University of Almería
Department of Informatics

Ph.D. Thesis

High Performance Computing for solving large sparse systems. Optical Diffraction Tomography as a case of study. (Computación de altas prestaciones para la resolución de sistemas dispersos de grandes dimensiones. Tomografía Óptica Difraccional como caso de estudio)

Gloria Ortega López
Almería, May 2014

Ph.D. Thesis

High Performance Computing for solving large sparse systems. Optical Diffraction Tomography as a case of study. (Computación de altas prestaciones para la resolución de sistemas dispersos de grandes dimensiones. Tomografía Óptica Difraccional como caso de estudio)

University of Almería
Department of Informatics

Author: Gloria Ortega López
Supervisors: Inmaculada García Fernández
Gracia Ester Martín Garzón

Almería, May 2014

A mis padres y a mi hermano

Agradecimientos

Esta tesis ha sido posible gracias a la financiación recibida del Ministerio de Educación, Cultura y Deporte a través de la beca FPU (AP2009-4797), del Ministerio de Economía y Competitividad a través de los proyectos de investigación TIN2008-01117 y TIN2012-37483 y a la Consejería de Economía, Innovación, Ciencia y Empleo de la Junta de Andalucía a través de los proyectos P10-TIC6002 y P11-TIC7176, cofinanciados por el Fondo Europeo de Desarrollo Regional (FEDER).

Igualmente, quiero expresar mi agradecimiento al Programa HPC-Europa2 y al Plan Propio de Investigación de la Universidad de Almería, por darme la oportunidad de realizar estancias en el extranjero, que han sido muy importantes en el desarrollo de esta tesis y en mi propia formación científica y humana. Este agradecimiento se hace extensivo a los centros de Supercomputación *Edinburgh Parallel Computing Centre (EPCC)* y *High Performance Computing Center Stuttgart (HLRS)* por poner a mi disposición sus recursos, tanto técnicos como científicos. En particular, me gustaría agradecer a los Drs. D. David Henty y D. José Gracia su soporte y profesionalidad.

La realización de esta Tesis Doctoral ha resultado ser un reto, tanto personal como intelectual, que se ha hecho posible gracias al apoyo recibido de todas aquellas personas que me han acompañado en este largo camino.

En primer lugar me gustaría agradecer a mis dos supervisoras, Dras. Dª Inmaculada García Fernández y Dª Gracia Ester Martín Garzón, su consejo, dedicación y confianza depositada en mí. El hecho de compatir con ellas este tiempo ha sido un aspecto clave en mi desarrollo personal y labor investigadora.

Notable reconocimiento merecen los compañeros del grupo de investigación TIC-146 “Supercomputación: Algoritmos”, con los que he compartido muy buenos momentos durante estos años. En especial me gustaría agradecer al Dr. D. Francisco Vázquez López, por poner a mi disposición el código de la valiosa rutina ELLR-T que fue la base de su tesis. Además, quiero expresar mi agradecimiento al Dr. D. José Antonio Martínez García por su estimable apoyo en tierras alemanas.

Quisiera hacer extensible mi sincera gratitud a las Dras. D^{a} M^{a} del Pilar Arroyo Grandes, D^{a} Julia Lobera Salazar y D^{a} Siham Tabik ya que su colaboración ha sido importante para mi formación.

Esta tesis se la dedico principalmente a mis padres, que son un pilar fundamental en mi vida. Ellos me han enseñado el valor de las cosas y que el esfuerzo no tiene límites cuando te propones alcanzar un sueño. Con este logro quiero devolver un poco de lo que me han dado desde que nací.

A mi hermano Luis, porque siempre ha estado a mi lado a pesar de la distancia. Gracias por ser mi apoyo incondicional y mi fuente de inspiración cuando más lo he necesitado.

Un agradecimiento muy especial merece la comprensión, paciencia y el ánimo recibidos de mi familia y amigos. En especial a aquellos que me han hecho compañía incluso en el extranjero.

A todos ellos, muchas gracias.

Prefacio

Esta tesis titulada "High Performance Computing for solving large sparse systems. Optical Diffraction Tomography as a case of study" (Computación de altas prestaciones para la resolución de sistemas dispersos de grandes dimensiones. Tomografía Óptica Difraccional como caso de estudio) investiga los aspectos computacionales asociados con la resolución de sistemas de ecuaciones lineales procedentes de la discretización de modelos físicos descritos mediante sistemas de Ecuaciones Diferenciales en Derivadas Parciales (PDEs). Estos modelos físicos se conciben para describir el comportamiento espacio-temporal de algún fenómeno físico $f(x, y, z, t)$ en términos de sus variaciones (derivadas parciales) con respecto a algunas de las variables de las que depende el fenómeno. Existe una gran diversidad de métodos de discretización de PDEs. Los dos más extendidos son el Método de las Diferencias Finitas (FDM) y el Método de los Elementos Finitos (FEM). Ambos métodos dan lugar a una descripción algebraica del modelo que se traduce en el planteamiento de un sistema de ecuaciones lineales del tipo ($Ax = b$), donde A es una matriz dispersa (porcentaje muy alto de elementos nulos) cuyo tamaño depende de la precisión con la que se desee modelar el fenómeno.

En esta tesis partimos de la descripción algebraica del modelo asociado al fenómeno físico, y nuestras contribuciones están relacionadas con el diseño de técnicas y modelos computacionales que permiten resolver estos sistemas de ecuaciones. Nuestro interés se centra en modelos que requieren un nivel de discretización muy fino y que por lo tanto generan matrices, A, que tienen una estructura dispersa y un gran tamaño. La literatura científica caracteriza este tipo de problemas por una alta demanda computacional (debido al grado de discretización) y por la dispersión de las matrices que

los definen, planteando que únicamente se pueden abordar mediante el uso de métodos y arquitecturas computacionales de alto rendimiento.

Actualmente, las arquitecturas de alto rendimiento más extendidas son los sistemas heterogéneos formados por clústeres de multiprocesadores de memoria distribuida, donde cada nodo posee una arquitectura multi-core con distinto número de núcleos. Además, estas arquitecturas pueden disponer de diversos elementos aceleradores, tales como unidades vectoriales, FPGAs, GPUs, Coprocesadores Intel Xeon Phi y una red de interconexión heterogénea compuesta por enlaces de distinto ancho de banda y latencia.

Uno de los principales objetivos de esta tesis es investigar las posibles alternativas que permitan la implementación de rutinas capaces de resolver sistemas lineales dispersos de grandes dimensiones y basadas en el aprovechamiento de las modernas plataformas de altas prestaciones. El uso de plataformas masivamente paralelas (GPUs), permite la aceleración de estas rutinas, ya que presentan ventajas para esquemas de computación vectorial. Por otro lado, el uso de plataformas de memoria distribuida permite la resolución de problemas que pueden ser modelados mediante matrices de enorme tamaño. Finalmente, la combinación de ambas técnicas, computación distribuida y multi-GPUs, permitirá abordar problemas de interés donde intervienen matrices dispersas de gran tamaño, en un tiempo muy reducido. En este sentido, una de las aportaciones de este trabajo consiste en poner a disposición de la comunidad implementaciones optimizadas para clústeres multi-GPU que permitan resolver sistemas de ecuaciones lineales dispersos, que son un aspecto clave en la computación científica.

La segunda parte de esta tesis se centra en un problema físico real del campo de la Tomografía Óptica Difraccional basado en datos holográficos. La Tomografía Óptica Difraccional permite extraer información sobre la forma de los objetos con una gran precisión y sin someterlos a la agresión de intensas radiaciones, por lo que posibilita la investigación de tejidos, microorganismos, etc. en vivo, y hace posible el estudio de su dinámica. Un modelo físico preliminar basado en la reconstrucción bidimensional de la distribución de partículas sembradoras en fluidos fue propuesto por J. Lobera

y J.M. Coupland. Sin embargo, su alto coste computacional (memoria y tiempo de cómputo) hace que su extensión a un modelo tridimensional tenga que basarse, necesariamente, en el uso de técnicas de computación de altas prestaciones. En la segunda parte de esta tesis se aborda la implementación y validación de este modelo físico para el caso de reconstrucciones tridimensionales. En dicho desarrollo es necesaria la resolución de grandes sistemas de ecuaciones dispersos. Por lo tanto, algunas de las rutinas algebraicas que hemos implementado en esta tesis han sido utilizadas para la implementación de estrategias computacionales capaces de resolver el problema de la reconstrucción 3D de Tomografía Óptica Difraccional.

Esta tesis está organizada en seis capítulos. El primero de ellos es una introducción a las áreas de investigación en las que se enmarca esta tesis. En dicho capítulo se presentan los materiales y métodos utilizados. En primer lugar, se revisan brevemente las arquitecturas paralelas así como los modelos de programación más utilizados actualmente. A continuación se describen de forma sucinta los aspectos relacionados con los métodos para resolver sistemas de ecuaciones lineales, una de las claves más importantes del Álgebra Lineal. Finalmente, se indican las plataformas computacionales que se han utilizado para evaluar las distintas implementaciones llevadas a cabo en esta tesis.

El Capítulo 2 se centra en la computación de matrices dispersas sobre plataformas GPU. En primer lugar, se lleva a cabo una descripción de los formatos de almacenamiento de matrices dispersas que más se utilizan en la actualidad. A continuación, se describen los detalles de una implementación de la operación matriz dispersa vector (SpMV) basada en el formato ELLR-T, que será utilizada en los siguientes capítulos. Finalmente, se propone una rutina algebraica para el cálculo de la multiplicación matriz dispersa matriz densa (SpMM), que se evalúa sobre plataformas GPUs. Esta rutina ha demostrado obtener mejores resultados, en términos de rendimiento, que otros kernels presentes en la literatura.

El Capítulo 3 está dedicado al desarrollo de un método capaz de resolver grandes sistemas de ecuaciones lineales dispersos utilizando GPUs. Concre-

tamente nos hemos centrado en el método del Gradiente Biconjugado (BCG) para resolver estos sistemas de ecuaciones. BCG ha sido desarrollado para su ejecución en plataformas GPU, utilizando dos rutinas alternativas para realizar la operación SpMV que incluye dicho método. Estas rutinas son: una rutina incluida en la librería CUSPARSE, ya existente en la literatura, y una rutina del SpMV basada en el formato ELLR-T. Al final del capítulo se muestra una evaluación comparativa de los dos métodos para un conjunto de matrices de prueba, donde se observa que la resolución del método BCG basado en el formato ELLR-T (para resolver la operación SpMV) obtiene un mejor rendimiento que el BCG que se implementa utilizando la librería CUSPARSE.

En el Capítulo 4 se estudia la ecuación de Helmholtz como un caso particular de sistemas de ecuaciones en derivadas parciales cuya discretización da lugar a un gran sistema de ecuaciones lineales, que en este caso está caracterizado por una matriz dispersa de grandes dimensiones y que exhibe un patrón regular particular en cuanto a la localización de los elementos no nulos. Las características de la ecuación de Helmholtz se detallan al comienzo del capítulo. Posteriormente, se analizan varias implementaciones multi-GPU que se han diseñado para acelerar la resolución de la ecuación de Helmholtz.

En el Capítulo 5 se describe el desarrollo e implementación de un nuevo método de reconstrucción tomográfica 3D basada en el procesado de datos holográficos con técnicas de computación de altas prestaciones. Este nuevo método requiere la resolución de la ecuación de Helmholtz cuya discretización da lugar a un sistema de ecuaciones disperso, regular, de tipo complejo y dimensiones elevadas, cuya solución puede obtenerse mediante el método BCG. El desarrollo del modelo mencionado se basa en una implementación del método BCG sobre GPUs que explota las regularidades de la matriz, tal como se describe en el Capítulo 4.

En el Capítulo 6 se resumen las conclusiones y las principales aportaciones de estas tesis. Además, se plantean algunas de las lineas de investigación que quedan abiertas y que se abordarán en el futuro.

Preface

This thesis, entitled "High Performance Computing for solving large sparse systems. Optical Diffraction Tomography as a case of study" investigates the computational issues related to the resolution of linear systems of equations which come from the discretization of physical models described by means of Partial Differential Equations (PDEs). These physical models are conceived for the description of the space-temporary behavior of some physical phenomena $f(x, y, z, t)$ in terms of their variations (partial derivative) with respect to the dependent variables of the phenomena. There is a wide variety of discretization methods for PDEs. Two of the most well-known methods are the Finite Difference Method (FDM) and the Finite Element Method (FEM). Both methods result in an algebraic description of the model that can be translated into the approach of a linear system of equations of type $(Ax = b)$, where A is a sparse matrix (a high percentage of zero elements) whose size depends on the required accuracy of the modeled phenomena.

This thesis begins with the algebraic description of the model associated with the physical phenomena, and the work herein has been focused on the design of techniques and computational models that allow the resolution of these linear systems of equations. The main interest of this study is specially focused on models which require a high level of discretization and usually generate sparse matrices, A, which have a highly sparse structure and large size. Literature characterizes these types of problems by their high demanding computational requirements (because of their fine degree of discretization) and the sparsity of the matrices involved, suggesting that these kinds of problems can only be solved using High Performance Computing techniques and architectures.

Nowadays, the High Performance Computing architectures that are most widely used are the heterogeneous platforms based on distributed memory systems, where every node has a multi-core architecture with a different number of cores. Moreover, these architectures can also provide several accelerators, such as vector units, FPGAs, GPUs, Intel Xeon Phi coprocessors and a heterogeneous interconnection network composed of links with different bandwidth and latency.

One of the main goals of this thesis is the research of the possible alternatives which allow the implementation of routines to solve large and sparse linear systems of equations using High Performance Computing (HPC). The use of massively parallel platforms (GPUs) allows the acceleration of these routines, because they have several advantages for vectorial computation schemes. On the other hand, the use of distributed memory platforms allows the resolution of problems defined by matrices of enormous size. Finally, the combination of both techniques, distributed computation and multi-GPUs, will allow faster resolution of interesting problems in which large and sparse matrices are involved. In this line, one of the goals of this thesis is to supply the scientific community with implementations based on multi-GPU clusters to solve sparse linear systems of equations, which are the key in many scientific computations.

The second part of this thesis is focused on a real physical problem of Optical Diffractional Tomography (ODT) based on holographic information. ODT is a non-damaging technique which allows the extraction of the shapes of objects with high accuracy. Therefore, this technique is very suitable to the in vivo study of real specimens, microorganisms, etc., and it also makes the investigation of their dynamics possible. A preliminary physical model based on a bidimensional reconstruction of the seeding particle distribution in fluids was proposed by J. Lobera and J.M. Coupland. However, its high computational cost (in both memory requirements and runtime) made compulsory the use of HPC techniques to extend the implementation to a three dimensional model. In the second part of this thesis, the implementation and validation of this physical model for the case of three dimensional

reconstructions is carried out. In such implementation, the resolution of large and sparse linear systems of equations is required. Thus, some of the algebraic routines developed in the first part of the thesis have been used to implement computational strategies capable of solving the problem of 3D reconstruction based on ODT.

This thesis is organized into six chapters. The first is an introduction to the main research areas involved in this thesis and it also presents the materials and methods that were used. Firstly, several parallel architectures and parallel programming models are briefly described. This is followed by the discussion of the issues related to the methods for solving linear systems of equations, one of the keys of Linear Algebra. Finally, the platforms used to evaluate the experiments are described.

Chapter 2 is dedicated to the computational aspects of sparse matrices on GPU platforms. First of all, a description of the most widely used compressed storage formats is made. Next, details of an implementation of the Sparse Matrix vector product (SpMV) based on ELLR-T format is shown. Finally, a routine for computing the Sparse Matrix Matrix product (SpMM) is proposed and evaluated on GPUs platforms. The implementation of the Sparse Matrix Matrix product in this study has shown to outperform other existing approaches.

Chapter 3 deals with the development of a method to solve large and sparse linear systems of equations on GPUs, namely, the BiConjugate Gradient Method (BCG). This chapter discusses the implementation of the BCG method on GPUs using two different approaches to compute the SpMV operation. These routines are: a routine included in the CUSPARSE library and an implementation of the SpMV based on ELLR-T format. At the end of the chapter the experimental results are provided from an extensive evaluation carried out using a set of test matrices. Experiments have shown that the implementation of the BCG based on ELLR-T outperforms the other approach.

In Chapter 4 the Helmholtz Equation is explored. It is a particular case of Partial Differential Equation whose discretization results in a large and

sparse linear system of equations. This linear system of equations is characterized by the regularities of the large and sparse matrix involved in the resolution. The features of the Helmholtz Equation are described at the beginning of the chapter. Later, several multi-GPU implementations proposed to accelerate the resolution of the 3D Helmholtz Equation are discussed.

In Chapter 5 the development and implementation of a new tomographic reconstruction technique based on holography and HPC techniques and architectures is described. This new method requires the resolution of the 3D Helmholtz Equation, whose discretization results in a complex, regular, large and sparse linear system of equations. BCG is the proposed solver to obtain the solution of the Helmholtz Equation. As a result, the development of the model is based on the implementation of BCG on GPUs exploiting the regularities of the sparse matrix as described in Chapter 4.

In Chapter 6 a summary where the main results are outlined is provided. Moreover, future lines of research are presented.

Contents

Prefacio v

Preface ix

List of Figures xiii

List of Tables xvii

List of Algorithms xxi

1 Introduction 1

1.1 High Performance Computing issues 1

1.1.1 High Performance Computing platforms 1

1.1.1.1 Shared memory multiprocessors 2

1.1.1.2 GPU 3

1.1.1.3 Xeon Phi 6

1.1.1.4 Distributed memory cluster 7

1.1.2 HPC development methodologies for scientific applications . . . 9

1.1.2.1 Parallel programming interfaces and models 10

1.1.2.2 Optimizing Code Performance 13

1.1.2.3 Algebraic libraries 18

1.1.2.4 MATLAB 20

1.1.3 Performance metrics 23

1.1.3.1 Speed up and efficiency . . . 23
1.1.3.2 Performance bounds: Roofline model . . . 23
1.2 Mathematical issues . . . 25
1.2.1 Solution of sparse linear systems of equations . . . 26
1.2.1.1 Computational study of Krylov methods . . . 30
1.3 Platforms used in this thesis . . . 35

2 Sparse matrix computation on GPUS 39
2.1 Compressed storage formats . . . 40
2.2 Sparse matrix vector product . . . 41
2.3 Sparse Matrix Matrix product. FastSpMM . . . 44
2.3.1 SPMM product on GPUs . . . 46
2.3.1.1 Streaming computation for FastSpMM . . . 50
2.3.2 Evaluation . . . 52
2.4 Conclusions . . . 63

3 BiConjugate Gradient for complex matrices on GPUs 65
3.1 Introduction . . . 65
3.2 BiConjugate Gradient Method . . . 67
3.2.1 Preconditioning BCG . . . 69
3.3 GPU Implementation of the BCG Method . . . 69
3.4 Evaluation . . . 71
3.5 Conclusions . . . 76

4 The 3D Helmholtz equation on multi-GPU clusters 79
4.1 Mathematical and Physical view of the 3D Helmholtz equation . . . 80
4.2 BCG-3D Helmholtz on multi-GPU clusters . . . 81
4.2.1 Related works . . . 83
4.3 BCG-3DH algorithm . . . 85
4.4 Compressed Regular Format . . . 87
4.5 BCG-3DH Implementations: CPU, GPU, Hybrid versions . . . 89
4.5.1 MPI tasks . . . 91
4.5.2 GPU computing . . . 92
4.6 Evaluation . . . 94

4.6.1 CPU and GPU versions . . . 96
4.6.2 Hybrid implementation: Fast-Helmholtz . . . 101
4.7 Conclusions . . . 103

5 Optical Diffraction Tomography as a case of study 105
5.1 Introduction . . . 106
5.2 Description of NLODT-P model . . . 108
5.3 Experimental validation of the model . . . 113
5.4 GPU-Based implementation of the model . . . 117
5.4.1 Software resources . . . 118
5.4.2 Numerical methods and memory optimizations for computing Forward . . . 118
5.5 Computational experiments . . . 120
5.6 Conclusions . . . 121

6 Contributions and future work 123

Glossary 130

Bibliography 131

Publications arising from this thesis 150

Other publications produced during the elaboration of this thesis 152

List of Figures

1.1 Shared memory architecture. 2
1.2 Example of a GPU acting as coprocessor of a CPU. 3
1.3 CUDA programming model (source [130]). 5
1.4 Distributed memory architecture. 8
1.5 Multi-GPU cluster architecture. 9
1.6 Modern HPC architecture (multi-GPU cluster) and interfaces used by the programmer to optimize the program performance. 14
1.7 Example of the simulation of a physical model using MATLAB interface and two calls to MEX files to compute the most computationally demanding task of the model. 21
1.8 C/C++ MEX cycle (source [11]). 22
1.9 Roofline model for AMD Opteron X2 (source [157]). 25
1.10 Candidate BLAS operations for fusion in the CG method. Applying the fusion optimization to CG consists of performing Fusion 1−3 that correspond to fusing two dot operations, two saxpy operations and one saxpy and dot operations, respectively. 32
1.11 Candidate BLAS kernels for fusion in the BCG method. Applying the kernel fusion optimization to BCG consists of performing Fusion 1−4 that correspond to fusing two saxpy operations, two matrix vector products, three saxpy operations and two dot operations, respectively. 33

1.12 Candidate BLAS kernels for fusion in the BCG stabilized method. Applying the kernel fusion optimization to BCG stabilized consists of performing Fusion 1−3 that correspond to fusing two dot operations, two saxpy operations and two dot operations, respectively. 34
1.13 Structure of the Bullx cluster, which is comprised of four compute nodes (16 cores) and eight Tesla M2075 GPU devices. 37

2.1 The ELLR-T memory storage of the sparse matrix fulfilling the coalescence and alignment conditions for $T = 2$. 43
2.2 Storage format of the sparse and dense matrices and threads mapping of SSFastSpMM (Algorithm 2) for the SpMM operation on GPUs. . . . 48
2.3 Timeline of FastSpMM execution using two independent streams (Stream 0 and 1). 52
2.4 Performance of FastSpMM as function of L^* ($L^* = L_c = L_{GPU}$) for test sparse matrices on Tesla C2050 (left) and GTX480 (right). 56
2.5 Performance evaluation of the sparse matrices using the approaches: CUSPARSE, SetSpMVs, FastSpMM* and FastSpMM to compute SpMM on Tesla C2050 (top) and GTX480 (bottom). 61

3.1 Performance of BCG approaches ($CuBCG_{CS}$ and $CuBCG_{ET}$) on GPU Tesla C2050 using real (left)/complex (right) matrices and single/double precision (SP/DP). 74

4.1 Illustration of discretization stencil and the linear system (source [52]). . 81
4.2 Halos swapping among 4 processors (P0, P1, P2 and P3) where every processor has a chunk of $M + 2 \cdot D2 = M'$ and M is the local vector v. . 92
4.3 Percentage of the runtime for each call to function using 4GPUs (a) and 8GPUs (b). 98
4.4 Acceleration factors of operations of 2Ax, saxpies and dots routines with 4GPUs (a) and 8GPUs (b) versus the sequential time of these routines. 98

5.1 Matched-Filtering sample image of an isolated particle: 3D view of the surface draw by the voxels above 0.6 of the maximum value and 2D view at the central plane. 114
5.2 3D view of the particle distribution problem. 114

5.3 Gradient (left) and filtered gradient (right) computed at the initial iteration: 3D view of the surface at 0.6 the maximum value (top) and 2D views at $z = 57$ pixels (middle) and at $z = 104$ pixels (bottom). 115
5.4 3D view of the particle distribution problem. 116
5.5 Linear ODT image after computing the corresponding Match-filter of the particle distribution of Figure 5.4 for a three-in-line hologram configuration (left) and three illumination and one observation direction configuration (right). 116
5.6 Evolution of the runtime of NLODT-P using different values for Vol (from 200^3 to 280^3). The remaining parameters of the example are: $N_h = 3$ and $iterMax = 4$. 120

List of Tables

1.1 Parallelism levels provided by HPC platforms and interfaces. 11

1.2 Brief description of some algebraic software, classified from the most basic to the most advanced one (from BLAS1 to ScaLapack). 19

1.3 Brief description of some algebraic software using GPU devices. 19

1.4 Characteristics of the GPU platforms considered for the evaluation. (Sorted by peak performance in double precision). 36

2.1 Characteristics of test sparse matrices (denoted as A in our definition of SpMM), where n is the number of rows, nz is the total number of non-zero elements, Av and $Mnzr$ are the average and the maximum number of entries per row, respectively, and $NFLOP = 2n \cdot nz/2^{30}$ is the estimated GFlops. 54

2.2 Processing Time (seconds) of the SpMM operation using two configurations of the shared and L1 memory over the FastSpMM approach. Columns PT_s and PT_{L1} refers to the configuration with 48 KB of shared memory and with 48 KB of L1 cache, respectively. Column AF_{L1} identifies the Acceleration Factor ($AF_{L1} = TP_s/TP_{L1}$). Special matrices (dense and tridiagonal matrices) are typed in italics. 57

2.3 Profiling of SpMM based on FastSpMM* and FastSpMM on Tesla C2050. The following notation is used, Av and $Mnzr$: average and the maximum number of entries per row, respectively; PT: Processing Time; CT: Communication Time; Runt: total Runtime (all in seconds); % Com: percentage of Communication Time with respect to the total runtime for the FastSpMM* approach; AF: Acceleration Factor of the FastSpMM with respect to FastSpMM*. Special matrices (dense and tridiagonal matrices) are typed in italics. 59

2.4 Profiling of SpMM based on FastSpMM* and FastSpMM on GTX 480. The following notation is used, Av and $Mnzr$: average and the maximum number of entries per row, respectively; PT: Processing Time; CT: Communication Time; Runt: total Runtime (all in seconds); % Com: percentage of Communication Time with respect to the total runtime for the FastSpMM* approach; AF: Acceleration Factor of the FastSpMM with respect to FastSpMM*. Special matrices (dense and tridiagonal matrices) are typed in italics. 60

2.5 Runtime executions in seconds of the SpMM multicore version using MKL with 1, 2, 4 and 8 cores (1C, 2C, 4C and 8C) and Acceleration Factors (AF) of FastSpMM on Tesla and GTX480 ($AF\ Tesla$ and $AF\ GTX480$, respectively) over the MKL version with 8 cores 62

3.1 Characteristics of real and complex test matrices. 71

3.2 GFlops of 1000 iterations of both implementations of BCG (columns $CuBCG_{CS}$ and $CuBCG_{ET}$) using the two sets of matrices in single and double precision (typed in bold). IP column shows the GFlops achieved by inner products and Ap_{CS}, $A^T p'_{CS}$, Ap_{ET} and $A^T p'_{ET}$ columns show the GFlops measured for both kinds of SpMV using CUSPARSE and ELLR-T libraries, respectively. 73

3.3 Runtimes (s) of 1000 iterations of the BCG method based on ELLR-T on a GPU card Tesla C2050, in single (SP) and double precision (DP). Values in parentheses show the percentage of the total runtime which is spent in the calculation of the inner products. 75

3.4 Acceleration factor for the $CuBCG_{ET}$ method against the BCG multicore version using MKL with 1, 2, 4 and 8 cores (1C, 2C, 4C and 8C). The two sets of matrices have been considered in single and double precision (typed in bold). 76

4.1 Characteristics of the test matrices (complex and double precision numbers). 95

4.2 Profiling for 1000 iterations of the sequential code of the BCG to solve the 3D Helmholtz equation. Columns 2Ax, saxpies and dots represent the runtime (in seconds) for the calls to these routines and total runtime identifies the total execution runtime (s). 96

4.3 Range of GFlops for the set of test matrices considering 1000 iterations of the BCG-3DH with 4GPUs and 8GPUs. Additionally, values for sequential, 4CPUs and 8CPUs implementations are shown. Columns 2Ax, saxpies and dots show the range of GFlops achieved by theses operations. 97

4.4 Runtimes, in seconds, of 1000 iterations of the BCG-3DH using the CPU version with 1, 2, 4, 8 and 16 processors (1CPU, 2CPUs, 4CPUs, 8CPUs and 16CPUs) (Column Runt). Column %C identifies the percentage of the total runtime that consume the communication processes for every execution. 99

4.5 Runtimes, in seconds, of 1000 iterations of the BCG-3DH using the GPU version with 2, 4 and 8 GPU devices (2GPUs, 4GPUs and 8GPUs) (Column Runt). Column %C identifies the percentage of the total runtime consumed by the communication processes for every execution. 100

4.6 Acceleration factor (AF), where AF CPUs represents the AF of the implementation with 8 CPUs versus the sequential code (1 CPU core), AF GPUs identifies the AF of the use of GPU computing versus the implementation using 8 CPUs, and AF totalGPUs represents the AF of the GPU approach over the sequential code (1 CPU core). 101

4.7 Profiling of the resolution of the 3D Helmholtz Equation based on 1000 iterations of the BCG method using the hybrid version and two different configurations: 4GPUs+8CPUs and 8GPUs+8CPUs. Column Runt identifies the total runtime in seconds of the execution, GPU(s) and CPU(s) show the runtime of the GPU and the CPU, respectively, F column denotes the factors to balance the workload in the hybrid approach and, finally, %I represents the acceleration factor of the hybrid implementation versus the GPU version with 4 GPUs (for 4GPUs+8CPUs implementation) and 8 GPUs (for 8GPUs+8CPUs implementation). . . 102

5.1 Notation used in this chapter 111

5.2 Comparison of features of available CUDA-based numerical linear algebra packages that can be combined with the MATLAB environment. . . 118

5.3 Memory requirements (GB) for storing A using two formats: COO and CRF. 119

List of Algorithms

1 Pseudocode of ELLR-T algorithm for computing SpMV on GPUs for $T = 32$ . . . 43

2 Pseudocode of SSFastSpMM, a Single Step of FastSpMM, to compute SpMM on GPUs. It computes the product $C^{L_c} = AB^{L_c}$ where A is sparse matrix of dimensions $n \times m$, B^{L_c} and C^{L_c} are dense matrices of dimensions $m \times L_c$ and $n \times L_c$, respectively . . . 47

3 FastSpMM*, general scheme to compute $C = A \cdot B$ based on SSFastSpMM 50

4 FastSpMM: code with streaming computation CPU-GPU to compute the SpMM operation . . . 51

5 BiConjugate Gradient Method . . . 68

6 Preconditioned BCG-3DH Method . . . 87

7 Algorithm of the NLODT-P model. . . . 110

1 Introduction

This chapter discusses the basic foundations of the two main research areas concerning this thesis: high performance computing and mathematical issues related to the resolution of sparse linear systems of equations. A section is devoted to each of them, where the main issues related to the current work are described.

1.1 High Performance Computing issues

This section analyses the state of the art in High Performance Computing (HPC) and describes the development methodology designed to create HPC approaches for a set of real applications. This section covers aspects such as hardware and software models from the programmer's point of view, performance evaluation tools, description of the considered HPC development methodologies for scientific applications, as well as a summary of the characteristics of the set of parallel and distributed computers which have been used in the evaluation of the parallel algorithms proposed in this work.

1.1.1 High Performance Computing platforms

The increasing computational demand of the next generation applications has driven computer designers to adopt new approaches in designing and constructing large High Performance Computing platforms, sparking the development and deployment of new technologies. Those technologies include the use of multicore and/or many-core architecture such as GPUs or the modern Xeon Phi platforms and multi-GPU clusters to speed up algorithms with high computational requirements.

Since many years Flynn's taxonomy [65] has proven to be useful for the classification of high-performance computers. This classification is based on the different ways of managing instructions and data streams and comprises four main architectural classes: Single Instruction, Single Data (SISD); Single Instruction, Multiple Data (SIMD); Multiple Instruction, Single Data (MISD); and Multiple Instruction, Multiple Data (MIMD) [140].

In this subsection, the modern HPC architectures and their connections to the Flynn's taxonomy are described with a certain level of detail.

