will be measured in million rows per second as described in Expression 1.25. The batch size was defined as $batch = 2^{22}/N$, therefore all the tests will process the same number of rows. Two reference points are provided for comparison purposes: one is *NVIDIA's CUSPARSE* library (v5.0),

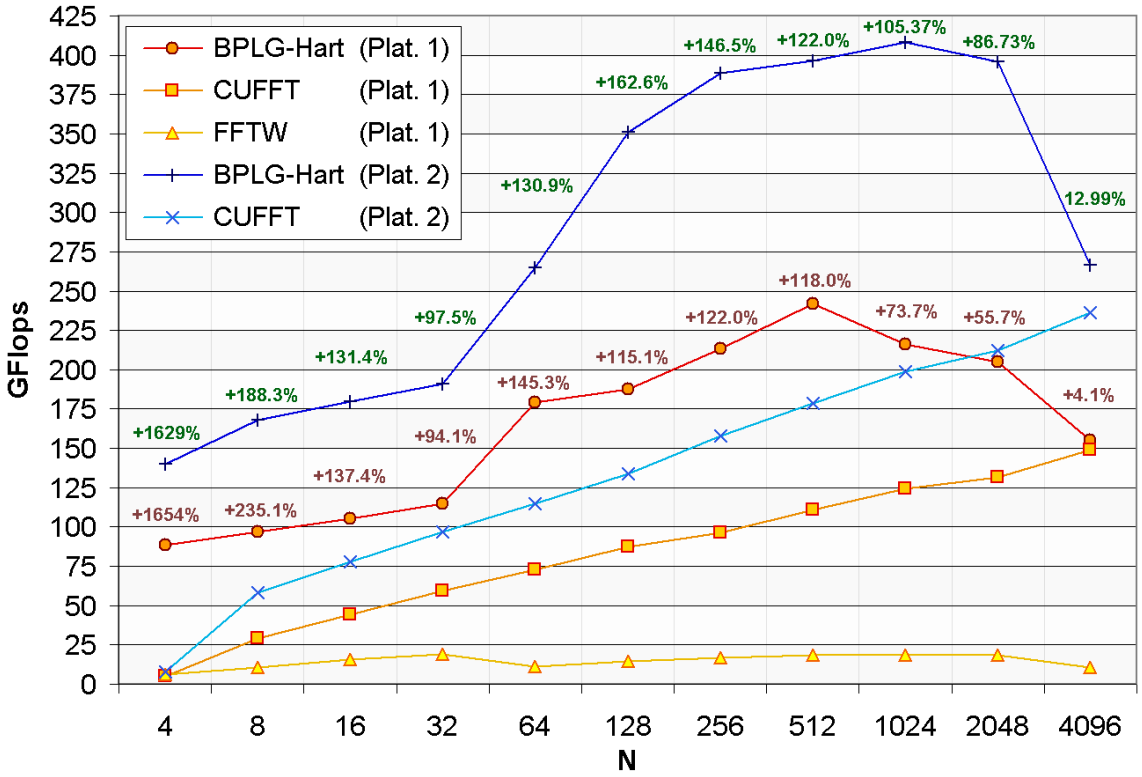


Figure 4.10: Algorithm performance for the Hartley Transform algorithm

and the other is the algorithm presented in [142], which is available as part of the *CUDPP* library (v2.1).

4.4.2.1. Balancing warp and block parallelism

Once again it is important to find the optimal balance between warp-level parallelism and block-level parallelism to maximize *SM* utilization. Table 4.5 shows the tridiagonal system resolution performance of *BPLG-TS* on *Platform 2* depending on the radix size R, the problem size N and the amount of tasks per block L. Unavailable configurations are shaded in gray, while the best cases are marked in bold. Observe that $L = 128$ tends to offer better performance for radix-2, only the two first system sizes ($N = 4$ and $N = 8$) perform faster with $L = 64$. If R is increased to radix-4 $L = 64$ is the preferred option, though $L = 128$ is competitive at the beginning, as size increases the gap between $L = 64$ and $L = 128$ becomes wider. In this case radix-8 results are not provided because the algorithm requires too many

Table 4.5: Impact of the task number for tridiagonal systems using Radix-2 and Radix-4 for BPLG-TS (Platform 2)

	Radix-2						Radix-4				
N	L32	L64	***L128***	L256	L512	L1024	L32	L64	L128	L256	L512
4	6003	9272	***9610***	9512	9228	5511	7635	***9314***	6326	6365	6478
8	5459	7412	***7568***	7535	7072	4433	5724	***6509***	6468	6216	4098
16	3887	5367	***5479***	5474	5231	3634	5390	***7396***	7299	7094	4463
32	3194	4670	***4786***	4700	4550	3257	5545	***6744***	6641	6265	4264
64	3252	5289	***5656***	5431	5014	3384	5292	***6993***	6845	6353	3870
128		4652	***5013***	4872	4061	3054	4666	***5838***	5721	5288	4568
256			***4501***	4037	3677	2777		***6030***	5768	5420	3195
512				***3582***	3348	2551			***4522***	4482	3072
1024					***3092***	2350				***4332***	2814
2048						***2179***					***2693***

registers, which incurs in local memory spilling and the consequent performance degradation. Notice that there is a similar resource balance when using radix-2 and $L = 128$ compared to radix-4 and $L = 64$, as both process 256 equations per block. Nonetheless, radix-4 is usually better because it minimizes shared memory exchanges and synchronizations, which are more expensive compared to the *BPLG* signal processing algorithms because more data is exchanged in each stage.

4.4.2.2. Tridiagonal system resolution performance

As it can be seen in the Figure 4.11, *BPLG-TS* implementation offers excellent performance compared with other two state-of-the-art solutions. The batch support provides a very good startup, which makes the library suitable for small problems (up to 9609.8 Mrows/sec for $N = 4$ running on Platform 2). The percentages near the lines of *BPLG-TS* represent the performance improvement over the fastest one from the other two implementations (*CUDPP* or *CUSPARSE*) running on the same platform. Even in the worst case *BPLG-TS* achieves over an 83.6% advantage over *CUSPARSE*. Furthermore, *BPLG-TS* executed on Platform 1 is usually faster than the other two running on Platform 2. The jagged outline, like in $N = 32$ or $N = 128$, is due to the mixed-radix stage. Performance tends to decrease with the number of stages because more synchronizations are required and the shared memory becomes a scarce resource, which reduces *SM* block parallelism. As in the case of the signal transforms, for equation systems where $N > 2048$ a different approach would be

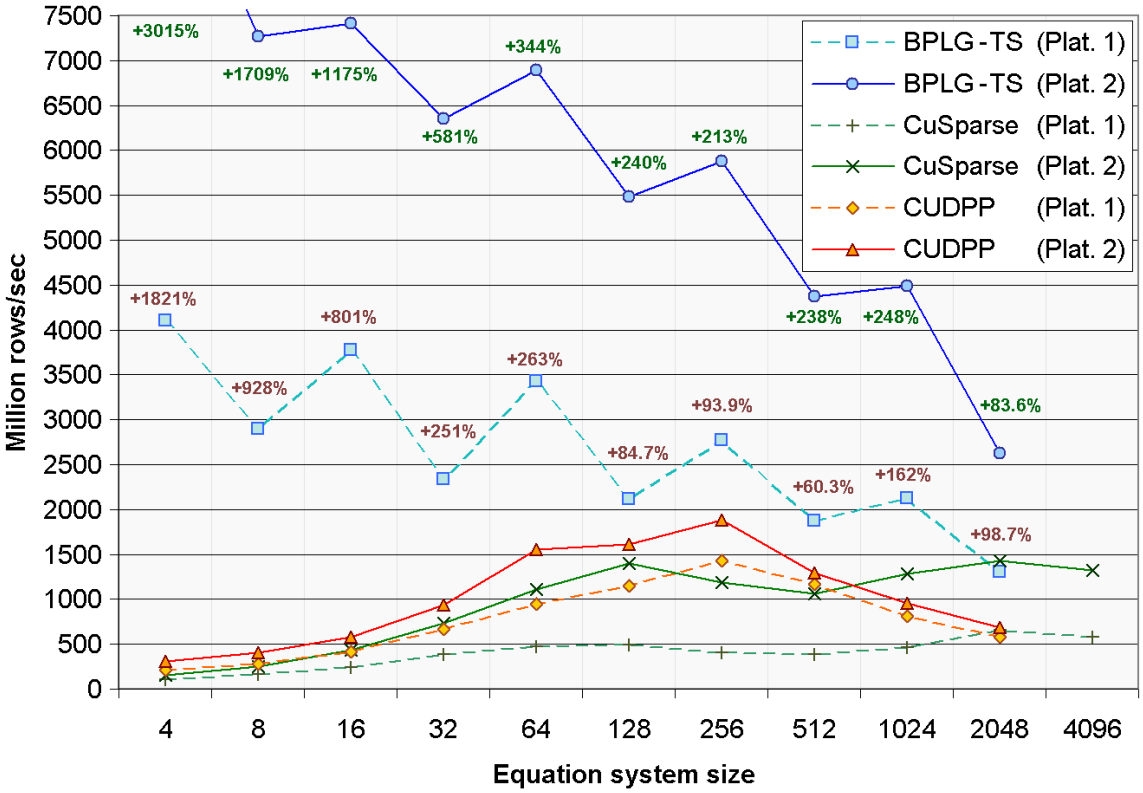


Figure 4.11: Algorithm performance for tridiagonal equation system resolution

required, like using a multi-kernel algorithm. This will be considered on future work.

Regarding the other two libraries, *CUSPARSE* and *CUDPP*, the observed performance difference between the two *GPU* architectures is smaller, specially in the case of *CUDPP*. Surprisingly, *NVIDIA's* own library usually offers the worst performance. Profiling *CUSPARSE* reveals that it is launching several kernels to solve the batch of problems, therefore the global memory bandwidth becomes a limiting factor. Only for the larger problems it is able to amortize the cost of the multi-kernel approach, which hints that it was probably designed with large equation systems in mind. Last, regarding *CUDPP*, at the beginning is quite similar to *CUSPARSE*, however as the problem size increases it gets better, being able to improve *NVIDIA's* library performance. Nonetheless, for $N > 512$ it becomes slower. This behavior is easily explained by the fact that each system is assigned to a single *CUDA* block. For small systems the kernel will be launched with a single warp per block, furthermore

some of the threads will be idle. For larger problems the shared memory becomes a limiting factor.

As a brief summary of this chapter, remark that *BPLG* outperforms other state-of-the-art solutions for all the considered algorithms but for *BPLG-cFFT* case. Next chapter of this thesis shows how even for the complex *FFT*, the proposed methodology is able to be competitive respect to the *NVIDIA CUFFT*.

Chapter 5

Efficient Index-Digit Algorithms Design for GPU Architectures

Detailed hardware specification and theoretical performance study provide information that can be used by the programmer in the optimization of applications. Many works were proposed about the elaboration of models for *GPU* performance analysis [116, 121], while other works study concrete *GPU* architectures through micro-benchmarks [43, 126, 141].

Autotuning [53, 65, 136, 137] is also a very interesting topic for *GPU* code development, although additional programming efforts are required. Autotuning requires writing code in a parametrized way to accommodate various performance tuning parameters, a pruned search-based strategy is commonly used to select the most efficient solution.

Regarding *FFT* autotuning, in [136] a wide range of variants are generated and the best solution is selected, however the methodology is centered only in the *FFT*. In [137] a different methodology of autotuning for *OpenCL* is used, it is based on compiler technology with a two-stage adaptation approach in different levels. In [4] 3D *FFTs* are performed on *GPU* with autotuning, but this approach does not explicitly consider some main performance factors such as the right balance between the high number of simultaneous tasks and the proper utilization of the shared resources.

Regarding autotuning for tridiagonal solvers, [1] uses an autotuning design for large systems that cannot be stored on the shared memory and require a multi-stage method.

In contrast, in this chapter we provide an alternative tuning optimization solution based on resource analysis, obtaining a limited search space by pruning it according to a series of performance features. More precisely, our approach is based on a two-stage methodology suitable for algorithms described as index-digit permutations. In the first stage a set of factors, that characterize the behavior of *GPU* in terms of performance, is obtained from a resource analysis. In the second stage, operator string manipulation [30] combined with tuning mapping vector is used to describe and adjust the data distribution in the *GPU* resources according to the resource analysis made at the first stage. Furthermore, the operator string manipulation enables the design of modular yet efficient kernels tuned for the *GPU* architecture, with a compact notation to represent the operations carried by the algorithm. Our design makes extensive usage of template metaprogramming [104], so many operations are performed at compile-time reducing any performance penalty, with the advantage of designing a flexible implementation while minimizing code replication. Thanks to the template functions the main kernel code can be easily obtained from the corresponding operator strings following simple conversion rules.

Specifically, our methodology has been applied to develop flexible Index-Digit algorithms for *CUDA GPUs* such as the *FFT* (*Fast Fourier Transform*) algorithm (ID-FFT) and a tridiagonal solver algorithm (ID-TS), showing that our work is able to surpass the performance of *NVIDIA's* libraries and other efficient proposals from the bibliography.

5.1. A 2-stage methodology for efficient index-digit algorithms design

Our approach is different from other tuning methodologies because only a few kernels are generated according to a small number of parameters but providing an efficient solution. Specifically, our methodology is based on two stages: *GPU* resources utilization analysis and operators string manipulation.

In the *GPU* resource utilization analysis stage, different features are analyzed in order to obtain the main factors that affect to performance. Concretely, a key property is to establish a balanced ratio between the number of parallel threads and the amount of shared resources that can be assigned to each thread. This can be described quantitatively through what we call *resource factors*.

In the operators string manipulation stage, algorithms are formulated with a set of operators which allow us to describe the operations carried out and redistribution of data among GPU resources. The distribution of data among GPU resources is described as the combination of two techniques: *mapping vector* and *index-digit permutations* [20], which allow us to define first the initial data distribution and then the data redistribution in each stage of the algorithm among resources of each *SM* of a GPU. The objective is to design efficient algorithms that can be easily adapted to architecture characteristics while optimizing all the features described in Section 1.1.3 using the tuning mapping vector explained in Section 5.1.1.

5.1.1. Applying Mapping Vector Techniques to GPUs

This section describes the data distribution on the *SM* resources using a mapping vector based on the index-digit notation [30].

Let us consider the mapping of a one dimensional data sequence of size $N = R^n$, where R depends on the selected radix size. We will denote this data sequence using the index-digit representation. That is, data item $x(t)$ with index $t = t_n \cdot R^{n-1} + \cdots + t_2 \cdot R + t_1$ is written as $[t_n \cdots t_2 t_1]$. In the *GPU*, kernel data is divided among $B = r^b$ execution blocks, and each of these blocks executes $L = r^l$ threads. A thread performs the calculation of $P = r^p$ data stored in private registers and threads within a block have access to $S = r^s$ data stored in shared memory. Then, the mapping of signal data of size $N = r^n$ within of r^{batch} number of signals that will be simultaneously processed by the kernel in a single invocation is identified with a 5-tuple of the form (n, p, s, l, b). Specifically, our proposal is based only on three parameters of these (n, p, s) because $s = p + l$, as all the data stored in registers also has a copy in shared memory to perform the inter-stage memory exchanges, and $b = batch - (s - n)$ is given by the batch size, which is only known at runtime.

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

$$[\ t_{n+batch} \cdots t_{n+1}\ t_n \cdots t_1\ ] \tag{5.1}$$

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

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

$$[\ \underbrace{t_{n+batch} \cdots t_{s+1}}_{b}\ \overbrace{\underbrace{t_{l+p} \cdots t_{p+1}}_{s}\ \overbrace{t_p \cdots t_1}^{p}}^{l}\ ] \tag{5.2}$$

This means first that each block $i = [t_{n+batch} \cdots t_{s+1}]$ processes S data that are stored in shared memory and second that thread $j = [t_{l+p} \cdots t_{p+1}]$ within a block processes P data where datum $[t_{l+p} \cdots t_{p+1}\ t_p \cdots t_1]$ is stored on the register $[t_p \cdots t_1]$ of thread j.

For example, Figure 5.1 displays the mapping to the GPU resources where $s = 9$ and $p = 4$ for the case $r = 2$ and $n = 4$. In this case, signals are of $N = 16$ data size, so observe how each block receives a set of input signals, in this case $512/16 = 32$ signals. Data are stored in shared memory, and evenly distributed to the registers of the 32 threads within each block; and each thread processes only a signal of the batch of 32 signals. The mapping vector for the example is:

$$[\ \underbrace{\cdots t_{11}\ t_{10}}_{b}\ \overbrace{\underbrace{t_9\ t_8\ t_7\ t_6\ t_5}_{s}\ \overbrace{t_4\ t_3\ t_2\ t_1}^{p}}^{l}\ ] \tag{5.3}$$

For illustration purposes, suppose that data 1 from input signal 65 (element 1041 of Figure 5.1) is assigned to GPU's global memory position:

$$[\ \underbrace{\cdots 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 1}_{batch}\ \underbrace{0\ 0\ 0\ 1}_{n}\ ] \tag{5.4}$$

applying the mapping of Expression 5.3, it will result in the following distribution

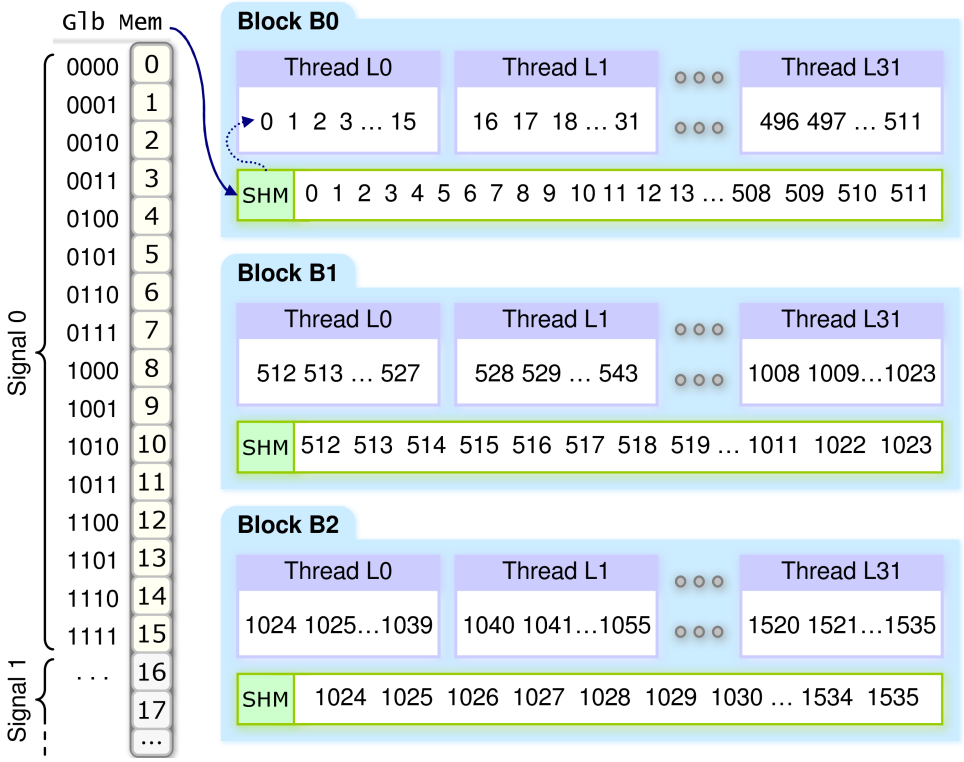


Figure 5.1: Input data mapping on the GPU resources where $r = 2$, $n = 4$, $s = 9$ and $p = 4$

in the SM:

$$[\underbrace{\cdots 0\ 1\ 0}_{b}\ \overbrace{\underbrace{0\ 0\ 0\ 0\ 1}_{s}}^{l}\ \overbrace{0\ 0\ 0\ 1}^{p}\ ] \tag{5.5}$$

This data is processed by thread 1 of block 2, stored in the register 1 and also stored in shared memory location 17.

5.1.2. Index-digit permutations

In this section we present a set of algebraic operators [30] that allow the description and simplification of index-digit algorithms. Index-digit permutations show the rearrangement of a data array according to a common permutation of the digits of each element's index. This formulation provides a clear and precise description of the data reordering, being useful in the design and simplification of the different

algorithms.

For writing the expressions of the operators we will follow the convention of composing operators from left to right. For example in the operator string $\phi_1\phi_2$, we first execute ϕ_1 and then, ϕ_2. We will define two types of operators, which correspond to computations and data permutations respectively. First, we define the operator that represents the computations.

Definition 1. *The butterfly operator, B_i^r, with $1 \leq i \leq n$, $r = log_2 R$, and $n = log_R N$, reads those sets of R data items whose position differs precisely in their i-th digit, performs the required radix operation over them and writes the R results.*

In general, to simplify the notation when using the basic radix-2 algorithm, the expression of this operator will be simply referred to as B_i instead of B_i^1. Furthermore, in order to clarify the explanation, we keep the index-digit representation with $R = 2$ and $n = log_2 N$. Therefore, the butterfly operator B_i^r, with $1 \leq i \leq n - r + 1$, reads those sets of R data items whose position differs precisely in their $[t_{i+r-1} \cdots t_i]$ digits, performs the required radix operation over them and writes the R results.

The second type of operators represent memory access patterns and data permutations.

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

$$\Gamma_{i,j}[t_n \cdots t_1] = [t_n \cdots t_{i+1} \underline{t_j t_i \cdots t_{j+1}} t_{j-1} \cdots t_1]. \tag{5.6}$$

We also define a generalization of this operator, $\Gamma_{i,j}^m$. Instead of performing a single cyclic shift to the right, it will perform m consecutive shift operations, as an example $\Gamma_{i,j}^2 = \Gamma_{i,j}\Gamma_{i,j}$. For instance, $\Gamma_{7,2}^2[\ t_8\ t_7\ t_6\ t_5\ t_4\ t_3\ t_2\ t_1\ ] = [\ t_8\ \underline{t_3\ t_2\ t_7\ t_6\ t_5\ t_4}\ t_1\ ]$.

Definition 3. *The perfect 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 $\{i \ldots j\}$ and $\{k \ldots l\}$ of the index-digit representation,*

$$\begin{aligned} \Gamma_{i,j,k,l}[t_n \cdots t_1] = \quad & [t_n \cdots t_{i+1} \underline{t_l t_i \cdots t_{j+1}} t_{j-1} \cdots t_{k+1} \\ & \underline{t_j t_k \cdots t_{l+1}} t_{l-1} \cdots t_1]. \end{aligned} \tag{5.7}$$

Therefore data in the range $\{t_{j-1} \cdots t_{k+1}\}$ remains unmodified. For instance, $\Gamma_{8,6,2,1}[\ t_8\ t_7\ t_6\ t_5\ t_4\ t_3\ t_2\ t_1\ ] = [\ \underline{t_1\ t_8\ t_7}\ t_5\ t_4\ t_3\ \underline{t_6\ t_2}\ ]$. An analogous operator $\sigma_{i,j,k,l}$ with two subfields is defined for the shuffle operator.

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,*

$$\rho_{i,j}[t_n \cdots t_1] = [t_n \cdots t_{i+1} \underline{t_j t_{j+1} \cdots t_i} t_{j-1} \cdots t_1]. \tag{5.8}$$

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 as $\rho_{i,j}\rho_{i,j} = 1$.

5.1.3. Operator string algebraic properties

Following we will define some algebraic properties of the operators that will allow us to manipulate the sequences and perform expression transformations [30]:

Lemma 1. *Two or more butterfly operators can be grouped together increasing the radix:*

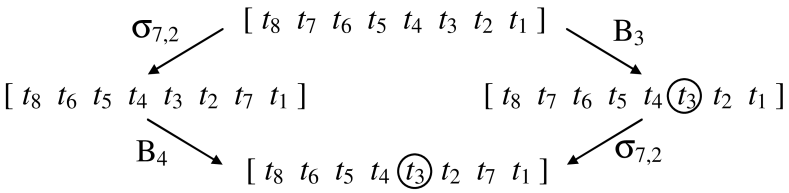
$$B^r_{i \cdot r-1} B^r_{i \cdot r} = B^{r+r}_i \tag{5.9}$$

Proof: Immediate, having two bases δ and λ with $\delta = \lambda^k$ the following equality is also true: $(\cdots a_3 a_2 a_1)_\lambda = (\cdots q_3 q_2 q_1)_\delta$ where $q_j = (a_{kj+k-1} \cdots a_{kj+1} a_{kj})$

Lemma 2. *The butterfly operator B_i commutes with all those operators that do not modify the* i*-th digit, for example, $B_i \sigma_{j,k} = \sigma_{j,k} B_i$ when ($i > j$ or $i < k$). Otherwise, if an operator modifies the* i*-th digit:*

$$\sigma_{k,j} B_i = \begin{cases} B_{i-1} \sigma_{k,j} & j < i \leq k \\ B_k \sigma_{k,j} & j = i, i \leq k \end{cases} \tag{5.10}$$

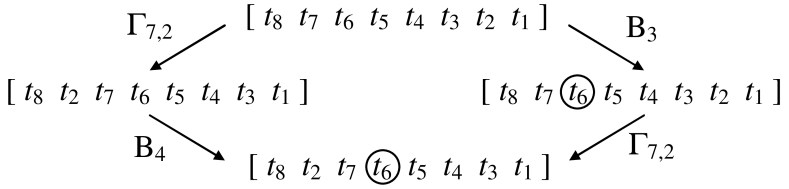
For instance, suppose that $k = 7$, $j = 2$ and $i = 4$:



And in the case of the perfect unshuffle operator:

$$\Gamma_{k,j}B_i = \begin{cases} B_{i+1}\Gamma_{k,j} & j \leq i < k \\ B_j\Gamma_{k,j} & j \leq i, i = k \end{cases} \tag{5.11}$$

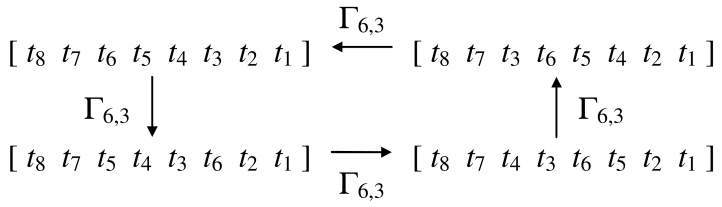
For example, suppose that $k = 7$, $j = 2$ and $i = 5$:



Lemma 3. *When the exponent m of a shuffle operator is equal to the space between i and j, the operator becomes the identity and does not modify the expression:*

$$\sigma^m_{i,i-m+1} = \Gamma^m_{i,i-m+1} = 1 \tag{5.12}$$

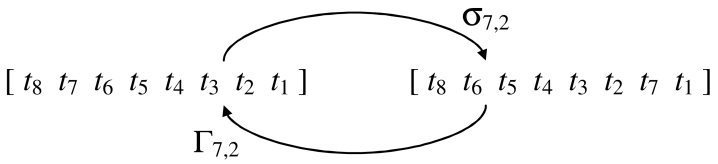
For instance, suppose that $m = 4$ and $i = 6$, $\Gamma^4_{6,3}$ can be decomposed in four consecutive $\Gamma^1_{6,3}$ operators:



Lemma 4. *The perfect shuffle operator σ and the perfect unshuffle operator Γ perform inverse displacements, therefore consecutive σ and Γ operators with the same parameters results in the identity:*

$$\sigma^m_{i,j}\Gamma^m_{i,j} = \Gamma^m_{i,j}\sigma^m_{i,j} = 1 \tag{5.13}$$

For example, suppose that $m = 1$, $i = 7$, $j = 2$:

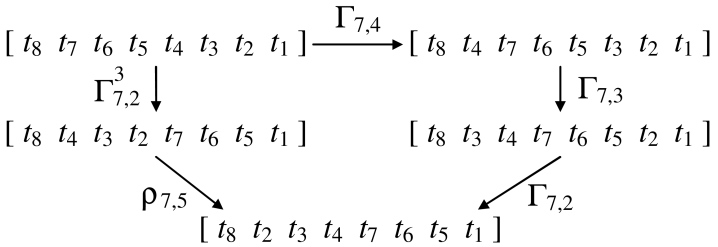


The proofs of the last three lemmas are immediate by the definition of the corresponding operators.

Lemma 5. *A sequence of consecutive* Γ *operators can be rewritten using a single* Γ^m *operator as follows:*

$$\prod_{k=0}^{m-1} \Gamma_{i,j-k} = \Gamma^m_{i,j-m+1}\rho_{i,i-m+1} \tag{5.14}$$

The product symbol is just a notation used to generate the sequence, remember that in general the operators do not satisfy the commutative property. This lemma can be proven by induction on m. For instance, suppose that $i = 7$, $j = 4$ and $m = 3$:



As it will be shown, thanks to the detailed operator string algebra it is possible to derive other algorithms from a given expression.

5.1.4. Optimized algorithm mapping using operator strings

To show how the algebraic manipulation works we present an example.

The *Stockham* [127] algorithm can be represented using the following expression:

$$\prod_{i=1}^{n} \Gamma_{n,n-i+1} B_n^r. \tag{5.15}$$

Thanks to the intermediate Γ operators the output will be in bit-reversed order without requiring an explicit ρ operator like the *Cooley-Tukey* [66] algorithm does. Supposing that $n = 6$ and $r = 1$, the corresponding operator sequence would be:

$$\Gamma_{6,6} B_6 \Gamma_{6,5} B_6 \Gamma_{6,4} B_6 \Gamma_{6,3} B_6 \Gamma_{6,2} B_6 \Gamma_{6,1} B_6 \tag{5.16}$$

According to Lemma 2 the Γ operators in even positions can be moved before their previous B_6 operator, accordingly modifying the butterfly index to B_5:

$$\Gamma_{6,6} \Gamma_{6,5} B_5 B_6 \Gamma_{6,4} \Gamma_{6,3} B_5 B_6 \Gamma_{6,2} \Gamma_{6,1} B_5 B_6 \tag{5.17}$$

For each sequence of the form $B_5 B_6$ the property described in Lemma 1 is applied, the expression is simplified replacing the radix-2 butterfly operators with more compact and efficient radix-4 operators:

$$\Gamma_{6,6} \Gamma_{6,5} B_5^2 \Gamma_{6,4} \Gamma_{6,3} B_5^2 \Gamma_{6,2} \Gamma_{6,1} B_5^2 \tag{5.18}$$

Applying the property described in Lemma 5 it is possible to replace each sequence of two Γ operators by a single Γ^2 followed by a $\rho_{6,5}$ operator. Notice that this property does not change the data communication pattern between butterfly steps:

$$\Gamma_{6,5}^2 \rho_{6,5} B_5^2 \Gamma_{6,3}^2 \rho_{6,5} B_5^2 \Gamma_{6,1}^2 \rho_{6,5} B_5^2 \tag{5.19}$$

Using Lemma 3 the first operator of the sequence can be omitted, because $\Gamma_{6,5}^2$ is an identity. The sequence $\rho_{6,5} B_5^2$ will be treated as our basic building block. The *FFT* is finally expressed using three identical blocks which operate on the most significant bits of the signal, separated by two unshuffle operators that perform the intermediate data exchanges:

$$\boxed{\rho_{6,5} B_5^2} \Gamma_{6,3}^2 \boxed{\rho_{6,5} B_5^2} \Gamma_{6,1}^2 \boxed{\rho_{6,5} B_5^2} \tag{5.20}$$

The last operator string only takes into account the problem data, but for the final expression the batch within the shared memory space should be also considered. In our example, using Lemma 4 we can add two opposite shuffles at the beginning of the last expression, therefore:

$$\Gamma^6_{s,1}\sigma^6_{s,1}\boxed{\rho_{6,5}B^2_5}\Gamma^2_{6,3}\boxed{\rho_{6,5}B^2_5}\Gamma^2_{6,1}\boxed{\rho_{6,5}B^2_5} \tag{5.21}$$

And now $\sigma^n_{s,1}$ can be displaced to the end of the string using Lemma 2, just updating the affected indexes:

$$\Gamma^6_{s,1}\rho_{s,s-1}B^2_{s-1}\Gamma^2_{s,s-3}\rho_{s,s-1}B^2_{s-1}\Gamma^2_{s,s-3}\rho_{s,s-1}B^2_{s-1}\sigma^6_{s,1}$$

Hereby, batch data is included, using the shared memory space to perform $s-n$ shifts at the beginning of the process, so the register data is in the higher bits. Omitting the first and last displacements for simplicity, the expression can be rewritten as:

$$\prod_{i=1}^{3}\Gamma^2_{s,s-2i+1}\rho_{s,s-1}B^2_{s-1} \tag{5.22}$$

Expression 5.22 can also be obtained from the *ID-FFT* algorithm that we propose, which has the following expression:

$$\prod_{i=1}^{\lfloor n/r\rfloor}\Gamma^r_{n,n-i\cdot r+1}\ \rho_{n,n-r+1}\ B^r_{n-r+1} \tag{5.23}$$

If generalized to use shared memory, assuming that $s \geq n$ and $n\ mod\ r = 0$, it can be rewritten as:

$$\prod_{i=1}^{\lfloor n/r\rfloor}\Gamma^r_{s,s-i\cdot r+1}\ \rho_{s,s-r+1}\ B^r_{s-r+1} \tag{5.24}$$

For the particular case of the example, when $n = 6$ and $r = 2$, the resulting expression is:

$$\prod_{i=1}^{3}\Gamma^2_{s,s-2i+1}\ \rho_{s,s-1}\ B^2_{s-1} \tag{5.25}$$

This is equal to Expression 5.22, which was originally derived from Expression 5.15 using the algebraic properties introduced in Section 5.1.3.

5.2. GPU resources utilization analysis stage

GPU performance depends on the right balance between the high number of simultaneous tasks and the proper utilization of the shared resources in the hardware. An optimal balance can be expressed by a series of resource factors that will be established in this section.

Trying to improve performance by increasing the number of warps per *SM* may not result the best option in many cases. In fact, the *GPU* hardware is able to provide very good performance even at lower occupancies [131, 132]. Lowering the occupancy provides more resources for each thread, thus, more work can be done in a single thread and there will be more opportunities to hide both instruction and memory latency. Instead of aiming to maximize the number of warps per *SM* (*SM* warp parallelism) we will focus on the number of blocks per *SM* (*SM* block parallelism), trying to keep processing the maximum amount of simultaneous blocks per *SM* (8 in the case of *Fermi* and 16 in the case of *Kepler* based *GPUs*).