1.1.1.1 Shared memory multiprocessors

This architecture consists of several processors connected to the same bus, through which, they share a common memory device. The problem to solve is mapped over the available processors with the goal of minimizing the total execution runtime. With this model, shared variables of a program are available for any processor at anytime. Communications among processors are carried out by means of these shared variables, coordinating the accesses by means of synchronization processes that allow the system to solve data dependencies in the program. Shared memory systems can be both SIMD or MIMD. Single-CPU vector processors can be regarded as an example of the former, while the multi-CPU models of these machines are examples of the latter. We will sometimes use the abbreviations SM-SIMD and SM-MIMD for the two subclasses [140]. Figure 1.1 shows the traditional organization of an architecture based on shared memory.

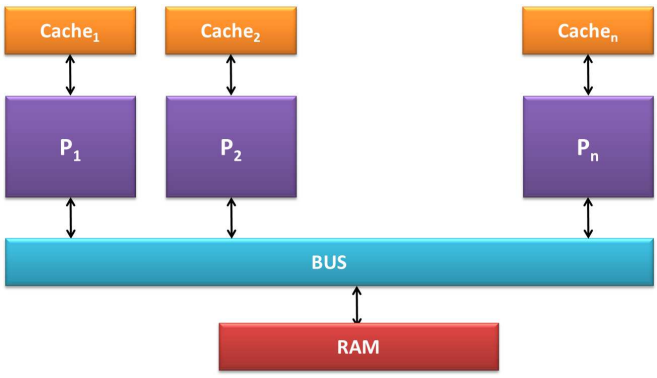


Figure 1.1: Shared memory architecture.

In this architecture a second classification is established depending on the way to access to the common memory space. Uniform Memory Access systems (UMA) provides the same latency for accessing to any memory word for any processor. On the other hand, in Non-Uniform Memory Access systems (NUMA) the accesses to certain memory words are much faster than the accesses to others depending on the processor that requests them. NUMA multiprocessors are more difficult to exploit, but their scalability is better than for UMA systems.

1.1.1.2 GPU

Graphics Processing Units (GPUs) are devices that act as coprocessors of CPUs (see Figure 1.2). Its great calculation power comes from the enormous quantity of computational resources that GPUs include.

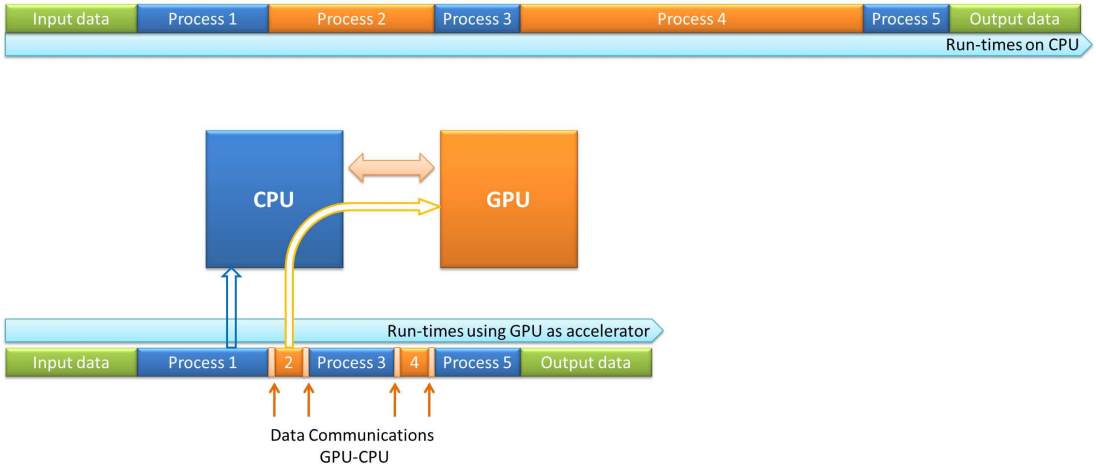


Figure 1.2: Example of a GPU acting as coprocessor of a CPU.

On GPU devices the area of the chip devoted to arithmetic calculations is maximized, whereas the area devoted to control is minimized. According to Flynn's taxonomy they are very near to SIMD systems since GPUs follow a vector programming model or Single Instruction, Multiple Threads (SIMT), in which thousands of threads collaborate for the resolution of the problem. The thread of the main execution is governed by the CPU, which makes calls to the parallel routines (kernels) that are executed in the GPU. This model has generated a new concept of programming known as General-Purpose computing on Graphics Processing Unit (GPGPU).

In the last years, the use of GPUs for general purpose applications has extraordinarily increased thanks to the availability of Application Programming Interfaces (APIs), such as Compute Unified Device Architecture (CUDA) [14] and OpenCL [10], which greatly facilitate the development of applications with GPUs. The use of GPUs for accelerating algorithms where sparse algebraic operations are involved, has been and is currently being studied by several research groups on the international stage [37, 43, 48, 148].

The GPU architecture consists of several processing units called Streaming Multiprocessor (SM) or multiprocessors, where every SM contains a particular number of Scalar Processors or cores that share the control logic and instructions cache according to the specific GPU architecture generation [98, 107]. Every SM contains:

1. A set of registers for Scalar Processors, whose size depends on the GPU architecture.

2. A space of read-write memory shared among all the Scalar Processors called shared memory.

3. Two areas of read-only memory (constant memory and texture memory) shared among all Scalar Processors of all SMs.

In addition, all SMs of the GPU share a global memory area named device memory. The global memory is DRAM GDDR (Graphics Double Data Rate) whose size depends on the GPU model. Each kernel is executed by a grid of thread blocks and each thread block contains a number of threads that are organized as SIMT groups called warps, which are simultaneously executed on SMs (see Figure 1.3). The total size of a block (BS) is defined by the programmer.

Theoretically, multiple grids can be available at the same time. Kernel invocations are also asynchronous. New GPU generations offer the possibility of running several kernels at the same time. This way, for algorithms which exhibit coarse-grained granularity, implementations on MIMD architectures are also possible. Hence, these new GPU generations could be classified as both SIMD and MIMD systems.

With the launch of the Fermi GPU in 2009, NVIDIA ushered in a new era in the HPC industry based on a hybrid computing model where CPUs and GPUs work together to solve computationally–intensive algorithms. In just a couple of years, NVIDIA

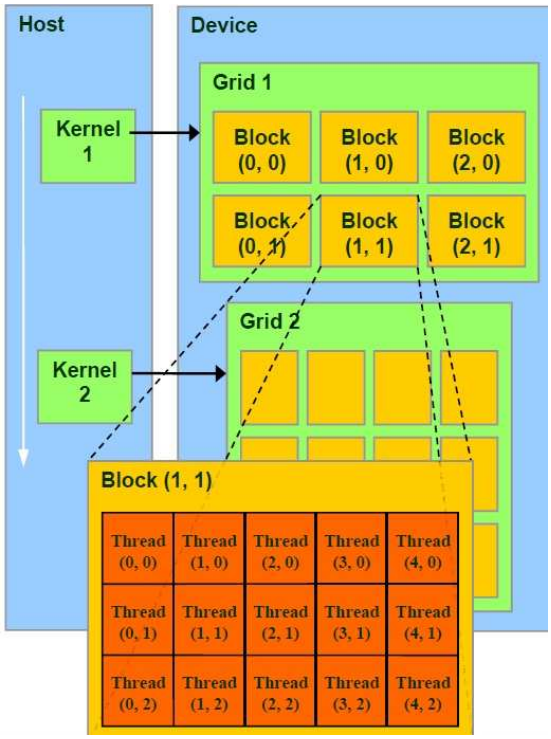


Figure 1.3: CUDA programming model (source [130]).

Fermi GPUs powers some of the fastest supercomputers in the world as well as tens of thousands of research clusters. The chip GT300, better known as Fermi architecture, contains from 14 to 16 SMs (depending on the model) with 32 Scalar Processors per SM, coming to a total from 448 to 480 Scalar Processors, obtaining peak performance values higher than 1.3 TFLOPS. Provided that every SM is massively multithreading, it can execute thousands of threads per application. A typical application can simultaneously execute between 5.000 – 12.000 threads. This Fermi architecture has been considered for all the experiments in this thesis.

The next generation of GPUs, which appeared in 2012, was the Kepler architecture (GK-codenamed chips). NVIDIA's Kepler was designed to vastly increase parallelism across the GPU. Unlike the old Fermi-class GPUs, which used Streaming Multiprocessors (SMs) of 32 cores each, Kepler had 192 cores in each of its next-generation Streaming Multiprocessors (SMXs).

Next generation of GPUs, referred as Maxwell, will appear at the end of 2014. Maxwell walks this trend back a bit, and returns to some design elements that Fermi

used but with a new design for the Streaming Multiprocessor (called SMM). Compared to any previous GPU in this price bracket, Maxwell has a much larger L2 cache. In Kepler, 192 cores shared a contiguous L1 and a separate "Unified Cache". With Maxwell, each pair of blocks within the SMM splits a combined L1/texture cache. According to NVIDIA, the new, larger L2 acts as a buffer for slower caches and for data sharing across the entire core [9].

Volta generation is scheduled to arrive after Maxwell GPUs (around 2016). NVIDIA plans to used stacked DRAM on future graphics chips in Volta GPUs. Volta GPUs will have access to up to 1TB per second of bandwidth by stacking the DRAM on top of the GPU itself, with a silica substrate between them (3D memory stacking) [2].

1.1.1.3 Xeon Phi

Intel Many Integrated Core Architecture or Intel MIC is a multiprocessor architecture developed by Intel incorporating earlier work on the Larrabee many core architecture, the Teraflops Research Chip multicore chip research project, and the Intel Single-chip Cloud Computer multicore microprocessor. In June 18, 2012, Intel announced that Xeon Phi will be the brand name used for all products based on their Many Integrated Core architecture.

Intel Xeon Phi platforms are coprocessors with architecture x86 orientated to the accomplishment of calculations in parallel processes. Every coprocessor obtains a higher performance than 1 Teraflop in double precision floating point.

Intel Xeon Phi coprocessors are designed to extend the reach of applications that have demonstrated the ability to fully utilize the scaling capabilities of Intel Xeon processor-based systems and to fully exploit available processor vector capabilities or memory bandwidth. For such applications, the Intel Xeon Phi coprocessors offer additional power-efficient scaling, vector support, and local memory bandwidth, while maintaining the programmability and support associated with Intel Xeon processors [81]. Getting minimal data movement supposes an algorithmic endeavor, but it can be eased through the higher bandwidth between memory and cores that is available on the Intel Xeon Phi coprocessors. This leads to parallel programming using the same programming languages and models across Intel products, which are generally also shared across all general-purpose processors in the industry.

The Intel Xeon Phi coprocessor is both generally programmable and tailored to tackle highly parallel problems. As such, it is ready to deal with very demanding parallel applications. These keys are not specifically for Intel Xeon Phi coprocessors but for any general-purpose parallel computer. The challenges of parallel computing are simple to enumerate: expose lots of parallelism and minimize communication. Thus, there are several reasons to expect that nearly all algorithms that work well on current GPGPUs can, with a minimal amount of restructuring, work equally well on the Intel Xeon Phi coprocessor. Additionally, algorithms requiring fine-grain concurrent control should be significantly easier to express on the coprocessor than on GPGPU. The main task in this type of architectures is to efficiently exploit the wide vector units, which are the most significant resources.

Here, it is important to highlight that we have explored several implementations of BLAS operations on Xeon Phi. However, they have not been reported in this thesis because this is part of our current research and future works.

1.1.1.4 Distributed memory cluster

The alternative to share the memory space among all the processors is the architecture based on distributed memory. This kind of systems refers to a multiple-processor computer system in which each processor has its own private memory. Computational tasks can only operate on local data, if a processor requires data from another processor's memory, it must exchange messages with the other processor. This system includes routines for the sending and receiving of messages, being implicit the coordination among the processors in the exchange. Every processor knows when it sends a message and the receiver knows when it receives it, however, if the sender needs confirmation from the receiver, the receiver can send a confirmation message.

As mentioned in [140], distributed memory systems may be either SIMD or MIMD. The first class of SIMD systems mentioned which operate in lock step, all have distributed memories associated with the processors. Distributed-memory MIMD systems exhibit a large variety in the topology of their connecting network. The details of this topology are largely hidden from the user, which is quite helpful with respect to portability of applications.

This architecture allows an unlimited scalability, but it has the disadvantage of the limitations of the interconnection network used to connect the different processors.

Few parallel applications can justify the use of an interconnection network of high performance computing due to its high cost. For this reason, the use of a standard set of servers interconnected by means of a local area network (clusters) has turned into the most widespread example of a system multiprocessor based on message passing. Figure 1.4 shows this type of architecture.

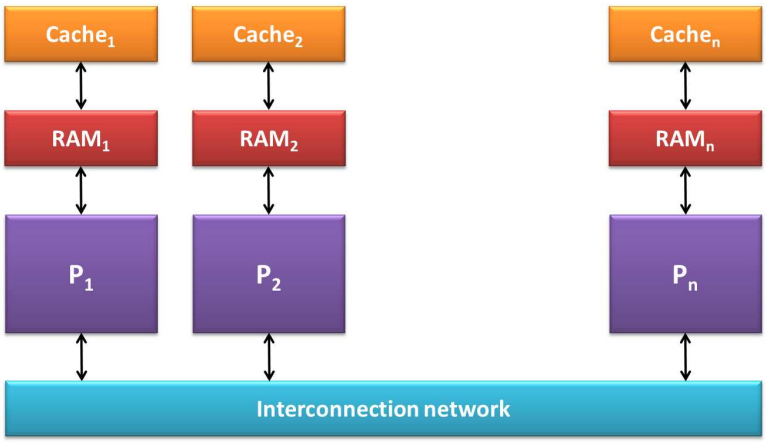


Figure 1.4: Distributed memory architecture.

Modern clusters can use some different types of computational units. A computational unit could be a general-purpose processor, a special-purpose processor (i.e. digital signal processor (DSP) or graphics processing unit (GPU)), a co-processor, a General Purpose GPU (GPGPU), or custom acceleration logic (application-specific integrated circuit (ASIC) or field-programmable gate array (FPGA)). So, current clusters are characterized by their heterogeneous processing elements since they include different kinds of processors with different instruction set architectures (ISAs). With the increasing adoption of GPUs in HPC, GPU devices are becoming part of some of the world's most powerful supercomputers and clusters [20].

A heterogeneous computing architecture based on GPUs can be characterized as a scalable cluster of shared memory nodes with multicore processors and PCI-attached GPUs, interconnected by a high-performance network fabric for fast, high-bandwidth inter-node communication. These heterogeneous clusters are referred as Multi-GPU clusters. Figure 1.5 shows a scheme of a multi-GPU cluster with n multicore processors of four cores and two GPUs per node.

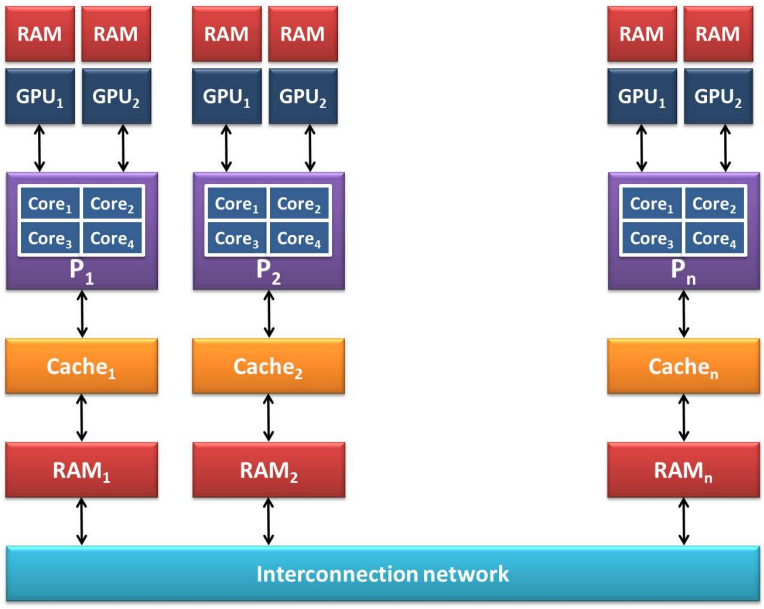


Figure 1.5: Multi-GPU cluster architecture.

Heterogeneous computation with this Multi-GPU clusters involves three general types of operational interactions. First, there are the interactions among nodes that take place through communications among processes. These involve libraries that implement message passing or global address space semantics. Second, there are the intra-node interactions among threads as part of the node's CPU multicore parallel execution. These involve shared memory programming and multithreaded runtime systems. Lastly, there are interactions among the nodes of the CPUs and the attached GPU devices. These involve DMA memory transfers to/from the GPU device over a PCI bus, launching of GPU kernels, and other functions provided by the device and driver libraries supporting the operations [103].

1.1.2 HPC development methodologies for scientific applications

The use of HPC is necessary when a problem demands more computational requirements than those supplied by a conventional laptop or workstation or it runs too slowly (because the algorithm is complex, the dataset is large, or data access is slow). However, to develop software for HPC is complicated and requires a good comprehension of algorithms, applications, and architectures. As a result, to efficiently design applications using HPC techniques and architectures requires to follow appropriate methodologies.

Applications should be executed on HPC resources by efficiently exploiting different levels of parallelism, i.e., the Instruction-Level Parallelism (ILP), data-based parallelism and/or task-based parallelism. Improvements of the performance for these kind of systems directly depends on the ability of the programmer to optimally exploit these three levels of parallelism. There is a wide variety of Application Programming Interfaces (APIs) that the programmer should confidently know how to manage in order to assemble all the parallelism levels.

Two main research directions have been taken to increase programmer's productivity and enhance the performance of their applications. The first alternative consists of completely relying on the compiler to automatically generate optimized codes for the target architecture. Some relevant advances have been achieved in the last decades, but they showed to be applicable only for very simple loops. One of the examples in this context is the automatic parallelization of loops implemented under Intel C Compiler (icc) [17].

The second alternative to ease the development of parallel applications consists of using parallel libraries that efficiently implement the most frequent operations on a wide range of target high-performance architectures, so that programmers can express their codes in terms of a set of optimized routines without worrying about the specific details of each architecture. In this line, the performance of a given application does not only depend on the programmers' skill to appropriately express his code in terms of the available routines, but also on the quality of the design and implementation of the set of basic operations to fully exploit the underlying architecture.

This subsection is devoted to describing different interfaces and optimization techniques used for developing HPC programs. Furthermore, some well-known algebraic libraries, that can make the development of HPC code easier, are discussed.

1.1.2.1 Parallel programming interfaces and models

One of the challenges to design efficient HPC codes is the exploitation of the different parallelism levels supplied by complex supercomputer architectures. From the programmers' point of view, HPC platforms are hierarchical systems where every level is related to a kind of parallelism and a specific programming interface. So, the HPC programmer

has to combine several interfaces since there is not a unified development interface. Table 1.1 summarizes the platforms and the interfaces for each level of parallelism (core, multicore and distributed system).

Table 1.1: Parallelism levels provided by HPC platforms and interfaces.

Processor Level	Program level	Platforms and Interfaces	
		CPU	GPU
Core	ILP	Optimization techniques/ compiler	Optimization techniques/ compiler
Multi/Many-core	Tasks/ data parallelism	OpenMP, PThreads	CUDA, OpenCL
Distributed system	Tasks/ data parallelism	MPI	

Two main kinds of granularity can be distinguished according to the size of the workload for every parallel process: coarse-grained and fine-grained. In the context of distributed systems, granularity is a qualitative measure of the ratio of computation to communication. While every processing element in a coarse-grained parallelism model calculates relatively big chunks of data, between consecutive communications, a fine-grained parallelism model computes very small pieces of code between consecutive communications. Fine-grained parallelism is used when communications among processing elements happen very frequently. Every type of parallelism of the applications will be exploited by particular resources of the HPC platforms.

As can be observed in Table 1.1, three main levels of parallelism can be distinguished in HPC. They are related to the computation on core, multicore and/or distributed system.

Let's assume that a core is the minimal processing element supplied by the HPC platform. Modern cores, based on superescalar architecture, can simultaneously execute a reduced set of instructions [121]. This parallelism, referred as Instruction-Level Parallelism (ILP), can be exploited to improve the performance of programs. When a particular program is executed, the exploitation of ILP strongly depends on its translation to assembly language and its memory management. Usually, HPC programmers

do not take into account this level because modern compilers add automatic optimizations. Nevertheless, to achieve an optimum ILP the programmer should facilitate the action of compilers by means of the development of regular codes with high locality memory access patterns.

In multicore and distributed systems, there are two perspectives, task and data parallelism, to extract the parallelism from the codes.

From the point of view of task parallelism, the key notion is that the programmer has to decompose the work into several tasks which can be easily mapped onto physical threads that are scheduled by the operating system.

The task of writing parallel programs can also be faced from the point of view of data parallelism, where tasks perform the same operation on different pieces of data [24].

The term data parallelism refers to the concurrency that is obtained when the same operation is applied to some or all elements of a data ensemble. A data-parallel program is a sequence of such operations. A parallel algorithm is obtained from a data-parallel program by applying domain decomposition techniques to the data structures. Computations are then partitioned, often according to the "owner computes" rule, in which the processor that "owns" a value is responsible for updating that value. Typically, the programmer is responsible for specifying the domain decomposition, but the compiler is in charge of partitioning the computation automatically [66].

Focusing our attention on "Processor Level" column in Table 1.1, we have defined three different elements (core, multicore and distributed system) and their corresponding type of parallelism at "Program Level" (ILP, Tasks or data parallelism). Each "Processor Level" makes use of specific interfaces which also depend on the platform (CPU / GPU) they are based on (see column "Platforms and Interfaces").

For the CPU architecture, at multi-core level, several optimizations techniques/compiler can be found. They will be described in the Subsection 1.1.2.2. At multicore level, to take advantage of the shared-memory feature of the nodes, two main parallel programming interfaces are widely used: Posix threads (Pthreads) [44] and OpenMP [124]. Task-parallel API is OpenMP which is mainly based on simple compiler directives used to guide mostly the parallelization of regular loops although the recently proposed extension [39] with a task-enqueuing mechanism extends its scope of application. Using OpenMP, multiple threads for a process can be easily created. These threads are dis-

tributed among all the cores of the CPU sharing the memory. So, there is no need to duplicate data nor to transfer information between threads.

For the case of distributed systems, Message Passing Model (MPI) is the standard for exchanging messages among processors which are in the same node or at different nodes [132]. MPI is particularly suitable to distribute memory systems where each processing unit has access only to a portion of the system memory and processes need to exchange data by message passing.

For the GPU architecture, at core and multicore processor levels several optimizations techniques/compiler can be applied. They will be described in the Subsection 1.1.2.2. Moreover, at the same levels, CUDA [14] and OpenCL [10] offer two different interfaces for programming GPUs. OpenCL is an open standard that can be used to program CPUs, GPUs, and other devices, while CUDA is specific to NVIDIA GPUs. Although OpenCL promises a portable language for GPU programming, its generality may entail a performance penalty. Finally, for the case of distributed systems, MPI is also the portable API for communicating data via messages (both point-to-point & collective) between distributed processes. In this way, MPI is frequently used in HPC to build applications that can scale on multi-GPU clusters.

1.1.2.2 Optimizing Code Performance

Bearing in mind all the considerations described in Table 1.1, Figure 1.6 shows a modern HPC architecture (multi-GPU cluster) and interfaces used from the programmers' point of view. It can be seen in Figure 1.6 that a HPC application can be executed into a multi-GPU cluster, where several parallelism levels can be exploited on core, multicore and/or many-core and distributed systems.

Focusing on the core level, the main optimizations for improving performance rely on compilers since it is important for programs to be independent of the specific platforms. It is thought, in general, that optimizations for a specific architecture should be done through the compiler. However, compilers do not always generate the most efficient assembly language code. Therefore, optimizations of the source code by programmers can improve the work of the compiler. Some of the optimizations to be applied to every CPU core can be described as follows:

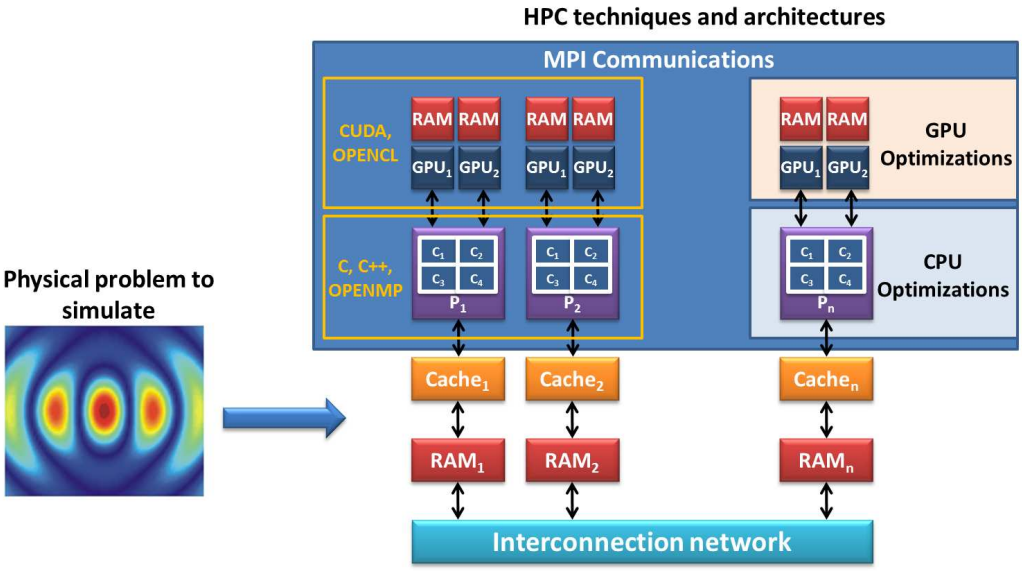


Figure 1.6: Modern HPC architecture (multi-GPU cluster) and interfaces used by the programmer to optimize the program performance.

1. **Using SSE instructions**. The current CPUs have special instruction sets (SSE, SSE2, SEE3) of SIMD type (Single Instruction, Multiple Data) that allow performing operations on data sets. A basic operation (addition, subtraction, multiplication, division, comparison, etc.) of four real numbers (in single precision) can be executed simultaneously. Another advantage is the straightforward translation to machine-code providing a higher performance.

2. **Array-Bounds Checks** which is a compiler optimization useful in programming languages or runtimes that enforce bounds checking, the practice of checking every index into an array to verify that the index is within the defined valid range of indexes. Its goal is to detect which of these indexing operations do not need to be validated at runtime, and eliminating those checks.

3. **Induction-variable** consists of identifying induction variables and relations among them and replacing expensive operations (e.g., multiplications) by cheap operations (e.g., additions).

4. **Code-block reordering** consists of changing the order of the basic blocks in a program for reducing the number of conditional branches and improving the

locality of references.

5. **Dead code elimination**. Removes instructions that will not affect the behaviour of the program, for example definitions which have no uses, called dead code. This reduces the size of the code and eliminates unnecessary computation.

6. **Loop optimizations**. Loops are often the performance bottlenecks of an application. The key to speed up the program is to make the loops run faster. Some of the most important loop optimizations are shown in [46] and can be summarized as follows:

   - Loop fission or loop distribution. Loop fission attempts to break a loop into multiple loops over the same index range but each taking only a part of the loop's body. This can improve locality of reference, both of the data being accessed in the loop and the code in the loop's body.
   - Loop fusion is another technique which attempts to reduce loop overhead. When two adjacent loops would iterate the same number of times (whether or not that number is known at compile time), their bodies can be combined as long as they make no reference to each other's data. This technique simplifies loops management. Although it could eventually suppose a loss of data locality, in platforms such as GPUs could increase the performance.
   - Loop unrolling. Unrolling duplicates the body of the loop multiple times, in order to decrease the number of times the loop condition is tested and the number of jumps, which hurt performance by impairing the instruction pipeline. A "fewer jumps" optimization. Completely unrolling a loop eliminates all overhead, but requires that the number of iterations be known at compile time.
   - Loop splitting. Loop splitting attempts to simplify a loop or eliminate dependencies by breaking it into multiple loops which have the same bodies but iterate over different contiguous portions of the index range. A useful special case is loop peeling, which can simplify a loop with a problematic first iteration by performing that iteration separately before entering the loop.

- Collapsing loops. Collapsing loops combines two or more nested loops into a single loop, producing longer vectors, less loop overhead, and better vector load balance.

Focusing our attention on the optimizations on the GPU architecture (referred as many-core level), several important optimizations are the following:

1. **Full implementation on GPU** to minimize the CPU-GPU transfers. This means that all the code can be implemented on GPU keeping data in the GPU memory. Therefore, the CPU-GPU communications are drastically reduced and only some particular results will be recovered from GPU at some time steps.

2. **Maximizing the occupancy of GPU**. Occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU or Streaming Multiprocessor (SM). To maximize the occupancy, it is essential to have the largest number of active warps in order to hide the latencies of memory access and maintain the hardware as busy as possible. The number of active warps depends on the registers required for the kernel, the GPU specifications and the number of threads per block. Using this optimization, the size of the block should be adjusted according to the registers of the kernel and the hardware specifications.

3. **Loop fusions**. The consideration of loop optimizations could have several advantages or disadvantages depending on the volume and type of the involved data, calculation and synchronization points. All the possible advantages can be summarized as follows [137]:

   - *Reduces the number of global barriers:* Merging two or more kernels that include explicit barriers, e.g., dot kernels, allows eliminating a number of these synchronization points.
   - *Reduces GPU global memory accesses:* Fusing multiple kernels may reduce data movements between on-chip and off-chip memories of the GPU. That is, fused kernels store intermediate data in on-chip memories (i.e., registers and shared memory) of the GPU, which can be accessed much faster than global memory.

- *Enhances locality at GPU memories level:* This benefit occurs only in the case where one or more arrays are used by multiple fused kernels. This fact eliminates multiple accesses to the same data.
- *Improves ILP and increasing the possibilities for compiler optimizations.* Fusing multiple kernels can achieve higher utilization and much more balanced demand for hardware resources which implies an increase of the workload volume per thread. In addition, a kernel of larger scope gives more opportunities for optimization to the compiler.
- *Reduces the number of data movements between CPU and GPU memories:* Fusing multiple kernels may reduce the number of movements of intermediate data through the Peripheral Component Interconnect express (PCIe). For example, merging multiple dot operations allows copying the final partial sums from GPU to CPU memories in one movement whereas non-fused dot kernels make as many movements as the number of dot kernels.
- *Increases concurrency:* Increasing the number of independent instructions by threads increases the workload volume per thread and consequently improves the overall management of the threads by the scheduler which can be translated into an enhancement of the concurrency, i.e., a higher number of active threads per active cycle.
- *Reduction of kernel launch overhead.* Fusing multiple kernels implies that a lower number of kernels are launched.

4. **Configuration of L1 cache**. From Fermi generations, the L1 cache size can be configured by the programmer. This flexibility of the architecture allows the programmer to set different configurations according to the particularities of the applications (higher use of L1 cache or shared memory). In the case of Fermi architectures, two configurations are possible: 48 KB of shared memory and 16 KB of L1 cache (the default option), or 16 KB of shared memory and 48 KB of L1 cache memory are possible [139]. However, the newest GPU Kepler architecture offers a reconfigurable L1 cache per Streaming Multiprocessor with different cache size and cache associativity [125].

1.1.2.3 Algebraic libraries

Apart from the development of efficient codes based on the ability of the programmer to optimally exploit the aforementioned three levels of parallelism by means of the use of optimizations in the HPC platforms, libraries are presented as another alternative to enhance the performance and make easier the development of HPC applications.

The methodology of developing HPC applications supported by libraries is based on the concept of abstraction. This methodology is useful in abstracting the complexity of the computation by means of several calls to well known libraries described in the literature. Although, there is a wide variety of libraries, our interest in this thesis is focused on algebraic libraries.

The last decades have known a rapid development of algebraic libraries for the efficient parallel solution of a wide variety of mathematical problems. These libraries provide subprograms that have been implemented and tested on a number of parallel computers and, very often, they are very high quality both from the point of view of numerical properties as well as the parallel performance. Thus, the use of these libraries is essential in the context of scientific parallel implementations. This approach can be called libraries–based parallelization.

In other words, libraries–based parallelization consists of integrating optimized blocks in the code, where optimized blocks are subprograms or routines that can vary from a simple product vector–vector to a solver that implements an iterative or a direct method, preconditioners, etc. There exists a vast number of algebraic software (see reference [53]).

Currently, Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK) are the most extended libraries. They are used as baseline for the development of high level libraries in this field. Table 1.2 gives a brief description of the most important low–level algebraic software. Furthermore, it is possible to find specific libraries BLAS and LAPACK which are optimized for certain processor types such as Math Kernel Library (MKL) of Intel [1].

In the last years, the number of new libraries which can efficiently exploit different levels of parallelism has considerably increased. This is the consequence of the appearance of new heterogeneous systems and the inclusion of accelerators such as

Table 1.2: Brief description of some algebraic software, classified from the most basic to the most advanced one (from BLAS1 to ScaLapack).

Library	Functionalities	Based on	For architectures
PBLAS	Parallel vec x vec, mat x vec & mat x mat operations	BLACS	distributed memory
BLACS	Dense linear algebraic problems	BLAS, MPI & PVM	machine-independent
LAPACK	Dense linear algebraic problems	BLAS, LINPACK & EISPACK	shared memory
EISPACK	Dense linear eigenproblem	BLAS 1	shared memory
LINPACK	Dense linear eigenproblem	BLAS 1	shared memory
BLAS 3	mat x mat operations	BLAS 2	shared memory
BLAS 2	mat x vec operations	BLAS 1	shared memory
BLAS 1	vec x vec operations	-	shared memory

GPUs. Focusing our attention on the algebraic software where GPU devices are involved, Table 1.3 shows some of the most commonly used libraries supplied by CUDA (CUSP [16], CUBLAS [4], CUSPARSE [5], CULA [15] and MAGMA [18]) for solving algebraic operations on GPUs and multi-GPUs clusters.

Table 1.3: Brief description of some algebraic software using GPU devices.

Library	Functionalities	Based on	For architectures
MAGMA	Dense linear algebraic problems	BLAS, LINPACK & EISPACK	heterogeneous/hybrid architectures
CULA	Dense and Sparse linear algebraic problems	BLAS, LINPACK & EISPACK	GPU architectures
CUBLAS (CUSPARSE)	Dense (Sparse) linear algebraic problems	BLAS	GPU architectures
CUSP	Sparse linear algebraic problems	BLAS	GPU

The basic libraries provide subprograms for the most frequent operations and solvers that exploit the parallelism at processor level on single processor high-performance computers, in addition they are usually provided with the compiler. However, most advanced libraries like MUMPS [22], SuperLU [97] and PETSc [12] offer an high level of abstraction by providing their own environment, i.e., data structures, solvers, storage formats, etc.

Besides the basic and the advanced algebraic libraries aforementioned, interactive mathematical tools can be understood as stand-alone applications that can be used for numerical analysis, data analysis, and display. These tools have both a GUI and a command-line interface. Among the most important interactive math tools we can highlight: Mathematica [19], Octave [58] and MATLAB [6]. MATLAB is one of most important software used in this thesis, so a deeply description of CUDA integration within MATLAB is carried out in the next subsection.

1.1.2.4 MATLAB

MATLAB is a high-level language and interactive environment for numerical computation, visualization, and programming. MATLAB can be used for the development of a wide range of applications, including signal processing and communications, image and video processing, control systems, test and measurement, computational finance, and computational biology. More than a million engineers and scientists in industry and academia use MATLAB, the language of technical computing. Although MATLAB is a complete, self-contained environment for programming and manipulating data, it is often useful to interact with data and programs external to the MATLAB environment. MATLAB provides an API to support these external interfaces using user defined C or Fortran subroutines from MATLAB as if they were built-in functions [6].

MATLAB interface can be used to perform behind-the-scenes parallel computations on the GPU. The combination of high arithmetic and memory bandwidth with the programmability provided by Compute Unified Device Architecture (CUDA), makes it very suitable for High Performance Computing.

There are several toolboxes/libraries capable of taking advantage of the sheer power of GPUs for Algebra from MATLAB interface, such as Jacket [13] from AccelerEyes and GPUmat [21] from GPYou. These toolboxes are independent resources that can be easily integrated by the user into MATLAB. Jacket supplies a library for developers called ArrayFire for the exploitation of the GPU in a transparent way with high level programming languages, such as C, C++ or Fortran. Jacket has two versions: the basic one (computation with only one GPU) and the professional one (multi-GPUs). Nevertheless, nowadays in both toolboxes, Jacket and GPUmat, there is a lack of algebraic routines in which sparse and complex matrices are involved. Moreover, there

are strategies to call specific C, C++ or Fortran routines or even CUDA kernels from MATLAB interface. The most widespread strategy are MEX-files.

A MEX-file (also written as MEX file) provides an interface between MATLAB and subroutines written in C, C++ or Fortran. When compiled, MEX files are dynamically loaded allowing non-MATLAB code to be invoked from within MATLAB as if it were a built-in function. To support the development of MEX files, MATLAB offers external interface functions that facilitate the transfer of data between MEX files and the workspace [11]. These MEX files could be used within CUDA by using tools provided by NVIDIA. Figure 1.7 shows an example of the simulation of a physical model using MATLAB and two calls to MEX files to compute the most computationally demanding task of the model.

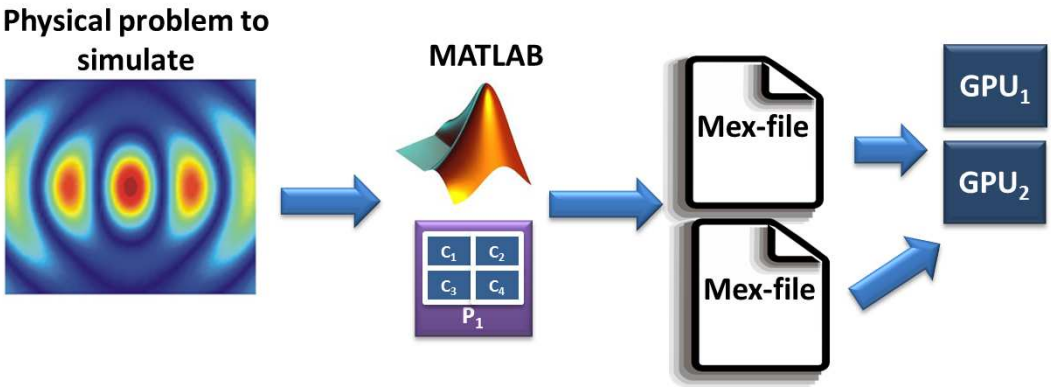


Figure 1.7: Example of the simulation of a physical model using MATLAB interface and two calls to MEX files to compute the most computationally demanding task of the model.

The source code for a MEX file consists of two different parts: the computational routine and the gateway routine. The computational routine contains the code for performing the computations. In the CUDA MEX scenario this would most likely be the computation kernel that is executed on the device. The gateway routine, which interfaces the computational routine with MATLAB by the entry point mexFunction and its parameters *prhs*, *nrhs*, *plhs*, *nlhs*. Where, *prhs* is an array of right-hand input arguments, *nrhs* is the number of right-hand input arguments, *plhs* is an array of left-hand output arguments, and *nlhs* is the number of left-hand output arguments. The gateway calls the computational routine as a subroutine.

The name of the gateway routine must always be mexFunction. In the gateway routine, the data access is through the mxArray structure and then this data is modified in the C computational subroutine. For MATLAB to recognize output from the MEX file, a pointer of type mxArray is to be set to the data returned by the computational routine or computational kernel, in the case of CUDA.

Figure 1.8 shows the C/C++ MEX Cycle where three main steps can be observed: (1) How inputs enter a MEX file; (2) What functions the gateway routine performs; and (3) How outputs return to MATLAB.

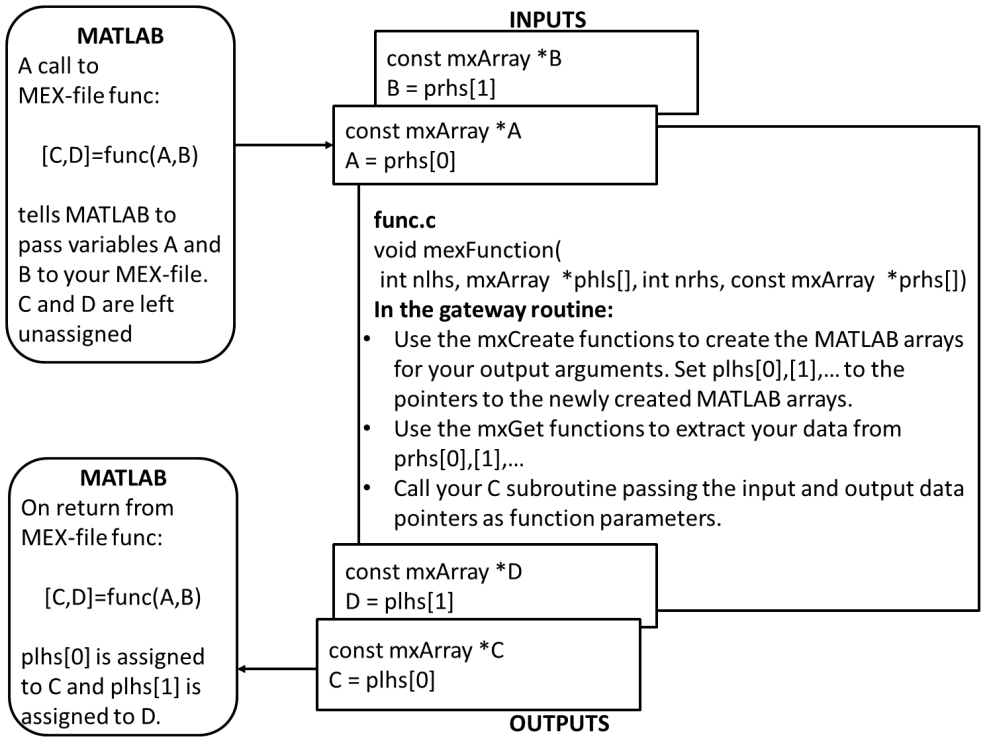


Figure 1.8: C/C++ MEX cycle (source [11]).

In Chapter 5, the integration of MATLAB and GPU computing has been used to accelerate a model based on Optical Diffraction Tomography. MEX files are used to invoke code on the GPU and to handle the data transfer between the host and GPU.

1.1.3 Performance metrics

The exploitation of the parallelism is an increasingly common approach for improving the performance of computer systems. In terms of hardware, this typically means providing multiple, simultaneously active architectures. In terms of software, this typically means structuring a program as a set of largely independent subtasks that can be executed at the same time [57].

1.1.3.1 Speed up and efficiency

For evaluating a parallel system running a parallel program, two performance measures of particular interest are *speed up* and *efficiency*. *Speed up* is defined for each number of processors P as the ratio of the elapsed time when executing a program on a single processor (the single processor execution time) to the execution time when P processors are available. In the notation that we shall use throughout this thesis:

$$Sp(P) = \frac{T_1}{T_P}. \tag{1.1}$$

In 1967, Amdahl gave a simple bound on the speed up that could be obtained by parallel processing as a function of the fraction of sequential code in a computation. This bound has proven usefulness in shaping our understanding of parallel systems because it strikes a useful balance between simplicity and precision.

Efficiency is defined as the average utilization of the p allocated processors. In general, the relationship between *efficiency* and *speed up* is given by the following equation:

$$Ef(P) = \frac{Sp(P)}{P}. \tag{1.2}$$

Ignoring I/O, the efficiency of a single processor system is 1. If efficiency remains at 1 as processors are added, a *linear speed up* is reached.

1.1.3.2 Performance bounds: Roofline model

The Roofline model, proposed in [157], provides a visual estimation of the performance bounds for processors according to their resources and the characteristics of the programs which are being executed. The hypothesis of this model points to the main bottleneck for the architectures is either the connection between processor and memory or

the Peak floating point performance. To quantify the performance bounds the parameter "operational intensity" is introduced. The Roofline model proposes "operational intensity" instead of "arithmetic intensity" related to the algorithms or applications to be programmed. It is defined as operations per byte of DRAM traffic, where this traffic is related to the caches and memory rather than between the processor and the caches. Thus, operational intensity predicts the DRAM bandwidth needed by a routine on a particular computer, and programmers can modify the operational intensity by changing the memory access pattern in their programs. So, the Roofline sets an upper bound on performance of a routine depending on the kernel's operational intensity according to the following expression:

$$\text{Attainable } \frac{\text{GFlops}}{\text{sec}} = min \begin{cases} \text{Peak Floating Point Performance,} \\ \text{Peak Memory Bandwidth} \times \text{Operational Intensity} \end{cases} \tag{1.3}$$

The proposed Roofline model ties together floating-point performance, operational intensity, and memory performance in a 2D graph. So, two regions in this graph can be distinguish: the first region is defined by the operational intensity values which have a performance bounded by the peak memory bandwidth; the second region is related to the operational intensity values whose performance is bounded by the peak floating-point performance of the computer. Consequently, if the operational intensity of a program is in the first region it means its performance is memory bound and if it is in the second region it is compute bound.

Figure 1.9 outlines the model for a 2.2 GHz AMD Opteron X2 model 2214 in a dual-socket system. The graph is on a log-log scale. The y-axis is attainable floating-point performance. The x-axis is operational intensity, varying from 0.25 Flops/DRAM byte-accessed to 16 Flops/DRAM byte-accessed. The slope of the linear function is related to the peak memory bandwidth where this parameter is the steady-state bandwidth potential of the memory in a computer, not the pin bandwidth of the DRAM chips. Note that in such figure, a routine with operational intensity higher than 1.0 Flops/Byte is compute-bound and a routine with operational intensity lower than 1.0 Flops/Byte is memory-bound.

Roofline model proposes a simple approach to estimate the performance limits for a particular combination of routine/processor. Recently, it has been widely used in

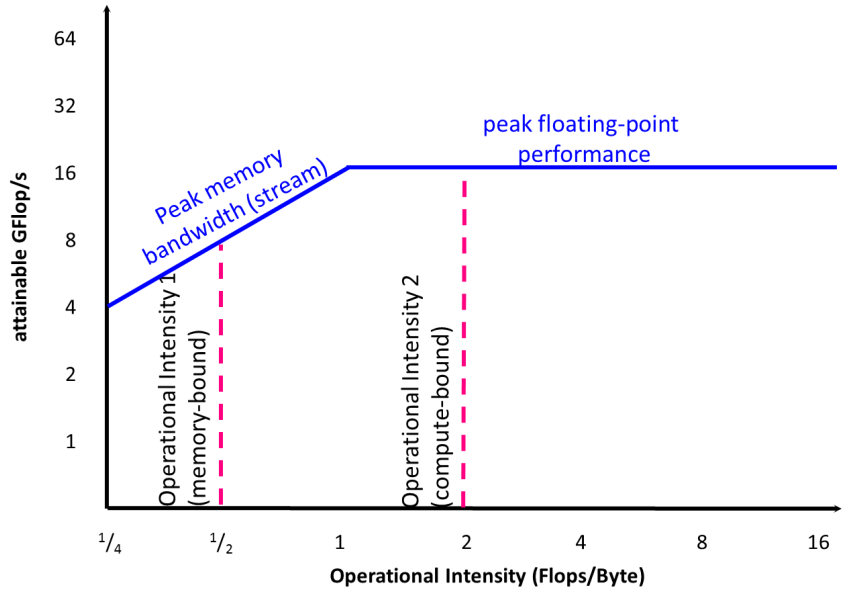


Figure 1.9: Roofline model for AMD Opteron X2 (source [157]).

the literature, however it is necessary to underline that it provides an estimation of performance bound, not a performance prediction.

1.2 Mathematical issues

A wide variety of mathematical models of physical phenomena is described by systems of Partial Differential Equations (PDEs). In most cases, it is impossible to obtain analytical solutions of these PDEs, and researchers have to resort to numerical methods, i.e., finite difference methods, finite element techniques, finite volume schemes, moment method etc., using computational techniques, where both the space and time variables are ultimately discretized, thus transforming a continuum problem into a discrete one at selected time levels and grid locations. The result of such a discretization is a linear system of algebraic equations which is the one solved numerically on computers [136].

One of the most important applications of numerical linear algebra is the resolution of these linear system of equations $Ax = b$ [104, 135]. When the linear system in question arises due to the discretisation of a partial differential equation (PDE) or a coupled system of PDEs, then the system matrix inherits many features from the underlying PDE operator and the chosen discretisation scheme. Solving such linear systems

usually require significant computational resources and the use of High Performance Computing techniques is necessary.

This section reviews the most important issues with relation to the resolution of sparse linear systems of equations used in this thesis.

1.2.1 Solution of sparse linear systems of equations

A sparse linear system can be expressed as $Ax = b$, where A is the sparse matrix of the known coefficients of dimension $n \times n$, b is a column vector of n values and x is a column vector of n unknown variables, where $A \epsilon \mathbb{R}^{n \times n}(\mathbb{C}^{n \times n})$ and x, $b \epsilon \mathbb{R}^{n}(\mathbb{C}^{n})$.

When the linear system of equations comes from the discretization of PDEs, they usually are sparse. If the matrix involved in the linear system of equation is sparse, there is only a relatively small number of non-zero entries. Unlike general methods, algorithms solving sparse systems are designed to handle the outweighing amount of zero elements effectively. Each method suits different types of sparse matrices. The precise pattern of sparsity is determined according to the position of the non-zero elements in the matrix. Special subclass of algorithms are those concentrating on band diagonal matrices, i.e. those with non-zero elements around the diagonal [83].

Methods for solving linear systems of equations can be divided in two groups:

1. *Direct methods.* Direct methods are based on the factorization of the sparse matrix A to translate the linear system in another with a resolution format much simpler. During the factorization, an element of the matrix with an initial null value could have a different value from zero; it suffers then a process of filling (fill). The more elements suffer a process of filling, the more operations the algorithm executes, therefore the computational load considerably increases. For this reason, direct methods need more memory requirements (because of the filling, apart from the original system, every new non zero entry has also to be stored). It supposes an important drawback for using direct methods in a wide variety of applications when the size of A is very high. Despite this difficulty, there are different factorization methods of a matrix, being one of the most commonly used the LU factorization [55, 97, 159], which have been adapted to the factorization of sparse matrices decreasing the filling process as far as possible. Presuming

all operations being performed exact, the gained result would be absolutely exact. Of course, because the performed computations are performed with rounding intermediate results, the final result is of limited exactness [51].

2. *Iterative Methods.* The iterative methods for solving general, large sparse linear systems have been gaining popularity in many areas of scientific computing. In the past, direct solution methods were often preferred to iterative methods in real applications because of their robustness and predictable behavior. However, a number of efficient iterative solvers were discovered and the increased need for solving very large linear systems triggered a noticeable and rapid shift toward iterative techniques in many applications. Currently, three-dimensional models are commonplace and iterative methods are almost mandatory. The memory and the computational requirements for solving three-dimensional Partial Differential Equations, or two-dimensional ones involving many degrees of freedom per point, may seriously challenge the most efficient direct solvers available today. Also, iterative methods are becoming more popular because they are easier to implement efficiently on high-performance computers than direct methods [127]. An inherent strategy to reduce the number of iterations of these methods is the use of preconditioners. These preconditioners are matrices that transform the original system and are specially relevant for ill-conditioned problems. An extensive set of literature about preconditioners can be found in [127].

   Iterative methods can be classified as:

   - Stationary methods. They are the simplest and the easiest to implement, but in general less efficient than the non stationary methods [85]. They are based on a relaxation of coordinates approach, beginning with an approximate solution and modifying the components of the approximation until the convergence is reached.

     Some stationary iterative methods are the following [144, 160]:

     – Jacobi. The Jacobi method is a method of solving a matrix equation on a matrix that has non-zero elements along its main diagonal [42]. Each diagonal element is solved for, and an approximate value plugged in. The process is then iterated until it converges. This algorithm is a

stripped-down version of the Jacobi transformation method of matrix diagonalization.

- Gauss Seidel. Method similar to the previous one excepting that it uses the updated values of the solution as soon as they are available. In general, it converges faster that Jacobi method [127].
- SOR (Successive Over-Relaxation). It comes from the method Gauss-Seidel method adding an extrapolation parameter w. With a correct choice of w, the method converges faster than Gauss-Seidel in one order of magnitude [127].
- SSOR (Symmetric Successive Over-Relaxation). This method does not have advantages as iterative method with regard to the SOR method. However, it is very useful as preconditioner for non-stationary methods [31].

- Non stationary methods. These methods are more complex than the stationary ones but highly efficient [126]. They differ from the stationary methods in which the computations involve information that changes at every iteration. Nowadays, the most popular belong to the set of the Krylov's subspace. Krylov's subspace $K^i(A, r_o)$ of i dimension, associated with a linear system $Ax = b$, for an initial solution vector x_0 and a residue vector $r_0 = b - Ax_0$ is defined as the subspace covered by the vectors $r_0, Ar_0, A^2r_0, \ldots, A^{i-1}r_0$. Depending on the characteristics of the matrix that defines the system of equations, we can classify these methods under several groups:
  - For symmetric positive definite matrices, the Conjugate Gradient (CG) is the most suitable [76, 91]. CG uses a sequence of orthogonal vectors x_i for which $(x_i - x)^T A(x_i - x)$ is minimized for all the vectors of the current Krylov space $K^i(A, r_0)$.
  - If the matrix is symmetric but is not positive definite, the Lanczos or MINRES methods can be considered [120]. In the MINRES methods, the elements $x_i \, \epsilon \, K^i(A, r_0)$ are determined by minimizing the quadratic norm of the residues $||b - Ax_i||^2$, whereas in Lanczos's methods the elements x_i are determined by the residues $b - Ax_i$ perpendicular to

the Krylov's subspace. In these cases, it is necessary to store the whole sequence, which carries a high consumption of memory.

- If the matrix is non symmetric, in general, it is not possible to determine an ideal set of solutions $x_i \epsilon K^i(A, r_0)$ with few sequences of vectors. Nevertheless, we can calculate the set of vectors $x_i \epsilon K^i(A, r_0)$ for those in which the condition $b - Ax_i \perp K^i(A^T, r_0)$ was reached. This way, two sequences of vectors are generated, one for the coefficients matrix A and another for A^T, and instead of the orthogonalization of every sequence, they do it mutually. This method is called BiConjugate Gradient (BCG). It requires a limited storage although the convergence could be irregular. In the BCG method, operations with A^T can be replaced by operations with A bearing in mind that $\langle A^T x, y\rangle = \langle x, Ay\rangle$, where the operator $\langle , \rangle$ represents the dot of two vectors. In some cases A^T can be replaced by A allowing to expand Krylov's subspace and to find better approximations to the solution with the same computational cost per iteration. This idea leads to iterative methods known as hybrid methods: the Quadratic Conjugate Gradient (QCG), the BiConjugate Gradient Stabilized (BCGSTAB) [141], transpose-free QMR (TFQMR) [67], etc. A variant of the BCG method is the Quasi-minimal Residue (QMR), which uses a least squares solver and an updated solution of the BCG, smoothing the behavior of the convergence and doing these methods more robust.
- If A is non-symmetric, we can also calculate the sequence of vectors $x_i \epsilon K^i(A, r_0)$ for obtaining minimal residues using an Euclidean norm (least squares). This method is called Generalized Minimal Residual method (GMRES) [128, 154]. This implementation needs to store the whole sequence, which leads to high memory requirements. Another version of the GMRES is the Flexible Generalized Minimal Residual method (FGMRES) which allows the preconditioning to vary at every step.

There are several applications in which direct methods have been commonly used against the iterative methods, because of the robustness and accuracy of the solutions. However, when the linear systems of equations come from PDEs, the discretization used

has a wide variety of points, therefore solving linear systems of equations turns out time and memory consuming approaches, and direct solvers often become ineffective. On the other hand, iterative methods have proven to be more effective with the appearance of methods such as the Conjugate Gradient, which combined with iterations over Krylov Subspace can supply easy and efficient general purpose procedures, achieving the quality of direct methods. Furthermore, iterative methods have also been characterized as being simpler to implement over high performance computing processors than direct methods [127].

1.2.1.1 Computational study of Krylov methods

Due to the advantages of the Krylov methods, among them their simplicity for being implemented on parallel computers, they are being widely used in HPC environments during the last few years. In this thesis we have focused our attention on the study of methods based on the Krylov subspace. To get an approximate solution at every step of the iterative process, these methods use projection processes onto Krylov-subspaces [127]. Krylov subspace methods form an orthogonal base of the sequence of powers of the matrix for the initial residue (Krylov's sequence). The approximations to the solution are formed by minimizing the residual value in the formed subspace.

Usually, when developing or porting iterative solvers, programmers use basic libraries such as the aforementioned in Section 1.1.2.3. Indeed, programmers implement iterative methods for the multicore/GPU using these libraries as baseline but in most cases the achieved performance is far from the optimum.

Based on our experience, the use of several isolated routines or kernels of different complexities degrades the performance on CPU and GPU architectures [113]. This degradation can be very important for iterative methods that solve linear systems of equations since they include several vector−vector operations (level 1 BLAS) and a small number of sparse matrix−vector products (level 2 BLAS). This is mainly due to the fact that these operations are memory bound and have a very low operational intensity which reduce their opportunities to obtain a balanced overlapping between memory accesses and computation [70]. Consequently, level 1 BLAS become the operations that dominate the overall runtime of Krylov methods.

Our approach to this problem is to design new routines by combining or fusing multiple BLAS operations to alleviate the memory bottleneck and to increase the op-

erational intensity. So that they can benefit from the optimization based on fusion as explained in Section 1.1.2.2.

After an extended analysis of a large set of sparse solvers based on Krylov subspace methods and all their variants, several operations which are fully independent can be identified and they can be merged together into one single routine. In particular, multiple saxpy operations, multiple dot operations and multiple matrix vector operations are usually performed in a specific order and can be merged together into one larger procedure [137].

Next, a methodology to identify elemental routines that could be fused is illustrated. So, three Krylov methods have been analyzed: the Conjugate Gradient Method, the BiConjugate Gradient Method and the Stabilized BiConjugate Gradient Method; they can be considered as representative Krylov methods to evaluate the benefits of kernel fusion optimizations on HPC platforms.

The **Conjugate Gradient** (CG) method is a nonstationary iterative method to solve linear systems of equations $Ax = b$, where A is a sparse symmetric positive definite matrix; b indicates the independent term and x is the unknown vector ($A \in R^{n \times n}$ and $x, b \in R^n$) [76, 91].

As can be observed in Figure 1.10, in each iteration of the method, seven vector−vector operations (four dot operations and three saxpy operations) and one SpMV are computed. The computational cost associated with the SpMV product increases with respect to the set of vector−vector operations as the ratio nz/n increases, where nz and n denote the number of non-zero elements and the number of rows and columns of A, respectively.

Figure 1.10 shows CG algorithm and the potential candidate vector−vector operations that can benefit from the optimization based on fusion. This figure also represents the data dependence graph of the main iteration of CG after a reordering process, where the candidate BLAS operations for fusion are shown in rectangles. The BLAS operations that can be fused in CG are:

- two dot operations,
- two saxpy operations,
- one saxpy operation and one dot operation.

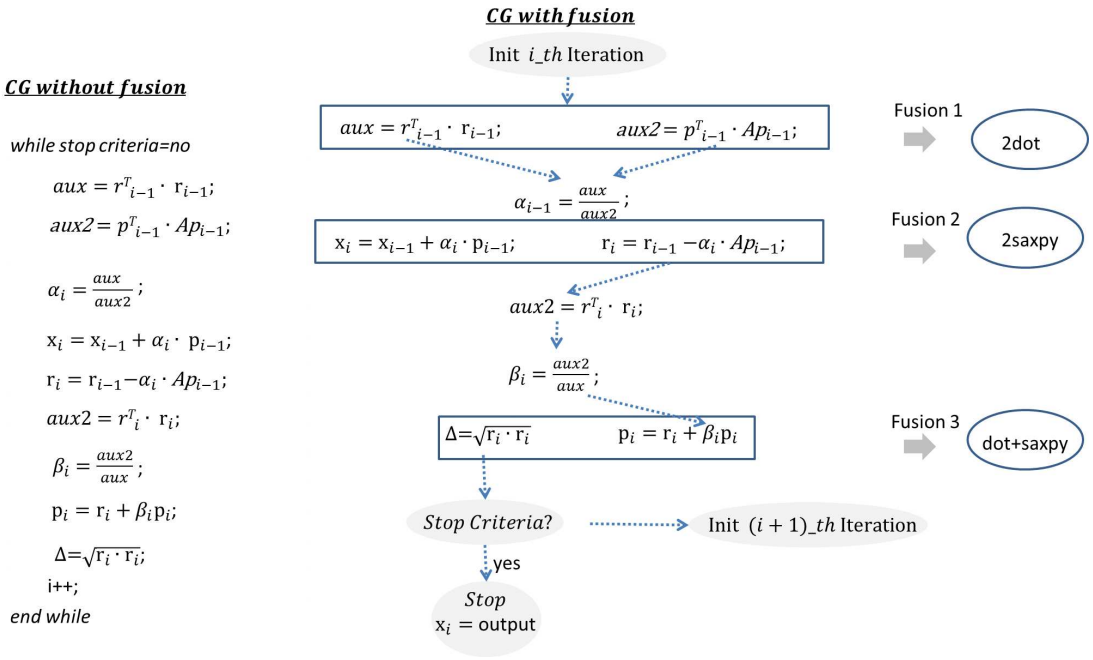


Figure 1.10: Candidate BLAS operations for fusion in the CG method. Applying the fusion optimization to CG consists of performing Fusion 1−3 that correspond to fusing two dot operations, two saxpy operations and one saxpy and dot operations, respectively.

Each iteration of CG is expressed using a set of three fused operations. The main benefits of expressing each iteration of CG with less routines thanks to the fusion of more elemental kernels are: the reduction of the number of global barriers, the reduction of GPU global memory accesses, improvement of locality at GPU memory levels, improvement of ILP and increases possibilities for compiler optimizations, reduction of the number of data movements between CPU and GPU memories and an increase of the concurrency.

Another method where this methodology can be applied is the **BiConjugate Gradient Method** (BCG). It is a nonstationary iterative method to solve linear systems of equations $Ax = b$, where the matrix $A \in C^{n \times n}$ can be non-symmetric [91]. Figure 1.11 shows the BCG method in a schematic way, where at every iteration eight vector−vector operations (three dot operations and five saxpy operations) and two SpMVs are computed. The computational cost associated with SpMV products increases with respect to the set of vector−vector operations as the ratio nz/n increases.

Figure 1.11 shows BCG algorithm and the potential candidate vector−vector opera-

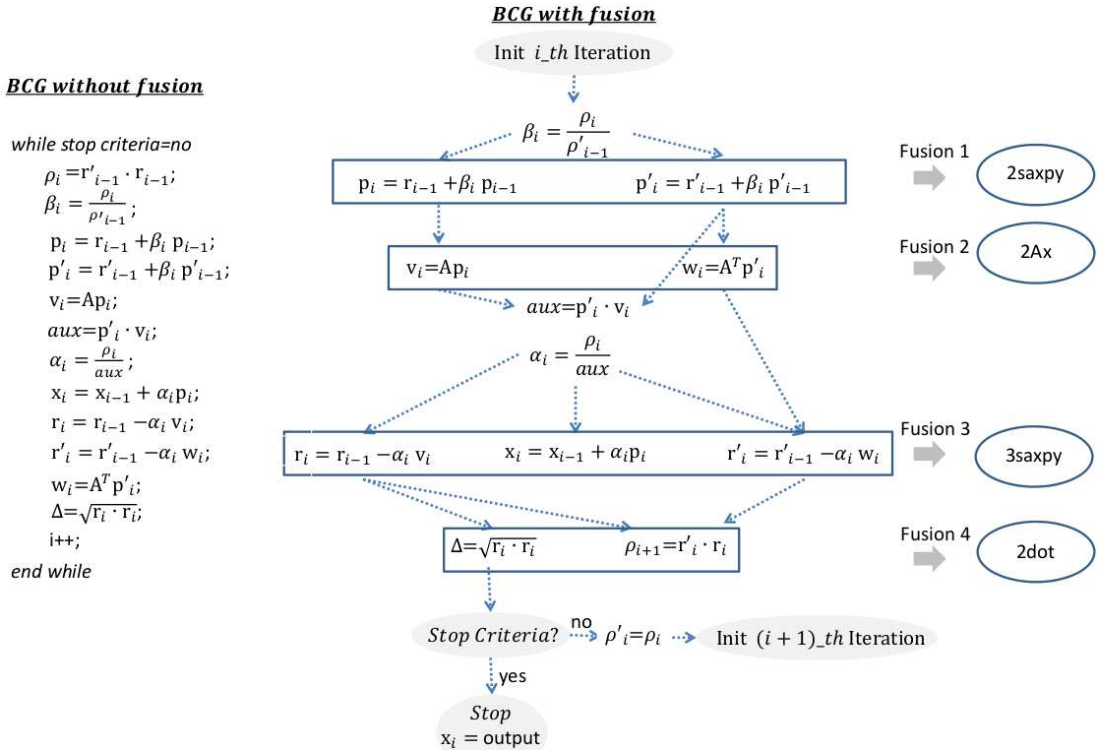


Figure 1.11: Candidate BLAS kernels for fusion in the BCG method. Applying the kernel fusion optimization to BCG consists of performing Fusion 1−4 that correspond to fusing two saxpy operations, two matrix vector products, three saxpy operations and two dot operations, respectively.

tions that can benefit from the optimization based on fusion. This figure also represents the data dependence graph of the main iteration of BCG after a reordering process, where the candidate BLAS kernels for fusion are shown in rectangles. Each iteration of BCG is expressed using a set of four fused operations. The BLAS operations that can be fused in BCG are:

- two saxpy operations,
- two matrix products,
- three saxpy operations,
- two dot operations.