Figure 5.2 analyzes how in the *Fermi* architecture the number of blocks that can be managed by each *SM* affects the maximum number of registers per thread in terms of the warps within the block. In the case of *Kepler* the total amount of registers per *SM* is doubled from 32768 to 65536, but so is the amount of maximum blocks per *SM* from 8 to 16, therefore maintaining the same optimal proportions. The number of registers used by each thread is assigned at compile-time. In hardware with *CUDA* capabilities 2.x or 3.0 it is not possible to assign more than 63 to the same thread. Hardware with *CUDA* capabilities 3.5 supports up to 255 registers per thread, however this heavily impairs *GPU* parallel execution usually resulting in bad performance. If the kernel requires more registers than supported by the architecture, local memory spilling will be generated. Regarding the optimal register balance three significant cases are highlighted in Figure 5.2 using a vertical line. Creating 4 warps per block (128 threads) it is possible to maintain 8 blocks per *SM* while allocating 32 registers. Alternatively, if the code is complex and additional

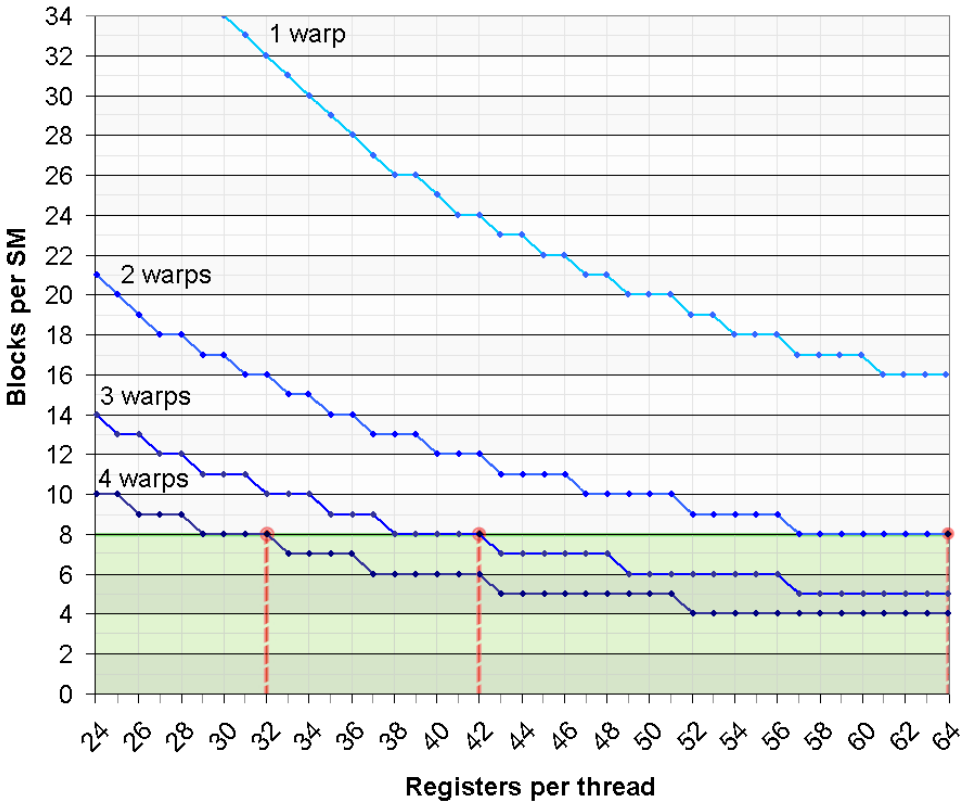


Figure 5.2: Simultaneous hardware blocks per SM in Fermi architecture depending on registers per thread

registers are required, it is possible to maintain 8 blocks per *SM* using either 3 warps (96 threads) and 42 registers or 2 warps (64 threads) and 64 registers.

In applications that require thread collaboration or communication, the availability of shared memory is essential. Figure 5.3 presents detailed information about how the number of blocks that can be managed by each *SM* affects the amount of shared memory bytes per thread in terms of the warps within the block. The threshold to maintain 8 blocks per *SM* in the *Fermi* architecture or 16 blocks per *SM* in the *Kepler* architecture depending on the amount of warps per block is highlighted in the figure. Choosing 4 warps only 24 bytes in *Fermi* and 12 bytes in *Kepler* of shared memory can be used by each thread. To fit more data per thread the programmer is allowed to use 2 warps and 96 bytes of shared memory in *Fermi* or 48 bytes in *Kepler*.

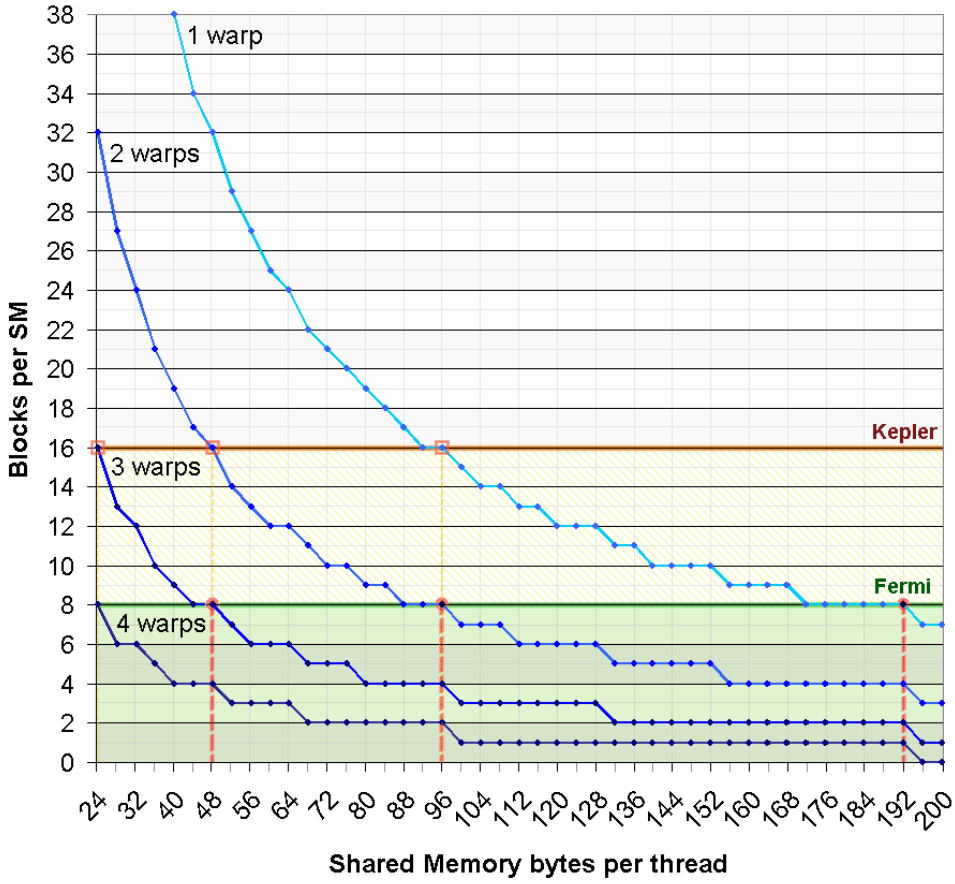


Figure 5.3: Simultaneous hardware blocks per SM in Fermi architecture depending on the shared memory bytes per thread

In summary, as the goal of our proposal is to obtain a good resource utilization and high available threading without increasing synchronization costs too much, it is recommended to maintain a 100% *SM* block occupancy, even at the cost of some *SM* warp parallelism. Table 5.1 summarizes the resource distribution imposed by this requirement for *Fermi* and *Kepler GPU* architectures. The table shows the maximum amount of registers per thread, shared memory elements per thread (assuming 32-bit values), and total *SM* warp occupancy ratio depending on the selected number of warps per block. Notice that *Kepler GPUs* with compute capabilities 3.5 can allocate up to 255 registers for a single thread without spilling, however 128 registers is the maximum that can be allocated for 1 warp per block without decreasing *SM* parallelism.

Table 5.1: Resource factors table depending on warps/block

	Warps per block	Registers per thread	Shared mem. bytes/thread	SM warp occupancy
Fermi	1	63	192	16.6%
	2	63	96	33.3%
	3	42	48	50.0%
	4	32	24	66.6%
Kepler	1	63 / 128	96	25.0%
	2	63	48	50.0%
	3	42	24	75.0%
	4	32	12	100.0%

Following the optimal values for those parameters (p, s, l) as defined in Section 5.1.1 are obtained, first for the *FFT* and then for the tridiagonal system algorithm.

5.2.1. Resource utilization analysis for FFT

Regarding the FFT, to find out the optimal resource factors (p, s, l) we rely on the fact that the optimal choice is to calculate an *FFT* Butterfly with data stored in the private registers of a task [57]. Therefore, the radix size will be determined by p, being at most radix-P. The number of operations per thread increases with the size of the butterfly, therefore the GPU can make better usage of the execution resources. Therefore, the objective is choosing a p maximum but achieving the right balance between the high number of simultaneous tasks and the proper utilization of the shared resources.

The maximum number of registers that can be effectively allocated on *NVIDIA's Fermi* platform for a single thread without register spilling to local memory is 63, therefore $p < 5$. For $p = 5$ the registers would be set to 32 complex elements, reserving up to 64 registers for the algorithm and leaving no space to accommodate the required temporary variables. For $p = 4$ the amount of reserved registers is usually between 62 and 45 (it depends on the algorithm), including problem data, as well as temporary variables for counters and strides, data reordering or twiddles. Therefore, p has to be equal to or less than 4, and considering Figure 5.2 the available options to have 8 simultaneous blocks per *SM* are to use blocks composed by 1 or 2

warps. From this information and based on Figure 5.3, it is possible to calculate that in *Fermi* the most adequate options for shared memory are either 192 bytes/thread for 1 warp, or 96 bytes/thread for 2 warps. In both cases:

$$\begin{aligned} 192\ bytes/thread \times 32\ threads/block = \\ 96\ bytes/thread \times 64\ threads/block = \\ 6144\ bytes/block \end{aligned}$$

Using $s = 9$ it is possible to store up to 2^9 complex elements in shared memory and we obtain

$$\begin{aligned} 512\ complex/block \times 8\ bytes/complex = \\ 4096\ bytes/block,\ where\ 512 = 2^s = 2^9 \end{aligned}$$

however, if we were to use $s = 10$

$$\begin{aligned} 1024\ complex/block \times 8\ bytes/complex = \\ 8192\ bytes/block,\ where\ 1024 = 2^s = 2^{10} \end{aligned}$$

Accordingly, for maximum *SM* block occupancy $s \leq 9$, so the shared memory space will remain under the 6144 bytes limit. In this work we will consider $s = 9$, as working with a smaller value will not provide any performance benefit. Shared memory padding will be used to minimize bank conflicts, hence the blocks may require slightly more than 4096 bytes.

Regarding the tuning targeted for the *Kepler GPUs*, the architecture doubles the register file and can execute up to 16 blocks per *SM* (twice than *Fermi*, therefore maintaining the register-per-block ratio), but the available shared memory remains the same. Thus, the proposed solution is to split the permutations into two steps, exchanging first the real part and then exchanging the imaginary part. This halves the amount of shared memory required by the algorithm at the cost of one additional synchronization, which has a negligible impact thanks to the small block size. This can also be applied to *Fermi* to enable $s = 10$ but, depending on the implementation, registers would become a limiting factor or the increased code complexity to take advantage of it would probably negate any benefit.

Regarding p, two options are available (see Table 5.1), and as stated before this parameter will also impose an upper limit to the maximum radix size. The options are to use 1 warp ($l = 5 \Rightarrow p = s - l = 9 - 5 = 4$) and process data in radix-16 steps at most, or to use 2 warps ($l = 6 \Rightarrow p = 9 - 6 = 3$) and process data in radix-8 steps at most. In those cases where $p = 3$ and $p = 4$ are viable the option with the best *SM* warp parallelism is preferred as can be observed in the results (see Section 5.5). For $7 \leq n \leq 9$ cases it is possible to choose between using $p = 3$ and two warps favoring *SM* warp parallelism, or using $p = 4$ and just one warp minimizing block synchronizations. In both cases the number of simultaneous blocks per *SM* will be 8, however for $p = 4$ a single warp per block is created (which results in 1/6 of the *SM* occupancy), while $p = 3$ it is possible to use 2 warps, thus increasing the number of simultaneous warps per *SM* and improving the *GPU* parallelism (1/3 of the *SM* occupancy).

In *Kepler GPUs* according to Table 5.1, p could be 5 but this option is not viable because l would be 4 that is the half of a warp (16 threads). This options is on average 38% slower due to the best performances of current GPUs is obtained if the size of a thread block is multiple of warp size.

For $n \geq 10$ it was opted to use $s = 12$, thereby allowing to simplify the memory access pattern. This configuration violates the optimal resource factor $s = 9$, but the optimal implementation when the signal is larger than the shared memory space ($n > s$) will be further studied in a future work. This presents several mapping strategies, but in general the computation is divided into two or more kernels, which will process a smaller portion of the signal at the cost of more global memory accesses and additional kernel invocation time.

In summary according to our analysis for Kepler and Fermi GPUs, the resource factors for a *FFT* of size n could be chosen from the following options:

$$(p,\, s,\, l) = \{\ (4,\, 9,\, 5),\ (3,\, 9,\, 6),\ (4,\, 12,\, 8)\ \}.$$

5.2.2. Resource utilization analysis for tridiagonal systems

Regarding the optimal resource factors for the tridiagonal system solver most of the previous reasoning is still valid. However, in contrast to the complex input

data of the *FFT*, this algorithms uses equations as inputs (see Section 5.3.2 for a description more detailed). Each row of the tridiagonal matrix is an equation that requires four floating point values (three diagonals and the independent term), therefore it can be easily stored in a *float4* data type. Each *float4* requires at least 16 bytes of storage, which is twice the amount of *Complex* data, therefore the optimal shared memory is proportionally reduced to s=8. Regarding p the Wang and Mou algorithm internally operates with triads of equations, therefore the register requirements is not twice, but six times as high compared to the *FFT*. Thus, a radix-4 algorithm would require:

$$4 \times 3\ eqn/triad \times 4\ floats/eqn = 48\ registers \tag{5.26}$$

Notice that in the case of radix-8:

$$8 \times 3\ eqn/triad \times 4\ floats/eqn = 96\ registers \tag{5.27}$$

which will cause memory spilling in *CUDA GPUs* limited to 63 registers per thread.

GPUs with compute capabilities 3.5 allow up to 255 registers per thread, but $p = 3$ implies $l = 8 - 3 = 5$. The computational load of *ID-TS* kernel is too light per thread and the number of simultaneous warps should be increased in order to efficiently hide memory and arithmetic latencies. Accordingly, the best performance will be usually obtained using $s = 8$ and $l \geq 6$, that is 64 threads (2 warps).

For $n \geq 9$ it was opted to use a strategy similar to *ID-FFT* proposal. Nevertheless, *ID-TS* proposal using $s = n$ obtains an efficient memory access pattern due to the excluding of the bit-reversal ordering.

In summary, the resource factors for a *ID-TS* on Kepler or Fermi GPUs could be chosen from the following options:

$$(p,\ s,\ l) = \{\ (2,\ 8,\ 6),\ (2,\ n,\ n-2)\}.$$

5.3. Operators string manipulation stage

In this section, both efficient FFT algorithm, called *ID-FFT* (Index-Digit FFT), and our proposal of solution for tridiagonal systems, called *ID-TS* (Index-Digit TS), are presented. The objective is to design efficient algorithms that can be easily adapted to architecture characteristics while optimizing all resource factors described in the previous sections using the operators string manipulation. Remark that this methodology is standard enough to be easily employed in the design of other Index-Digit algorithms, for example extending previous developments for orthogonal transforms algorithms in [58].

5.3.1. The FFT Case

Following we present the formulation of the general *ID-FFT* algorithm which is able to overcome the *GPU* architecture performance features seen on Section 1.1.3.

$$\prod_{i=1}^{\lfloor n/r \rfloor} \Gamma^{r}_{n,n-i\cdot r+1}\ \rho_{n,n-r+1}\ B^{r}_{n-r+1} \tag{5.28}$$

See [60] for a demonstration of the operator properties and how using the operator algebra it is possible to derive Equation (5.28) from the *Stockham* expression. Equation (5.28) computes $\lfloor n/r \rfloor$ radix-R steps. When $m = n\ mod\ r \neq 0$ a mixed-radix approach is used, where any remaining work is performed by computing a radix-2^m step. For instance, the expression for the first formulation is:

$$\Gamma^{m}_{n,1}\ \rho_{n,n-m+1}\ B^{m}_{n-m+1} \tag{5.29}$$

Although a single version of the algorithm is possible, generating four different variants allows tweaking many specific details for optimal performance. These versions are derived from Equation (5.28) and match the cases for resource factors that were described in Section 5.2.1. Thanks to the usage of parametrized templates it was possible to obtain a compact and flexible algorithm for each case:

- *ID-FFT*.V1: The signal size is smaller or equal to the private register space and

fits in the shared memory space ($n \leq p$):

$$[\underbrace{t_{n+batch} \cdots t_{s+1}}_{b} \overbrace{\underbrace{t_{p+l} \cdots t_{p+1}}_{s}}^{l} \overbrace{t_p \cdots t_{n+1} \underbrace{t_n \cdots t_1}_{n}}^{p}] \tag{5.30}$$

The operator string used in these cases was obtained from Equation (5.28) and it is the following:

$$\rho_{n,1}\ B_1^n \tag{5.31}$$

In this case the radix size is always equal to the signal size ($r = n$) and each thread can compute a whole problem of the batch in a single computation step. In fact, if $n < p$ each thread computes n/p problems at once, increasing arithmetic workload and reducing scheduling overhead.

In this case the resource factors used are (p, s, l) = (4, 9, 5). A single warp will be used ($l = 5$), taking advantage of the maximum amount of registers that can be reserved while avoiding local memory spilling ($p = 4$). The shared memory space ($s = 9$) is filled with batch problems. To clarify the assignment between mapping vectors, consider the following example:

$$[\underbrace{\cdots t_{11}\ t_{10}}_{b} \overbrace{\underbrace{t_9\ t_8\ t_7\ t_6\ t_5}_{s}}^{l} \overbrace{t_4\ t_3\ t_2\ \underbrace{t_1}_{n}}^{p}] \tag{5.32}$$

where $n = 3$, $L = 2^5 = 32$ threads, $S = 2^9 = 512$ shared memory assigned to threads within a block and $P = 2^4 = 16$ registers. Thus, each thread is assigned to process data from $P/N = 16/8 = 2$ different signals in batch mode (this is given by the element in index-digit in location 4 which is inside p but not included in n). Data that differs in $[\ \cdots\ t_{11}\ t_{10}\ ]$ is processed by different *CUDA* computing blocks in batch mode.

- *ID-FFT*.V2: The signal is larger than the private register space and fits in the

shared memory space with $p < n \leq 2p$ and $2p + 1 = s$.

$$[\ \underbrace{t_{n+batch} \cdots t_{s+1}}_{b}\ \overbrace{\underbrace{t_{s-1} \cdots t_n \cdots t_{p+1}}_{s}\ \underbrace{t_s t_p \cdots t_1}_{n}}^{p}\ \overbrace{}^{l}\ ] \tag{5.33}$$

The following operator string was obtained from Equation (5.28):

$$\begin{aligned} &\left(\rho_{s-p+m,s-p+1}\ B^{m}_{s-p+1}\right) \Gamma^{p}_{s,s-p+1,p,1} \\ &\left(\rho_{s,s-p+1}\ B^{p}_{s-p+1}\right) \end{aligned} \tag{5.34}$$

In this case, from the L threads of each *CUDA* block, 2^{n-p} threads will collaborate in the same signal, while the remaining will work over different signals in batch mode. When $n - p = l$, the L threads within the block process the same signal. This version of the algorithm performs two butterfly steps, addressing problems in the range $p < n \leq 2p$ so $\lfloor (n-1)/r \rfloor = 1$.

As in the previous case, this version uses $l = 5$ and $p = 4$. Therefore, each block is composed by a single warp and it does not require explicit thread synchronizations to access shared memory. The operator string is:

$$[\ \underbrace{\cdots t_{11}\ t_{10}}_{b}\ \overbrace{\underbrace{t_8\ t_7\ t_6\ t_5}_{s}\ \underbrace{t_9\ t_4\ t_3\ t_2\ t_1}_{n}}^{p}\ \overbrace{}^{l}\ ] \tag{5.35}$$

For example, for $n = 6$, $s = 9$ and $p = 4$ the data mapping in the *GPU* resources will be as follows:

$$[\ \underbrace{\cdots t_{11}\ t_{10}}_{b}\ \overbrace{\underbrace{t_8\ t_7\ t_6\ t_5}_{s}\ \underbrace{t_9\ t_4\ t_3\ t_2\ t_1}}^{p}\ \overbrace{}^{l}\ ] \tag{5.36}$$

the signal data $[\, t_6 \cdots t_1 \,]$ does not fit in the private space of a single thread. Thus, each signal is processed among 4 threads ($[\, t_6\, t_5 \,]$), while the other 28 threads in the same block work in seven different signals.

- *ID-FFT*.V3: The signal is larger than the private register space and fits in the

shared memory space and ($2p < n \leq 3p \leq s$).

$$[\underbrace{t_{n+batch}\cdots t_{s+1}}_{b}\overbrace{\underbrace{t_n\cdots t_{n-p+1}}^{p}\ \underbrace{t_s\cdots t_{n+1}t_{n-p}\cdots t_p\cdots t_1}^{l}}^{n}] \tag{5.37}$$

The following operator string was obtained from Equation (5.28)

$$\begin{aligned}&\left(\rho_{s,s-p+1}\ B^p_{s-p+1}\right)\Gamma^p_{s,s-2\cdot p+1}\\&\left(\rho_{s-p+m,s-p+1}\ B^m_{s-p+1}\right)\Gamma^p_{s,s-3\cdot p+1}\\&\left(\rho_{s,s-p+1}\ B^p_{s-p+1}\right)\end{aligned} \tag{5.38}$$

The algorithm computes the *FFT* in three butterfly steps. Like in the previous case, from the L threads of each *CUDA* block, 2^{n-p} threads will collaborate in the same signal, while the remaining will work over different signals in batch mode. This version was designed to address larger problems in the range $2p < n \leq 3p$ so $\lfloor (n-1)/r \rfloor = 2$. In this version, resource factors used are (p, s, l) = (3, 9, 6). The operator string with $p = 3$ and $s = 9$ and $m = n\ mod\ 3$:

$$(\rho_{9,7}\ B^3_7)\Gamma^3_{9,4}(\rho_{6+m,7}\ B^m_7)\Gamma^3_{9,1}(\rho_{9,7}\ B^3_7) \tag{5.39}$$

For example, for $n = 7$ the data mapping in the *GPU* resources will be as follows:

$$[\underbrace{\cdots t_{11}\ t_{10}}_{b}\overbrace{\underbrace{t_7\ t_6\ t_5}^{p}\ \underbrace{t_9\ t_8\ t_4\ t_3\ t_2\ t_1}^{l}}^{n}] \tag{5.40}$$

the signal data are distributed among 16 threads, while the other 48 threads in the same block work in different signals. Observe how the middle part of the expression contains the batch elements t_9, t_8.

- *ID-FFT*.V4: The signal does not fit in registers nor shared memory ($n > s$). In this case there would be three different implementation strategies. The computation could be divided into two or more kernels, which will process a smaller portion of the signal at the cost of more global memory accesses and additional kernel invocation time. Another option is to use shared memory multiplexing [140] to perform the data

communication in several stages, which will result in additional synchronizations. The third option, which was the one used in this work, is to increase the shared memory space above the recommended value (see Table 5.1). Therefore, this case falls back into the previous version ($p \leq n \leq s$). Due to its complexity, the different options for the $n > s$ case will be studied with more detail in a future work.

5.3.2. The Tridiagonal System case

ID-TS is based on the Wang and Mou algorithm [135], that offers excellent performance due to its suitability for *GPU* architectures, thanks to the regular structure based on a successive doubling method. The algorithm offers good numerical stability for diagonal dominant matrices or when no pivoting is needed.

The formulation of the general *ID-TS* algorithm with operator string is the following:

$$\left[\prod_{i=1}^{\lfloor n/p \rfloor} \Gamma^p_{i\cdot p,(i-1)\cdot p+1,p,1} B^p_1\right] \Gamma^p_{n,1} \tag{5.41}$$

After the application of the B^p_1 operator in the Wang and Mou algorithm, the resulting 2^p left equations will acquire the same value. Analogously, the 2^p right equations will also have the same value. Thus, it is possible to re-use some partial results and save a few registers during the computation of the operator.

Although a single version of the algorithm is possible, generating three different variants allows tweaking many specific details for optimal performance. These versions are derived from previous equation. Thanks to the usage of parametrized templates it was possible to obtain a compact and flexible algorithm for each case:

- *ID-TS*.W1: The signal size is smaller or equal to the private register space and fits in the shared memory space ($n \leq p$):

$$[\ \underbrace{t_{n+batch} \cdots t_{s+1}}_{b} \overbrace{\underbrace{t_s \cdots t_{p+1}}^{l}\ \overbrace{t_p \cdots t_n \cdots t_1}^{p}}^{s}\ ] \tag{5.42}$$

Each task can process up to P/N signals because when $P > N$ the registers can

accommodate data from more than one signal. A single system is processed in a radix-N step and the radix size will be always equal to the signal size ($R = N$). As explained in Section 5.2.2 $p = 3$ at most, therefore $n \leq 3$, therefore the resource factors are (p, s, l) = {(3, 8, 5),(2, 8, 6)}. The operator string used in these cases is the following one: B_1^n.

- *ID-TS*.W2: The system of equations is larger than the private register space and fits in the shared memory space ($p < n \leq s$).

$$[\ \underbrace{t_{n+batch} \cdots t_{s+1}}_{b} \overbrace{\underbrace{t_s \cdots t_n \cdots t_p}_{s}}^{l} \overbrace{t_p \cdots t_1}^{p}\ ] \tag{5.43}$$

In this version, resource factors used are (p, s, l) = (2, 8, 6). The operator string used in these cases is the following:

$$\left[\prod_{i=1}^{\lfloor n/2 \rfloor} \Gamma^2_{i\cdot 2,(i-1)\cdot 2+1,2,1} B_1^2\right] \Gamma^2_{n,1} \tag{5.44}$$

from the 64 threads of each *CUDA* block, 2^{n-2} threads will collaborate in the system of equations, while the remaining will work over different system of equations in batch mode. For example, for $n = 6$ the data mapping in the GPU resources is as follows:

$$[\ \underbrace{\cdots t_{10}\ t_9}_{b}\ \underbrace{\overbrace{t_8\ t_7\ t_6\ t_5\ t_4\ t_3}^{l}}_{s}\ \overbrace{t_2\ t_1}^{p}\ ] \tag{5.45}$$

from the 64 threads of each block, $2^{6-2} = 16$ threads collaborate in the system of equations. In this example, the operator string used is:

$$B_1^2\ \Gamma^2_{4,3,2,1}\ B_1^2\ \Gamma^2_{6,5,2,1}\ B_1^2\ \Gamma^2_{6,1} \tag{5.46}$$

- *ID-TS*.W3: The system of equations is larger than the private register space and fits in the shared memory space ($s < n$). In this case, it is used a strategy similar

to *ID-FFT* proposal, where shared memory above recommended value is increased:

$$[\ \underbrace{t_{n+batch}\cdots t_{n+1}}_{b}\overbrace{\underbrace{t_n\cdots t_{p+1}}_{s}\overbrace{t_p\cdots t_1}^{p}}^{l}\ ] \tag{5.47}$$

Specifically, resource factors used are $(p,\ s,\ l) = (2,\ n,\ n-2)$. All threads of each block collaborate in the same system of equation, and each block works over different system of equation in batch mode.

5.4. Algorithm implementation strategies and optimizations

Following we will study some details in the design and implementation of the algorithms *ID-FFT* and *ID-TS* described in Section 5.3.

5.4.1. Implementation of operators

The efficiency of the proposed algorithm for *GPU* architectures lies on the fact that shuffle and digit reversal operators do not involve any explicit data movement within the shared memory or registers. Data permutations are implicitly performed when exchanging thread data in shared memory by using different write and read access patterns. The algorithm controls these access patterns in terms of offsets and strides, which are very fast and easy to compute.

For instance, digit reversal operator ρ in *ID-FFT* only implies modifications in the index-digits associated with the register private space p. The advantage is that it does not require moving 2^r complex data, it just changes the way data is accessed or written by each task. The *CUDA* compiler can optimize the ρ operator at compile-time replacing it by a simple register renaming when referencing the data after the digit reversal.

On the other hand, the shift operator Γ for *ID-FFT* and *ID-TS* implies modifications in the shared memory access pattern when threads exchange data after

each butterfly step. Figure 5.4 illustrates how the data exchange works. Instead of performing a redundant copy from a shared memory buffer to another shared memory buffer (see the $\Gamma^2_{6,3}$ permutation on the left), and then to private registers, the operator can be directly implemented from shared memory to registers. This way, each thread directly controls the offset and stride based on its thread identifier (see the equivalent implicit permutation on the right).

5.4.2. GPU memory access optimization examples

In a direct mapping of the operator strings, when the first operator the string requires consecutive data to be computed by a single thread, this kind of global memory access would lead to bad performance due to noncoalesced access.

In the case of the tridiagonal solver *ID-TS* which usually has a small radix, coalescence issues can be easily solved by combined loads using the *float2* or *float4* data types, which are quite efficient in the *CUDA* architecture. Each component of the equation is stored in a different vector, and each memory access reads up to four consecutive elements from up to four equations. This implementation has the advantage that it does not require any communication among the threads. Nevertheless, *ID-FFT* may use larger radix values, therefore a different strategy is recommended.

ID-FFT.V1 requires consecutive data elements by a single thread, however this pattern potentially results in uncoalesced memory access that wastes global memory bandwidth. Following we will describe a strategy to prevent this issue. For simplicity and a more compact representation of the examples, in the figures of this section it will be assumed that due to register constraints $p = 2$, the memory segment size for coalesced operation is 4 complex values (32 bytes), the warp size is composed by 4 threads, and the shared memory provides 4 independent memory banks.

ID-FFT.V1 is designed to process small signals, where each thread can compute a whole problem of the batch in a single computation step (in fact, if $n < p$ each thread can compute N/P problems at once). If balance between computation and memory operations is adequate a single warp can be used ($l = 2$), eliminating block synchronization overhead and taking advantage of the maximum amount of registers ($p = 2$) that can be reserved while avoiding local memory spilling. Even if

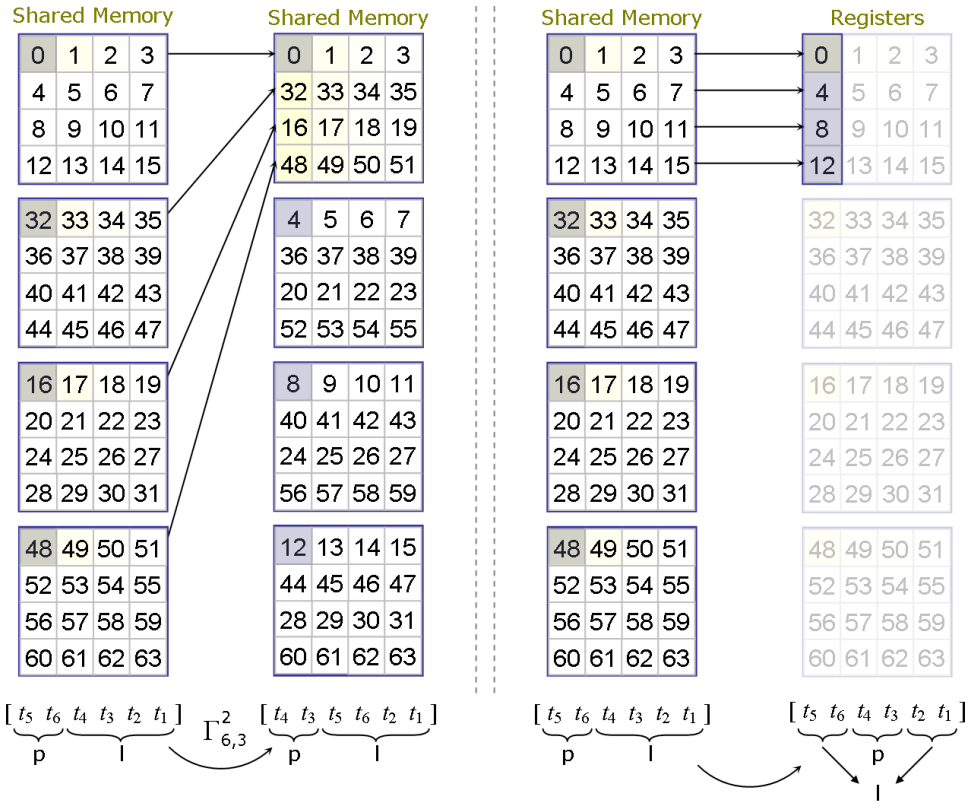


Figure 5.4: Data exchange, from shared memory directly to registers

there are multiple computation steps like in *ID-FFT.V2*, when using a single warp the blocks do not require any thread synchronizations to access shared memory, because the threads within a warp will not present race conditions on the *GPU*. Fence instructions may be still required to prevent the compiler from applying certain optimizations. As we will see in the results section, despite the small block size this solution will offer good performance thanks to the high number of simultaneous blocks scheduled by the hardware. The shared memory space ($s = 4$) is filled with batch problems.

In a straightforward implementation, the butterfly operator requires consecutive data and this kind of global memory access would lead to bad performance due to noncoalesced access. Figure 5.5 displays an illustrative example that presents this coalescence problem. Observe how the first thread T_1 reads the consecutive data $\{0, 1, 2, 3\}$ to its registers. The second thread T_2 would also read consecutive elements $\{a, b, c, d\}$ presenting coalescence issues, in fact this problem affects all the

threads. The problem with this kind of memory access is that it generates more transactions than needed. Furthermore, most of the data from each transaction is discarded, thus wasting memory bandwidth. Even in *GPU* architectures with global memory cache this access pattern will have overhead, reducing the maximum effective bandwidth.

Figure 5.5 also presents many shared memory bank conflicts. The *GPU* shared memory is divided in banks, and when several threads from the same warp try to simultaneously access different addresses within the same memory bank the involved accesses are serialized. In the example, when performing the data exchange in shared memory each thread will write to a different memory bank without any issue. For instance, T_1 will write data $\{0, 2, 1, 3\}$ to $Bank1$, T_2 will write data $\{a, c, b, d\}$ to $Bank2$, and so on. However, when reading for shared memory, all threads will try to access simultaneously the same bank generating a 4-way bank conflict. For instance, T_1 will access data in location 0 in $Bank1$, while at the same time T_2 will be trying to access data in location 1, which is also located in $Bank1$. Accordingly, the memory requests will be serialized and each thread will read from shared memory in a different cycle.

Figure 5.6 presents how to avoid the explained issues. At the beginning, instead of loading data to registers, the signal is directly copied to shared memory (which also prevents intermediate copies). Observe how initially threads access data in a perfectly coalesced pattern. For example, thread T_1 loads $\{0, a, x, \alpha\}$, starting in the first array location with a stride of 4 data locations (for coalescent access it should be a multiple of the segment size). Remember that to perform useful computation in the butterfly stage threads require consecutive data, for example, T_1 needs to operate on $\{0, 1, 2, 3\}$. To achieve this data distribution, the offset and stride of the shared memory read pattern will be different, implicitly performing a data exchange. Bank conflicts are easily avoided using padding, observe how T_1 writes its data ($\{0, a, x, \alpha\}$) with a stride of $banks + 1 = 5$ data locations to banks $\{0, 1, 2, 3\}$. Next, it reads data elements $\{0, 1, 2, 3\}$ consecutively from banks $\{0, 1, 2, 3\}$. After the exchange, each thread works on the assigned problem using private registers. Next, the data exchange is reversed using shared memory and data is written to global memory using a coalescent access pattern like at the beginning of the process. For example, thread T_1 writes $\{0, a, x, \alpha\}$ to locations $\{0, 4, 8, 12\}$ of

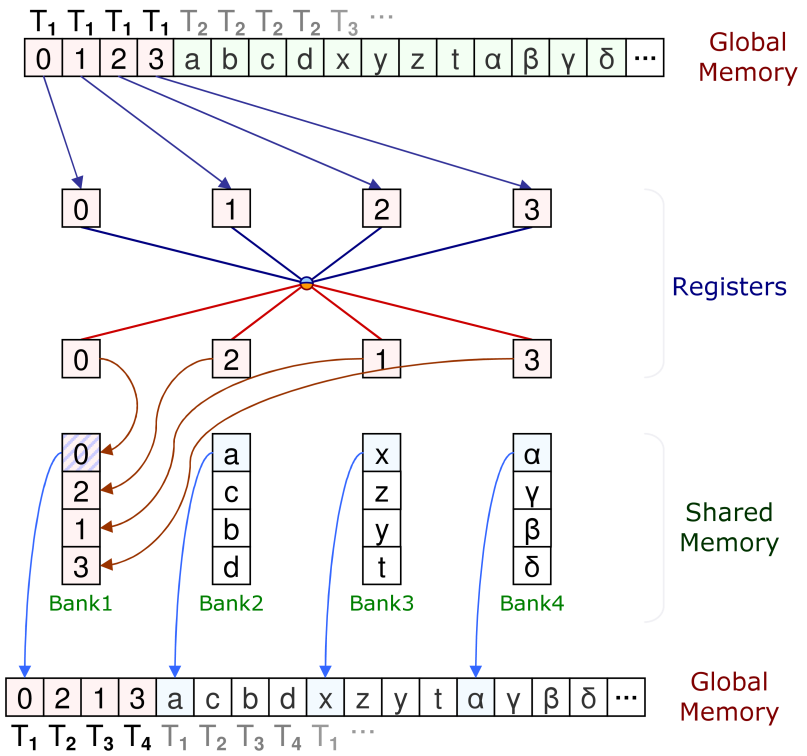


Figure 5.5: Problematic global and shared memory access pattern for $n = l = p = r = 2$, $s = 4$

global memory.