Experimentally, using several matrices from Matrix Market for the fusion of two matrix products, we found that this fusion does not provide noticeable improvement

on the performance specially when there is no shared vector between the SpMV kernels. Let us remark that, according to the experimental analysis developed in [137] the fusion of the remaining operations allows to improve the performance of the BCG.

The **Stabilized BiConjugate Gradient Method** (BCGSTAB) is an iterative method developed by H. A. van der Vorst for the numerical solution of nonsymmetric linear systems. It is a variant of the BiConjugate Gradient Method (BCG) and has faster and smoother convergence than the original BCG as well as other variants such as the Conjugate Gradient Squared method (CGS) [141].

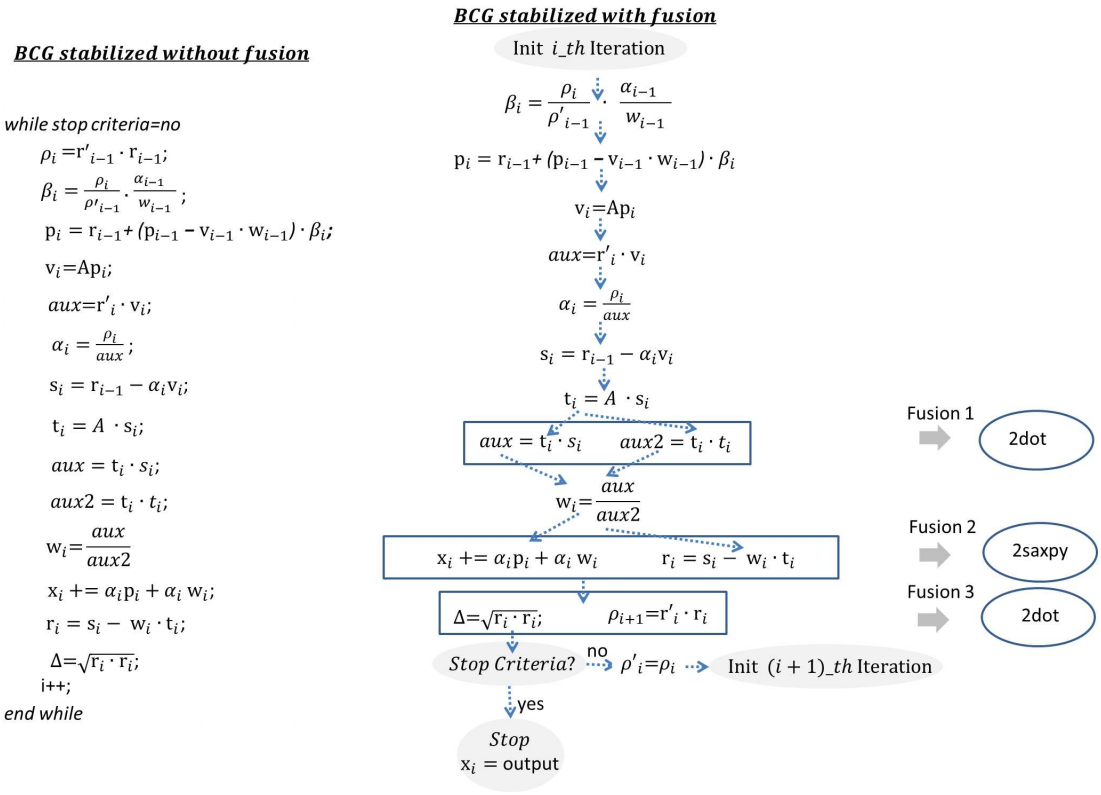


Figure 1.12: Candidate BLAS kernels for fusion in the BCG stabilized method. Applying the kernel fusion optimization to BCG stabilized consists of performing Fusion 1−3 that correspond to fusing two dot operations, two saxpy operations and two dot operations, respectively.

Figure 1.12 shows BCG algorithm and the potential candidate vector−vector operations that can benefit from the optimization based on fusion. This figure also represents the data dependence graph of the main iteration of BCGSTAB after a reordering

process, where the candidate BLAS kernels for fusion are shown in rectangles. Each iteration of BCGSTAB is expressed using a set of four fused operations. The BLAS operations that can be fused in BCGSTAB are:

- two dot operations,
- two saxpy products,
- two dot operations.

This way, each iteration of BCGSTAB is expressed using a set of three fused operations.

The experimental evaluation of this methodology using Krylov methods on GPU devices has demonstrated that fusion optimization enhances the overall performance (up to 1.27×) using reasonable large size problems. Therefore this methodology is particularly interesting for this type of methods. Moreover, these results underline the interest of extending the CUBLAS library with new versions of multiple fused vector operations for the GPU.

1.3 Platforms used in this thesis

All performance results shown later in Chapters 2, 3, 4 and 5 have been obtained from evaluations on several parallel platforms. Details of the two shared memory multiprocessors and the distributed memory cluster used in this thesis are explained in this subsection. All of these platforms have GPU devices whose main characteristics are shown in Table 1.4.

- ***Shared memory multiprocessors:***
  - DaVinci:
    - CPU: 2×Intel Xeon Quad-core E5640 (2.67 GHz, 16 GB RAM)
    - GPU: 1×NVIDIA GeForce GTX 480 (Fermi). 2×NVIDIA Tesla C2050 (Fermi)
    - Linux distribution x64
  - Hermes:

  - CPU: Intel Xeon E5620 (16 cores, 2.40 GHz, 47 GB RAM)
  - GPU: 1× NVIDIA Tesla M2090 (Fermi)
  - Linux distribution x64

- ***Distributed memory cluster:***
  - ***Bullx***. Figure 1.13 depicts the structure of the Bullx cluster.
    - CPU: 18× Bullx R424-E3. Intel Xeon E5 2650 (16 cores), 64 GB RAM and 128 GB SSD
    - GPU: 2 × 4 NVIDIA Tesla M2070 (Fermi)
    - Linux distribution x64
    - InfiniBand interconnect and Ethernet

Table 1.4: Characteristics of the GPU platforms considered for the evaluation. (Sorted by peak performance in double precision).

	Hermes	Bullx	DaVinci		
	Tesla M2090	Tesla M2070	Tesla C2050	Geforce GTX 680	GeForce GTX 480
Peak performance (double precision) (GFlops)	665	515	515	168	129
Peak performance (simple precision) (GFlops)	1331	1030	1030	1350	3090.4
Device memory (GB)	6	5.2	2.6	1.5	2
Clock rate (GHz)	1.3	1.2	1.2	1.4	3
Memory bandwidth (GBytes/sec)	177	150	144	177.4	192.2
Multiprocessors	16	14	14	15	8
CUDA cores	512	448	480	448	1536
Compute Capability	2	2	2	2	3
Year architecture	2011	2010	2010	2010	2012
DRAM TYPE	GDDR5	GDDR5	GDDR5	GDDR5	GDDR5

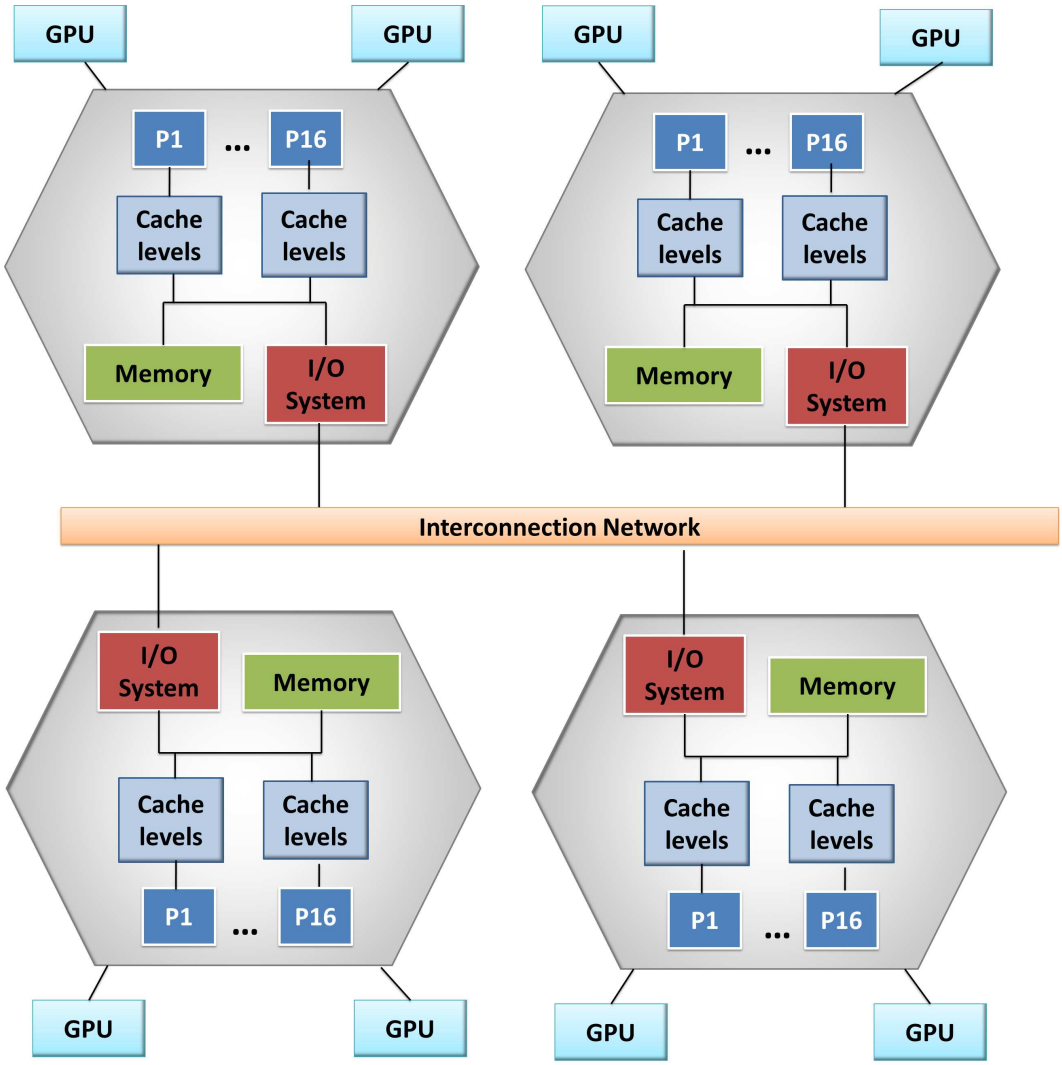


Figure 1.13: Structure of the Bullx cluster, which is comprised of four compute nodes (16 cores) and eight Tesla M2075 GPU devices.

2

Sparse matrix computation on GPUS

As shown in Chapter 1, sparse matrix operations are a key point in the resolution of sparse linear systems of equations, which are the main issues in this thesis.

The Matrix-Vector product is a key operation for a wide variety of scientific applications, such as image processing, simulation, control engineering and so on. For many applications based on Matrix-Vector product, matrices are large and sparse. Sparse matrices are involved in linear systems, eigensystems and partial differential equations from a wide spectrum of scientific and engineering disciplines [38].

For these problems the optimization of the Sparse Matrix Vector product (SpMV) is a challenge because of the irregular computation of large sparse operations. GPUs have emerged as platforms that yield outstanding acceleration factors. SpMV implementations for GPUs have already appeared on the scene, however they need some additional efforts to accelerate/optimize their performance. This effort is focused on the design of appropriate data formats to store the sparse matrices, since the performance of SpMV is directly related to the used format as shown in [105, 108, 138].

This chapter is devoted to investigate the optimization of some sparse matrix computations on GPUs. The discussion of this issue has been organized as follows. In Section 2.1, the main compressed storage formats which have been devised to manage multidimensional sparse arrays are introduced. Section 2.2 describes an implementation of SpMV for NVIDIA GPUs based on a new format, ELLPACK-R, that allows the storage of sparse matrices in a regular manner. The main findings in the chapter are highlighted in Section 2.3, where we propose a strategy (FastSpMM) to compute the Sparse matrix matrix product (SpMM). The performance of FastSpMM has been

evaluated and compared to the CUSPARSE library (supplied by NVIDIA), which also includes routines to compute SpMM on GPUs. Experimental evaluations based on a representative set of test matrices have shown that, in terms of performance, Fast-SpMM outperforms the CUSPARSE routine. Finally, Section 2.4 summarizes the main conclusions and future works of this chapter.

2.1 Compressed storage formats

Several formats have been proposed in the literature to optimize the computation with sparse matrices for a specific architecture. These formats define the locality or the coalescence of memory access for the SpMV. The pattern of memory access to read the elements of the sparse matrix has a strong impact in the performance of SpMV. Thus, every specific SpMV algorithm designed to exploit the computational resources of a particular architecture is related to a specific storage format of the sparse matrix. Then, an important key to increase the performance of SpMV on GPUs is the development of an appropriate algorithm with its corresponding storage format. Next, the main formats to compress sparse matrices and their corresponding algorithms are described. Our attention is focused on the formats specifically designed for SIMD architectures, such as vector architectures and GPUs [145]. Hereinafter, we assume that in the SpMV operation ($u = A\ v$), A is a sparse matrix with n rows, n columns and nz non-zero entries, and u and v are vectors with n elements.

The **COOrdinate storage scheme (COO)** to compress a sparse matrix is a direct transformation from the dense format. A typical implementation of COO uses three one-dimensional arrays of size nz. One array, $A[]$ of floating point numbers (hereafter referred to as floats), contains the non-zero entries. The other two arrays of integer numbers, $I[]$ and $J[]$, contain the corresponding row and column indices for each non-zero entry. The performance of SpMV may be penalized by COO because it does not implicitly include the information about the ordering of the coordinates, and, additionally, for multi-threaded implementations of SpMV atomic data access must be included when the elements of the output vector are written.

Compressed Row Storage (CRS) is the most commonly known format to store sparse matrices on superscalar processors [38]. The data structure consists of the following one dimensional arrays: (1) $A[]$, an array of floats of dimension nz which stores

the non-zero entries; (2) $J[]$, an array of integers of size nz, which stores their column index; and (3) $start[]$, an array of integers of size $n+1$ which stores the pointers to the first element of every row in $A[]$ and $J[]$, both sorted out by row index (by convention, $start[n+1] = nz+1$) [71, 90]. It is the most widely used storage format. Based on CRS, several libraries reported in the literature have improved the performance of sparse computation on current processors [1, 156]. In particular, the Intel Math Kernel Library (MKL) has improved the performance of sparse BLAS operations by optimizing the memory management and exploiting the fine grained parallelism on Intel processors. According to our experience, MKL can accelerate the SpMV up to 3× with respect to the standard implementation (and gcc compiler).

ELLPACK or **ITPACK** [86] was introduced as a format to compress a sparse matrix with the purpose of solving large sparse linear systems with ITPACKV subroutines on vector computers. This format stores the sparse matrix on two two-dimensional arrays, one float ($A[]$), to save the entries, and one integer ($J[]$), to save the column index of every entry. Both arrays are, at least, of dimension $n \times Mnzr$, where n is the number of rows and $Mnzr$ is the maximum number of non-zeroes per row in the matrix, with the maximum taken over all rows. Note that the size of all rows in these compressed arrays $A[]$ and $J[]$ is the same, because every row is padded with zeros. Therefore, ELLPACK can be considered as an approach to fit a sparse matrix in a regular data structure similar to a dense matrix. Consequently, this format is appropriate to compute operations with sparse matrices on vector architectures.

ELLPACK-R format, a variant of ELLPACK, has demonstrated to improve the performance reached by ELLPACK on GPUs. ELLPACK-R consists of two two-dimensional arrays, $A[]$ (float) and $J[]$ (integer) of dimension $n \times Mnzr$ and an additional one-dimensional integer array ($rl[]$) of dimension n (i.e. the number of rows) specifying the actual length of every row, regardless of the number of the zero elements padded. This ELLPACK-R format has demonstrated better performance than other formats in the literature to compute SpMV.

2.2 Sparse matrix vector product

The sparse matrix vector product (SpMV) is a key operation in engineering and scientific computing and, hence, it has been subjected to intense research for a long

time. The irregular computations involved in SpMV make its optimization challenging. Therefore, enormous effort has been devoted to devise data formats to store the sparse matrix with the ultimate aim of maximizing the performance.

Several implementations of SpMV for GPUs developed with CUDA have already been described in the literature [33, 37, 43, 48, 106]. This section is focused on some of these implementations; specifically on the ELLR-T algorithm which is based on the compressed storage format for the sparse matrix, ELLPACK-R [149, 152], described in the previous section.

Several formats have been proposed to optimize the computation with sparse matrices for a specific architecture. These formats define the locality or the coalescence of memory access for the SpMV, which are essential to optimize the performance on CPU or GPU architectures.

Let $u = Av$ be a sparse matrix vector product where A is the sparse matrix, v and u are the input and output vectors, respectively. According to the mapping of threads in the computation of every row, several implementations of SpMV based on ELLPACK-R can be developed. Thus, when T threads compute the element $u[i]$ accessing to the $i-th$ row, the implementation of SpMV is referred to as ELLR-T. In ELLR-T, the $i-th$ row of A is split in sets of T elements. Then, in order to compute the element $u[i]$, T threads compute $\lceil rl[i]/T \rceil$ iterations of the inner loop of SpMV. Every thread stores its partial computation in the shared memory of the GPU. Finally, to generate the value of $u[i]$, one reduction of the T values computed and stored in shared memory has to be included. The value of parameter T can be explored in order to obtain the best performance with every kind of sparse matrices. Figure 2.1 illustrates the characteristics of the code of ELLR-T ($T = 2$), showing the specific storage for the sparse matrix which ensures that the device memory accesses are coalescent and aligned. This characteristic is very relevant for ELLR-T due to the large number of memory accesses related to the SpMV computation.

Algorithm 1 shows the pseudoce for ELLR-T in order to compute SpMV on GPUs. Note that in this algorithm lines 9 and 10, the parameter n_{align} is used to redefine the dimension of $A[]$ and $J[]$ such that they fulfill the memory alignment requirements.

ELLR-T algorithm takes advantage of:

1. *Coalesced and aligned global memory access.* The access to read the elements of $A[]$,$J[]$ and $rl[]$ are coalesced and aligned thanks to the column-major ordering

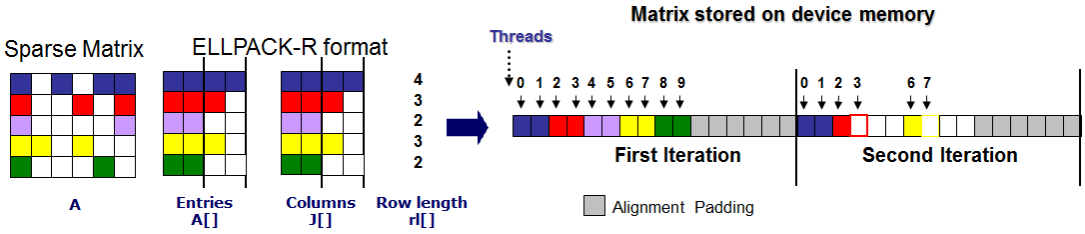


Figure 2.1: The ELLR-T memory storage of the sparse matrix fulfilling the coalescence and alignment conditions for $T = 2$.

Algorithm 1 Pseudocode of ELLR-T algorithm for computing SpMV on GPUs for $T = 32$

```
1: idx=global thread index
2: idb= local thread index into its block
3: i = ⌊idx/T⌋                                   #Row index of matrix A
4: idp = idb mod T                  #Thread index into S_i (set of T Threads for row i)
5: if i < n then
6:     svalue = 0.0
7:     max = ⌈rl[i]/T⌉
8:     for k = 0 → k < max do
9:         value = A[k · n_align · T + i · T + idp]
10:        col = J[k · n_align · T + i · T + idp]
11:        svalue+ = value · v[col]
12:    end for
13:    shared[idb] = svalue
14:    if idp < 16 then
15:        shared[idb]+ = shared[idb + 16]
16:        if idp < 8 then shared[idb]+ = shared[idb + 8]
17:        if idp < 4 then shared[idb]+ = shared[idb + 4]
18:        if idp < 2 then shared[idb]+ = shared[idb + 2]
19:        if idp == 0 then u[i] = shared[idb] + shared[idb + 1]
20:    end if
21: end if
```

used to store the matrix elements and the zeros-padding to complete the length of every row as multiple of 16 (alignement). Consequently, the highest possible memory bandwidth of GPU is exploited.

2. *Homogeneous computing within the warps.* The threads belonging to one warp do not diverge when executing the kernel to compute SpMV. The code does not include flow instructions that cause serialization in warps since every thread executes the same loop, but with different number of iterations. Every thread stops as soon as its loop finishes, and the remaining threads continue the execution.

3. *Reduction of useless computation and unbalance of the threads of one warp.* Let S_i be the set of T threads which are collaborating on the computation of $u[i]$. The $k - loop$ reaches the maximum value of $k = \lceil rl[i]/T \rceil \leq \lceil Mnzr/T \rceil$ for specific sets, S_i, into the warp. Then, the runtime of every warp is proportional to the maximum element of the subvector $\lceil rl[i]/T \rceil$ related to every warp, and it is not necessary for the $k - loop$ reaches $k = \lceil Mnzr/T \rceil$ for all threads, then, there are not useless iterations and the control of loops of this implementation is reduced comparing with SpMV based on ELLPACK. However, if the value of T increases too much, relevant number of threads are unloaded, the unbalance increases and the kernel achieves a poor performance.

Consequently, ELLR-T is able to exploit the GPU architecture for computing the SpMV operation. However, in order to reach the best performance it is very relevant to select the optimum values of two parameters: (1) T, the number of threads which collaborate to compute one element of the output vector, which is related to every matrix row, then, T is a specific parameter of ELLR-T; and (2) BS, the block size of the CUDA code, which is a general parameter to optimize the CUDA programs.

In [152] an extensive performance evaluation of ELLR-T using a representative set of test matrices has been reported. The comparative study has drawn the conclusion that ELLR-T outperforms the most common routines for SpMV on GPUs used so far. An additional advantage of ELLR-T is related to the model which allows to optimally configure it according to the particular combination of input sparse matrix/GPU architecture [146].

2.3 Sparse Matrix Matrix product. FastSpMM

Sparse matrix matrix multiplication is involved in a wide range of scientific and technical applications. Some libraries to compute this matricial operation can be found

in the literature [5]. The computational requirements for this kind of operation are enormous, specially for large matrices. This section analyzes and evaluates a method (FastSpMM) to efficiently compute the Sparse Matrix Matrix product (SpMM) in a computing environment which includes Graphics Processing Units (GPUs). We experimentally show that FastSpMM outperforms the existing CUSPARSE routine as well as the implementation of the SpMM as a set of SpMVs.

The contribution of our work in this area has been focused on a different and more complex matrix operation which includes sparse and dense matrices: the sparse matrix matrix product (SpMM). Our goal is intended to the optimization of the SpMM code and its implementation on GPUs. This kind of matricial operation is involved in very relevant applications. For example, in area of tomography, the reconstruction methods Weighted Back Projection (WBP) and Simultaneous Iterative Reconstruction Technique (SIRT) can be expressed in terms of this kind of matrix product [147, 148]. SpMM computes the matrix $C = A \cdot B$, where A is a sparse matrix, B is a dense matrix and consequently C is also dense. This operation is classified as level 3 routine for sparse matrix computation [5]. This kind of routines are able to efficiently exploit the Instruction Level Parallelism (ILP) on modern architectures [23]. The computational advantages of this kind of routines in terms of performance are well known [70]. The level 3 BLAS has been implemented on different target architectures for dense matrices [1, 4]. However, few references of SpMM implementations on GPUs can be found. For example, CUSPARSE library supplies one sparse level 3 function which computes $C = \alpha \cdot A \cdot B + \beta \cdot C$, where α, β are scalars, it is available for real or complex matrices [5] but its performance on GPUs is poor.

The goal of this work is to analyze strategies intended to improve the performance of SpMM on GPUs. Our proposal supplies an approach, FastSpMM, which efficiently combines the computational advantages of a level 3 matrix computation and the exploitation of several GPU computing resources: (1) the thread level parallelism [150] and (2) the streaming computation that allows the overlapping of CPU-GPU communication/computation.

Details of the implementation of FastSpMM and its performance evaluation are given in the following subsections. Section 2.3.1 is devoted to analyzing the FastSpMM implementation. Finally, in Section 2.3.2, a comparative performance evaluation of

FastSpMM and other routines for computing SpMM is carried out, showing that FastSpMM clearly outperforms the corresponding CUSPARSE routine in terms of performance on GPUs.

2.3.1 SPMM product on GPUs

Let us assume that in the matrix matrix product ($C = A \cdot B$), A is sparse with n rows and m columns, while B and C are dense matrices with sizes $m \times L$ and $n \times L$, respectively. Special attention should be paid to the storage of sparse matrix A. To optimize the computation of operations which involve sparse matrices, several storage formats have been devised for specific architectures. These formats define the locality or the coalescence of memory access of the sparse matrix, which are essential to optimize the performance on CPU or GPU architectures.

Our interest is focused on the format ELLPACK-R [86] because it allows the storage of sparse matrices in a regular manner as aforementioned.

The kernel ELLR-T has already been used to compute SpMV on GPU. It is based on ELLPACK-R and it outperforms other approaches in terms of performance [56, 146].

In this section, our efforts are oriented to optimize the performance of SpMM operations on GPUs. For this goal we have created a SpMM routine based on an extension of ELLR-T, called FastSpMM. In our description of the GPU computation of FastSpMM we have assumed that matrices B and C have $L_{GPU} \leq L$ columns, where the value of L_{GPU} is chosen such that the memory requirements to store the matrices (A, B and C) are available on the device memory of the GPU architecture. Later on we will explain how to deal with larger matrices.

In FastSpMM, every set of T threads carries out L_{GPU} reduction operations, those involved in the computation of every row of the output matrix, C. In order to accelerate them, every thread stores its partial results in the shared memory of the GPU [14]. Bearing in mind the small size of the shared memory of GPU architectures, every set of T threads can only compute a very limited number of reductions. Therefore, a new parameter is introduced, L_c ($L_c \leq L_{GPU} \leq L$), as the number of columns of C which are computed by a set of T threads, as described by Algorithm 2 (Single Step of FastSpMM denoted by SSFastSpMM).

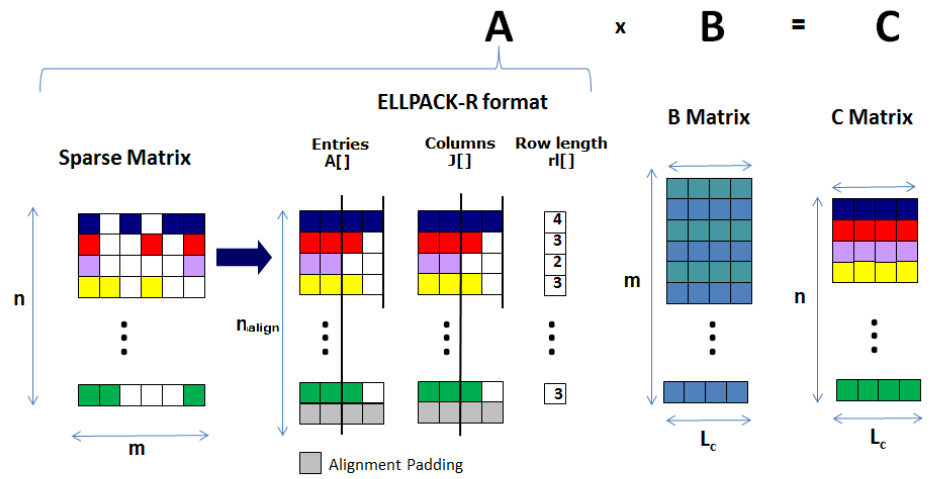
FastSpMM will iteratively compute $C_l^{L_c} = AB_l^{L_c}$ ($0 \leq l \leq \lceil L_{GPU}/L_c \rceil$), where $C_l^{L_c}$ and $B_l^{L_c}$ denote submatrices of C and B with L_c columns. Note that in Algorithm

2 line 8, the parameter n_{align} ($n_{align} \geq Mnzr$, where $Mnzr$ is the maximum number of non-zeroes per row in the matrix, with the maximum taken over all rows) is used to redefine the dimension of $A[]$ and $J[]$ such that they fulfill the memory alignment requirements. So, rows of zeroes are padded up to the number of elements of T columns of $A[]$ (and $J[]$) is a multiple of 16 [14]. Consequently, accesses to the matrix A from the device memory satisfy coalescence and alignment requirements. More details about this approach to optimize the reading of matrix A from the device memory of GPU can be found in [146]. Figure 2.2 illustrates the organization of threads for SSFastSpMM with $T = 2$ and $L_c = 4$, where a row has been added for memory alignment in the data structure of matrix A (grey color).

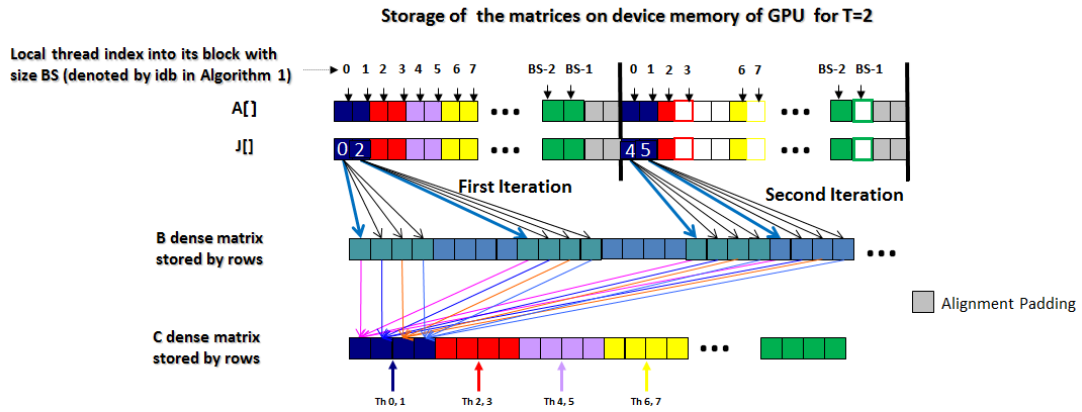
Algorithm 2 Pseudocode of SSFastSpMM, a Single Step of FastSpMM, to compute SpMM on GPUs. It computes the product $C^{L_c} = AB^{L_c}$ where A is sparse matrix of dimensions $n \times m$, B^{L_c} and C^{L_c} are dense matrices of dimensions $m \times L_c$ and $n \times L_c$, respectively

1: idx=global thread index
2: idb= local thread index into its block
3: $i = \lfloor idx/T \rfloor$ #Row index of matrix A
4: $idp = idb \mod T$ #Thread index into the set of T threads for row i of A
5: **if** $i < n$ **then**
6: $sv_0 = sv_1 = \ldots = sv_{L_c-1} = 0$
7: **for** $j = 0 \rightarrow j < \lceil rl[i]/T \rceil$ **do**
8: $index$=$T \cdot (j \cdot n_{align}$+$i)$+$idp$
9: v=$A[index]$
10: col=$J[index]$
11: sv_0+=$v \cdot B[col, 0]$
12: sv_1+=$v \cdot B[col, 1]$
13: $\cdots$
14: sv_{L_c-1}+=$v \cdot B[col, L_c - 1]$
15: **end for**
16: Reduction of $sv_0, sv_1, \ldots, sv_{L_c-1}$ on shared memory to compute $C[i, 0], C[i, 1], \ldots, C[i, L_c - 1]$ for the specific value of T
17: **end if**

Algorithm 2 describes the pseudocode for a single step of FastSpMM, where several sets of T threads are defined and every set computes L_c elements of the i-th output row



(a) Format of matrices involved in a single step of FastSpMM (SSFastSpMM)



(b) Matrices storage schemes for T=2 on device memory and threads mapping of SSFastSpMM on the GPU

Figure 2.2: Storage format of the sparse and dense matrices and threads mapping of SSFastSpMM (Algorithm 2) for the SpMM operation on GPUs.