In the remaining *1D-FFT* algorithms, the threads read data from global memory with a coalescent access pattern thanks to the constant stride multiple of the segment size. After loading the signal data, each thread can directly compute a radix *FFT* step using data in registers without requiring any permutation with other threads.

5.4.3. Obtaining the code from the operator strings

Once the expression are generated, obtaining the code is quite straightforward. The perfect shuffle and the perfect unshuffle operators are implemented using different strides and offsets when transferring data between shared memory and registers or global memory and registers. The bit-reverse and the butterfly operators are obtained from their definition. The implementation does an extensive usage of template functions to create several optimized versions for each function depending on

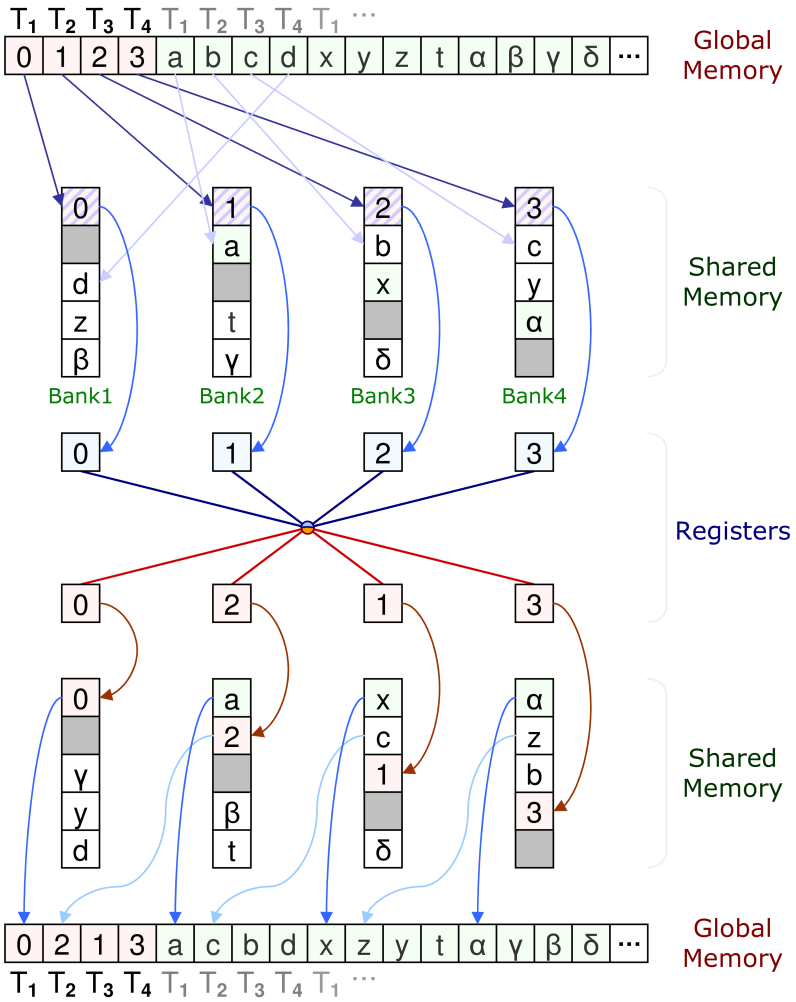


Figure 5.6: Coalescent and bank conflict-free access pattern for $n = l = p = r = 2$, $s = 4$

the problem size. Most of the function calls, register loops and redundant move operations will be fully optimized at compile-time.

Figure 5.7 presents a simplified version of the pseudocode used to implement the 3D case Equation (5.22). At the beginning of the algorithm (lines 1 to 3) the different stride and offset are obtained based on the thread and block identifier. These values are obtained from the corresponding operator string, although several factors like padding to prevent shared memory bank conflicts have to be considered. Next, in line 6 data is loaded from global memory to registers, which maps to the first shuffle $\Gamma^6_{s,1}$. Each call to the *radix* function in lines 10, 19 and 28 compute one radix

```
01:   // Compute stride and offsets based on thread and block identifier
02:   // Must consider block dimensions, padding data and mixed-radix stage
03:   ...
04:
05:   // Load data from global memory to registers
06:   copy<DIMY>(registers, glbMem + glbOffset, glbStride);
07:
08:   // Compute the first radix stage
09:   if(DIR < 0) scale<DIMY, N>(reg);
10:   radix<DIMY, DIR>(reg);
11:
12:   // Exchange register data using shared memory
13:   shmExchange<DIMY>(registers,
14:         shMem + shOff_w1, shStr_w1,
15:         shMem + shOff_r2, shStr_r2);
16:
17:   // Obtain the twiddle and compute the second radix stage (mixed-radix)
18:   float ang2 = getAngle<DIR, DIMY, MIXRAD>(locY);
19:   radix<DIMZ, MIXRAD, DIR>(reg, ang2);
20:
21:   // Exchange register data using shared memory
22:   shmExchange<DIMX>(registers,
23:         shMem + shOff_w2, shStr_w2,
24:         shMem + shOff_r3, shStr_r3);
25:
26:   // Obtain the twiddle and compute the third radix stage
27:   float ang3 = getAngle<DIR, DIMY * MIXRAD, DIMX>(loc);
28:   radix<DIMX, DIR>(reg, ang3);
29:
30:   // Write data from registers to global memory
31:   copy<DIMX>(glbMem + glbOffset, glbStride, registers);
```

Figure 5.7: Example of pseudocode used for the 3D version of the algorithm

stage, composed by a *bit-reversal* and a *butterfly* operator which maps to $\rho_{s,s-1}$ and B^2_{s-1}, respectively. The shared memory exchanges are performed by the function *shmExchange* in lines 13 and 22. The first exchange corresponds to $\Gamma^2_{s,s-3}$ and the second to $\Gamma^2_{s,s-3}$. Finally, the result is written to global memory in line 31, which maps to $\sigma^6_{s,1}$ and reverses the access pattern used in line 6. Regarding the function *getAngle* used in lines 18 and 27, they just compute the corresponding twiddle factor for each thread based on the current stage and the effect of the previous exchanges. Observe how the operator string was easily mapped to the code, providing flexibility and a compact representation.

5.5. Experimental results

All the tests were run in single precision using batch execution to compute several problems each time. The size of the batch depends on the input size and is given by the expression $batch = 2^{24}/N$. There are no data transfers to *CPU* during the benchmarks to prevent interactions with other factors. The performance of the experiments for the *FFT* is measured using the *GFLOP* metric given by Equation (1.4). In the case of tridiagonal system resolution the performance is expressed in million rows processed per second as given by Expression 1.25.

The test platforms used in our experiments are described in Table 5.2. The kernels were executed setting the *cudaFuncCachePreferShared* cache configuration flag, so it is possible to use up to 48 *KB* of shared memory, but only 16 *KB* of $L1$ cache are available. The *GeForce Titan* is based on the *Kepler* architecture, and its configurable bank size was set to four bytes (*cudaSharedMemBankSizeFourByte*). Using eight bytes resulted in slightly slower performance for many cases. The algorithms were also designed to take advantage of the *Kepler's* read-only data cache, which slightly improves global memory effective read bandwidth. Different driver versions were tested with little impact on performance, however in some cases older versions of the *CUDA SDK* obtained around 2.4% better performance for the *CUFFT* on the *Fermi* architecture.

5.5.1. Complex FFT

Table 5.3 presents the performance results for complex signals and the kernel profile analysis of the different algorithms for Platform 1, highlighting the best result for each signal size.

For small problems, even with only 1/6 of the maximum *SM* occupancy, the *ID-FFT.V1* algorithm offers very good results, up to 209.64 *GFlops* for $n = 4$. Due to hardware limitations, if 32 or more complex values are simultaneously used by the same thread, there would be a substantial performance drop due to register spilling, being even slower than the $n = 2$ case. The *ID-FFT.V2* algorithm also has 1/6 of the *SM* occupancy and starts where the *ID-FFT.V1* algorithm left off, scaling perfectly as the signal size increases, up to 411.26 *GFlops* for $n = 8$. Nonetheless, for the last

Table 5.2: Description of the test platforms

	Platform 1	Platform 2
CPU	Core i7 2600	C2 Duo E8400
Memory	8 GB DDR3 1333	8 GB DDR3 1333
OS	Win 7 x64 SP1	WinXP x64 SP2
Compiler	MSVC 2010 SP1	MSVC 2010 SP1
GPU	GeForce 580	GeForce Titan
Driver	v310.90, SDK 5.0	v320.17, SDK 5.0
GPU Peak	1581 GFLOPS	4500 GFLOPS
GPU Bw.	192.4 GB/s	288.4 GB/s

two cases ($n = 7$ and $n = 8$) the *ID-FFT.V3* version provides better computational balance and improved parallelism (1/3 of the *SM* occupancy), achieving 418.59 *GFlops* for $n = 8$ with radix-8. The last signal size that can be processed by the *ID-FFT.V3* algorithm using radix-8 is $n = 9$, obtaining 471.97 *GFlops*. For bigger signals the *ID-FFT.V4* algorithm is used and the amount of available shared memory becomes a limiting factor (for $s = 12$, 17408 bytes are required), hindering the *GPU SM* parallelism. Nonetheless, the *ID-FFT.V4* algorithm reaches 492.43 *GFlops* for $n = 12$. The efficiency drop for $n \geq 10$ confirms that extending the parameters beyond the recommended value has a noticeable effect diminishing the performance, and other alternative parallelization strategies should also be studied, however these problems are beyond the scope of this work.

Regarding the kernel profiler analysis, Table 5.3 has detailed information about the block size, the number of registers, the amount of shared memory (in bytes), the global memory bandwidth (in GB/s), the instruction replay rate and the *SM* occupancy. Notice that the maximum number of registers for all cases is 52 and there will be no local memory spilling. This is confirmed by the local memory replay rate, which was not included in the table because its value is always 0. To take advantage of the maximum *SM* block parallelism, the shared memory is always below 6 *KB* except for the last group, *ID-FFT.V4*. Observe the efficient access to global memory, most configurations use more than 84% of the total memory bandwidth. The instruction replay rate is fairly low, being the first test in each group the worst obtained value because one of the steps performs fewer computations due to the mixed-radix. Even using batch processing within that step there are not enough independent computations to hide memory or instruction latency, specially considering the small block sizes used by the algorithm. Last, observe that due to

Table 5.3: Complex FFT kernel performance and profiler analysis for the different versions (Platform 1)

Alg	n	p	GFLOPS	Threads /block	Reg	Shared memory	Gl. mem. bandw.	Instruction replay (%)	Occupancy (%)
ID-FFT.V1	2	4	***104.75***	32	52	2112	162.50	4.4	∼16
	3	4	***157.09***		50		162.42	3.2	
	4	4	***209.64***		49		162.53	2.2	
ID-FFT.V2	5	4	***261.82***	32	42	2112	162.56	1.7	∼16
	6	4	***312.78***		45		162.11	0.9	
	7	4	363.29		47		161.40	0.5	
	8	4	411.26		44		159.95	0.3	
ID-FFT.V3	7	3	***365.30***	64	28	2304	162.45	1.2	∼33
	8	3	***418.59***		29		162.45	0.8	
	9	3	***471.97***		30		163.05	0.4	
ID-FFT.V4	9	4	413.12	256	46	17408	143.01	5.4	∼33
	10	4	***448.35***		45		139.19	5.0	
	11	4	***475.01***		46		134.18	4.6	
	12	4	***492.43***		45		126.26	1.7	

the small block size (see *threads/block* column) the *SM* occupancy is low, 1/3 or even 1/6 depending on the test. Nevertheless, thanks to the *SM* block parallelism and thorough resource allocation, the algorithms are able to use the hardware efficiently. A similar profiler analysis was made for *NVIDIA's CUFFT 5.0*. In contrast to the algorithm proposed in this work, the profiling of *NVIDIA*'s implementation reveals bank conflicts for some of the problem sizes (up to 17.5% for $n = 6$). In most cases the kernels are executed with 128 or 256 threads per block, hence the *GPU* occupancy (typically around 60%) is higher compared to our implementation. Nevertheless, the lower computational load per thread usually increases the instruction replay rate resulting in lower performance.

Figure 5.8 shows a global overview of the complex *FFT* performance. The results are compared to *NVIDIA's CUFFT 5.0*. Also, the *NukadaFFT v1.0-20130518.15* was executed on Platform 1 and was added to offer a second comparison with another recent *GPU* implementation. The *Spiral 6.0* library was executed on Platform 1 and is also included to provide a reference point. As can be observed, *GPU* based solutions offer a clear advantage over *CPU* implementations (in this case *Spiral*, but other solutions like the *Intel IPP* library [49] offer very similar performance), resulting about ten times faster. For signal sizes between $N = 4$ and $N = 1024$, our algorithm is on average 5.9% faster than the *NVIDIA* library on Platform 1, and

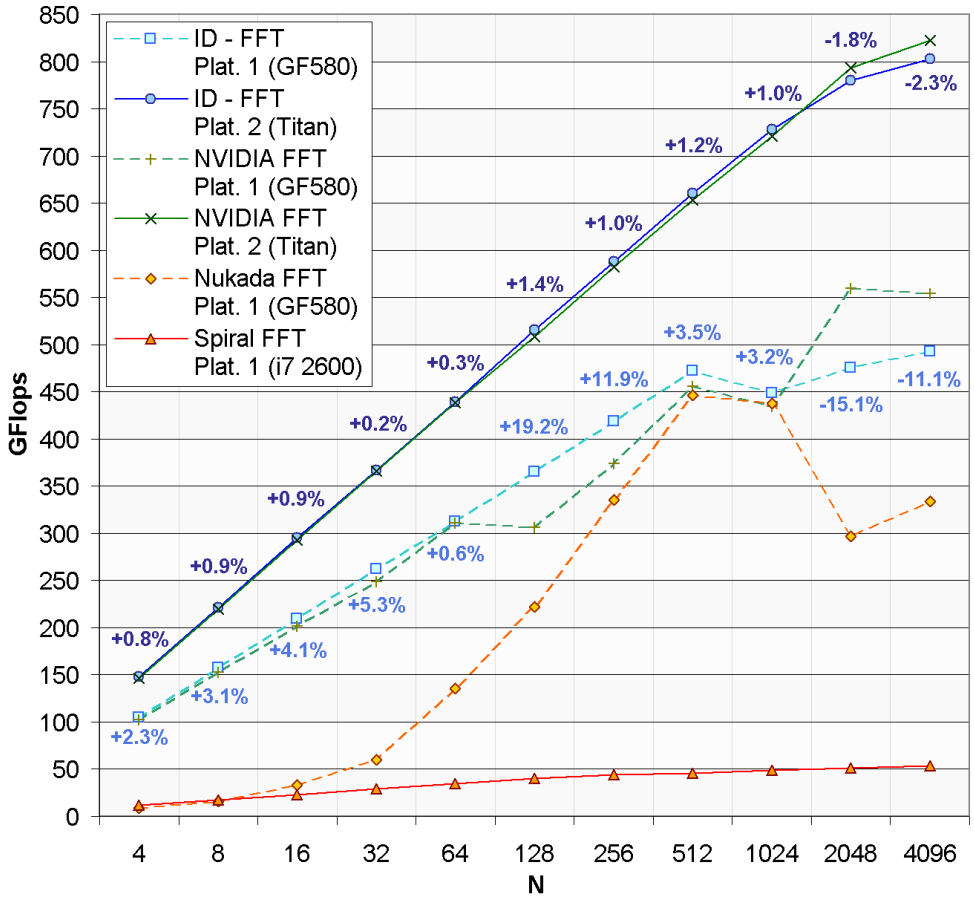


Figure 5.8: Performance comparison of Complex ID-FFT proposal

around 0.8% faster on Platform 2. It also obtained quite competitive results for signal sizes above $N = 1024$ (these larger problems will be addressed in a future work). In the best case, for Platform 1 and N=128 we achieve a 19.2% improvement, while for Platform 2 and $N = 128$ we achieve a 1.4% improvement. The pronounced performance drop experienced by the *CUFFT* for $N = 128$ and $N = 256$ on Platform 1 seems to be a regression in the *CUDA SDK v5.0*, because in previous releases it resulted in a smoother line. The generated code in the latest release is probably more tuned for the *Kepler* architecture. Regarding the difference between the two *GPU* architectures, the results of our library shows identical tendency for most problem sizes. Thanks to the efficient tuning of the mapping vector and the algorithm design based on operators string, our complex *FFT* implementation offers near perfect scaling with signal size up to $N = 512$. The work is well distributed among the threads,

and the latency of read operations is perfectly hidden by arithmetic instructions. Nevertheless, for larger problems with $N \geq 512$ the best performance was obtained using the *Titan GPU* (up to 802.9 *GFlops* for $N = 4096$), followed by the *GeForce 580* with 492.4 *GFlops*. Judging from the straight slope given by the evolution of the graph scaling with the problem size, performance may be reaching some kind of asymptotic limit imposed by the architecture. Observing the best results from Table 5.3, the performance grows while the global memory bandwidth remains between 161 and 163 *GB/s*, therefore the theoretical memory bandwidth seems to be the main limiting factor. Considering that the maximum memory bandwidth of the *GeForce 580* is 192.4 *GB/s* the algorithm is achieving almost 85% bandwidth efficiency, which is quite impressive considering how difficult is to achieve nearly peak efficiency on *GPUs* for real-world scenarios. The *Titan* has nearly three times the raw computing power, but also seems limited by the total memory bandwidth, which peaks at 288 *GB/s* achieving approximately a 82% bandwidth efficiency.

5.5.2. Tridiagonal Equation System

Regarding the performance of the tridiagonal system resolution algorithm, Table 5.4 presents the profiler analysis for the different versions and problem sizes. The performance is expressed in million rows processed per second ($Mrow/s$), and the best solution for each size is highlighted using bold text.

In practice, as it is shown in the profiling of Table 5.4, the actual register requirement per thread is a bit lower than estimated in Section 5.2.2, but not enough to increase the radix size. Notice that, as previously commented, the resulting 2^n left equations $E^{t-1}(q \cdot 2^{t-1})$ acquire the same value and the 2^n right equations $E^{t-1}((q+1)2^{t-1}-1)$ also have the same value. This property means that there will be only N unique equations per signal, therefore during the data exchanges just the center equation $E^{t-1}(i)$, will be stored in shared memory. This also means that instead of wasting registers to store both equations, this data can be directly accessed from shared memory when performing the butterfly operator. As long as dynamic register addressing is avoided (which leads to local memory spilling), the programmer can use high level code to implement the operators because the $NVCC$ compiler does a good job reducing temporal variables and intermediate copy operations.

Table 5.4: Tridiagonal system performance (Platform 2)

Alg	n	r	Mrow/s	Threads /block	Reg	Shared mem.	Occup. (%)
ID-TS.W1	2	2	9381.2	64	28	0	47.1
	2	3	5253.3	64	42	0	45.7
	3	3	***8013.0***	32	46	0	24.5
ID-TS.W2	2	1	***9643.5***	128	26	4096	70.9
	3	1	7588.1	128	26	4096	71.8
		2	6653.8	64	43	4096	36.1
	4	1	7691.0	128	26	4096	72.2
		2	***7940.3***	64	43	4096	36.0
		3	3894.2	32	68	4096	18.5
	5	1	***7130.6***	128	26	4096	72.6
		2	7032.9	64	53	4096	36.3
		3	5678.4	32	66	4096	18.4
	6	1	6675.4	128	26	4096	72.9
		2	***7561.7***	64	54	4096	36.3
		3	6808.0	32	66	4096	18.3
	7	1	6021.1	128	26	4096	73.1
		2	***6134.8***	64	50	4096	36.5
		3	3934.6	32	82	4096	18.5
	8	1	5478.3	128	26	4096	73.2
		2	***6770.5***	64	50	4096	36.5
		3	5271.4	32	82	4096	18.4
ID-TS.W3	9	1	4511.4	256	26	8192	73.3
		2	***5337.2***	128	54	8192	36.5
		3	4921.5	64	79	8192	18.4
	10	1	3782.3	512	29	16384	73.6
		2	***5137.0***	256	53	16384	36.5
		3	3437.9	128	79	16384	18.4
	11	1	2398.5	1024	29	32768	49.6
		2	***2948.8***	512	54	32768	24.8
		3	2926.6	256	80	32768	12.4

As computed in Section 5.2.2 the optimal shared memory reserved for each block is 4096 bytes. Like in the case of the *FFT*, large problem sizes require to extend s beyond the recommended limit. Processing problems where $N > 2048$ would require a different approach, like a multi-kernel algorithm, which is beyond the scope of this work.

On the other hand, in order to prove the advantage of our proposal Table 5.4 also displays the results with non-optimal resource factors (column p). For example, for *ID-TS.W2* the results with $p = 1$ and $p = 3$ obtain a lower performance than optimal resource factor $p = 2$. The option $p = 1$ implies a high occupancy but a

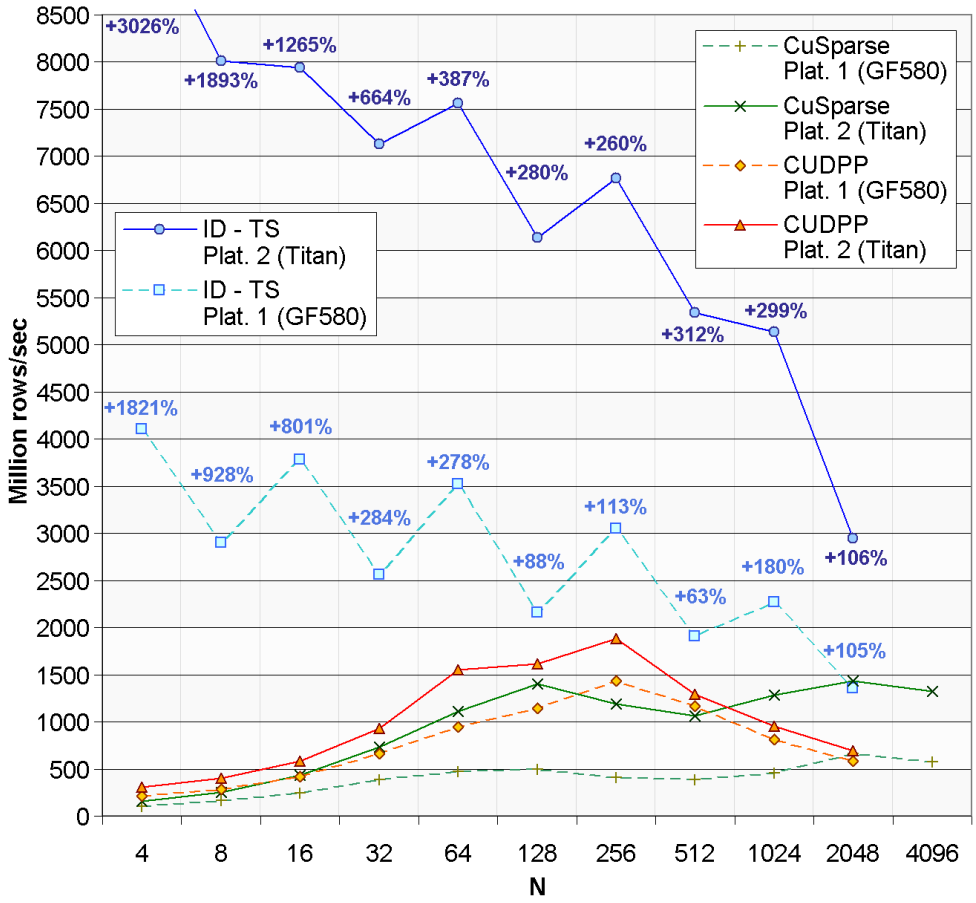


Figure 5.9: Comparison of performance of *ID-TS* proposal

low number of operations per thread. Meanwhile, the resource factor $p = 3$ does not allow to maintain the minimum of occupancy.

Figure 5.9 compares our best results to *CUDPP* [142] and *NVIDIA's CuSparse* library. The percentages near the lines represent the performance increase over the fastest of the other two solutions. As it can be observed, for batches of small equation our library is much faster, for instance when solving systems of 64 equations it obtains a 387% improvement in the case of the *Platform 2* and a 278% improvement in the case of *Platform 1*. Even for $N = 2048$ which is the worst case scenario, our algorithm provides over 100% performance improvement. In fact, the difference is so large that our algorithm running on *Platform 1* readily outperforms the other libraries running on *Platform 2*.

In summary, our methodology based on operators string manipulation directed by the study of hardware features achieves good performance, independently of the selected GPU architecture or Index-digit algorithm.

Chapter 6

Conclusions and Future Work

One of the main goals of this thesis was to propose a generalized methodology to solve many types of divide-and-conquer problems which can be formulated through the Index-Digit representation. Our aim was to be able to produce optimized implementations with competitive performance, assisting the programmer in the design of new libraries through the use of our methodology. Next, the main conclusions are presented.

At the beginning we have studied the parallelization of two applications using a manual approach that relied on hand-tuning and well-known optimization techniques. We implemented the *FFT* operation on a *GPU* using the *Brook+* programming language and a shallow water simulator using *CUDA*. Even though the inability to properly exploit shared memory in *Brook+* and the restrictions of its stream programming model have been experienced, it has been proved that the *Fourier* transform can be successfully mapped to the streaming model. We described several optimization strategies, like stage blending and fusion using a recomputation approach and scheduling layouts. This allowed us to achieve up to 180 *GFlops* for small *FFTs*, and even for the worst case scenario, the performance is above 50 *GFlops*, outperforming the *CPU* implementations.

Regarding the shallow water simulator the described model is able to handle wet-dry zones as well as the transport of inert substances such as a pollutant on am esturary. Our aim was to develop a *GPU* implementation allowing quick studies of the behavior of a pollutant in a realistic environment under different hypothesis,

and fast enough to take all the required decisions to deal with it. We started from a naive implementation based on a recomputation solution in which redundant computations and many accesses to global memory were performed. Following, an optimized version that significantly reduces the number of computations was implemented. A ghost cell decoupling strategy which exploits shared memory and textures has been implemented, achieving an average speedup of 19% with respect to the naive implementation for the five largest problem sizes, and a speedup of 24x with respect to a parallel execution on a recent multicore *CPU*. Such performance, measured using a realistic problem, enabled the calculation of solutions not only in real-time, but orders of magnitude faster than the simulated time.

Following, we have analyzed several performance aspects of the *GPU* architecture, like the influence of the memory access pattern, the texture cache, the shared memory performance or the impact of register pressure. The results of this analysis will be useful for the tuning of other future applications. We used an *FFT* kernel to benchmark the different *GPUs*. The *FFT* offers good balance between computational cost and memory bandwidth, thus the results can be extrapolated to other parallel algorithms with regular structure.

We studied the performance impact of uncoalesced memory access, as well as the influence of texture cache or the global memory cache introduced in the *Fermi* architecture. Access coalescence resulted a critical factor, even when performing cached memory access. Texture cache proved to be useful to improve effective memory bandwidth in some scenarios. We also analyzed the effect of the configurable *L1* cache for different execution configurations and the cost of enabling *ECC* for error correction. Regarding the thread data storage, shared memory provided less effective bandwidth than registers even in the absence of bank conflicts. For the largest signal size, the high register pressure generated local memory spilling, which caused a significant performance reduction.

Although the primary objective of the analysis was to study the architecture and several aspects of the memory hierarchy, the resulting implementation offered competitive performance for batches of small problems. For example, we were able to obtain up to 185 *GFlops* for the *GeForce 480* and up to 154 *GFlops* for the *Tesla S2050* (in both cases the same performance as *NVIDIA's CUFFT*).

Next, we used the knowledge gathered by the research conducted so far to design a *CUDA* library for several well-known butterfly algorithms, namely the complex and real version of the *FFT*, the *DCT* and the *Hartley* signal transform algorithms, as well as a tridiagonal equation system solver. The resulting implementations were tested on two different platforms, comparing the efficiency with other state of the art libraries. Our approach provided excellent performance in many cases, like an average 60.4% advantage in the real *FFT* or over 200% advantage for tridiagonal equation systems, and as far as we know it is also the fastest general purpose implementation of the *DCT* and the *Hartley* transforms. Regarding the complex *FFT*, *BPLG* still offers competitive results with respect to the *CUFFT*, especially considering that our primary focus on this work has been to provide a library flexible enough to improve *GPU* programmability.

One of the most interesting features of the proposed library is the modular design based on small building blocks. The building blocks of the algorithms where implemented using high-level *C++* templates, with emphasis on flexibility and configurability. The library parameters were adjusted to obtain good efficiency on two recent *GPU* architectures.

Finally, we presented a tuning methodology which allowed us to create efficient and flexible *GPU* algorithms. Our methodology addresses the parallelization of divide and conquer problems which can be formulated through the Index-Digit representation. This methodology is based on 2-stages: resource analysis and operator string manipulation. A small number of performance factors were used to tune the operator strings of two algorithms, *ID-FFT* for the complex *Fourier* transform and *ID-TS* for the resolution of tridiagonal systems.

In summary, our methodology based on operators string manipulation directed by the study of hardware features achieves good performance, independently of the selected *GPU* architecture or Index-Digit algorithm.

The performance of the resulting proposals was analyzed using two different architectures, obtaining very good efficiency over other well-known *GPU* libraries. Compared to *NVIDIA's CUFFT*, *ID-FFT* is up to 19.2% faster in the case of the *GeForce 580* and 1.4% faster in the case the *GeForce Titan*, being *ID-TS* up to 1821% (*GeForce 580*) and 3026% (*GeForce Titan*) faster respectively than other

state of art implementations like *CuSparse* or *CUDPP*.

One question remains regarding the performance improvement obtained thanks to the proposed two-stage methodology and the algorithm tuning. Figure 6.1 compares the final performance results of *BPLG-cFFT* (from Chapter 4) and *ID-FFT* (from Chapter 5). Compared to the *BPLG* library proposed in the fourth chapter the complex *FFT* algorithm experiences a significant speedup for most signal sizes. In the case of the *GeForce Titan* there is on average an 11.3% performance improvement, however in the case of the *GeForce 580* the tuning advantage is even larger, around a 22.8% improvement. The most relevant gains are concentrated in the small signal sizes ($8 \leq N \leq 32$) and also the larger signals ($1024 \leq N \leq 4096$), precisely those areas where the performance was more distant from the ideal straight line given by the theoretical memory bandwidht.

Figure 6.2 repeats a similar analysis, but comparing *BPLG-TS* (from Chapter 4) and *ID-TS* (from Chapter 5). As it can be observed, for the tridiagonal solver the performance advantage in the case of the *GeForce 580* is smaller, nonetheless on average there is a 3.6% performance improvement thanks to the algorithm tuning. The *GeForce Titan* experiences more benefit, on average a 11.5% performance gain. In the tridiagonal case the performance tuning benefit seems more evenly distributed, with the exception of the smaller problems which can be efficiently executed in few stages, thus there is less room for improvement, and the largest size, where the shared memory becomes a limiting factor reducing the parallelism. Due to the amount of data required by the equation triads, the tridiagonal algorithm uses a smaller radix size, hence larger blocks and more stages are required for the same N, leading to a higher number of synchronizations which becomes a performance limiting factor.

6.1. Future work

There are many interesting topics as future work, but considering the good results one of the most promising topics is to apply the presented methodology to address the design of other signal transforms or Index-Digit algorithms. Moreover, the automatization of the methodology could further reduce the programming effort,

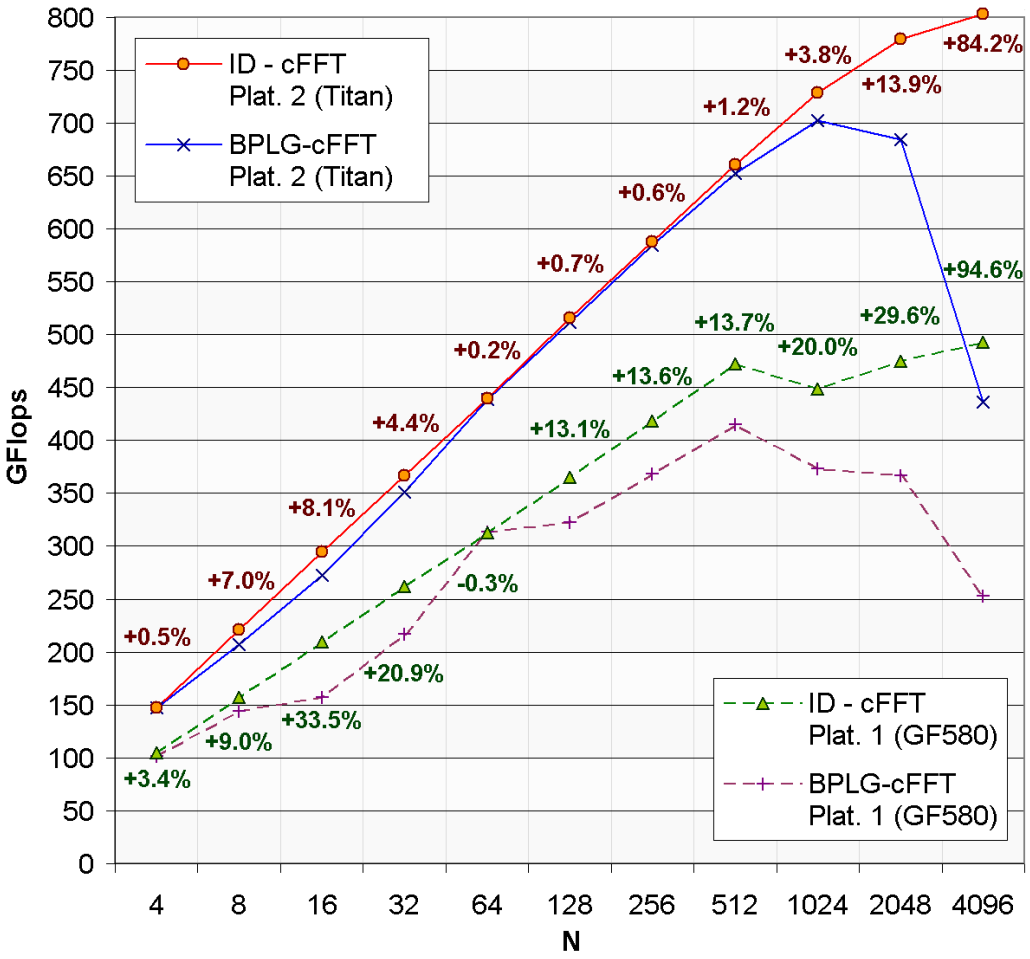


Figure 6.1: Performance comparison: BPLG-cFFT vs ID-FFT

enabling the automatic generation of the kernel skeletons from the operator strings. Another interesting topic is to extend the methodology for other generic manycore architectures supported by *OpenCL*.

In the case of the signal processing algorithms, we also plan to extend them for larger signals and enable support for 2D transforms. Efficient processing of large problems requires an in-depth study of the different implementation strategies to allow appropriate work distribution and collaboration among different blocks. Provisional tests show positive results, although further analysis is required to propose a general solution.