(i.e. $C[i,0],\ldots,C[i,L_c-1]$). So, the i-th row of A is split in sets of T elements. Each thread computes $\lceil rl[i]/T \rceil$ iterations of the inner loop of SSFastSpMM (lines 8-17), they compute their partial reductions which are stored in shared memory (lines 12-16). Finally, to generate the values of $C[i,0],\ldots,C[i,L_c-1]$, L_c reductions of the T values computed and stored in shared memory by every thread are carried out.

It is remarkable that the data structure to store the sparse matrix and the mapping

of threads of SSFastSpMM provides the following advantages: (i) It optimally exploits the memory bandwidth of GPU for the reading of the sparse matrix ($A[]$, $J[]$ and $rl[]$) since the storage format allows the coalesced and aligned global memory access and the reduction of threads load unbalance [146]; (ii) SSFastSpMM is able to take advantage of the high ratio computation/memory access of the SpMM operation compared to the SpMV operation. Note that the SpMM operation ($C = A \cdot B$) could be computed by a set of SpMV operations ($c_i = A \cdot b_i$ with $0 \leq i \leq L_{GPU}$). However, in this case, the sparse matrix is read L_{GPU} times from the device memory to compute C. In contrast, FastSpMM reads the sparse matrix once and computes L_c columns of C, so the ratio computation/memory access for FastSpMM is L_c times higher than for SpMM computed as a set of SpMV. Furthermore, FastSpMM has another computational advantage since the temporal locality for reading B is increased and the indirect addressing to read the elements of B is evaluated just once to compute L_c elements of C, thus the ILP is better exploited by FastSpMM. Consequently, FastSpMM combines the computational advantages above described, with the advantages of ELLR-T format to exploit the GPU architecture.

It is relevant to emphasize that the performance of FastSpMM on GPUs is high only for very large sparse matrices [152]. So, the size of the corresponding dense matrices is also very large, with huge memory requirements not provided by the device memory of the GPU. Consequently, the number of columns of B involved in one step of FastSpMM cannot be increased excessively and should be tuned to the size of the device memory of the GPU.

Bearing in mind these GPU limitations, FastSpMM has finally been designed as a routine which computes $C^{L_{GPU}} = A \cdot B^{L_{GPU}}$ (the value of L_{GPU} is chosen such that the memory requirements to store the matrices, A, $B^{L_{GPU}}$ and $C^{L_{GPU}}$, are available on the device memory of the GPU architecture). Thus, to compute $C = A \cdot B$ on GPU, it is necessary to include $\lceil L/L_{GPU} \rceil$ successive communications between CPU-GPU with sets of L_{GPU} columns of B and C. It is noteworthy that the above mentioned limitations to compute SpMM on GPUs are associated to all the implementations of the SpMM on GPUs. Consequently, for the CUSPARSE routine it is also necessary to include the same communications between GPU and CPU. This library is taken as a reference to evaluate our proposal.

A basic algorithm called FastSpMM* is considered as the starting point to compute $C = A \cdot B$ with a basic CPU-GPU communication scheme. The CPU-GPU communication scheme to compute $C = A \cdot B$ on GPU is described in Algorithm 3 (FastSpMM*), where the parameter L_{GPU} is an upper bound of L_c for FastSpMM* and a multiple of L_c. This scheme is also valid for the approach based on the SpMV operation ($L_c = 1$).

From the description of Algorithm 3 it is clear that the cost of the CPU-GPU communications is very high because every iteration requires the exchange of L_{GPU} columns of B and C between the CPU and the GPU memory. Therefore, the streaming computation for overlapping CPU-GPU communication and computation is also considered in the optimized version of the SpMM operation as described in the next subsection.

Algorithm 3 FastSpMM*, general scheme to compute $C = A \cdot B$ based on SSFastSpMM

1: Copy the sparse matrix A to the GPU memory
2: **for** $i = 0 \rightarrow \lceil L/L_{GPU} \rceil - 1$ **do**
3: Copy L_{GPU} columns of B from CPU to GPU memory
4: **for** $j = 0 \rightarrow \lceil L_{GPU}/L_c \rceil - 1$ **do**
5: SSFastSpMM ($C_j^{L_c} = AB_j^{L_c}$ on GPU) #Algorithm 2
6: **end for**
7: Copy L_{GPU} columns of C from GPU to CPU memory
8: **end for**
9: output C matrix is stored on CPU memory

2.3.1.1 Streaming computation for FastSpMM

The CPU-GPU communications play an important role in FastSpMM* (Algorithm 3) and have a strong impact on the performance of this routine. However, the scheme of FastSpMM* includes iterative steps of communications and computations which could be overlapped at runtime. The idea of the overlapping communication/computation consists of doing the computational work while the communication infrastructure simultaneously performs data transfers, with the goal of hiding the latency and transfer costs of the inter-process communication [155].

The strategy for overlapping communication/computation between the host and the GPU platform is based on the stream processing supplied by CUDA [130, 155].

In this context, a stream represents a queue of GPU operations that are executed in a specific order. Operations in different streams can be interleaved and in some cases overlapped, a property that can be used to hide data transfers between the host and the device. Therefore, the scheme described in Algorithm 3 can be expressed by two streams which interleave their execution as shown in Algorithm 4.

Algorithm 4 FastSpMM: code with streaming computation CPU-GPU to compute the SpMM operation

1: Streams creations 0 and 1
2: **Stream 1:** Asynchronous copy of B_0 (column chunk of B) to the GPU memory
3: **for** $i = 0 \rightarrow \lceil L/L_{GPU} \rceil - 1$ **do**
4: $p = i \mod 2$; $q = (p+1) \mod 2$
5: **Stream p:** $\lceil L_{GPU}/L_c \rceil$ executions of SSFastSpMM to compute $C_i = A \cdot B_i$
6: **Stream q:** Asynchronous copy of B_i to the GPU memory
7: **Stream p:** Asynchronous copy of C_i to the CPU memory
8: **end for**
9: Streams delete

Our approach relies on two points: the "chunked" computation (for overcoming the GPU memory limitation and executing SpMM for large matrices) and overlapping the memory read/write operations (memcpy) with the kernel executions (for decreasing the communication penalization CPU-GPU and viceversa). Assuming that our memory read/write operations and kernel executions take roughly the same amount of time, in Figure 2.3 we have schematically represented the execution timeline of FastSpMM, where: stream 0 sends its input buffers to the GPU; later, stream 1 executes the same operation while stream 0 is executing its kernel; then, stream 1 will execute its kernel while stream 0 copies its results to the host; and, finally, a new input buffer is copied to the GPU and the obtained results are sent to the host memory. This process will be repeated for the next chunks of data.

From the interleaved execution of both streams (0 and 1), in Figure 2.3 we can identify an iterative sequence of four time steps with six operations (enclosed in brackets). These six operations would consume only four time steps instead of six. Therefore, if we define the Acceleration Factor (AF) as the ratio between sequential and overlapped runtimes, then AF can achieve the maximum value of 1.5.

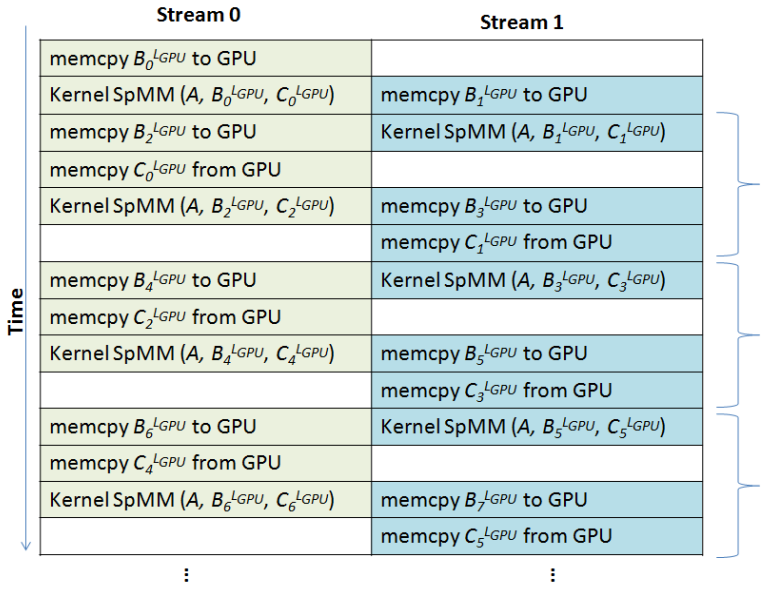


Figure 2.3: Timeline of FastSpMM execution using two independent streams (Stream 0 and 1).

It is necessary to underline that the previous analysis has been simplified, since it assumes that SpMM kernel (A, B_i, C_i) in Figure 2.3 spends the same time that every communication GPU-CPU and CPU-GPU (memcpy in the same figure), i.e. the computation steps waste the same time than the communication steps. However, on the current GPU platforms the bandwidth is higher for the CPU-GPU communication than in the GPU-CPU direction. Moreover, the time of SpMM kernel (A, B_i, C_i) increases as $nz \cdot L_{GPU}$ increases and the communication time increases as n and/or m increase. Therefore, the impact of the streaming included in FastSpMM in terms of performance depends on the characteristics of the matrices involved in the SpMM.

2.3.2 Evaluation

This section is devoted to evaluating several approaches for computing the SpMM operation using a wide set of test sparse matrices. These matrices come from a broad spectrum of disciplines of science and engineering and exhibit different characteristics, from those that are well-structured and regular to highly irregular matrices with large unbalances in the distribution of non-zeroes per matrix row. Table 2.1 shows the set of test matrices and the characteristic parameters related to their specific patterns: num-

ber of rows (n), total number of non-zero elements (nz), average (Av) and maximum ($Mnzr$) number of entries per row and the estimated giga floating point operations for the SpMM computation ($NFLOP = 2n \cdot nz/2^{30}$). In this table, some special matrices are considered: three dense matrices (dense2, dense5000 and dense10000) and two tridiagonal matrices (tridiagonal1 and tridiagonal2). It is relevant to underline that these special matrices do not represent the prototype of unstructured sparse matrices, so FastSpMM has not been specifically designed to accelerate the SpMM for these types of matrices. However, they are included in our evaluation because they help to analyze computational characteristics of our library for matrices with extreme values of Av. Note that all matrices are ordered according to the value of the Av parameter and all of them verify $n = m$.

The evaluation has been carried out on two GPUs platforms with the Fermi architecture of NVIDIA [7]: a Tesla C2050 and a GeForce GTX480. The main characteristics of both GPUs are shown in Chapter 1, Table 1.4.

Four routines to compute the SpMM on GPUs have been evaluated:

1. **SpMM CUSPARSE**, based on the level 3 function of this library, computes $C = \alpha \cdot A \cdot B + \beta \cdot C$ with $\alpha = 1$ and $\beta = 0$. This routine prevents the useless computation of βC when $\beta = 0$.

2. **SetSpMVs** (ELLR-T) computes the SpMM as a set of n SpMV operations based on the kernel ELLR-T [146].

3. **FastSpMM*** is based on the ELLR-T format without overlapping communication/computation (Algorithm 3).

4. **FastSpMM** is based on ELLR-T format and takes advantage of overlapping communication/computation (Algorithm 4).

The CUSPARSE library provides a set of basic linear algebra subroutines used for handling sparse matrices based on the CRS format to compress the sparse matrix [5, 38]. The paradigm of CUSPARSE is to define multiple blocks where each block is in charge of processing a group of rows. Each row is assigned to a set of threads. Moreover, a few additional approaches for improving performance are considered: (1) adjust the number of threads per row to minimize the unbalance among threads; (2) align threads per row

Table 2.1: Characteristics of test sparse matrices (denoted as A in our definition of SpMM), where n is the number of rows, nz is the total number of non-zero elements, Av and $Mnzr$ are the average and the maximum number of entries per row, respectively, and $NFLOP = 2n \cdot nz/2^{30}$ is the estimated GFlops.

Matrix	$n[10^3]$	$nz[10^6]$	Av	$Mnzr$	$NFLOP$
tridiagonal1	200	0.6	3	3	223.5
tridiagonal2	500	1.5	3	3	1396.9
cop20k_A	121	2.6	22	81	592.4
qcd5_4	49	1.9	39	40	175.5
Si87H76	240	10.6	45	361	4773.4
msdoor	416	19.1	47	77	14851.6
pwtk	218	11.6	53	181	4722.5
shipsec1	141	7.8	55	102	2050.2
cant	62	4.0	64	78	466.2
Ga41As41H72	268	18.4	69	702	9232.5
consph	83	6.0	72	81	933.0
x104	108	8.7	81	324	1759.1
RM07R	382	37.4	99	295	26635.8
pdb1HYS	36	4.3	119	204	294.7
wbp128	16	3.9	240	256	120.0
nd3k	9	3.2	365	515	55.0
wbp256	66	31.4	480	512	3834.7
dense2	2	4.0	2000	2000	14.9
dense5000	5	25.0	5000	5000	232.8
dense10000	10	100.0	10000	10000	1862.6

for coalescing and (3) use shared and texture memory [5]. However, CUSPARSE does not allow the user to select values of configuration parameters such as threads block size (BS) or the number of threads (T), to achieve the optimal performance.

A comparative evaluation of the performance achieved by the four versions of the SpMM operation has been carried out. In order to fairly compare the performance values of both FastSpMM versions with the CUSPARSE routine, all of them compute the operation $C = \alpha \cdot A \cdot B$ with $\alpha = 1$. It is relevant to note that SpMM CUSPARSE, SetSpMVs and FastSpMM* versions include the same communications scheme (as de-

scribed in Algorithm 3). In our evaluation the size of B and C is $n \times n$ ($L = m = n$).

Let us remark that the values of L_c (number of columns of C computed in the execution of SSFastSpMM) and L_{GPU} (number of columns of B and C involved in a single CPU-GPU transfer) are strongly related to the performance of the SpMM for all the approaches. So, $L_c = 1$ for SetSpMVs and L_c can be considered as a variable parameter for FastSpMM* and FastSpMM, with $L_c \leq L_{GPU}$. Experiments to explore the best results in terms of performance according to the values of L_c have been carried out and we have concluded that the highest performance is achieved when $L_c = L_{GPU}$.

A preliminary evaluation of the performance of FastSpMM for several values of L^* ($L^* = L_{GPU} = L_c$) has been carried out. The result of this evaluation will help us to determine the value of L^* (for each platform, Tesla C2050 and GTX480) which will be later used in our comparative evaluation of the performance of FastSpMM and FastSpMM* with respect to CUSPARSE and SetSpMVs. Figure 2.4 shows the experimental results of our evaluation (performance of FastSpMM as a function of L^*). The values of the performance have been optimized with respect to the configuration parameters BS and T, taking also into account the memory limitations of both platforms. It is remarkable that for the *Ga41As41H72* matrix and the GTX480 platform, only the configurations with $L^* \leq 4$ have been possible because of the high device memory requirements of this matrix. The complete set of test matrices has been evaluated, however for the sake of clarity dense and tridiagonal matrices have not been included in Figure 2.4. For tridiagonal matrices the performance of FastSpMM is very low and decreases as L^* increases. For dense matrices the performance is very high and the larger the value of L^*, the better the performance. From this study we can conclude that for most of the test matrices, appropriate values of L^* for the Tesla C2050 and GTX480 platforms are 8 and 4, respectively. For larger values of this parameter the performance does not improve significantly. Note that the less the value of L^* the less memory resources are required, facilitating the portability of FastSpMM to GPU platforms with limited resources or the allocation of additional data structures required when FastSpMM is assembled with other kernels on the GPU. Bearing in mind all the previous considerations, in the rest of our experiments the value of the parameter L^* has been set to 4.

An additional evaluation of the impact of the configurable cache on our SpMM approach has been carried out. The purpose of this evaluation is to optimize our

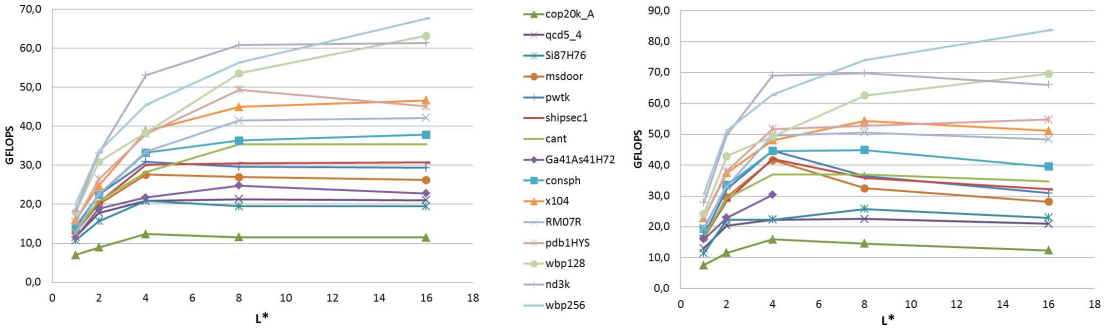


Figure 2.4: Performance of FastSpMM as function of L^* ($L^* = L_c = L_{GPU}$) for test sparse matrices on Tesla C2050 (left) and GTX480 (right).

SpMM kernel in terms of performance. Note that for CUDA devices with compute capability of 2.0 or greater such as the Fermi GPUs used in this section, the memory for each multiprocessor is 64 KB. This per-multiprocessor on-chip memory is split and used for both shared memory and L1 cache [7]. Thus, we have tuned the scratchpad-cache for making a better use of on-chip memory. The tested configurations have been: (1) 48 KB of shared memory with 16 KB of L1 cache (default configuration); and (2) 16 KB of shared memory with 48 KB of L1 cache.

Note that both configurations applied to FastSpMM can achieve different processing times but the GPU-CPU communication times do not depend on the cache configuration. So, the analysis of the impact of both configurations in the performance of FastSpMM has been done in terms of the values of the processing time. Table 2.2 shows the processing times of FastSpMM for the set of test matrices and both configurations. As can be observed in this table, the configuration using a higher capacity of the L1 obtains the best results. Bearing in mind that FastSpMM requires a maximum of 16 KB for shared memory for all test matrices (for $L_c = 4$ and float data type), both configurations provide these share memory requirements. FastSpMM is dominated by device memory accesses to read and write the matrices involved in the operation $A \cdot B = C$. The access pattern to read the matrix B is related to the location of the entries of the sparse matrix A, therefore, the management of the device memory can be improved with a larger cache memory [139]. According to these considerations, results in Table 2.2 show that the configuration with a larger L1 achieves better performance. This is the configuration used in the remaining evaluations.

Table 2.2: Processing Time (seconds) of the SpMM operation using two configurations of the shared and L1 memory over the FastSpMM approach. Columns PT_s and PT_{L1} refers to the configuration with 48 KB of shared memory and with 48 KB of L1 cache, respectively. Column AF_{L1} identifies the Acceleration Factor ($AF_{L1} = TP_s/TP_{L1}$). Special matrices (dense and tridiagonal matrices) are typed in italics.

	Tesla C2050			GTX480		
Matrix	PT_s	PT_{L1}	AF_{L1}	PT_s	PT_{L1}	AF_{L1}
tridiagonal1	19.3	19.3	1.0	14.0	13.7	1.0
tridiagonal2	76.9	76.2	1.0	46.3	45.4	1.0
cop20k_A	37.1	29.0	1.3	28.3	20.2	1.4
qcd5_4	5.6	5.0	1.1	4.2	3.7	1.1
Si87H76	173.4	157.8	1.1	123.5	113.2	1.1
msdoor	361.4	325.1	1.1	251.2	210.0	1.2
pwtk	87.4	81.9	1.1	69.0	56.5	1.2
shipsec1	42.0	40.9	1.0	32.8	28.5	1.2
cant	12.2	10.9	1.1	9.3	8.3	1.1
Ga41As41H72	336.4	320.0	1.1	243.4	240.8	1.0
consph	20.9	19.0	1.1	16.2	13.6	1.2
x104	40.8	39.9	1.0	36.0	27.0	1.3
RM07R	678.5	623.2	1.1	557.3	413.1	1.3
pdb1HYS	6.8	6.0	1.1	5.1	4.5	1.1
wbp128	3.3	3.1	1.1	2.5	2.2	1.1
nd3k	1.3	0.9	1.4	1.0	0.7	1.4
wbp256	108.0	82.0	1.3	80.2	65.5	1.2
dense2	0.3	0.2	1.3	0.2	0.1	1.3
dense5000	3.7	3.0	1.2	2.6	2.0	1.3
dense10000	24.2	22.9	1.1	16.6	15.2	1.1

We have also carried out a comparative analysis of FastSpMM* and FastSpMM to assess the improvement of the performance due to the overlapping communication/computation in the SpMM. Tables 2.3 and 2.4 show the values for the Communication Time (CT), Processing Time (PT), total Runtime (Runt), percentage of the Communication Time in the total runtime (% Com) and Acceleration Factor of the FastSpMM versus FastSpMM* (AF) for every test matrix on both GPUs. It is

necessary to highlight that the values of runtime related to the FastSpMM* algorithm include the computation and communication time. Communications represent a relevant percentage of runtime, according to the scheme of Algorithm 3.

Experimental results from Tables 2.3 and 2.4 show that the values of communication time are very relevant in this kind of operation and depend on the matrix dimension (n). From Tables 2.3 and 2.4, we can conclude that FastSpMM outperforms FastSpMM* in terms of runtime for all test matrices due to the exploitation of overlapping communication/computation. This advantage is less relevant if the communication time with respect to the total runtime (see % Com column of Tables 2.3 and 2.4) achieves extreme values. So, for tridiagonal matrices (with %Com > 84) and for dense matrices (with %Com < 13) the Acceleration Factor, AF, is nearly one. Therefore, these experimental results confirm that the approach to overlap communication/computation considerably improves the performance of SpMM on GPU.

A comparative analysis of the performance achieved by the CUSPARSE, SetSpMVs (ELLR-T), FastSpMM* and FastSpMM versions of SpMM has been carried out. Experimental results for all the sparse matrices and both the Tesla C2050 and GTX480 platforms have been depicted in Figure 2.5. These results clearly show that the performance strongly depends on:

1. *The characteristics of the sparse matrices.* For instance, *wbp256*, *nd3k* and *wbp128* matrices achieve the best performance for all approaches due to their high average number of elements per row (see Av column in Tables 2.3 and 2.4). However, the product of the *Ga41As41H72* matrix achieves a poor performance due to the irregular filling of its rows which produces a high value of $Mnzr$ (specific parameter of the ELLPACK-R format described in Section 2.3.1.1). So, the memory requirements to store the matrix A are enlarged with ELLPACK-R and the memory management is also penalized. The regular pattern of the *pwtk* matrix allows to achieve a better performance despite its low value of Av.

2. *The approach for computing the SpMM on GPU.* We can observe that, for most of the test matrices, the CUSPARSE library achieves the poorest performance, on both GPUs. This is mainly due to the fact that this library does not allow the user to set up the parameters according to the pattern of the matrix. The SetSpMVs version achieves slightly better performance since suitable parameters

Table 2.3: Profiling of SpMM based on FastSpMM* and FastSpMM on Tesla C2050. The following notation is used, Av and $Mnzr$: average and the maximum number of entries per row, respectively; PT: Processing Time; CT: Communication Time; Runt: total Runtime (all in seconds); % Com: percentage of Communication Time with respect to the total runtime for the FastSpMM* approach; AF: Acceleration Factor of the FastSpMM with respect to FastSpMM*. Special matrices (dense and tridiagonal matrices) are typed in italics.

				FastSpMM*			FastSpMM	
Matrix	Av	$Mnzr$	PT	CT	% Com	Runt	Runt	AF
tridiagonal1	3	3	19.3	108.8	84.9	128.1	109.9	1.17
tridiagonal2	3	3	76.2	619.8	89.1	695.9	623.5	1.12
cop20k_A	22	81	29.0	41.9	59.1	70.9	51.5	1.38
qcd5_4	39	40	5.0	8.1	62.0	13.0	9.0	1.44
Si87H76	45	361	157.8	158.6	50.1	316.4	244.9	1.29
msdoor	47	77	325.1	431.5	57.0	756.6	610.5	1.24
pwtk	53	181	81.9	125.2	60.4	207.1	164.2	1.26
shipsec1	55	102	40.9	56.8	58.1	97.7	73.2	1.33
cant	64	78	10.9	13.1	54.5	23.9	17.7	1.35
Ga41As41H72	69	702	320.0	188.9	37.1	508.9	457.1	1.11
consph	72	81	19.0	21.2	52.8	40.2	30.2	1.33
x104	81	324	39.9	32.5	44.9	72.3	56.8	1.27
RM07R	99	295	623.2	379.1	37.8	1002.3	855.1	1.17
pdb1HYS	119	204	6.0	4.6	43.4	10.5	8.3	1.27
wbp128	240	256	3.1	1.0	24.0	4.1	3.4	1.20
nd3k	365	515	0.9	0.3	26.5	1.3	1.1	1.15
wbp256	480	512	82.0	14.5	15.0	96.5	90.5	1.07
dense2	2000	2000	0.2	0.0	10.3	0.2	0.2	1.06
dense5000	5000	5000	3.0	0.1	3.5	3.2	3.0	1.05
dense10000	10000	10000	22.9	0.4	1.9	23.3	23.0	1.01

to obtain the best performance of SetSpMVs have been selected according to the GPU platform and the characteristic of the sparse matrices. However, its performance is very poor compared to FastSpMM* and FastSpMM. Focusing our attention on the performance improvements of FastSpMM* versus CUSPARSE

Table 2.4: Profiling of SpMM based on FastSpMM* and FastSpMM on GTX 480. The following notation is used, Av and $Mnzr$: average and the maximum number of entries per row, respectively; PT: Processing Time; CT: Communication Time; Runt: total Runtime (all in seconds); % Com: percentage of Communication Time with respect to the total runtime for the FastSpMM* approach; AF: Acceleration Factor of the FastSpMM with respect to FastSpMM*. Special matrices (dense and tridiagonal matrices) are typed in italics.

				FastSpMM*			FastSpMM	
Matrix	Av	$Mnzr$	PT	CT	% Com	Runt	Runt	AF
tridiagonal1	3	3	13.7	89.4	86.7	103.0	99.2	1.04
tridiagonal2	3	3	45.4	519.6	92.0	565.0	533.5	1.06
cop20k_A	22	81	20.2	36.0	64.0	56.2	40.0	1.40
qcd5_4	39	40	3.7	7.0	65.3	10.7	8.4	1.27
Si87H76	45	361	113.2	145.1	56.2	258.3	210.7	1.23
msdoor	47	77	210.0	356.6	62.9	566.5	404.2	1.40
pwtk	53	181	56.5	103.2	64.6	159.6	113.5	1.41
shipsec1	55	102	28.5	47.1	62.3	75.6	52.5	1.44
cant	64	78	8.3	11.1	57.1	19.4	13.6	1.43
Ga41As41H72	69	702	240.8	152.4	38.8	393.2	326.2	1.21
consph	72	81	13.6	18.9	58.2	32.5	22.5	1.45
x104	81	324	27.0	28.7	51.5	55.7	45.9	1.21
RM07R	99	295	413.1	357.5	46.4	770.6	577.2	1.34
pdb1HYS	119	204	4.5	3.9	46.5	8.4	6.1	1.37
wbp128	240	256	2.2	0.8	26.8	3.0	2.6	1.16
nd3k	365	515	0.7	0.3	27.8	1.0	0.9	1.15
wbp256	480	512	65.5	12.5	16.0	78.0	65.6	1.19
dense2	2000	2000	0.1	0.0	12.2	0.2	0.2	1.05
dense5000	5000	5000	2.0	0.1	4.5	2.0	2.0	1.04
dense10000	10000	10000	15.2	0.4	2.5	15.6	15.2	1.03

and SetSpMVs, we can remark that they are related to the suitable exploitation of the high ratio computation/memory access in combination with the advantages of ELLPACK-R format to exploit the GPU architecture. Finally, we can conclude that FastSpMM achieves the best performance thanks to the use of the schema

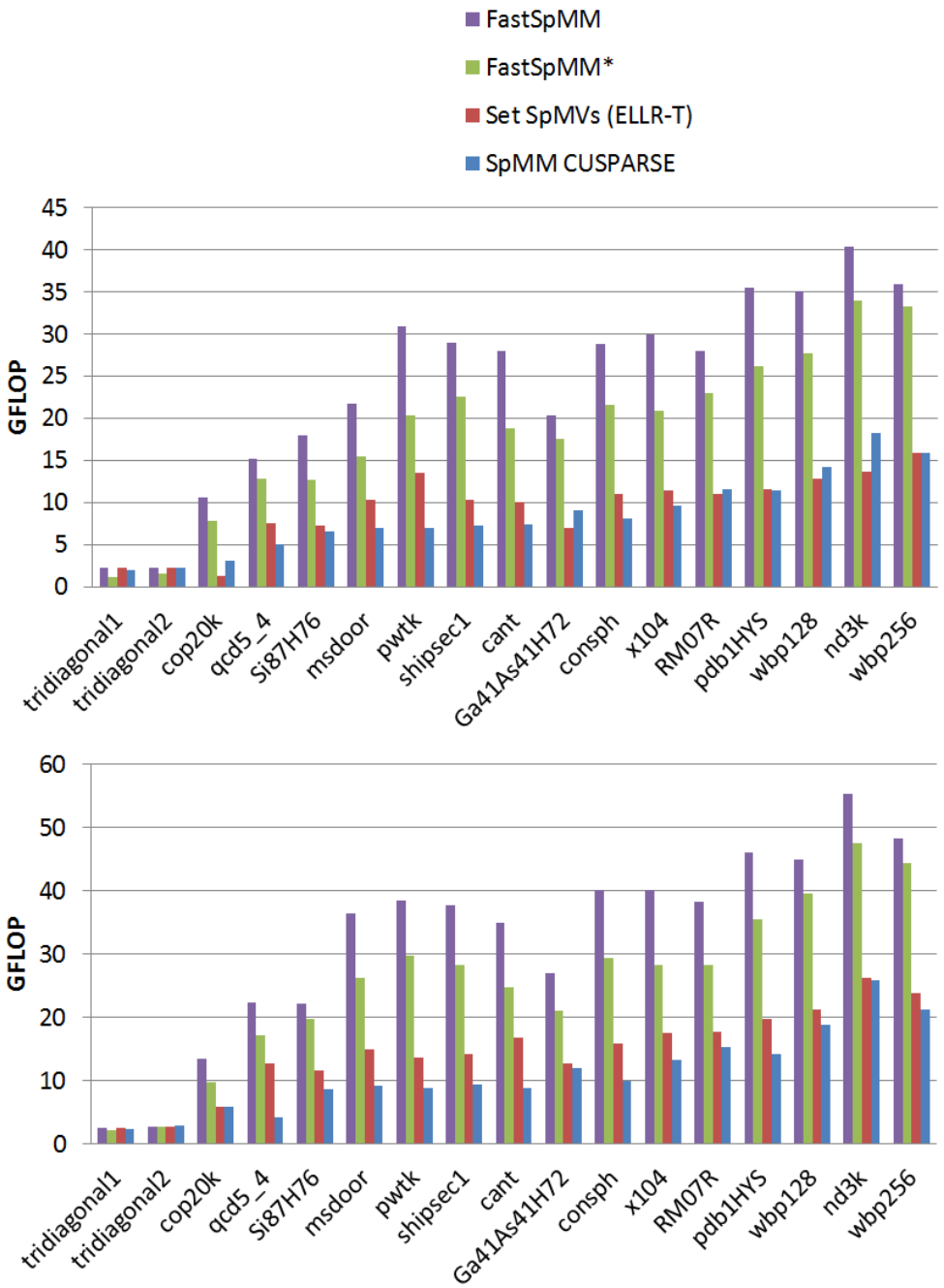


Figure 2.5: Performance evaluation of the sparse matrices using the approaches: CUSPARSE, SetSpMVs, FastSpMM* and FastSpMM to compute SpMM on Tesla C2050 (top) and GTX480 (bottom).

for overlapping communication/computation based on CUDA streaming and the advantages of FastSpMM*.