Larger problems may cause numerical instability or result in lower than expected

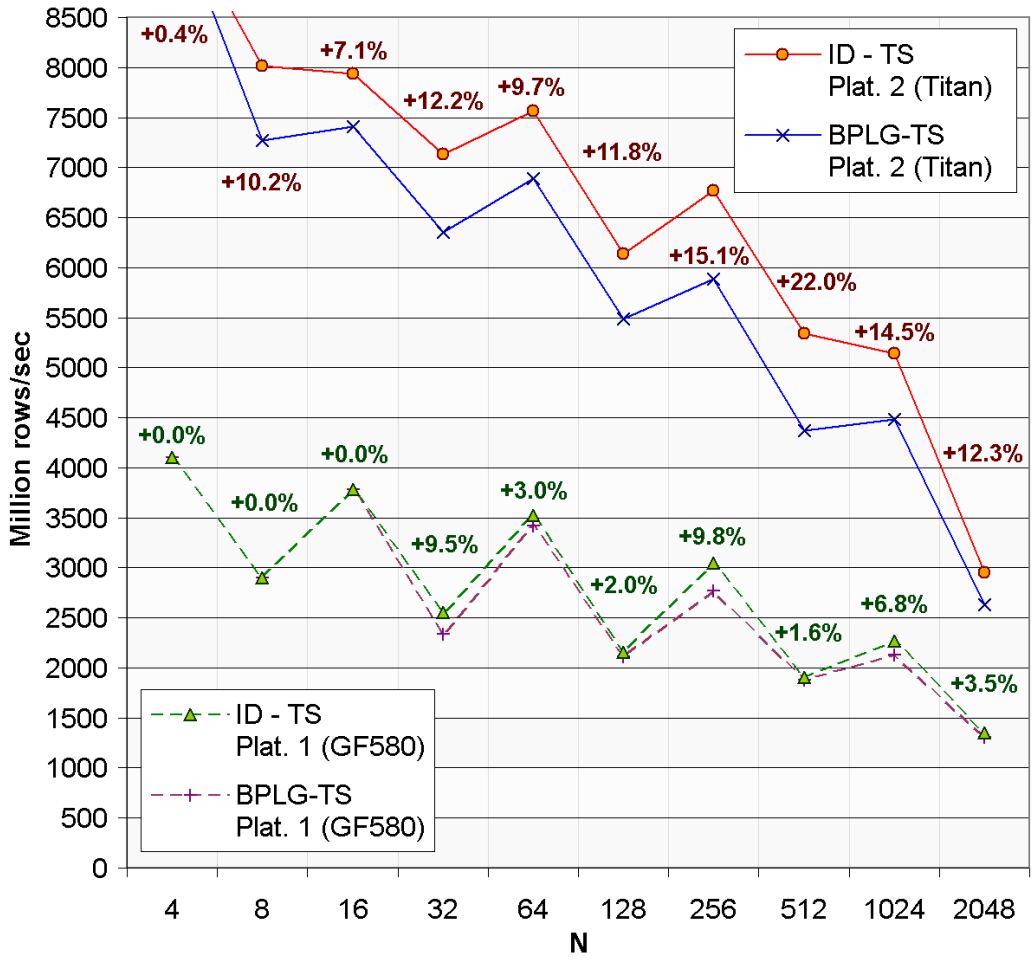


Figure 6.2: Performance comparison: BPLG-TS vs ID-TS

precision, thus, a related topic is the development of double precision versions of the algorithms. This approach can be further refined: taking advantage of the template-based approach it is possible to create type-independent algorithms (for instance to enable quadruple precision arithmetic). The performance tuning of this kind of kernels is a complex subject due to the variability in the size of the data type and the operator cost.

6.2. Publications from the Thesis

Journal Papers (5)

- Jacobo Lobeiras, Margarita Amor, Ramón Doallo. BPLG: A Tuned Butterfly Processing Library for GPU Architectures. International Journal of Parallel Programming (IJPP). (Accepted for publication).
  JCR Impact Factor: 0.404, Q4 in Computer Science, Theory & Methods.

- Jacobo Lobeiras, Margarita Amor, Ramón Doallo. FFT Algorithm Design for GPU Architectures Based on Operator String Representation. IEEE Transactions on Parallel and Distributed Systems (TDPS). (Submitted, under second review).

- Jacobo Lobeiras, Moisés Vinãs, Margarita Amor, Basilio B. Fraguela, Manuel C. Arenaz, Jose A. García, Manuel J. Castro. Parallelization of Shallow Water Simulations on Current Multi-Threaded Systems. International Journal of High Performance Computing Applications (IJHPCA). Vol. 27, no. 4, pages 493-512. November 2013.
  JCR Impact Factor: 1.295, Q2 in Computer Science, Theory & Methods.
  DOI: 10.1177/1094342012464800.

- Moisés Viñas, Jacobo Lobeiras, Basilio B. Fraguela, Manuel C. Arenaz, Margarita Amor, Jose A. García, Manuel J. Castro, Ramón Doallo. A Multi-GPU Shallow-Water Simulation with Transport of Contaminants. Concurrency and Computation: Practice and Experience. Vol. 25, issue 8, pages 1153-1169. June 2013.
  JCR Impact Factor: 0.845, Q2 in Computer Science, Theory & Methods.
  DOI: 10.1002/cpe.2917.

- Jacobo Lobeiras, Margarita Amor, Ramón Doallo. Influence of Memory Access Patterns to Small-Scale FFT Performance. The Journal of Supercomputing. Vol. 64, issue 1, pages 120-131. April 2013.
  JCR Impact Factor: 0.917, Q2 in Computer Science, Theory & Methods.
  DOI: 10.1007/s11227-012-0807-5

International Conferences (6)

- Jacobo Lobeiras, Margarita Amor, Ramón Doallo. SPLG: A Tuned Signal Processing Library for GPU Architectures. 25th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2013), pages 184-191. October 2013.
  DOI: 10.1109/SBAC-PAD.2013.30

- Jacobo Lobeiras, Margarita Amor, Ramón Doallo. GPU Performance Analysis using the FFT. Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES), pages 215-218. July 2011.
  ISBN: 978-90-382-1798-7

- Moisés Viñas, Jacobo Lobeiras, Basilio B. Fraguela, Manuel C. Arenaz, Margarita Amor, Ramón Doallo. Simulation of Pollutant Transport in Shallow Water on a CUDA Architecture. II Workshop on Exploitation of Hardware Accelerators, WEHA 2011, held in conjunction with the 2011 International Conference on High Performance Computing and Simulations (HPCS 2011), pages 664-670. July 2011.
  DOI: 10.1109/HPCSim.2011.5999890

- Jacobo Lobeiras, Margarita Amor, Ramón Doallo. Performance Evaluation of GPU Memory Hierarchy using the FFT. International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE), pages 750-791. June 2011.
  ISBN: 978-84-614-6167-7

- Jacobo Lobeiras, Margarita Amor, Ramón Doallo. FFT Implementation on a Streaming Architecture. Euromicro International Conf. on Parallel, Distributed and Network-Based Processing (PDP), pages 119-126. February 2011.
  DOI: 10.1109/PDP.2011.31

- Jacobo Lobeiras, Margarita Amor, Manuel C. Arenaz, Basilio B. Fraguela. Streaming-Oriented Parallelization of Domain-Independent Irregular Kernels. UnConventional High Performance Computing (UCHPC), Euro-Par 2010 Parallel Processing Workshops, pages 381-388. August 2010.
  DOI: 10.1007/978-3-642-21878-1_47

National Conferences (2)

- Jacobo Lobeiras, Margarita Amor, Manuel C. Arenaz, Basilio B. Fraguela. Análisis del Rendimiento de Núcleos Computacionales sobre una GPU con Brook+. Workshop de Aplicaciones de Nuevas Arquitecturas de Consumo y Altas Prestaciones (ANACAP). Digital publication, pages 1-10. November 2009.
  ISBN: 978-84-692-7320-3

- Jacobo Lobeiras, Margarita Amor, Manuel C. Arenaz, Basilio B. Fraguela. Simulación de Aguas poco Profundas en una GPU mediante Brook+. XX Jornadas de Paralelismo, pages 327-332. September 2009.
  ISBN: 84-9749-346-8

Bibliography

[1] A. Davidson, Y. Zhang and J.D. Owens. An Auto-tuned Method for Solving Large Tridiagonal Systems on the GPU. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 956–965. IEEE Comp. Society, 2011. Cited in p. 156

[2] A. Greenbaum. *Iterative Methods for Solving Linear Systems*. Society for Industrial and Applied Mathematics (SIAM), 1997. Cited in p. 58

[3] A. Kurganov and G. Petrova. A Second-Order Well-Balanced Positivity Preserving Central-Upwind Scheme for the Saint-Venant System. *Commun. Math. Sci.*, 5(1):133–160, 2007. Cited in p. 91

[4] A. Nukada and S. Matsuoka. Auto-tuning 3-D FFT Library for CUDA GPUs. In *Proc. of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, pages 1–10, 2009. Cited in p. 155

[5] A.B. Watson. Image Compression Using the Discrete Cosine Transform. *Mathematica Journal*, 4:81–88, 1994. Cited in p. 16, 57

[6] A.D. Poularikas. *The Transforms and Applications Handbook*. Electrical Engineering Handbook. CRC Press, 2010. Cited in p. 16, 55

[7] AMD. *AMD Stream Computing User Guide*, 2009. v1.4.1. Cited in p. 17, 44, 63

[8] AMD. *AMD Math Libraries, OpenCL Fast Fourier Transform (clAmdFft)*, 2012. Cited in p. 54

[9] A.R. Brodtkorb, M.L. Sætra and M. Altinakar. Efficient Shallow Water Simulations on GPUs: Implementation, Visualization, Verification and Validation. *Computer and Fluids*, 55:1–12, 2012. Cited in p. 82, 90

[10] B. Chapman, G. Jost and R. der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation).* The MIT Press, 2007. Cited in p. 14, 33

[11] B. Sander. *Bolt: A C++ Templater Library for HSA*. Presented in AMD Fusion Developer Summit '12, 2012. Cited in p. 18, 64

[12] B. Wang, M. Álvarez-Mesa, C. Ching and B. Juurlink. An Optimized Parallel IDCT on Graphics Processing Units. In *18th International Conference on Parallel Processing Workshops (EuroPar '12)*, pages 155–164. Springer-Verlag, 2013. Cited in p. 57

[13] C. Boyd and M. Schmit. The Direct3D11 Compute Shader. Microsoft WinHEC, 2008. Cited in p. 44

[14] C.-T. Ho and S.L. Johnsson. Optimizing Tridiagonal Solvers for Alternating Direction Methods on Boolean Cube Multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 11(3):563–592, 1990. Cited in p. 16, 58

[15] C.G. Kim and Y.S. Choi. A High Performance Parallel DCT with OpenCL on Heterogeneous Computing Environment. *Multimedia Tools and Applications*, 64(2):475–489, 2013. Cited in p. 57

[16] C.L. Cox and J.A. Knisely. A Tridiagonal System Solver for Distributed Memory Parallel Processors with Vector Nodes. *Journal of Parallel and Distributed Computing*, 13(3):325–331, 1991. Cited in p. 59

[17] C.M. Rader. Discrete Fourier Transforms when the Number of Data Samples is Prime. *Proceedings of the IEEE*, 56(6):1107–1108, 1968. Cited in p. 51

[18] *CUDA Data Parallel Primitives Library*, 2013. v2.1. Cited in p. 59

[19] D. Egloff. High Performance Finite Difference PDE Solvers on GPUs. *QuantAlea GmbH, Technical Report*, 2010. Cited in p. 59

[20] D. Fraser. Array Permutation by Index-Digit Permutation. *J. ACM*, 23(2):298–309, 1976. Cited in p. 157

[21] D. Göddeke and R. Strzodka. Cyclic Reduction Tridiagonal Solvers on GPUs Applied to Mixed Precision Multigrid. *IEEE Transactions on Parallel and Distributed Systems (TPDS), Special Issue on High Performance Computing with Accelerators*, 22(1):22–32, 2011. Cited in p. 59

[22] D. Ribbrock, M. Geveler, D. G oddeke and S. Turek. Performance and Accuracy of Lattice-Boltzmann Kernels on Multi- and Manycore Architectures. *International Conference on Computational Science. Procedia Computer Science*, 1(1):239–247, 2010. Cited in p. 82

[23] D. van Dyk, M. Geveler, S. Mallach, D. Ribbrock, D. Göddeke and C. Gutwenger. HONEI: A Collection of Libraries for Numerical Computations Targeting Multiple Processor Architectures. *Computer Physics Communications*, 180(12):2534–2543, 2009. Cited in p. 82

[24] D. Vandevoorde and N.M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2002. Cited in p. 18, 64

[25] D.B. Kirk and W.W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2nd edition, 2012. Cited in p. 39

[26] E. Chu and A. George . *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*. Computational Mathematics Series. CRC Press, 2000. Cited in p. 16, 54, 135

[27] E. Ömer, K. Cetin and J.A. Knisley. A Recursive Doubling Algorithm for Solution of Tridiagonal Systems on Hypercube Multiprocessors. *Journal of Computational and Applied Mathematics, Special Issue on Parallel Algorithms for Numerical Linear Algebra*, 27(1-2):95–108, 1989. Cited in p. 58, 59

[28] E. Polizzi and A.H. Sameh. A Parallel Hybrid Banded System Solver: The SPIKE Algorithm. *Parallel Computing*, 32(2):177–194, 2006. Cited in p. 59

[29] E.F. Toro. *Shock-Capturing Methods for Free-Surface Shallow Flows*. John Wiley & Sons, 2001. Cited in p. 81

[30] F. Argüello, M. Amor and E.L. Zapata. FFTs on Mesh Connected Computers. *Parallel Computing*, 22(1):19–38, 1996. Cited in p. 19, 64, 156, 157, 159, 161

[31] F. Franchetti and M. Puschel. Generating High Performance Pruned FFT Implementations. In *ICASSP '09: Proceedings of the 2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 549–552. IEEE Computer Society, 2009. Cited in p. 75

[32] F. Franchetti and M. Püschel. Fast Fourier Transform. In *Encyclopedia of Parallel Computing*. Springer, 2011. Cited in p. 51

[33] F. Franchetti, M. Püschel and Y. Voronenko. Discrete Fourier Transform on Multicore. *IEEE Signal Processing Magazine*, 26(6):90–102, 2009. Cited in p. 51

[34] F. Randima and M.J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003. Cited in p. 44

[35] F.C. Lin and K.K. Chung. A Cost-Optimal Parallel Tridiagonal Solver. *Parallel Computing*, 15(1-3):189–199, 1990. Cited in p. 58

[36] Freescale Semiconductor. *Complex Fixed-Point Fast Fourier Transform Optimization for AltiVec*, 2013. Cited in p. 53

[37] G. Bruun. Z-Transform DFT Filters and FFT's. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 26(1):56–63, 1978. Cited in p. 51

[38] G. Spaletta and D.J Evans. The Parallel Recursive Decoupling Algorithm for Solving Tridiagonal Linear Systems. *Parallel Computing*, 19(5):563–576, 1993. Cited in p. 59

[39] G.R. Halliwell. Evaluation of Vertical Coordinate and Vertical Mixing Algorithms in the HYbrid-Coordinate Ocean Model (HYCOM). *Ocean Modelling*, 7(3):285–322, 2004. Cited in p. 16, 58

[40] GSIC Center, Tokyo Institute of Technology. TSUBAME 2.5. `http://www.gsic.titech.ac.jp/en/tsubame`, 2013. Cited in p. 44

[41] H. Bantikyan. Implementation of Parallel Fast Hartley Transform (FHT) Using CUDA. *Journal of Computer Sciences and Applications*, 2(1):6–8, 2014. Cited in p. 56

[42] H.-S. Kim, S. Wu, L.-W. Chang and W.W. Hwu. A Scalable Tridiagonal Solver for GPU. In *International Conference on Parallel Processing*, pages 444–453. IEEE Comp. Society, 2011. Cited in p. 59

[43] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi and A. Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. In *2010 IEEE Int. Symp. on Performance Analysis of Systems Software (ISPASS)*, pages 235–246, 2010. Cited in p. 155

[44] H.V. Sorensen, D.L. Jones, C.S. Burrus and M. Heideman. On Computing the Discrete Hartley Transform. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 33(5):1231–1238, 1985. Cited in p. 56

[45] H.V. Sorensen, D.L. Jones, M. Heideman and C.S. Burrus. Real-Valued Fast Fourier Transform Algorithms. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 35(6):849–863, 1987. Cited in p. 54

[46] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics*, 23(3):777–786, 2004. Cited in p. 17, 63, 68

[47] I.J. Good. The Interaction Algorithm and Practical Fourier Analysis. *Journal of the Royal Statistical Society*, 20(2):361–372, 1958. Cited in p. 51

[48] Intel. *Intel Math Kernel Library, Reference Manual*, 2009. v10.2. Cited in p. 53, 59

[49] Intel. *Intel Integrated Performance Primitives for Intel Architecture, Reference Manual*, 2012. Volume 1: Signal Processing. Cited in p. 53, 125, 188

[50] Intel. *Intel Architecture Instruction Set Extensions Programming Reference*, 2014. Cited in p. 145

[51] J. Douglas. Alternating Direction Methods for Three Space Variables. *Numerische Mathematik*, 4(1):41–63, 1962. Cited in p. 16, 58

[52] J. Duato, A.J. Pena, F. Silla, R. Mayo and E.S. Quintana-Ortí. rCUDA: Reducing the Number of GPU-based Accelerators in High Performance Clusters. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 224–231, 2010. Cited in p. 44

[53] J. Kurzak, S. Tomov and J. Dongarra. Autotuning GEMM Kernels for the Fermi GPU. *IEEE Transactions on Parallel and Distributed Systems*, 23(11):2045–2057, 2012. Cited in p. 155

[54] J. Lamas-Rodríguez, F. Argüello, D.B. Heras and M. Bóo. Memory Hierarchy Optimization for Large Tridiagonal System Solvers on GPU. In *EEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 87–94, 2012. Cited in p. 59

[55] J. Lobeiras, M. Amor and R. Doallo. FFT Implementation on a Streaming Architecture. In *PDP '11: Proc. of the 19th Euromicro Conference On Parallel, Distributed and Network-based Processing*, pages 381–388. IEEE Computer Society, 2011. Cited in p. 17, 54, 63, 67, 125

[56] J. Lobeiras, M. Amor and R. Doallo. Performance Evaluation of GPU Memory Hierarchy using the FFT. In *Proc. of the 11th Int'l Conf. on Computational and Mathematical Methods in Science and Engineering (CMMSE)*, volume 2, pages 750–761, 2011. Cited in p. 17, 63, 109

[57] J. Lobeiras, M. Amor and R. Doallo. Influence of Memory Access Patterns to small-scale FFT Performance. *Journal of Supercomputing*, 64(1):120–131, 2013. Cited in p. 17, 63, 100, 109, 169

[58] J. Lobeiras, M. Amor and R. Doallo. SPLG: A Tuned Signal Processing Library for GPU Architectures. In *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2013)*, volume 1, pages 184–191, 2013. Cited in p. 18, 64, 128, 173