Table 2.5: Runtime executions in seconds of the SpMM multicore version using MKL with 1, 2, 4 and 8 cores (1C, 2C, 4C and 8C) and Acceleration Factors (AF) of FastSpMM on Tesla and GTX480 ($AF\ Tesla$ and $AF\ GTX480$, respectively) over the MKL version with 8 cores

	1C	2C	4C	8C	$AF\ Tesla$	$AF\ GTX480$
cop20k_A	721.8	444.0	211.6	112.0	2.2	2.8
qcd5_4	148.8	72.2	38.0	19.6	2.2	2.3
Si87H76	4112.9	2191.3	1219.8	639.6	2.6	3.0
msdoor	14049.4	7191.6	3936.9	2282.7	3.7	5.6
pwtk	3781.4	1975.1	1043.7	566.3	3.4	5.0
shipsec1	1651.1	865.5	469.3	251.0	3.4	4.8
cant	381.0	244.6	103.8	55.8	3.1	4.1
Ga41As41H72	7629.0	4022.5	2298.3	1211.7	2.7	3.7
consph	734.6	385.1	218.6	109.6	3.6	4.9
x104	1563.6	866.8	447.1	258.3	4.5	5.6
RM07R	21263.3	11166.8	5900.3	3390.4	4.0	5.9
pdb1HYS	242.6	121.2	68.4	36.8	4.4	6.0
wbp128	106.4	52.4	27.6	14.4	4.3	5.5
nd3k	42.9	22.0	12.8	7.3	6.6	8.5
wbp256	4145.6	2208.1	1033.9	518.5	5.7	7.9

In order to estimate the net gain provided by GPUs over modern processors in the SpMM, we have chosen the optimized versions of SpMM routines for both kind of architectures. In our experiments a computer based on a state-of-the-art multicore processor, Intel Xeon E5640 with 8 cores, has been used to compute the SpMM operation based on the MKL library [1]. It is remarkable that for very large matrices, the memory requirements of SpMM are not supplied by the CPU architecture. In these cases the "chunked" computation has been also applied for overcoming the CPU memory limitation. Table 3.4 shows the experimental results in terms of runtime (in seconds) of the SpMM multicore version using MKL with 1, 2, 4 and 8 cores (1C, 2C, 4C and 8C) and the acceleration factors of FastSpMM on Tesla and GTX480 ($AF\ Tesla$ and $AF\ GTX480$, respectively) with respect to the MKL version with 8 cores. As can be

observed, the increase of the number of cores means a considerable reduction in the total runtime of the MKL version. The results obtained using MKL and 8 cores have been compared to the runtimes of the FastSpMM algorithm, showing the acceleration factors for every matrix and GPU considered. These acceleration factors range from 2.2× (2.3×) to 6.6× (8.5×) on Tesla C2050 (GTX480) for the set of test matrices. So, we can conclude that the GPU turns out to be an excellent accelerator of SpMM.

2.4 Conclusions

In this section the FastSpMM approach to accelerate the SpMM operation on GPUs has been analyzed. FastSpMM combines two considerations: (1) a high ratio computation/memory access with the advantages of ELLPACK-R format to exploit the GPU architecture and (2) streaming computation for overlapping communication/computation. The comparative evaluation with other proposals (CUSPARSE, SpMM as a set of SpMV operations based on ELLR-T and FastSpMM*) has proved that FastSpMM is the best approach in term of performance. A comparison of FastSpMM on two GPUs (Tesla C2050 and GeForce GTX480) has revealed that acceleration factors of up to 6.6× and 8.5× can be achieved in comparison to an optimized implementation of SpMM which exploits 8 cores of a state-of-the-art multicore processor. However, for very large and extremely sparse matrices, the SpMM achieves poor performance on the GPU, mainly due to the relevance of the CPU-GPU communications of the dense matrices (B and C).

According to the previous results, several aspects of the FastSpMM can be improved so, our future work will be focused on them. Concretely, our next work will consist of extending FastSpMM to matrices with complex elements instead of real ones and broadening the kinds of platforms which can be exploited by FastSpMM. Moreover, we are particularly interested in the reduction of the Processing Time through improving the memory management. In order to achieve this goal, FastSpMM will be re-written according to the GPU programming tool CudaDMA [35] which allows to efficiently managing data transfers between the on-chip and off-chip memories of GPU platforms. Finally, due to the importance of the L^* parameter in the performance of the SpMM product, it will be interesting to develop a model to optimize the value of this parameter according to the specific matrix characteristics and the GPU platform.

FastSpMM for computing SpMM on GPU platforms is freely available through the following web site: https://sites.google.com/site/mcfastsparse/

3

BiConjugate Gradient for complex matrices on GPUs

In a wide variety of applications from different scientific and engineering fields, the solution of complex and/or nonsymmetric linear systems of equations is required. To solve this kind of linear systems the BiConjugate Gradient method (BCG) is specially relevant. Nevertheless, BCG has a enormous computational cost. GPU computing is useful for accelerating this kind of algorithms but it is necessary to develop suitable implementations to optimally exploit the GPU architecture. In this chapter, we show how BCG can be effectively accelerated when all operations are computed on a GPU. The BiConjugate Gradient method has been implemented with two alternative routines of the Sparse Matrix Vector product(SpMV): the CUSPARSE library and the ELLR-T routine. Although our interest is mainly focused on complex matrices, our implementations have been evaluated on a GPU for two sets of test matrices: complex and real, in single and double precision data. Experimental results show that BCG based on ELLR-T routine achieves the best performance, particularly for the set of complex test matrices. Consequently, this method can be useful as a tool to efficiently solve large linear system of equations (complex and/or nonsymmetric) involved in a broad range of applications.

3.1 Introduction

It is widely accepted that the BiConjugate Gradient algorithm (BCG) is one of the best known iterative algorithms for solving nonsymmetric and/or complex systems with high accuracy [70, 127]. It can be used in a wide variety of applications such as electromag-

netism or tomography [100, 148]. However, each iteration of the BCG method involves a set of high computational cost operations. Therefore, High Performance Computing (HPC) can be a suitable way to accelerate this method. Currently, Graphics Processing Units (GPUs) are considered to be HPC platforms that offer massive parallelism, which are also useful for accelerating this kind of algorithms.

In the literature, several developments of the BCG method over GPUs have been implemented and evaluated for solving systems of equations which involve real matrices [68, 69]. In contrast, this chapter is focused on the acceleration of BCG based on complex arithmetic by means of GPU computing. Recently, De Donno et al. [54] have described a GPU implementation of the complex BCG algorithm based on the HYB kernel to compute the SpMVs included in the BCG method (the HYB kernel has been proposed by Bell et al. [37]). Nevertheless, more efficient kernels to compute the SpMV on GPUs have been designed; among them, ELLR-T has proved to outperform the aforementioned approaches [152].

GPU computing is suitable for accelerating the two kinds of vector operations involved in BCG: (1) inner products and (2) Sparse Matrix Vector products (SpMVs); where the SpMV is the operation with the highest computational cost in the BCG algorithm. Thus, it could be considered as the key in the development of this method on GPUs. In our implementation of BCG, both operations are computed on the GPU to reduce data transfer overheads. In this way, only an initial and final communication is necessary to transfer the input and output data, respectively.

In order to accelerate the SpMV on GPUs there are several CUDA libraries such as ELLR-T or HYB among others, but most of them deal with real arithmetic [34, 48, 152]. In contrast, CUSPARSE [5] is an example of a library which can compute SpMV with complex numbers. Our work has been focused on accelerating the SpMVs included in the BCG method using two alternative kernels: (a) the SpMV kernel from the library CUSPARSE [5], and (b) the ELLR-T kernel which is based on the ELLPACK-R format [145, 152], which has been adapted for complex numbers. In this chapter both alternatives are evaluated in depth and a quantitative analysis of the BCG profiling is carried out. Additionally, runtime performance gains are measured by executing our implementation on the GPU versus an analogous implementation in a current multicore architecture.

The remainder of this chapter is structured as follows: Section 3.2 briefly reviews the BCG method. Sections 3.3 and 3.4 are devoted to describing and evaluating our implementation of the BCG method on GPU. Comparisons to a multicore CPU implementation are also shown. Finally, Section 3.5 summarizes the main conclusions.

3.2 BiConjugate Gradient Method

The BCG method (proposed by Lanczos [91]) is a nonstationary iterative method to solve systems of linear equations $Ax = b$, where the matrix $A \in \mathbb{C}^{n \times n}$ is a sparse matrix which can be nonsymmetric, b indicates the independent term and x is the unknown vector.

The Conjugate Gradient method is not suitable for nonsymmetric systems because the residual vectors cannot be made orthogonal with short recurrences (for proof of this see Voevodin [153] or Faber and Manteuffel [64]). The GMRES method retains orthogonality of the residuals by using long recurrences, at the cost of a larger storage demand. The BiConjugate Gradient method takes another approach, replacing the orthogonal sequence of residuals by two mutually orthogonal sequences, at the price of no longer providing a minimization [53].

The update relations for residuals in the Conjugate Gradient method are augmented in the BiConjugate Gradient method by relations that are similar but based on A^T instead of A. Thus, two sequences of residuals

$$r_i = r_{i-1} - \alpha_i A p_i, \qquad r'_i = r'_{i-1} - \alpha_i A^T p'_i, \tag{3.1}$$

and two sequences of search directions have to be updated:

$$p_i = r_{i-1} + \beta_{i-1} p_{i-1}, \qquad p'_i = r'_{i-1} + \beta_{i-1} p'_{i-1}. \tag{3.2}$$

The choices

$$\alpha_i = \frac{r'^T_{i-1} r_{i-1}}{p'^T_i A p_i}, \qquad \beta_i = \frac{r'^T_i r_i}{r'^T_{i-1} r_{i-1}} \tag{3.3}$$

ensure the bi-orthogonality relations

$$r'^T_i r_j = p'^T_i A p_j = 0, \qquad \text{if } i \neq j. \tag{3.4}$$

Algorithm 5 describes a pseudocode for the BCG method. At every iteration, several inner products operations and two SpMVs are computed. Estimations of the

computational complexity (floating point operations) of the most expensive operations are provided in the right-hand columns (for real and complex numbers), where nz and n denote the number of non-zero elements and the number of rows and columns of A, respectively. At each BCG iteration, two SpMV operations are computed with matrices A and A^T (lines 9 and 17 of Algorithm 5). The computational cost of the SpMV operation is related to the value of nz, so it is higher than the remaining operations which mainly consist of inner products. Moreover, the SpMV for the transpose matrix (A^T) consumes more runtime due to the penalties related to the locality loss in the access to the elements of A^T. To overcome these penalties and to take advantage of the large amount of available memory resources on current computational platforms, our CPU and GPU implementations store both A and A^T as two different sparse matrices.

Algorithm 5 BiConjugate Gradient Method

		Real data	*Complex data*
Require:	Define $\epsilon =$ *Accuracy Threshold*		
Ensure:	The value of x_i		
1:	Compute $r_0 = b - Ax_0$ for some initial guess x_0		
2:	Choose $r'_0 = r_0$; $p'_0 = 0$; $p_0 = p'_0$ $\rho'_0 = 1$		
3:	Calculate $\Delta_0 = \|r_0\|$)		
4:	**for** $i = 1, 2, ...$ *until convergence* **do**		
5:	$\rho_i = (r'_{i-1}, r_{i-1})$	$O(2n)$	$O(8n)$
6:	$\beta_i = \rho_i / \rho'_{i-1}$		
7:	$p_i = r_{i-1} + \beta_i p_{i-1}$	$O(2n)$	$O(8n)$
8:	$p'_i = r'_{i-1} + \beta_i p'_{i-1}$	$O(2n)$	$O(8n)$
9:	$v_i = Ap_i$	$O(2nz)$	$O(8nz)$
10:	$\alpha_i = \rho_i / (p'_i, v_i)$	$O(2n)$	$O(8n)$
11:	$x_i = x_{i-1} + p^i \alpha_i$	$O(2n)$	$O(8n)$
12:	$r_i = r_{i-1} - v^i \alpha_i$	$O(2n)$	$O(8n)$
13:	$r'_i = r'_{i-1} - \alpha_i (A^T p'_i)$	$O(2nz + 2n)$	$O(8nz + 8n)$
14:	$\Delta_i = \|r_i\|$	$O(2n)$	$O(4n)$
15:	**if** $\Delta_i < \epsilon \Delta_0$ **then**		
16:	**return** x_i		
17:	**else**		
18:	$\rho'_i = \rho_i$		
19:	**end if**		
20:	**end for**		

3.2.1 Preconditioning BCG

The BCG method, similarly to other Krylov methods, could suffer from nasty side-effects such as stagnation or breakdown therefore, in some cases, it could not guarantee to lead to acceptable approximate solutions within modest computing time and storage.

The trick is then to try to find some nearby operator K such that $K^{-1}A$ has better (but still unknown) spectral properties. This is based on the observation that for $K = A$, we would have the ideal system $K^{-1}Ax = Ix = K^{-1}b$ and all subspace methods would deliver the true solution in one single step. The hope is that for a K in some sense close to A, a properly selected Krylov method applied to, for instance, $K^{-1}Ax = K^{-1}b$, would need only a few iterations to yield a good enough approximation for the solution of the given system $Ax = b$. An operator that is used for this purpose is called a preconditioner for the matrix A [142].

The general problem of finding an efficient preconditioner, is to identify a linear operator K (the preconditioner) with the properties that: (1) K is a good approximation to A in some sense; (2) the cost of the construction of K is not prohibitive; and (3) the new system of equations is much easier to solve than the original system.

There is no a general theory on which we can safely base an efficient selection. Except for some trivial situations, the matrix $K^{-1}A$ is never formed explicitly. In many cases this would lead to a dense matrix and destroy all efficiency that could be obtained for sparse matrices.

There are different ways of implementing preconditioning (Left, Right and Two-sided preconditioning); for the same preconditioner these different implementations lead to the same eigenvalues for the preconditioned matrices [142]. In Chapter 4 an algorithmic expression of a preconditioning BCG is shown.

3.3 GPU Implementation of the BCG Method

Our efforts have been focused on improving the performance of the BCG method by accelerating both: inner products and SpMV operations using GPU computing and CUDA. According to this model, each kernel is executed as a batch of threads organized as a grid of thread blocks whose configuration is defined by the programmer [7].

As previously mentioned, in the BCG method the SpMVs represent the operations with the highest computational cost, although inner products consume a significant

percentage of runtime. So, devoting some effort towards accelerating both kinds of operations on GPUs is a worthwhile undertaking. For this purpose, NVIDIA supplies a set of basic routines or libraries intended to speed up a wide variety of matrix operations on GPUs. Among them, several implementations of the SpMV based on CUDA have been reported in the literature [5, 16, 34, 48, 146].

Two CUDA implementations of the SpMV have been selected to develop the BCG method on GPUs: (1) the kernel included in the library CUSPARSE [5] and (2) the ELLR-T kernel, which is based on the format ELLPACK-R [145, 152]. We have used these kernels because the comparative evaluation described in [146] shows that ELLR-T achieves better performance than other approaches for the SpMV operation on GPUs for real arithmetic. However, the routine to compute SpMV of the CUSPARSE library is not included in the mentioned comparative evaluation, despite this library is currently the main tool provided by NVIDIA for the SpMV on GPUs.

The CUSPARSE library provides a set of basic linear algebra subroutines used for handling sparse matrices. Compress Row Storage (CRS) is the format to compress the sparse matrix [38]. The paradigm of CUSPARSE is to define multiple blocks where each block is in charge of processing a group of rows. Each row is assigned to a group of threads. Moreover, a few additional tweaks for improving performance are considered: (1) adjust the number of threads per row to minimize the imbalance among threads, (2) align threads per row for coalescing, and (3) use shared and texture memory [5]. However, CUSPARSE does not allow the user to select the value of the configuration parameter such as threads block size (BS) to achieve the optimal performance.

The ELLR-T routine with the format ELLPACK-R allows us to store the sparse matrix in a regular manner. In ELLR-T, every set of T threads computes one element of the output vector. The global memory access to the matrix is coalesced and aligned. According to the mapping of threads in the computation related to every row, several configurations of ELLR-T can be executed. To optimize the performance, the values of two parameters, number of threads (T) and threads block size (BS), have to be adjusted for each sparse matrix. These can be automatically determined by the specific configuration routine associated with ELLR-T [146, 152]. The central question to be examined in this chapter is the development and evaluation of a BCG implementation based on both kernels. The main features of our implementation are: (1) it is based on

Table 3.1: Characteristics of real and complex test matrices.

Real Mat	S	n	nz	Av	Complex Mat	S	n	nz	Av
wbp128	no	16384	3933097	240	kim1	no	38415	933195	24
cant	yes	62451	4007384	64	fem_filter	no	74062	1731206	23
pdb1HYS	yes	36417	4344766	119	NN70	yes	729000	5086618	7
consph	yes	83334	6010481	72	NN80	yes	1000000	6979798	7
shipsec1	yes	140874	7813404	55	NN90	yes	1331000	9292578	7
pwtk	yes	217918	11634425	53	kim2	no	456976	11330020	24
wbp256	no	65536	31413931	479	fem_hifreq	no	491100	20239237	41

CUBLAS library for accelerating inner products, and (2) two different kernels for SpMVs are tested (CUSPARSE and ELLR-T). Hereinafter, when CUSPARSE or ELLR-T routines are used, they will be referred to as $CuBCG_{CS}$ or $CuBCG_{ET}$, respectively.

The next section is devoted to evaluating our BCG implementation using both kernels: CUSPARSE and ELLR-T.

3.4 Evaluation

Two sets of sparse matrices (real and complex) have been considered for the evaluation. These matrices exhibit different characteristics, from those that are well-structured and regular to highly irregular matrices with large imbalances in the distribution of non-zeros per matrix row. Table 3.1 illustrates the set of real and complex test matrices considered with the characteristic parameters related to their specific patterns: symmetry of the matrix (S), number of rows (n), total number of non-zeros elements (nz) and average number of entries per row (Av). Notice that the dimension for all matrices is $n \times n$.

Our analysis is based on runtimes measured on a Tesla C2050 (see Chapter 1 Table 1.4). Reported results based on Tesla C2050 can be extrapolated to other Fermi platforms. This conclusion is based on similar results obtained from an additional study on a GTX 480 card (Fermi GPU with a higher peak performance), not reported here. In our experiments for evaluating the performance of CUSPARSE and ELLR-T kernels, we have used the optimal configuration of parameters T and BS for each test matrix. In the case of CUSPARSE, these parameters are determined by the CUSPARSE routine.

Table 3.2 shows the performance (GFlops) achieved for 1000 iterations of both implementations of BCG (columns $CuBCG_{CS}$ and $CuBCG_{ET}$). Table 3.2 also provides

values of the performance for the most computationally expensive procedures of BCG: (i) the performance of two SpMV operations (lines 9 and 17 of Algorithm 5) is shown in columns Ap_{CS}, $A^T p'_{CS}$, Ap_{ET} and $A^T p'_{ET}$, where the subindexes CS and ET are related to CUSPARSE and ELLR-T libraries, respectively; and (ii) the performance of inner products is shown in column IP. The IP column also shows, in parentheses, the percentage of inner products workload with respect to the total floating point operations of BCG. Moreover, for each matrix, results for executions using single and double floating point precision are provided (data in bold refers to double precision).

Experimental results show that performance of the SpMV operations for nonsymmetric matrices is quite different despite that both matrices, A and A^T, contain the same number of non-zeros elements. In fact, the performance is different because the row patterns of A and A^T are different. Experimental results have shown that data transfer overheads are negligible compared to the total GFlops ($\leq 1\%$), so they have not been included in Table 3.2. Results in Table 3.2 show that:

1. $CuBCG_{ET}$ outperforms $CuBCG_{CS}$ in terms of GFlops, specially for complex matrices where the computational burden of arithmetic operations is higher compared to real matrices.

2. The performance of SpMVs based on ELLR-T is better than the performance of CUSPARSE due to the fact that ELLR-T is tuned to the pattern of the sparse matrix through the parameters T and BS. It can be underlined that the advantages of ELLR-T over CUSPARSE are specially relevant for the SpMV with complex matrices (columns Ap_{CS}, $A^T p'_{CS}$, Ap_{ET} and $A^T p'_{ET}$).

3. Performance reached by all SpMV kernels is significantly higher than those of the inner products. This poor performance of inner products penalizes the performance of the BCG method (columns $CuBCG_{CS}$ and $CuBCG_{ET}$). It can be concluded that the low arithmetic intensity of inner products considerably reduces the performance of BCG despite the high performance of ELLR-T kernels.

4. The performance of inner products increases as n rises because the arithmetic intensity of inner products is proportional to the dimension of n. Therefore, the highest performance for the inner product is achieved for the largest values of n,

Table 3.2: GFlops of 1000 iterations of both implementations of BCG (columns $CuBCG_{CS}$ and $CuBCG_{ET}$) using the two sets of matrices in single and double precision (typed in bold). IP column shows the GFlops achieved by inner products and Ap_{CS}, $A^T p'_{CS}$, Ap_{ET} and $A^T p'_{ET}$ columns show the GFlops measured for both kinds of SpMV using CUSPARSE and ELLR-T libraries, respectively.

	Real matrices						
	IP	Ap_{CS}	$A^T p'_{CS}$	$CuBCG_{CS}$	Ap_{ET}	$A^T p'_{ET}$	$CuBCG_{ET}$
wbp128	0.3 (2%)	13.4	13.3	8.1	18.8	16.2	9.9
	0.3	**10.8**	**10.8**	**7.1**	**12.0**	**10.5**	**7.4**
cant	1.3 (6%)	8.8	8.8	6.6	17.9	17.9	10.2
	1.0	**7.9**	**7.9**	**5.7**	**10.7**	**10.7**	**7.4**
pdb1HYS	0.8 (3%)	11.6	11.6	8.0	16.8	16.8	10.4
	0.7	**10.0**	**10.0**	**7.2**	**11.1**	**11.1**	**8.0**
consph	1.6 (5%)	9.4	9.4	7.5	16.5	16.5	11.5
	1.4	**8.4**	**8.4**	**6.7**	**9.7**	**9.7**	**8.0**
shipsec1	2.4 (7%)	8.5	8.5	7.3	16.3	16.3	12.2
	2.0	**7.6**	**7.6**	**6.4**	**9.8**	**9.8**	**8.5**
pwtk	3.3 (7%)	8.5	8.5	7.7	18.8	18.8	14.1
	2.2	**7.6**	**7.6**	**6.7**	**11.6**	**11.6**	**9.9**
wbp256	1.3 (1%)	13.5	12.9	12.3	18.0	14.9	14.8
	1.2	**9.3**	**8.7**	**8.5**	**11.1**	**9.9**	**9.9**

	Complex matrices						
	IP	Ap_{CS}	$A^T p'_{CS}$	$CuBCG_{CS}$	Ap_{ET}	$A^T p'_{ET}$	$CuBCG_{ET}$
kim1	2.8 (14%)	14.5	14.5	9.3	45.4	45.8	15.5
	1.4	**8.4**	**8.3**	**4.9**	**26.3**	**26.5**	**8.1**
fem_filter	4.5 (14%)	13.4	13.5	10.6	28.9	28.9	16.7
	2.5	**8.1**	**8.1**	**6.2**	**15.9**	**15.9**	**8.9**
NN70	15.7 (35%)	17.4	17.4	16.9	44.1	44.1	27.5
	7.7	**12.4**	**12.4**	**10.2**	**24.9**	**24.9**	**14.7**
NN80	16.9 (35%)	17.8	17.8	17.5	44.4	44.4	28.8
	8.3	**12.5**	**12.5**	**10.6**	**25.1**	**25.1**	**15.4**
NN90	17.8 (35%)	18.3	18.3	18.2	43.5	43.5	29.3
	8.7	**12.8**	**12.8**	**11.0**	**23.8**	**23.8**	**15.4**
kim2	13.6 (13%)	18.3	18.3	17.5	51.6	51.6	37.7
	6.8	**9.9**	**9.9**	**9.3**	**29.0**	**29.1**	**20.2**
fem_hifreq	14.0 (8%)	26.0	26.0	24.3	42.2	42.2	35.6
	6.9	**14.8**	**14.8**	**13.5**	**25.3**	**25.3**	**20.8**

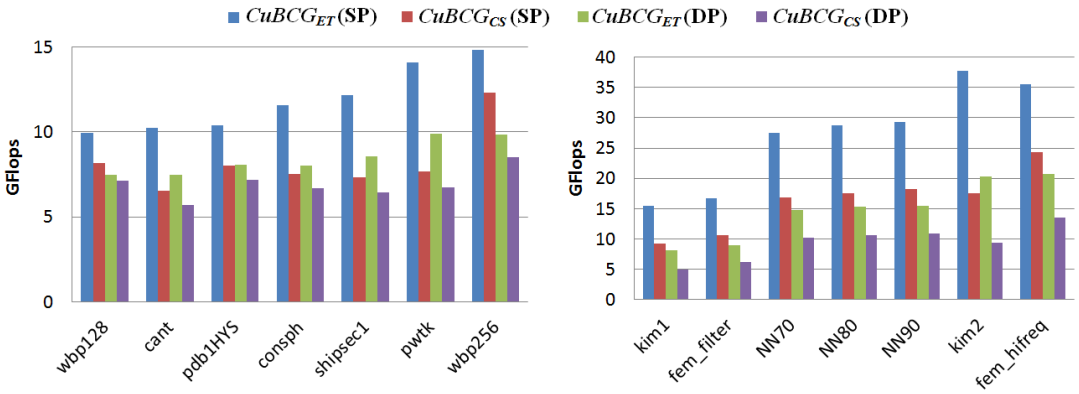


Figure 3.1: Performance of BCG approaches ($CuBCG_{CS}$ and $CuBCG_{ET}$) on GPU Tesla C2050 using real (left)/complex (right) matrices and single/double precision (SP/DP).

as is the case of *shipsec*1 and *pwtk* (for real matrices) and $NN70$ and $NN80$ (for complex matrices).

5. Bearing in mind the Amdahl's Law, the performance of BCG is dominated by inner products or SpMVs according to the values of n and nz; e.g., for the complex matrices $NN70$, $NN80$ and $NN90$, the percentage of workload of inner products is nearly 35% (see percentage in IP column), consequently penalties on the performances of inner products are more relevant for these matrices. However, for $kim2$ and fem_hifreq, the SpMV is the dominant computation due to the high values of nz. For real matrices, the impact of inner products performance is less due to their low percentage of workload, so the BCG performance is strongly determined by nz.

Figure 3.1 graphically shows the performance (GFlops) reached by $CuBCG_{CS}$ and $CuBCG_{ET}$ for real and complex data using single and double precision. It clearly shows that $CuBCG_{ET}$ outperforms $CuBCG_{CS}$. Hereinafter, our analysis is focused on the $CuBCG_{ET}$ because it achieves better performance.

Table 3.3 shows total runtimes of 1000 iterations of the BCG method based on ELLR-T on a GPU card Tesla C2050. Due to the high impact of the inner products in the total runtime of the BCG method, the percentage of the total runtime spent in the calculation of the inner products is also provided (in parentheses). Let notice that it is significantly higher than the percentage of inner products workload (see Table 3.2).

Results in Table 3.3 clearly shows that runtimes range from 1 to 10 seconds for single precision and from 1 to 16 seconds for double. These results are considered as reference in the following analysis which is devoted to evaluating the performance of BCG on multicore and GPU platforms.

Table 3.3: Runtimes (s) of 1000 iterations of the BCG method based on ELLR-T on a GPU card Tesla C2050, in single (SP) and double precision (DP). Values in parentheses show the percentage of the total runtime which is spent in the calculation of the inner products.

	Real matrix			Complex matrix	
	SP	*DP*		SP	DP
wbp128	1.50 (43.8%)	**2.00 (34.7%)**	kim1	1.03 (70.5%)	**1.98 (73.4%)**
cant	1.55 (46.1%)	**2.13 (34.2%)**	fem_filter	1.79 (50.1%)	**3.36 (51.6%)**
pdb1HYS	1.61 (40.4%)	**2.08 (29.9%)**	NN70	4.24 (59.4%)	**7.91 (61.6%)**
consph	2.04 (33.5%)	**2.95 (21.7%)**	NN80	5.55 (57.9%)	**10.39 (60.1%)**
shipsec1	2.56 (30.4%)	**3.64 (18.3%)**	NN90	7.27 (56.2%)	**13.78 (57.7%)**
pwtk	3.30 (30.1%)	**4.71 (20.9%)**	kim2	5.16 (36.6%)	**9.60 (39.5%)**
wbp256	7.96 (9.9%)	**11.96 (6.6%)**	fem_hifreq	9.25 (22.8%)	**15.82 (24.8%)**

In order to obtain the net gain provided by GPUs versus multicore architectures, a state-of-the-art processor [95] and a GPU platform based on Fermi architecture of NVIDIA [7] have been used in our experiments. The CPU architecture is an Intel Xeon E5640 (8 cores, 2.66 GHz, 12 GB RAM and under Linux) which uses the multithreaded Intel MKL library [1]. For the GPU implementation we have considered the above-mentioned GPU card (Tesla C2050) and the BCG implementation based on ELLR-T and CUBLAS.

Table 3.4 shows the acceleration factors obtained for $CuBCG_{ET}$ against the BCG executed on 1, 2, 4, and 8 cores. Results in Table 3.4 show that for 1 core, the acceleration factor ranges from 4.8× to 16.8× and from 4× to 15.7× for single and double precision, respectively; and for 8 cores, the acceleration factor ranges from 1.2× to 5.3× and from 1.8× to 6.8× for single and double precision, respectively.

Experimental results show that, for the test matrices used in the study (real/complex, single/double precision), our BCG method on GPUs clearly outperforms the optimized multicore implementation for up to 8 cores. These results constitute proof of the advances in double precision arithmetic in modern GPUs of NVIDIA based on Fermi

Table 3.4: Acceleration factor for the $CuBCG_{ET}$ method against the BCG multicore version using MKL with 1, 2, 4 and 8 cores (1C, 2C, 4C and 8C). The two sets of matrices have been considered in single and double precision (typed in bold).

	Real matrix			
	1C	2C	4C	8C
wbp128	8.4	6.3	3.9	2.8
	11.1	**7.4**	**4.6**	**3.2**
cant	8.6	6.3	3.8	2.6
	7.2	**6.4**	**4.9**	**3.0**
pdb1HYS	7.9	4.7	3.9	2.9
	8.3	**6.8**	**5.3**	**3.1**
consph	9.1	7.1	4.3	3.0
	7.6	**4.6**	**3.8**	**3.2**
shipsec1	10.2	7.6	4.6	3.2
	9.2	**7.4**	**4.2**	**3.6**
pwtk	11.8	8.7	6.2	3.3
	10.6	**8.5**	**6.7**	**4.0**
wbp256	16.8	10.3	7.0	3.4
	15.7	**9.7**	**6.4**	**3.3**

	Complex matrix			
	1C	2C	4C	8C
kim1	4.8	3.4	2.6	1.2
	4.0	**3.1**	**2.5**	**1.8**
fem_filter	6.5	4.3	3.2	2.0
	4.8	**4.0**	**3.0**	**2.2**
NN70	12.2	8.5	5.8	4.6
	9.0	**6.2**	**5.6**	**5.4**
NN80	11.9	7.1	7.2	5.0
	10.7	**6.6**	**6.0**	**5.7**
NN90	13.4	9.4	7.4	5.3
	9.6	**9.0**	**6.7**	**6.8**
kim2	11.5	9.0	7.0	4.5
	9.0	**8.4**	**6.9**	**5.0**
fem_hifreq	14.7	10.6	7.0	4.6
	11.7	**7.3**	**5.1**	**4.7**

architecture.

3.5 Conclusions

We have analyzed and experimentally evaluated a BCG implementation to solve complex and/or nonsymmetric linear systems of equations on GPUs. This work compares two different versions of the BCG method ($CuBCG_{CS}$ and $CuBCG_{ET}$), which are based on CUSPARSE and ELLR-T libraries to accelerate the SpMV, since it is the key operation in the BCG. Moreover, all inner products have been accelerated by means of the CUBLAS library.

After an extensive study with two sets of representative test matrices, it can be concluded that $CuBCG_{ET}$ clearly achieves better performance due to the fact that the kernel ELLR-T can be better adapted to the different patterns of the sparse matrices.

This advantage is more relevant for complex matrices since the consideration of complex arithmetic involves more floating point operations. Furthermore, a comparison of $CuBCG_{ET}$ on a Tesla C2050 has revealed that acceleration factors range from 1.2× to 5.3× for single precision and from 1.8× to 6.8× for double precision, in comparison to an optimized implementation of the BCG which exploits a state-of-the-art processor with eight cores. Finally, we can conclude that $CuBCG_{ET}$ exploits the GPU platform better than other approaches described in the literature.

The analysis has shown that despite the small percentage of workload related to inner products, their poor performance has a strong impact on the performance of the BCG. Therefore, in our future work, we plan to follow the methodology described in Chapter 1 (Section 1.2.1.1) with the objective of expressing every iteration of the BCG with the fusion of kernels. Several advances have been already carried out in this research line [137].

4 The 3D Helmholtz equation on multi-GPU clusters

The resolution of the 3D Helmholtz equation is required in the development of models related to a wide range of scientific and technological applications. The Helmholtz equation, named for the German physicist Hermann von Helmholtz, is a partial differential equation that governs the scattering of plane wave in acoustics and electromagnetism [73].

This chapter is focused on the study of this Helmholtz equation. In Section 4.1 Mathematical and Physical aspects of the 3D Helmholtz equation are introduced. Section 4.2 summarizes our contributions to the computational aspects of the BCG method for solving the 3D Helmholtz equation (BCG-3DH). This section also gives an overview of the state of the art with respect to this issue, reviewing previous implementations of iterative solvers for the Helmholtz equation. Section 4.3 describes the BCG-3DH algorithm which has been implemented for the resolution of the Helmholtz equation. Section 4.4 presents a new compressed storage format, called "Compressed Regular Format (CRF)", specially designed to take advantage of the regularities appearing in the sparse matrix associated to the Helmholtz equation. Section 4.5 describes the implementation details of three different strategies: CPU and GPU versions and a Hybrid CPU-GPU version (called Fast-Helmholtz), where several optimizations (fusion and reordering) have been considered for increasing performance. In Section 4.6 performance evaluations of the Fast-Helmholtz approach against the CPU and GPU versions are provided. Section 4.7 ends this chapter with some conclusions and future lines to work on.

4.1 Mathematical and Physical view of the 3D Helmholtz equation

Many mechanical, acoustical, thermal and electromagnetic wave problems can be modeled by means of the Helmholtz equation, and it is specifically relevant in problems of wave scattering and fluid-solid-interaction [50, 74, 82]. It represents time-harmonic wave propagation in the frequency domain. The 3D Helmholtz equation in the scalar approximation is defined by the following linear elliptic Partial Differential Equation (PDE):

$$(\nabla^2(\mathbf{r}) + k(\mathbf{r})^2)E(\mathbf{r}) = 0, \tag{4.1}$$

where ∇^2 is the Laplace operator; E is a complex scalar function defined at a spatial point $\mathbf{r} = (x, y, z) \in \mathbb{R}^3$ and $k(\mathbf{r})$ is some real or complex constant. This equation naturally appears in general conservation laws of physics and can be interpreted as a wave equation for monochromatic waves (wave equation in the frequency domain), where $k(\mathbf{r})$ is related to the wave-number [129]. This equation can be numerically solved by means of an appropriated transformation based on Green's functions and a spatial discretization [73, 129]. Both Finite Difference Method (FDM) and Finite Elements Method (FEM) are suitable for the discretization of this kind of differential operator [78]. In this thesis FEM is used.

Finite Element Methods (FEM) are based on the variational formulation of the problem which is obtained when the equation is multiplied by test functions and integrated over the domain using integration by parts [41, 61]. For discrete approximation of the problem, the function space S on which the variational formulation is defined is replaced by a finite dimensional subspace S_h. The approximation u_h of the solution u is expressed as a linear combination of the finite number of basis functions $\phi_j(x)$ which are defined to be continuous and non-zero only on small subdomains. In that way, this type of problems are transformed into a sparse system of equations. A great advantage of FEM is that it is mathematically well motivated with rigorous error estimates. FEM is thus also very well suited for the problems with complex geometries [123].

Finite element method leads to sparse systems of equations with the number of unknowns n that depends on the frequency/wavelenght, $n \backsim \omega$. For high frequencies,

those systems are large and direct methods like Gaussian elimination become computationally too expensive. The alternative is to use iterative methods. However, there is a difficulty in this approach, as well. Since the system of equations is indefinite and ill-conditioned, iterative methods have slow convergence rate. The convergence rate can be improved by preconditioning, but finding a good preconditioner for Helmholtz equation is still a challenge [62]. The indefiniteness makes it difficult to use multigrid and the conjugate gradient method. Typically Krylov subspace methods, like GMRES, BCG and BCGSTAB are used instead. Preconditioners can for instance be based on incomplete LU-factorization [101] or on fast transform methods [59, 63, 92, 119].

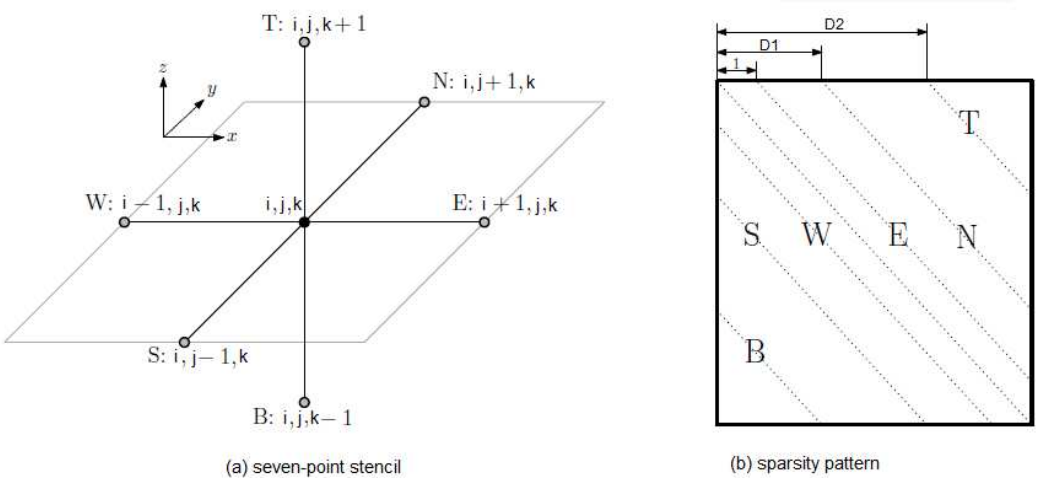


Figure 4.1: Illustration of discretization stencil and the linear system (source [52]).

4.2 BCG-3D Helmholtz on multi-GPU clusters

For solving this equation in complex arithmetic, the BiConjugate Gradient method is one of the most relevant solvers. However, this iterative method has a high computational cost because of the large sparse matrix and the vector operations involved. In this section, a specific BiConjugate Gradient Method (BCG), adapted to the regularities of the Helmholtz equation (BCG-3DH algorithm) is presented. This BCG method is based on the implementation of a novel format (named "Compressed Regular Format (CRF)") that allows the storage of the large sparse matrix involved in the SpMV in a compact form. The contribution of this chapter is twofold: (i) decreasing the memory

requirements of the 3D Helmholtz equation using the CRF and (ii) speeding up the resolution of the equation using High Performance Computing (HPC) resources.

To do a fair comparison between the benefit of accelerating the 3D Helmholtz equation using a heterogeneous multi-GPU cluster, this chapter describes three different parallelization schemes and also includes an evaluation of their performance. The three parallel schemes consist of using:

1. Multicore processors (CPU version).

2. GPU devices (GPU version);

3. The combination of CPU cores and GPU devices (Hybrid version). Experimental results show that this hybrid implementation (called Fast-Helmholtz) outperforms the other approaches. Fast-Helmholtz combines optimizations at MPI and GPU levels to reduce communications cost and to improve the exploitation of the GPU architecture. This strategy makes possible to increase the discretization level of the Helmholtz problem, thanks to the relevant reduction of memory requirements and runtime.

Our goal is to develop a parallel solution for the BCG-3DH which combines the exploitation of the high regularity of the matrices involved in the numerical methods and the massive parallelism supplied by heterogeneous architecture of modern multi-GPU clusters [20].

For programming heterogeneous multi-GPU clusters two parallel interfaces are used, MPI and CUDA [80]. Our hybrid version launches two MPI processes which exploit the CPU cores and the GPU devices of the multi-GPU cluster, respectively.

This way, both types of MPI processes can collaborate to exploit all the resources of the heterogeneous architecture. The main advantages of the hybrid implementation are related to two issues. On one hand, a hybrid version is capable of exploiting the different kinds of resources in multi-GPU clusters. This is due to the fact that CPU cores, despite being slower than the GPU devices, will be able to collaborate to the GPU device and accelerate the global process by the overlapping the computation on CPUs and GPUs. Note that the workload of CPU processes will be significantly less than the workload of the GPU processes. On the other hand, our hybrid approach will overcome the memory limitation of the GPU devices because it can exploit the large

memory resources of multicore nodes. GPUs are able to accelerate computations but the GPU workload is limited due to the small size of its memory. CPU processes can collaborate with the GPU processes to exploit the large memory of multicore nodes.

In this chapter, BCG-3DH, a parallel solver of the 3D Helmholtz equation based on BCG method, is proposed and evaluated on multi-GPU clusters. The main contributions of this implementation are:

- It translates the regularities of the 3D Helmholtz problem into computational regularities that allow a better exploitation of the memory resources of the platforms. That is, considering the regularities of the involved matrices, we have defined specific data structures to reduce (1) the number of floating point operations; (2) the memory requirements; and (3) the communication penalties.

- A hybrid multi-GPU cluster is exploited. This heterogeneous platform allows to extract the parallelism of the CPU and GPU devices available in the cluster. This implementation is based on the combination of two parallel interfaces: MPI and CUDA. MPI paradigm is used for distributing the linear system among the different nodes. In addition, the computation on every node is accelerated using CPUs and GPUs. Furthermore, fusion and reordering techniques specifically designed for the BCG method have been used in both CPUs and GPUs. As a result, local Sparse Matrix Vector products (SpMVs) and local vector operations included at every iteration of the BCG method have been accelerated by means of the massive parallelism of multi-GPU platforms.

Our parallel solution of the 3D Helmholtz equation considerably reduces memory requirements and runtime, extending the resolution of problems of practical interest to several different fields of Physics, such as the new Non-Linear Optical Diffraction Tomography model (NLODT-P) that will be described in Chapter 5.

4.2.1 Related works

The solution of large sparse linear systems has been widely studied as shown in the literature on the iterative methods [31, 72, 127, 142]. The resulting systems from the Helmholtz equation are complex and symmetric. In general, the Krylov subspace methods based on Lanczos biorthogonalization are effective for solving complex systems in

terms of convergence properties and memory requirements [27]. Examples of this are the BiConjugate Gradient method (BCG) and the BiConjugate Gradient Stabilized method (BCGSTAB), which are variants of the Conjugate Gradient (CG) for unsymmetric systems [127]. The Conjugate Orthogonal Conjugate Gradients method (COCG) is an adaptation of CG for solving symmetric complex systems [143]. BCG, BCGSTAB and COCG are suitable for solving the resulting systems from the Helmholtz equation. They are based on one-term recurrence which combines vector operations (dot, saxpy) and Sparse Matrix Vector products (SpMV).

In this context, the computational resources needed to solve the Helmholtz PDEs are enormous and require the use of High Performance Computing techniques (HPC). Literature describes a variety of parallel implementations for solving PDEs which exploit several different kinds of parallel platforms. For example, PETSc is a well-known parallel library for solving PDEs. It contains a few parallel versions supporting MPI, Pthreads, and NVIDIA GPUs, as well as hybrid parallelism [12]. However, currently, some PETSc routines based on CUDA are still being tested. There are no benchmarking results published for the multi-GPU implementation using complex numbers. Consequently, the solution of Helmholtz PDEs has not been still implemented to exploit multi-GPU clusters.

In [96], an efficient solution of the 3D Helmholtz equations based on the conjugate residual (GCR) algorithm is proposed to be included in a prediction model in Global/Regional Assimilation and Prediction System (GRAPES). The paper proposes improvements for the GCR iteration by the refactorization of the GCR algorithm, which decreases communication overhead on multicore clusters. The performance evaluation provided in [96] records speed-ups of 10 using 2048 cores with respect to 256 cores. In [87], the authors describe a multi-GPU implementation of the BCGSTAB solver preconditioned by a shifted Laplace multigrid method for the 3D Helmholtz equation. In the implementation, the sparse matrix is stored in a CRS matrix format which does not take any advantage of the regularities of the Helmholtz equation. Moreover, in the considered multi-GPU architecture all the GPUs are located in the same node using Pthreads for the parallelization, which results in a strong limitation on the number of GPUs available since it is not possible to use GPUs located in different nodes. In [40], the parallelization of the Fast Multipole Method (FMM), as applied to scattering problems, has been analyzed using the Helmholtz equation. This paper studies the

performance on shared memory architectures and makes some preliminary tests using a GPU device. The results obtained are promising, and as future works they propose the inclusion of MPI programming to use distributed memory. In [89], a specific MPI parallel approach for solving 2D Helmholtz equations is described. However, they do not take advantage of the regularity of the sparse matrices involved in the Helmholtz PDEs.

The solution of Helmholtz PDEs based on an iterative solver is expressed as a recurrence relation which includes several kinds of vector operations and sparse matrix-vector products (SpMV) that consume a relevant percentage of runtime. SpMV computation implies a challenge to exploit the architectures, and their performance strongly depends on the specific compressed storage format of the sparse matrix. Literature describes a wide variety of formats for optimizing the computation with sparse matrices on specific architectures (see Chapter 2, Section 2.1). These formats define the locality or the coalescence of memory access for the SpMV, which are essential in order to optimize performance on CPU or GPU architectures. Several optimizations have been proposed to improve the performance of sparse computation on current multicore platforms [1, 156] and on short-vector SIMD architectures [90]. Also, several implementations of SpMV have been developed with CUDA on GPUs [5, 37, 43, 106, 145, 152]. However, when the pattern of the sparse matrix exhibits any kind of regularity, the corresponding SpMV can improve its performance taking this regularity into account (Section 1.2 of [70] analyzes usual structured matrices involved in applications and [147] describes the advantages in terms of performance of the tomographic reconstruction when the regularity of the sparse matrix is considered). Focusing on the 3D Helmholtz equation, the pattern of the matrices consists of a main diagonal and six sub-diagonals with the same entries, so it is extremely regular. The exploitation of this regularity can strongly reduce the memory requirements and improve the performance of their computations.

4.3 BCG-3DH algorithm

Many numerical solvers of Equation 4.1 based on the Finite Differences Method (FDM) or the Finite Element Method (FEM) have been developed [29, 79]. FEM discretizes the region of interest in small elements, assuming the function $E(\mathbf{r})$ can be approximated to a constant value in each of these elements.

A similar system of equations can be obtained by a FDM approach assuming the spatial derivatives of the Laplace operator can be discretized with a seven-point stencil in 3D (see Figure 4.1). As a consequence, for a three-dimensional mesh, the linear system of equations resulting from Equation 4.1 is also described by a matrix with only seven non-zero diagonals.

Therefore, FDM and FEM transform the 3D Helmholtz equation into a linear system of equations $Ax = b$, where b indicates the independent term, x is the unknown vector, and matrix $A \in \mathbb{C}^{n \times n}$ is sparse, symmetric, very large (depending on the number of spatial discretization points, n) and only contains seven non-zero diagonals.

As mentioned above, there are several Krylov subspace methods which are suitable for solving the systems resulting from the Helmholtz equation. We have focused our attention on the BCG method and developed an approach to accelerate the resolution of the Helmholtz equation. The methodology and techniques developed in this chapter can also be straightforward applied to the resolution of the Helmholtz equation based on other Krylov solvers.

It is necessary to underline that the Krylov subspace methods may sometimes converge slowly. As a result, in practice, some kind of preconditioning has to be applied to improve their convergence [32, 142]. Focusing our attention on the BCG method for solving the 3D Helmholtz Equation, the standard BCG [142] can be preconditioned preserving the regularities of A. Algorithm 6 describes a pseudocode for the preconditioned BCG for solving the 3D Helmholtz Equation [31].

The key operations in the BCG method are two kinds of vector operations (dot and saxpy) and two sparse matrix vector products (SpMVs) computed at every iteration. It is remarkable that the symmetry of the Helmholtz equation has allowed the use of A instead of A^T.

In such algorithm, fusion and reordering optimization techniques have been used to improve the performance of the BCG method. These optimizations, referred as "Fusion 0-3", were explained in Chapter 1, Section 1.2.1.1, are the combination or fusing of multiple BLAS operations, to alleviate the memory bottleneck, to increase the operational intensity and to improve the total performance of the solver.

In Algorithm 6, the complexity of the most computationally expensive operations is shown in parenthesis, where n is the matrix size. In the BCG method, the complexity of the SpMV operation (line 9) is related to the number of non-zero elements of the matrix

Algorithm 6 Preconditioned BCG-3DH Method

Require: Define $\epsilon = Accuracy\ Threshold$ Complex data

Ensure: The value of x

1: Compute $r_0 = b - Ax_0$ for some initial guess x_0
2: Choose $r'_0 = r_0$; $p'_0 = 0$; $p_0 = p'_0$; $\rho'_0 = 1$
3: Calculate $\Delta_0 = \|r_0\|$
4: **for** $i = 1, 2, ...$ *until convergence* **do**
5: solve $Mx_{i-1} = r_{i-1}$; $M^T x'_{i-1} = r'_{i-1}$ **(Fusion 0: 2Msystem)** $O(f(M))$
6: $\rho_i = (x'_{i-1}, r_{i-1})$ $O(8n)$
7: $\beta_i = \rho_i / \rho'_{i-1}$
8: $p_i = x_{i-1} + \beta_i p_{i-1}$; $p'_i = x'_{i-1} + \beta_i p'_{i-1}$ **(Fusion 1: 2saxpy)** $O(16n)$
9: $q_i = Ap_i$; $q'_i = A^T p'_i$ **(Fusion 2: 2Ax)** $O(38n)$
10: $\alpha_i = \rho_i / (p'_i, q_i)$ $O(8n)$
11: $x_i = x_{(i-1)} + \alpha_i p_i$; $r_i = r_{i-1} - \alpha_i q_i$; $r'_i = r'_{i-1} - \alpha_i q'_i$ **(Fusion 3: 3saxpy)** $O(24n)$
12: $\Delta_i = \|r_i\|$ $O(4n)$
13: **if** $\Delta_i < \epsilon \Delta_0$ **then**
14: **return** $x = x_i$
15: **else**
16: $\rho'_i = \rho_i$
17: **end if**
18: **end for**

(nz). In our particular BCG the complexity order is $nz = 7n$. In the next section, the regularities of the sparse matrix that allow us to decrease the computational complexity of the algorithm are explained in detail.

4.4 Compressed Regular Format

In general, Compressed Row Storage (CRS) is the most widespread format for storing sparse matrices in the memory of multicore and clusters platforms [38]. For GPUs, other formats such as ELLR-T (see Chapter 2, Section 2.1) can be more appropriate. ELLR-T has shown to outperform other formats for the SpMV operation on GPUs for real/complex arithmetic [113, 145, 152].

For structure sparse matrices, the performance of SpMV can be improved when its regularity is taken into account (Section 1.2 of [70] analyzes this effect for a set of structured matrices taken from real applications). Bearing in mind that the matrix involved in the Helmholtz equation has several regularities, our proposal consists of

defining a specific storage format for these regular sparse matrices which leads to reduce both, the memory requirements and the number of float operations for computing SpMV. This kind of format has been referred to as CRF.

The regularities of the matrix involved in the 3D Helmholtz equation can be summarized as follows (see Figure 4.1(b)):

1. A is a symmetric matrix;
2. There is a maximum of seven non-zero elements per row;
3. Non-zero elements are located at seven diagonals in the matrix, where one is the main diagonal, two of them are the first lower and upper diagonals and four of them are located at $\pm D1$ and $\pm D2$ from the main diagonal;
4. All the elements of every lateral diagonal are equal (a, b, c);
5. The main diagonal is defined by a set of complex numbers.

From these characteristics, the CRF proposed in this chapter consists of:

1. One array, $A[]$ (complex) of dimension n;
2. Three integer values (a, b, c) to store all remaining entries;
3. Two additional integer values ($D1$, $D2$) which specify the displacements of the lateral diagonals with respect to the main diagonal.

CRF optimally minimizes the amount of data needed to store the sparse matrix. In this way, the computation time of SpMV is decremented due to the fact that: the number of memory accesses to read the elements of the sparse matrix is reduced; and the number of float operations when $a = b = c = 1$, because six complex multiplications can be avoided.

CRF significantly reduces the memory requirements with respect to the CRS format in a factor of 9. According to the Roofline performance model (see Chapter 1, Section 1.1.3.2) the total number of FLOPs performed divided by the total number of memory accesses (operational intensity) is an estimation of the maximum attainable performance of a computer algorithm. Therefore, the operational intensity of SpMV based on CRF is 1.6 times larger than for the CRS format if $a = b = c = 1$. As a result,

the use of the CRF has a strong impact on the performance of SpMV involved in the resolution of the 3D Helmholtz equation. These optimizations (derived from the use of this format) do not depend on the architecture model where SpMV is computed. They have been integrated into the Multi-GPU Cluster version of the Helmholtz equation.

4.5 BCG-3DH Implementations: CPU, GPU, Hybrid versions

Modern high performance architectures are characterized by the heterogeneity of their resources. So, parallel implementations have to be tuned for each particular heterogeneous architecture. In this context, the main challenge is to determine an appropriate workload distribution because the computational burden assigned to every processing element should be related to its computational power. Therefore, it is necessary to have a detailed knowledge of both the computing resources and the algorithm [93].

In this section, we study the parallel implementation of the 3D Helmholtz equation based on the exploitation of heterogeneous resources of multi-GPU clusters (multicores and GPU devices). Exploiting the heterogeneous platforms of a cluster has two main advantages: (1) larger problems can be solved because the code can be distributed among the available nodes; and (2) runtime is reduced since more operations are executed simultaneously at different nodes and accelerated by the parallel execution on CPU and GPU architectures. The three implementations considered are described as follows:

1. The CPU version is based on the distribution of the problem among several cores located into different nodes. MPI paradigm is used for processing tasks on cores which can belong to nodes with shared or distributed memory. Note that for solving the SpMV operations, we account the regularity of the matrix A for establishing a strategy that allows the reduction of the number and size of the messages among the available processors. So, to compute the SpMV, each MPI process adds redundant elements to the chunk of the vector (at the beginning and at the end). The number of additional elements is fixed ($2 \cdot D2$), hereinafter referred as "halo". These "halos" will be exchanged among the processors for updating the value of the vector before the computation of every SpMV. So, the use of redundant halos reduces the size of the messages and the number

of communications among processes to compute the SpMV. This fact allows to exploit the regularity of A optimizing the communication pattern.

2. The GPU version is based on the distribution of the problem among several GPU devices that can be located in the same or in a different node. Our proposal is the development of a parallel code capable of exploiting the benefits of distributed computing and GPU devices allocated in a multi-GPU cluster. Then, two different strategies of programming are considered: (1) the distribution of the computations by MPI [132]; and (2) the exploitation of the GPU devices using CUDA [14]. In this chapter, an algebraic library (CUBLAS [4]), which provides commonly used subroutines, has been considered. As it is well known, the key requirement for obtaining effective acceleration on GPU devices is the minimization of data transfer between the memories of the CPU and GPU. So, CPU-GPU communications are strongly reduced using the strategy based on "halos", also applied in the CPU version.

3. The Hybrid version has been designed to exploit all the computing resources of heterogeneous architectures. The idea behind this last implementation is the collaboration among CPU and GPU processes to accelerate the BCG algorithm by the exploitation of the full variety of available resources of multi-GPU clusters using also the "halos" strategy. Here it is important to highlight that a static workload balance scheduling has been considered because the workload of the application is known at compile time and constant during the execution. So, the distribution among processes can be done at compile time. This hybrid version presents two main advantages: (1) Despite CPU is slower than GPU, the parallel computation of GPU and CPU processes contributes to accelerate the algorithm execution. It has been done by setting up an efficient CPU-GPU collaboration, by an appropriate balance of the workload assigned to each process; and (2) the hybrid version can exploit the large memory resources of multicore nodes. This way, the memory limitation of the GPUs is overcome by the inclusion of CPU processes in the hybrid strategy.

The exploitation of the resources of multi-GPUs has been based on the definition of two kind of MPI processes: (1) Processes which exploit every core of the node

and (2) Processes which exploit accelerators (GPUs). Using different configuration of this implementation enables the exploitation of only CPUs, only GPUs or the hybrid (CPUs+GPUs).

4.5.1 MPI tasks

In the three implementations of BCG-3DH, MPI paradigm has been considered for the communication among processors [132]. It should be noted that one MPI process per CPU (GPU) is started. Parallelization has been carried out according to the data parallel concept. Following this concept, all the data structures (matrix, solution vector and subsidiary storage vectors) have been distributed among the processors of a cluster using a homogeneous partition.

The solution vector of the Helmholtz equation is performed in a parallel regime. Then, each MPI-process outputs a part of the solution vector which is locally stored into its own local memory. Output data can be merged into the global solution output file, if necessary.

The decomposition method used for the data structures is a kind of row-wise matrix decomposition which has been used for solving similar problems [87, 89].

Important issues are the communications among processing units which occur twice at every iteration: (1) when computing the two SpMV operations; and (2) during the reduction operations required for obtaining the results of dot operations.

Focusing our attention on the second case, dot operations require communication operations (MPI_AllReduce) to combine local results in order to obtain the global value. It implies a communication/synchronism point after each dot operation in the BCG algorithm.

The SpMV operations have been implemented using the CRF that allows the reduction of the number and the size of the communications among processors. Therefore, to compute the SpMV, $q = Ap$, each processor adds redundant elements to the chunk of p (at the beginning and at the end). The number of additional elements is fixed ($2 \cdot D2$) and is hereinafter referred to as "halo". These "halos" will be exchanged among the processors for updating the value of p before the computation of every SpMV (see Figure 4.2). Thus, the use of redundant halos reduces the size of the messages and the number of communications among processes to compute SpMV.

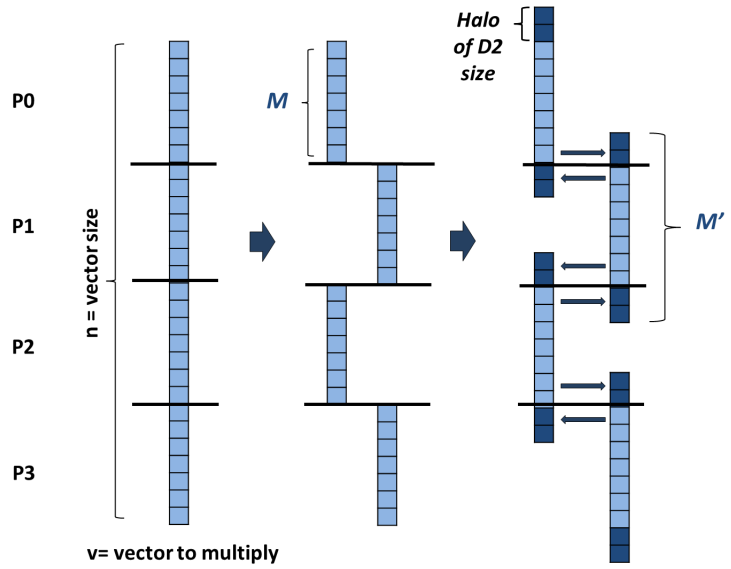


Figure 4.2: Halos swapping among 4 processors (P0, P1, P2 and P3) where every processor has a chunk of $M + 2 \cdot D2 = M'$ and M is the local vector v.

The idea of "halos" can be applied because the sparse matrix pattern of the Helmholtz equation is very regular [78]. However, it is worthy to notice that the "halo swap" step has a limitation and could be quite expensive as the number of MPI tasks are increased. It is fair to underline that this approach is advantageous only when the percentage of redundancy with respect to the total data of every process is small; i.e., when $P \ll n/D2$, where P is the number of MPI tasks, n the dimension of A and $D2$ half of the halo elements.

4.5.2 GPU computing

GPU and hybrid versions include the exploitation of one GPU device per processor. For this purpose, an initial mapping from each CPU memory to each GPU memory is performed. When each CPU has its own data chunk, these data structures are copied into the GPU associated to each CPU. In this way, all the operations are carried out in the GPUs, but when a communication process is required among cluster's processors, data chunks are copied to the CPU memory and the exchange among CPUs is executed.

Each GPU device is devoted to computing all the local vector operations (dot, saxpy) and local SpMVs which are involved in the BCG specifically suited for solving

the 3D Helmholtz equation.

Additionally, several optimization techniques have been considered for developing the local computation on GPU. The reading of the sparse matrix and data involved in vector operations are coalesced global memory access, maximizing the bandwidth of global memory. Moreover, share memory and registers are used to store any subsidiary data, despite their low reuse. These optimizations are very relevant for improving the performance of operations dominated by memory accesses (memory bounded) such as SpMVs, dots and saxpys, which are the keys in the resolution of the Helmholtz equation.

Previous evaluations of BCG implementations on GPUs have revealed that, although SpMV can be optimally accelerated on GPU, the performance of BCG can be penalized by the poor acceleration of vector operations [113]. BCG is expressed in [113] as a sequence of kernels related to the individual operations which define the iterations. An approach to better exploit the GPU by means of the vector operations consists of the fusion of several operations in one kernel. The fusion of several operations improves the exploitation of GPU resources, decreases the number of synchronization points and can introduce data reuse. According to our experience the fusion has strong impact on the BCG performance on one GPU. Also, the advantages of kernels fusions are analyzed in terms of performance and energy-aware for Conjugate Gradient in [25]. All these optimizations have been exploited at the level of the local GPU computations for the resolution of the Helmholtz equation.

In order to apply fusion of kernels, the following methodology has been applied: (1) identification of data dependencies; (2) reordering of operations; and (3) fusion of independent operations. Algorithm 6 shows the operations candidates for fusion in BCG-3DH (lines 5, 8, 9 and 11).

It is relevant to underline that the fusion of the two SpMVs can be executed at the same time, so avoiding the reading of A twice, since the Helmholtz matrix is symmetric $A = A^T$. Consequently, the operational intensity is improved by this fusion.

A specific kernel to compute the two SpMV operations ($q = Ap$ and $q' = Ap'$) on the GPU platforms has been developed. In this kernel, every thread computes one element of the output vectors (q and q'). Thus, the loop to compute every row has been unrolled. This way, the parallelism of this kernel is very high. In the kernel, the number of reads has been considerably reduced since the sparse matrix A is stored in

the memory of the GPU for both SpMVs. Moreover, this optimization is combined with the advantages, in terms of performance, introduced by the use of our CRF.