[59] J. Lobeiras, M. Amor and R. Doallo. BPLG: A tuned Butterfly Processing Library for GPU Architectures. *International Journal of Parallel Programming (IJPP), accepted for publication*, 2014. Cited in p. 128

[60] J. Lobeiras, M. Amor and R. Doallo. Operator Strings Algebraic Manipulation. Technical report, Computer Architecture Group, University of A Coruña, 2014. Cited in p. 173

[61] J. Lobeiras, M. Viñas, M. Amor, B.B. Fraguela, M. Arenaz, J.A. García, M.J. Castro. Parallelization of Shallow Water Simulations on Current Multi-Threaded Systems. *International Journal of High Performance Computing Applications (IJHPCA)*, 27(4):493–512, 2013. Cited in p. 17, 63, 67

[62] J.M. Gallardo, S. Ortega, M. de la Asunción and J.M. Mantas. Two-Dimensional Compact Third-Order Polynomial Reconstructions. Solving Non-conservative Hyperbolic Systems Using GPUs. *Journal of Scientific Computing*, pages 1–23, 2011. Cited in p. 82, 90

[63] J.R. Ohm, G.J. Sullivan, H. Schwarz, T. Keng Tan and T. Wiegand. Comparison of the Coding Efficiency of Video Coding Standards - Including High Efficiency Video Coding (HEVC). *IEEE Trans. on Circuits Systems for Video Technology*, 22(12):1669–1684, 2012. Cited in p. 57

[64] J.S. Jacaba. Audio Compression Using Modified Discrete Consine Transform: The MP3 Coding Standard. Technical report, University of Philippines. Research paper, 2001. Cited in p. 57

[65] J.W. Choi, A. Singh and R.W. Vuduc. Model-Driven Autotuning of Sparse Matrix-Vector Multiply on GPUs. In *Proc. of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP 2010)*, volume 45, pages 115–126, 2010. Cited in p. 155

[66] 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. Cited in p. 51, 110, 164

[67] K. Gregory and A. Miller. *C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++*. Microsoft Press, 2012. Cited in p. 44

[68] Keith Jones. *The Regularized Fast Hartley Transform*. Signals and Communication Technology. Springer, 2010. Cited in p. 56, 131, 135

[69] Khronos OpenCL Working Group. *The OpenCL Specification*, 2011. Cited in p. 44

[70] K.R. Rao and P.C. YIP. *The Transform and Data Compression Handbook*. The Electrical Eng. and Signal Processing. CRC Press, 2001. Cited in p. 16, 50, 57, 131, 135

[71] L. Chen, L. Liu, S. Tang, L. Huang, Z. Jing, S. Xu, D. Zhang and B. Shou. Unified Parallel C for GPU Clusters: Language Extensions and Compiler Implementation. In V. S. K. Cooper, J. Mellor-Crummey, editor, *Languages and Compilers for Parallel Computing*, volume 6548 of *Lecture Notes in Computer Science (LNCS)*, pages 151–165. Springer Berlin Heidelberg, 2011. Cited in p. 44

[72] L.-W. Chang, J.A. Stratton, H.-S. Kim and W.W. Hwu. A Scalable, Numerically Stable, High-performance Tridiagonal Solver Using GPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*, pages 27:1–27:11. IEEE Computer Society Press, 2012. Cited in p. 59, 60

[73] L.H. Thomas. Elliptic Problems in Linear Differential Equations over a Network. Technical report, Columbia University, 1949. Cited in p. 58

[74] L.H. Thomas. Using a Computer to Solve Problems in Physics. In *Applications of Digital Computers*. Ginn and Company, 1963. Cited in p. 51

[75] L.I. Bluestein. A Linear Filtering Approach to the Computation of Discrete Fourier Transform. *IEEE Transactions on Audio and Electroacoustics*, 18(4):451–455, 1970. Cited in p. 51

[76] M. Bueno and F. Dopico. Stability and Sensitivity of Tridiagonal LU Factorization without Pivoting. *Bit Numerical Mathematics*, 44(4):651–673, 2004. Cited in p. 58

[77] M. de la Asunción, J.M. Mantas and M.J. Castro. Simulation of One-Layer Shallow Water Systems on Multicore and CUDA Architectures. *Journal of Supercomputing*, pages 1–9, 2010. Cited in p. 100

[78] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proc. of the IEEE*, 93(2):216–231, 2005. Cited in p. 54, 145

[79] M. Geveler, D. Ribbrock, D. G oddeke, S. Turek. Lattice-Boltzmann Simulation of the Shallow-Water Equations with Fluid-Structure Interaction on Multi- and Manycore Processors. *Lecture Notes in Computer Science: Facing the Multicore Challenge*, 6310:92–104, 2010. Cited in p. 82

[80] M. Guptda and A.K. Garg. Analysis of Image Compression Algorithm using DCT. *International Journal of Engineering Research and Applications (IJERA)*, 2(1):512–521, 2012. Cited in p. 57

[81] M. Harris. *Optimizing Parallel Reduction in CUDA*. NVIDIA, 2012. v5.0. Cited in p. 94

[82] M. Lastra, J.M. Mantas, C. Ureña, M.J. Castro and J.A García-Rodríguez. Simulation of Shallow Water Systems using Graphics Processing Units. *Mathematics and Computers in Simulation*, 80(3):598–618, 2009. Cited in p. 82

[83] M. Mathew, V. Bhat, S.M. Thomas and Y. Changhoon. Modified MP3 Encoder using Complex Modified Cosine Transform. In *International Conference on Multimedia and Expo (ICME '03)*, volume 1, pages 709–712. IEEE Computer Society, 2003. Cited in p. 57

[84] M. Panella and L. Basset. An Efficient GPU Implementation of Modified Discrete Cosine Transform Using CUDA. *Int. Journal of Computer Science and Information Security*, 10(5):23–30, 2012. Cited in p. 57, 145

[85] M. Püschel, J.M.F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson and N. Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proc. of the IEEE, on "Program Generation, Optimization, and Platform Adaptation"*, 93(2):232–275, 2005. Cited in p. 54

[86] M. Viñas, J. Lobeiras, B.B. Fraguela, M. Arenaz, M. Amor and R. Doallo. Simulation of Pollutant Transport in Shallow Water on a CUDA Architecture. In *Proc. of Workshop on Exploitation of Hardware Accelerators (WEHA 2011)*

As part of the 2011 International Conf. on High Performance Computing and Simulation, HPCS2011, pages 664–670, 2011. Cited in p. 17, 63, 67, 82, 94

[87] M. Viñas, J. Lobeiras, B.B. Fraguela, M. Arenaz, M. Amor, J.A. García, M.J. Castro and R. Doallo. A Multi-GPU Shallow-Water Simulation with Transport of Contaminants. *Concurrency and Computation: Practice and Experience*, 25(8):1153–1169, 2013. Cited in p. 17, 63, 67

[88] M. Viñas, Z. Bozkus and B.B. Fraguela. Exploiting Heterogeneous Parallelism with the Heterogeneous Programming Library. *Journal of Parallel and Distributed Computing*, 73(12):1627–1638, 2013. Cited in p. 44

[89] M.C. Pease. An Adaptation of the Fast Fourier Transform for Parallel Processing. *Journal of the Association for Computing Machinery (ACM)*, 15(2):252–264, 1968. Cited in p. 51, 52, 110

[90] M.J. Castro, E.D. Fernández-Nieto, A.M. Ferreiro, J.A. García-Rodríguez and C. Parés. High Order Extensions of Roe Schemes for Two Dimensional Nonconservative Hyperbolic Systems. *Journal of Scientific Computing*, 39(1):67–114, 2009. Cited in p. 86, 90

[91] M.J. Castro, J.A. García-Rodríguez, J.M. González-Vida and C. Parés. A Parallel 2D Finite Volume Scheme for Solving Systems of Balance Laws with Nonconservative Products: Application to Shallow Flows. *Computer Methods in Applied Mechanics and Engineering*, 195(19-22):2788–2815, 2006. Cited in p. 82, 86

[92] M.J. Castro, J.A. García-Rodríguez, J.M. González-Vida and C. Parés. Solving Shallow-Water Systems in 2D Domains using Finite Volume Methods and Multimedia SSE Instructions. *J. Comput. Appl. Math.*, 221(1):16–32, 2008. Cited in p. 82

[93] M.J. Castro, T. Chacón, E.D. Fernández-Nieto, J.M. González-Vida, C. Parés. Well-Balanced Finite Volume Schemes for 2D non-Homogeneous Hyperbolic Systems. Application to the dam break of Aznalcóllar. *Computer Methods in Applied Mechanics and Engineering*, 197:3932–3950, 2008. Cited in p. 86, 87, 90, 91

[94] M.L. Sætra and A.R. Brodtkorb. Shallow Water Simulations on Multiple GPUs. In *Applied Parallel and Scientific Computing*, volume 7134 of *Lecture Notes in Computer Science*, pages 56–66. Springer, 2012. Cited in p. 82

[95] N. Ahmed, T. Natarajan and K.R. Rao. Discrete Cosine Transform. *IEEE Transactions on Computers*, C-23(1):90–93, 1974. Cited in p. 16, 57

[96] N. Bell. Thrust: A Productivity-Oriented Library for CUDA. In *GPU Computing Gems, Jade Edition*. Morgan Kaufmann, 2011. Cited in p. 18, 64

[97] N. Sakharnykh. Efficient Tridiagonal Solvers for ADI Methods and Fluid Simulation. In *GPU Technology Conference 2010 (GTC 2010 presentation)*, 2010. Cited in p. 59

[98] NVIDIA. *CUDA Compute Unified Device Architecture*, 2011. Cited in p. 44

[99] NVIDIA. *CUDA C Best Practices Guide (SDK Document.)*, 2012. v5.0. Cited in p. 100, 128, 133

[100] NVIDIA. *CUDA CUFFT Library*, 2012. v5.0. Cited in p. 19, 54

[101] NVIDIA. *CUSPARSE Library*, 2012. v5.0. Cited in p. 19, 59

[102] Oak Ridge National Laboratory. Titan. `http://www.olcf.ornl.gov/titan`, 2013. Cited in p. 43

[103] OpenACC. *The OpenACC Application Programming Interface*, 2013. v2.0. Cited in p. 44

[104] P. Wiemann, S. Wenger and M. Magnor. CUDA Expression Templates. In *WSCG Communication Papers Proceedings 2011*, pages 185–192, 2011. Cited in p. 19, 65, 156

[105] P. Ye, X. Shi and X. Li. CUDA Based Implementation of DCT/IDCT on GPU. Technical report, University of Delaware, 2008. Cited in p. 57

[106] P. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc., 1996. Cited in p. 14, 33

[107] Q. Hou, K. Zhou and B. Guo. BSGP: Bulk-Synchronous GPU Programming. *ACM Transactions on Graphics*, 27(3):1–12, 2008. Cited in p. 44

[108] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001. Cited in p. 105

[109] S. B. R. Dolbeau and F. Bodin. HMPP: A Hybrid Multi-core Parallel Programming Environment. In *Proceedings of the Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, pages 1–5, 2007. Cited in p. 44

[110] R.C. Singleton. A Method for Computing the Fast Fourier Transform with Auxiliary Memory and Limited High-Speed Storage. *IEEE Transactions on Audio and Electroacoustics*, 15:91–98, 1967. Cited in p. 72

[111] R.J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press, 2002. Cited in p. 81

[112] R.V.L. Hartley. A More Symmetrical Fourier Analysis Applied to Transmission Problems. *Proc. of the Institute of Radio Engineers (IRE)*, 30(3):144–150, 1942. Cited in p. 16, 55

[113] R.W. Hockney. A Fast Direct Solution of Poisson's Equation Using Fourier Analysis. *Journal of the ACM (JACM)*, 12(1):95–113, 1965. Cited in p. 58

[114] R.W. Hockney and C.R. Jesshope. *Parallel Computers Two: Architecture, Programming and Algorithms*. IOP Publishing Ltd., 1988. Cited in p. 58

[115] S. Bondeli. Divide and Conquer: A Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations. *Parallel Computing*, 17(4-5):419–434, 1991. Cited in p. 59

[116] S. Hong and H. Kim. An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness. In *Proc. of the 36th Int. Symposium on Computer Architecture (ISCA '09)*, volume 37, pages 152–163, 2009. Cited in p. 155

[117] S. Oraintara, Ying-Jui Chen and T.Q. Nguyen. Integer Fast Fourier Transform. *IEEE Transactions on Signal Processing*, 50(3):607–618, 2002. Cited in p. 53

[118] S. Winograd. On Computing the Discrete Fourier Transform. *Mathematics of Computation*, 32(141):175–199, 1978. Cited in p. 51

[119] S.C. Chapra and R. Canale. *Numerical Methods for Engineers*. McGraw-Hill, Inc., 5 edition, 2006. Cited in p. 16

[120] S.M. Müller and D. Scheerer. A Method to Parallelize Tridiagonal Solvers. *Parallel Computing*, 17(2-3):181–188, 1991. Cited in p. 59

[121] S.S. Baghsorkhi, M. Delahaye, S.J. Patel, W.D. Gropp and W.W. Hwu. An Adaptive Performance Modeling Tool for GPU Architectures. In *Proceedings of the 15 th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010).*, pages 105–114, 2010. Cited in p. 155

[122] Swiss National Supercomputing Centre (CSCS). Piz Daint. `http://www.cscs.ch/computers/piz_daint`, 2013. Cited in p. 44

[123] T. Han and T. Abdelrahman. hiCUDA: A High-level Directive-based Language for GPU Programming. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61, 2009. Cited in p. 44

[124] T. Morales de Luna, M.J. Castro, C. Parés and E.F Nieto. On a Shallow Water Model for the Simulation of Turbidity Currents. *Communications in Computational Physics*, 6:848–882, 2009. Cited in p. 82

[125] T. Runar, K.-A. Lie and J. R. Natvig. Solving the Euler Equations on Graphics Processing Units. In *Proceedings of the 6th International Conference on Computational Science*, volume 3994 of *Lecture Notes in Computer Science*, pages 220–227, 2006. Cited in p. 82

[126] R. Taylor and X. Li. A Micro-benchmark Suite for AMD GPUs. In *2010 39th International Conference on Parallel Processing Workshops (ICPPW)*, pages 387–396, 2010. Cited in p. 155

[127] T.G. Stockham. High-Speed Convolution and Correlation. In *Proceedings of the Spring Joint Computer Conference*, pages 229–233, 1966. Cited in p. 51, 52, 164

[128] Top500. Top 500 Supercomputer Sites. `http://www.top500.org`, 2013. Cited in p. 43

[129] T.R. Hagen, M.O. Henriksen, J.M. Hjelmervik and K.-A. Lie. How to Solve Systems of Conservation Laws Numerically Using the Graphics Processor as a High-Performance Computational Engine. In G. Hasle, K.-A. Lie, and E. Quak, editors, *Geometric Modelling, Numerical Simulation, and Optimization*, pages 211–264. Springer Berlin Heidelberg, 2007. Cited in p. 82

[130] V. Chauhan, P.K. Nathaney, M. Pandey and K.M. Rai. A Novel Approach to Video Compression Technique using Variable Block Sizes in Motion Estimation Process. *International Journal of Electronics and Computer Science Engineering (IJECSE)*, 1(3):1321–1327, 2012. Cited in p. 16, 57

[131] V. Volkov. Better Performance at Lower Occupancy. In *GPU Technology Conference 2010 (GTC 2010 presentation)*, 2010. Cited in p. 113, 166

[132] V. Volkov. Use Registers and Multiple Outputs per Thread on GPU. In *International Workshop on Parallel Matrix Algorithms and Applications 2010 (PMAA '10 presentation)*, 2010. Cited in p. 113, 166

[133] V. Volkov and B. Kazian. Fitting FFT onto the G80 Architecture. Technical report, University of California, Berkeley, 2009. Cited in p. 54

[134] W. Gander and G.H. Golub. Cyclic Reduction - History and Applications. In *Workshop on Scientific Computing*, volume 6, pages 73–88, 1997. Cited in p. 58

[135] X. Wang and Z.G. Mou. A Divide-and-Conquer Method of Solving Tridiagonal Systems on Hypercube Massively Parallel Computers. *IEEE Symposium on Parallel and Distributed Processing*, pages 810–817, 1991. Cited in p. 59, 60, 139, 177

[136] Y. Dotsenko, S.S. Baghsorkhi, B. Lloyd and N.K. Govindaraju. Auto-Tuning of Fast Fourier Transform on Graphics Processors. In *Principles and Practice of Parallel Programming (PPoPP '11)*, pages 257–266, 2011. Cited in p. 155

[137] Y. Li, Y. Zhang, Y. Liu, G. Long and H. Jia. MPFFT: An Auto-Tuning FFT Library for OpenCL GPUs. *Journal of Computer Science and Technology*, 28(1):90–105, 2013. Cited in p. 155

[138] Y. Wang and M. Vilermo. Modified Discrete Cosine Transform: Its Implications for Audio Coding and Error Concealment. *Journal of Audio Engineering Society*, 51(1):52–61, 2003. Cited in p. 16, 57

[139] Y. Wang, M. Baboulin, J. Dongarra, J. Falcou, Y. Fraigneau and O.P. Le Maître. A Parallel Solver for Incompressible Fluid Flows. In *International Conf. on Computational Science (ICCS 2013)*, volume 18 of *Procedia Computer Science*, pages 439–448. Elsevier, 2013. Cited in p. 16, 58

[140] Y. Yang, P. Xiang, M. Mantor, N. Rubin and H. Zhou. Shared Memory Multiplexing: A Novel Way to Improve GPGPU Throughput. In *Proc. of the 21st Int'l. Conf. on Parallel Architectures and Compilation Techniques*, PACT '12, pages 283–292. ACM, 2012. Cited in p. 139, 176

[141] Y. Zhang and J.D. Owens. A Quantitative Performance Analysis Model for GPU Architectures. In *Proc. of the 17th IEEE Int. Symposium on High-Performance Computer Architecture (HPCA 17)*, pages 382–393, 2011. Cited in p. 155

[142] Y. Zhang, J. Cohen and J.D. Owens. Fast Tridiagonal Solvers on the GPU. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010)*, pages 127–136, 2010. Cited in p. 59, 151, 192