4.6 Evaluation

This section is devoted to analyzing the computational advantages of the resolution of the Helmholtz equation on a heterogeneous multi-GPU cluster. Algorithm 6 has been implemented to solve this problem in complex and double precision. Preconditioning is introduced in BCG-3DH in line 5 of Algorithm 6 (the computational load of these operations depends on the specific preconditioner M). In general, it is desirable that these systems are easily solved and have a computationally light load [142]. Therefore, preconditioning can be considered as an additional vector operation which is included in the BCG iterative procedure. Since there is not a general preconditioner, for the sake of simplicity all the experiments have been conducted without a preconditioner.

The characteristics of the set of test matrices used in the experiments are described in Table 4.1. Let us remark that n is the dimension of the sparse matrix (A); $D1$ and $D2$ are locations of the lower and upper diagonals ($\pm D1-th$ and $\pm D2-th$ diagonals); and nz is the number of non-zero elements in A. We have considered instances with $n \geq 160^3$ because we have experimentally verified that for these problem sizes the GPU implementation has relevant gains in terms of performance compared to that of CPU. For problem sizes smaller than 160^3 the use of GPU computing was not very relevant. Furthermore, matrices larger than 520^3 have not been studied due to the limitations related to the global memory resources of the GPU.

For the evaluation, a Bullx cluster composed of 4 nodes whose main characteristics are described in Chapter 1, Section 1.3 has been considered. In the same section, the characteristics of the GPU devices can be found (see Table 1.4). Notice that the peak GFlops in single precision is twice the peak GFlops in double precision. Table 1.4 also provides features of the GPU global memory, which is actually the key parameter limiting the size of the problem that can be solved. The experiments have been compiled with NVIDIA CUDA C and mpicc compilers with -O2 as the optimization option. The three implementations of BCG-3DH aforementioned have been evaluated: CPU version using only the CPU processors, GPU version using only the GPU devices of the cluster

Table 4.1: Characteristics of the test matrices (complex and double precision numbers).

Complex Matrix	n	$D1$	$D2$	nz
m_160^3	4.10E+06	160	25600	2.87E+07
m_200^3	8.00E+06	200	40000	5.60E+07
m_240^3	1.38E+07	240	57600	9.68E+07
m_280^3	2.20E+07	280	78400	1.54E+08
m_320^3	3.28E+07	320	102400	2.29E+08
m_360^3	4.67E+07	360	129600	3.27E+08
m_400^3	6.40E+07	400	160000	4.48E+08
m_440^3	8.52E+07	440	193600	5.96E+08
m_480^3	1.11E+08	480	230400	7.74E+08
m_520^3	1.40E+08	520	270400	9.84E+08

and a hybrid version which exploits both CPU and GPU processes located into the multi-GPU cluster.

In Table 4.2, a preliminary study of the sequential approach on a CPU single core was carried out. We considered 1000 iterations of the BCG method based on the Helmholtz equation to identify the hot spots of the code. The *gprof* tool was used and showed that most of the execution time ($\approx$ 98%) is spent on four routines: 2saxpies, 2Ax, 3saxpies (lines 8, 9 and 11 of Algorithm 6) and dots (all the dots of Algorithm 6). As a result, we have focused our attention on the evaluation of these particular routines. Table 4.2 provides the experimental results of evaluating the execution time of these specific operations as well as the total runtime of BCG-3DH. Column Total runtime represents the execution runtime (in seconds) of 1000 iterations of the BCG-3DH. Columns 2Ax, saxpies and dots represent the runtime (in seconds) for the calls to these routines. As can be observed, most of the runtime is consumed by the saxpy operations and SpMV. One issue that can be observed in such table is that BCG-3DH could not be executed on a single core for matrix with the largest size due to the limitations related to the global memory resources of one core.

In [137] has been shown the improvements of fusing saxpy operations, which represents an acceleration of the $\approx$ 9% of the total runtime with respect to the sequential code. Moreover, the SpMV operations have been also optimized using the CRF. The runtime for saxpies is larger than for 2Ax because the FLOPs for 2Ax ($38n$) are slightly

Table 4.2: Profiling for 1000 iterations of the sequential code of the BCG to solve the 3D Helmholtz equation. Columns 2Ax, saxpies and dots represent the runtime (in seconds) for the calls to these routines and total runtime identifies the total execution runtime (s).

	2Ax	saxpies	dots	Total runtime
m_160^3	75.6	89.8	46.5	214.3
m_200^3	129.8	152.9	87.8	378.0
m_240^3	259.4	296.2	155.4	719.4
m_280^3	356.7	428.6	239.9	1038.4
m_320^3	631.9	731.1	353.5	1750.8
m_360^3	766.4	911.7	511.6	2217.7
m_400^3	1278.5	1376.0	718.4	3411.1
m_440^3	1475.7	1665.0	932.5	4124.3
m_480^3	2236.0	2381.8	1245.2	5929.4
m_520^3	-	-	-	-

lower than for saxpies operations ($40n$). Therefore, the experimental profiling for sequential BCG-3DH code is in accordance with the theoretical computational cost of the individual operations, as provided in Algorithm 6.

4.6.1 CPU and GPU versions

Focusing our attention on the CPU approach, a study of the acceleration factors using several (P) CPU processors against the sequential code has been carried out. $P = 2, 4, 8$ processors were used for the evaluation. The experimental results of the speed up for 2Ax, saxpies and dots were nearly the ideal. Here it is important to highlight that in the CPU version, most of the time ($> 96\%$) is spent on processing time, and the communication times (MPI routines) are negligible.

For the GPU version, several issues have to be taken into account, so a deeper analysis has been carried out. Firstly, the range of the performance (GFlops) for the set of test matrices considering 1000 iterations of the BCG-3DH using the GPU approach (with 4 and 8 GPUs) is shown in Table 4.3. The range of GFlops for the CPU version (4 and 8 CPUs) have also been included. It can be seen in Table 4.3 that the best performance is reached by the saxpy operations thanks to the fusion of kernels. Furthermore, good results in terms of performance are obtained for the 2Ax routine

thanks to the CRF and to the fusion of the two SpMV operations, which together improve the operational arithmetic of this kind of operations.

Table 4.3: Range of GFlops for the set of test matrices considering 1000 iterations of the BCG-3DH with 4GPUs and 8GPUs. Additionally, values for sequential, 4CPUs and 8CPUs implementations are shown. Columns 2Ax, saxpies and dots show the range of GFlops achieved by theses operations.

	2Ax	saxpies	dots
sequential	1.8-2.2	3.4-3.9	0.7
4CPUs	7.1-8.6	13.1-15.3	2.6-2.7
8CPUs	14.1-17.1	27.6-29.9	4.7-5.5
4GPUs	67.3-69.3	93.6-98.4	22.5-31.7
8GPUs	132.2-134.0	187.7-196.5	26.9-56.4

Figure 4.3 shows the distribution of the runtime devoted to the main operations for 4 and 8 GPUs. As can be observed, the percentage of the communication time is more relevant for 8 GPUs than for 4, because the aforementioned condition ($P \ll n/D2$) is stronger for $P = 4$ than $P = 8$. Note that for the GPU approach, halos swapping requires four communications between the GPU/CPU at every iteration. It can be seen that for small problems, the bottleneck is due to the communications CPU-GPU for the halos exchange. However, as the size of the problem increases, the communication time decreases since for a specific value of P the condition $P \ll n/D2$ is stronger as n increases. Therefore, this consideration agrees to the experimental results which show that the percentage of communications decreases as the size dimension of the matrix increases.

Figure 4.4 shows the acceleration factors of every operation for 4 and 8 GPU implementations versus the sequential profiling of Table 4.2. It can be seen that all operations are considerably accelerated thanks to the use of GPU computing. The saxpy operations have the lowest acceleration factor, representing a relevant percentage of the total runtime (see Figure 4.3). This is because the majority of the operations in Algorithm 6 are saxpies and they are less accelerated by the multi-GPU architecture than the 2Ax or dots operations.

A study of the runtime of the CPU and GPU versions has also been carried out. Tables 4.4 and 4.5 show the experimental results where columns Runt provide the total

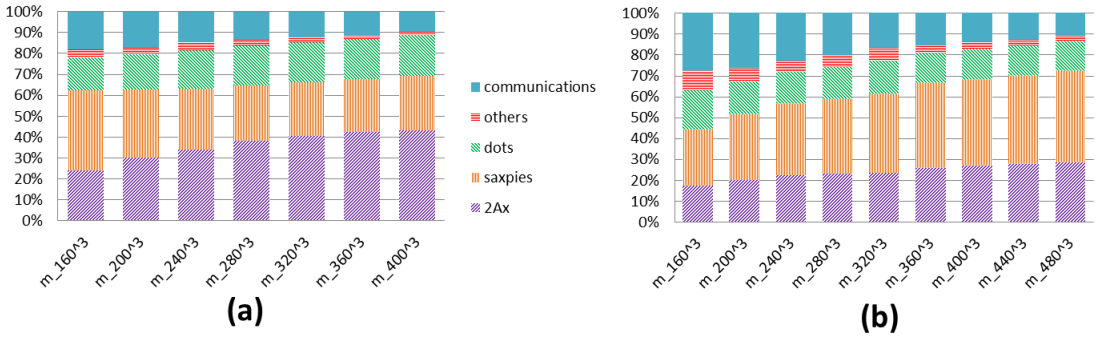


Figure 4.3: Percentage of the runtime for each call to function using 4GPUs (a) and 8GPUs (b).

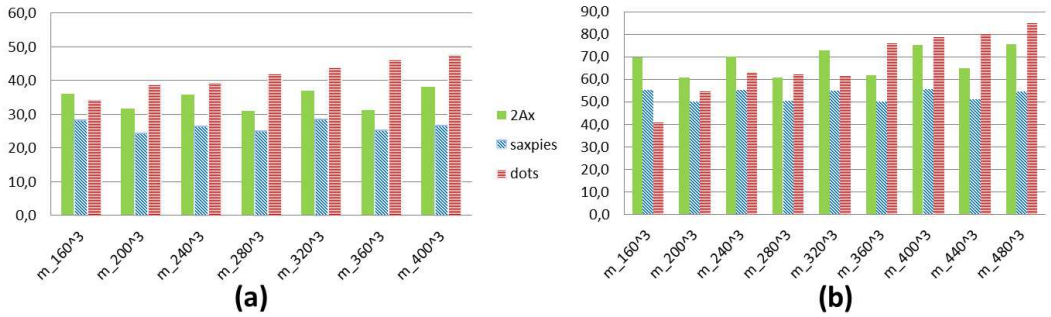


Figure 4.4: Acceleration factors of operations of 2Ax, saxpies and dots routines with 4GPUs (a) and 8GPUs (b) versus the sequential time of these routines.

runtime in seconds of the BCG-3DH execution and columns %C give the percentage of the communications penalties with respect to the total runtime. Note that CPU and GPU processes have been launched in different nodes since the number of MPI processes has been selected less or equal to the number of available cores on the cluster. To be more specific, the implementation using 2CPUs (2GPUs) has been launched using two nodes of the cluster and the implementations using 4CPUs (4GPUs), 8CPUs (8GPUs) and 16CPUs, using the four nodes of the cluster. Focusing our attention in Table 4.4, up to 16CPUs have been considered for the parallelization because using more CPU processors (e.g. 32CPUs) the size of the halos are relevant with respect to the local data and consequently the communication penalties deteriorate the performance.

Tables 4.4 and 4.5 clearly shown that the runtime (Runt) decreases as the number of CPU and GPU processes increases, because the BCG-3DH method is dominated by the computation. This fact is more evident for the CPU version with the low value

Table 4.4: Runtimes, in seconds, of 1000 iterations of the BCG-3DH using the CPU version with 1, 2, 4, 8 and 16 processors (1CPU, 2CPUs, 4CPUs, 8CPUs and 16CPUs) (Column Runt). Column %C identifies the percentage of the total runtime that consume the communication processes for every execution.

	1CPU	2CPUs		4CPUs		8CPUs		16CPUs	
	Runt	Runt	%C	Runt	%C	Runt	%C	Runt	%C
m_160^3	214.32	108.12	1.34	54.64	1.95	28.18	3.84	16.34	11.32
m_200^3	377.98	190.80	1.45	98.30	2.31	50.67	4.66	31.44	6.50
m_240^3	719.37	357.22	1.15	179.84	4.08	92.96	2.88	54.38	4.84
m_280^3	1038.38	520.67	1.10	264.55	0.83	140.53	6.22	85.24	4.77
m_320^3	1750.79	920.56	0.68	458.92	2.02	230.57	3.59	127.79	4.93
m_360^3	2217.68	1111.75	1.53	560.56	3.74	295.42	3.70	177.96	4.23
m_400^3	3411.10	1687.70	0.85	847.62	1.69	436.52	2.01	250.03	3.26
m_440^3	4124.25	2095.97	2.56	1035.28	0.43	538.26	2.74	324.65	3.84
m_480^3	5929.36	2963.20	0.49	1489.04	1.20	752.24	1.93	429.89	4.16
m_520^3	-	3410.84	0.61	1743.64	0.81	878.22	2.07	538.24	2.89

of %C column with values which range from 0.43 to 11.32. In the GPU version, the communication penalties are larger (range from 4.48 to 33.94) and correspond to the additional communication between the CPU and the GPU and to the runtime reduction of the GPU. The value of %C is directly related to the size of the problem to solve, therefore, the larger is the problem, the less impact of the communication time respect to the total runtime. One issue that can be seen in Table 4.5 is that BCG-3DH could not be run for the matrices with the largest size due to the limitations related to the global memory resources of the GPUs, even for 8GPUs.

The best results in terms of performance are always obtained by the GPU approach thanks to massive parallelism that GPU devices offer. So, the execution of BCG-3DH algorithm with 1000 iterations using 2, 4 and 8 GPUs are faster than the execution using 2, 4, 8 and 16 CPU processors, with acceleration factors which range from 5.6 to 8.6 when we compare the same number of CPU and GPU processes (2CPUs-2GPUs, 4CPUs-4GPUs and 8CPUs-8GPUs). Moreover, the communications penalties of the GPU version are higher than the CPU version due to the additional communication GPU-CPU and CPU-GPU in the "halos" swap.

Table 4.6 shows the acceleration factors (AF) achieved by the CPU and the GPU

Table 4.5: Runtimes, in seconds, of 1000 iterations of the BCG-3DH using the GPU version with 2, 4 and 8 GPU devices (2GPUs, 4GPUs and 8GPUs) (Column Runt). Column %C identifies the percentage of the total runtime consumed by the communication processes for every execution.

	2GPUs		4GPUs		8GPUs	
	Runt	%C	Runt	%C	Runt	%C
m_160^3	13.89	9.50	7.90	19.40	5.03	33.94
m_200^3	26.60	7.55	14.80	15.84	9.12	30.22
m_240^3	45.73	6.45	24.85	12.28	14.60	25.38
m_280^3	72.49	5.28	38.43	10.44	22.18	22.68
m_320^3	108.15	4.48	55.57	9.32	31.90	19.23
m_360^3	-	-	78.95	8.38	44.22	16.90
m_400^3	-	-	107.42	6.99	58.04	14.85
m_440^3	-	-	-	-	76.73	13.79
m_480^3	-	-	-	-	97.86	11.66
m_520^3	-	-	-	-	-	-

approaches considering $P = 8$ (columns AF CPUs and AF totalGPUs, respectively), with respect to the sequential version executed on one core of Intel Xeon E5 2650. It is relevant to underline that the CPU version achieves speed-ups which range from 7.5 to 8, therefore the CPU version exhibits good scalability. This is because more than the 98% of the total runtime of the execution is devoted to computation and communications are negligible. Focusing now our attention on column AF GPUs which represents the gain of using the GPU (AF GPUs = AF totalGPUs/ AF CPUs), it can be observed that GPUs are able to obtain accelerations in the range of 5.6 to 7.7 depending on the size of the problem. This is due to the ratio computation/communication increases as the size of the problem increases. As previously mentioned, the communication process depends on the size $(2 \cdot D2)$ of the halos size (n).

The largest speed-ups have been obtained by the GPU version (AF totalGPUs column), with values that range between 41.5 and 60.6. Notice that when the dimension of the problem to solve increases, the importance of the halos decreases since they represent a negligible percentage of the total dimension of the problem. Moreover, the AF totalGPUs increases as the problem size increases, which highlights the interest of using multi-GPU computing to considerably accelerate the BCG-3DH algoritm.

Table 4.6: Acceleration factor (AF), where AF CPUs represents the AF of the implementation with 8 CPUs versus the sequential code (1 CPU core), AF GPUs identifies the AF of the use of GPU computing versus the implementation using 8 CPUs, and AF totalGPUs represents the AF of the GPU approach over the sequential code (1 CPU core).

	AF CPUs	AF GPUs	AF totalGPUs
m_160^3	7.6	5.6	42.6
m_200^3	7.5	5.6	41.5
m_240^3	7.7	6.4	49.4
m_280^3	7.4	6.3	46.8
m_320^3	8.0	7.2	54.6
m_360^3	7.5	6.7	50.2
m_400^3	7.8	7.5	58.8
m_440^3	7.7	7.0	53.7
m_480^3	7.9	7.7	60.6
m_520^3	-	-	-

Results demonstrate the relevance of the use of multi-GPU computing to considerably accelerate the resolution of the 3D Helmholtz equation. Therefore, it is clear that thanks to the use of the CRF, which better exploits the architecture of the GPU, it is possible to efficiently solve larger problems. It implies that our GPU version efficiently exploits the distributed architecture resources such as one multi-GPU cluster and allows the Helmholtz equation to solve larger problems.

4.6.2 Hybrid implementation: Fast-Helmholtz

In the hybrid version (Fast-Helmholtz) we have considered the executions with 4 and 8 GPU processes as these resources reached the best results in terms of performance for the size of the test problems. Table 4.7 shows the runtime of 1000 iterations of the BCG-3DH method using two hybrid configurations: 4GPUs+8CPUs and 8GPUs+8CPUs. The following notation is used, Runt identifies the total runtime in seconds and $GPU(s)$ and $CPU(s)$ represent the runtime in seconds of the GPU and CPU processes, respectively. Column F identifies the ratio between the workload assigned the CPU and the GPU. The workload assigned to a GPU process is F times higher compared to the workload of the CPU process. Different benchmarking processes have been carried out for this cluster and this algorithm to determine the computational burden associated to

the hybrid version which provides the best workload balance. The value of F has been estimated by a preliminary benchmarking (not automatized) using values which range from 8 to 12 (these values are related to the acceleration factor of running BCG-3DH algorithm on one GPU with respect to one CPU). Column %I represents the acceleration factor of the hybrid implementation versus the GPU version with 4 GPUs (for 4GPUs+8CPUs implementation) and 8 GPUs (for 8GPUs+8CPUs implementation). As aforementioned, the workload of the GPU version has to be larger than the workload of the CPU version but taking into account the memory limitation of the GPUs. Therefore, for problems of dimension 440^3, 480^3 and 520^3 with 4GPUs+8CPUs, the memory requirements of BCG-3DH are larger than the memory resources of 4 GPUs and the values of F are less than 8.

Table 4.7: Profiling of the resolution of the 3D Helmholtz Equation based on 1000 iterations of the BCG method using the hybrid version and two different configurations: 4GPUs+8CPUs and 8GPUs+8CPUs. Column Runt identifies the total runtime in seconds of the execution, GPU(s) and CPU(s) show the runtime of the GPU and the CPU, respectively, F column denotes the factors to balance the workload in the hybrid approach and, finally, %I represents the acceleration factor of the hybrid implementation versus the GPU version with 4 GPUs (for 4GPUs+8CPUs implementation) and 8 GPUs (for 8GPUs+8CPUs implementation).

	4GPUs+8CPUs					8GPUs+8CPUs				
	Runt	GPU(s)	CPU(s)	F	%I	Runt	GPU(s)	CPU(s)	F	%I
m_160^3	7.24	5.29	3.31	11	8.45	4.81	3.03	2.48	10	4.24
m_200^3	13.66	10.74	7.49	12	7.73	8.62	5.80	4.22	11	5.45
m_240^3	21.63	17.47	16.17	10	12.96	14.4	9.66	10.09	11	1.40
m_280^3	33.44	27.36	26.75	8	12.98	21.42	15.00	13.67	10	3.43
m_320^3	49.13	41.74	37.95	12	11.58	30.52	22.18	18.58	11	4.35
m_360^3	69.73	60.10	50.61	10	11.67	41.73	29.80	31.36	10	5.61
m_400^3	93.75	83.41	68.05	11	12.73	55.80	43.70	42.81	9	3.87
m_440^3	131.99	101.85	123.08	7	-	73.35	56.62	56.58	11	4.41
m_480^3	329.82	102.76	306.90	3	-	93.38	75.65	73.67	9	4.58
m_520^3	473.82	15.45	459.82	2	-	123.49	97.83	98.14	9	-

As can be observed in Table 4.7, the runtimes of the CPU and GPU processes are well balanced. Note that in all experiments, 4 GPU or 8 GPU cards are exploited and the runtime decreases as the number of GPU processes increases. It is due to the fact

that CPU processes collaborate with the GPU processes to accelerate the computation. Note that the workload balance is not optimal for the matrices m_440^3, m_480^3 and m_520^3 due to the memory limitations of the GPU. So, although the workload of the CPU processes is high, the CPU memory resources allow to extend the size of problem with light penalties of performance. For the problem's sizes considered (from $n = 160^3$ to $n = 520^3$), the approach with 4GPUs+8CPUs reaches the best results in terms of performance compared to the 4GPUs version (acceleration factors range from 8.45 to 12.73). It results evident that this kind of platforms can only be efficiently exploited for very large problems, even larger than the instances considered in this chapter. It is also clear that the hybrid platform is appropriate to overcome the GPU memory limitations.

4.7 Conclusions

In this chapter we have studied the performance of BCG-3DH algorithm using heterogeneous multi-GPU clusters. This method is based on two main ideas: the collaboration of CPU and GPU platforms for accelerating the operations involved in the algorithm and the use of MPI for distributing the workload among the available processors. Moreover, we have used a specific format (CRF) that makes it possible to considerably reduce the memory requirements for the storage of the large sparse diagonal matrix involved in the solution of linear system of equations, like the BCG-3D method. This also implies an important reduction in the runtime of the method.

Experimental results have shown that Fast-Helmholtz outperforms both, the CPU and the GPU versions when several CPU cores collaborate with the GPU devices. However, we only obtain large acceleration factors using more GPUs when the size of the problem is large enough.

Optimizations of the code at different levels have been carried out: (1) at the sequential programming level, the use of the CRF has improved the operational intensity of the SpMV operations; (2) at MPI level, a distribution scheme which minimizes communications has been applied (bearing in mind the regular pattern of the Helmholtz matrix); (3) at the GPU level, the use of fusion optimizations for the exploitation of the memory and the improvement of the vector and sparse matrix operations.

Experimental results show the importance of using GPU-based clusters to design parallel software on the modern heterogeneous architectures. As a result, the resolution of the 3D Helmholtz equation for large real world applications has been made feasible. According to the above results, several aspects of the Fast-Helmholtz can be still improved, so our future work will be focused on them specifically. The difficulty of having the workload well-balanced is related to the appropriate distribution of the workload between the CPU and GPU processes, so our current on-going work involves to automatize the benchmarking process to determine the most suitable factor F to exploit the specific architecture considered. Furthermore, another future work will be focused on the development of a hybrid version MPI-OpenMP on the hybrid multi-GPU cluster. Apart from that, in order to increase the flexibility of the implementation, the use of rCUDA (a middleware which allows to make use of remote GPUs as if they were local ones) [3] will be part of a future work. This virtualization framework allows the execution of GPU-accelerated applications within virtual machines (VMs), enabling the sharing of a pool of centralized GPU resources that reside externally to the compute nodes. Finally, another line of future work is the development of implementations on multi-GPU clusters of BCGSTAB and COCG as alternative methods for solving the resulting systems from the Helmholtz equation.

5

Optical Diffraction Tomography as a case of study

This chapter is focused on a real physical problem of Optical Diffractional Tomography (ODT) based on holographic information. ODT is a non-damaging technique which allows the extraction of the shapes of objects with high accuracy. Therefore, this technique is very suitable to the in vivo study of real specimens, microorganisms, etc., and it also makes the investigation of their dynamics possible. Tomography has recently been incorporated to the field of fluid velocimetry to provide three dimensional information of the location of particles. Previous works have proven the potential of Optical Diffraction Tomography for biological and microfluidic devices.

Several approaches to tomographic reconstruction that are essentially based on linear and non-linear combinations of holographic reconstructions of the scattered fields observed under varied illuminating conditions have been considered to solve this problem. Linear ODT has shown to provide images of highest fidelity, however these methods cannot properly accounts for the effects of multiple scattering. A non-linear ODT physical model (NLODT-P) based on a two dimensional reconstruction of the seeding particle distribution in fluids was proposed by J. Lobera and J.M. Coupland. However, to address its extension to a three dimensional model makes compulsory the use of HPC techniques due to its high computational cost (both memory requirements and runtime).

This model requires the solution of the Helmholtz equation, whose discretization results in a complex, regular, large and sparse linear system of equations. BCG is the proposed solver to obtain the solution of the Helmholtz Equation. As a result, the development of the model is based on the implementation of BCG on GPUs exploiting

the regularities of the sparse matrix as described in Chapter 4. In this chapter, the implementation and validation of this physical model for the case of three dimensional reconstructions is carried out.

This chapter is organized as follows. In Section 5.1 the Optical Diffraction Tomography (ODT) problem is described. Section 5.2 studies and analyzes the proposed Non Linear ODT model for locating particles (NLODT-P). Section 5.3 validates the model. Section 5.4 is devoted to discussing our GPU implementation. Section 5.5 provides experimental evaluations of our model using a GPU device. Finally, Section 5.6 summarizes the main conclusions and future works.

5.1 Introduction

Holographic Particle Image Velocimetry (HPIV) provides simultaneous three components, three-dimensional (3C-3D) measurements of a seeded fluid flow [26, 49]. Classical analysis of HPIV recordings assumes that the particle illuminating and the scattered beam do not suffer multiple scattering. In practice, however, multiple scattering effects increase background noise, decreasing the number of velocity vectors that can be retrieved from a given flow field [88]. Tomographic methods using several recordings from different observation directions have been proposed in the last years to mitigate this problem [60, 131, 133]. In Tomographic Particle Image Velocimetry the particles within the entire volume need to be imaged in focus, which is obtained by setting proper numerical aperture (NA). The application of Optical Diffraction Tomography (ODT) in HPIV [100], and more specifically the non-linear ODT, would improve the spatial resolution. ODT is a non-damaging radiation technique that provides a 3D map of the object refractive index from holographic recordings of scattered fields with different illumination or observation directions. If the linear approximation is assumed, the spectral components of the field scattered by the object are directly related to the spectral components of the object refractive index field [36, 94, 158]. Some impressive advances have been done recently in coherent microscopy [84, 102, 161], even though these measurements are incomplete and the assumption of weak scattering is severely restrictive.

For the non-linear ODT approach, image reconstruction is considered the solution that better explains the scattered far field but it requires large and powerful compu-

tational resources. In the implementation of the optimization method, the Helmholtz equation (see Chapter 4, Section 4.1) needs to be solved for a known refractive index distribution and illuminating field - the forward problem. Appropriate sampling is roughly a tenth of the wavelength, and due to the size of volume of interest in biological flows the computational requirements are very demanding. Thus, the performance of the non-linear Optical Diffraction Tomography in HPIV is determined by the selected computing strategy, and this is particularly true for its implementation for 3D problems. The use of a priori information concerning the object can reduce instability in optimization and computation time [28, 47, 99, 100]. In fluid velocimetry we usually have additional information about the object, such as the diameter and the optical properties of the seeding particles, effectively reducing the particle imaging reconstruction problem to a particle location problem [134].

In this context, High Performance Computing (HPC) is required to implement and validate the aforementioned model. HPC allows the scientific community to extend their models and accelerate their simulations by the exploitation of a wide variety of computing resources [45]. However, the selection of HPC architectures and programming interfaces for the models to be developed require an important effort. Earlier works have shown the parallel computation capability of GPUs in performing ODT models [84, 114].

Thanks to technological and architectural advances [75], current standalone computers present tremendous power and they can be considered as desktop supercomputers if their heterogeneous resources (such as GPUs) are appropriately exploited. As aforementioned in Section 1.1.2.4, MATLAB is a popular framework used by the scientific and engineering community to develop software applications and to test models [6]. In this chapter, we discuss an implementation of our Non-Linear ODT model at a source-level MATLAB compiler calling MEX-files for using GPU routines, which is also experimentally evaluated. The main objective of this chapter is an attempt of the development and validation of the NLODT-P model using the issues studied in previous chapters of this thesis. More concretely, the BCG-3DH method with one GPU has been used to parallelize the most computationally demanding procedure of the model.

In this chapter some numerical experiments have been carried out showing very promising results for 3D volumes of $(10\mu m)^3$. This chapter shows the feasibility of this approach and the outstanding imaging capability of non-linear ODT compared to

linear tomographic approaches. In particular, we make the following contributions: (1) development and validation of a non-linear ODT 3D model for the location of particles in a multiple scattering environment. This model has been capable of locating particles from a set of small numerical experiments where the linear ODT could not find them; (2) an efficient exploitation of the parallel computing power of the GPU devices for performing optical diffraction tomography image reconstruction; and (3) illustrating how scientists can extend their models, based on MATLAB, to more complex applications by the use of HPC techniques. HPC resources can be used from MATLAB through the MEX-files, which provide an interface between MATLAB and HPC platforms.

5.2 Description of NLODT-P model

In fluid velocimetry the studied flow is seeded by small particles with a refractive index different from the background. Typically, a coherent source is used to illuminate the flow, and this illuminating beam is scattered by the seeding particles, which allows to determine their location at one instant of time. To determine the velocity field of the flow, two consecutive recordings are required. In particular, we will focus our attention on the combination of several simultaneous recordings to recover the position of each particle on a certain volume of interest at one time.

The proposed method consists in a minimization of the cost function defined by the square root difference between the measured and the computed scattering field by the seeding particles. This is a non-linear optimization problem that would be addressed by a modified Conjugated Gradient Optimization Method (CGM) [100]. The search direction at the first iteration is the negative gradient of the cost function. Although the gradient can be expressed by an analytical equation, it still requires the resolution of the forward problem twice. This forward problem consists of computing the scattered field $E_s(r)$ by a known object and an illuminating beam $E_r(r)$. According to scalar diffraction theory the (complex) amplitude of a monochromatic electric field, $E(r) = E_s(r) + E_r(r)$, propagating in a medium of (complex) refractive index, $n(r)$, obeys the Helmholtz equation [129] which can be rewritten in the following inhomogeneous form using the appropriate transformation described in [100]:

$$(\nabla^2 + k_0^2 n^2(r))E_s(r) = f(r)E_r(r) \tag{5.1}$$

where $f(r)$ is the scattering potential, defined by:

$$f(r) = -k_0^2(n^2(r) - 1) \tag{5.2}$$

Our objective is to find the particle field position that minimizes the square difference between the measured $E_m(r)$ and the computed field that should be measured, $cost(f)$, according to the available estimation of the refractive index, $E_c(f, r)$:

$$cost(f) = \sum_i |E_m^i(r) - E_c^i(f, r)|^2 \tag{5.3}$$

where for any hologram there is a separate contribution (i) to the cost function.

CGM has been chosen to minimize this cost function. Thus, given a scattering potential, the gradient of the cost function can be taken as an image of the difference between the real $n(r)$ and its available estimation. This image is typically a smooth distribution with several local maxima, even when some sharp refractive index changes are expected, as for our case of particles on a flow. Furthermore, in general the object in fluid velocimetry applications is a sparse distribution of particles of known refractive index and shape. Subsequently a relatively small matrix PL that stores the location of the particles could describe the 3D refractive index field.

Bearing in mind the previous considerations, the developed model referred as NLODT-P is detailed below. A procedure presenting the most important steps involved in the NLODT-P model are shown in Algorithm 7 and the notation used in this chapter is presented in Table 5.1.

This model is composed by a first procedure (step), where the location of the first particle is determined (see Step 1 of NLODT-P as shown in Algorithm 7), and an iterative process, where the remaining particles are located. The initial step does contain computations similar to the subsequent iterations, but no special computing resources are required, therefore it is considered separately. Next, the main details for every NLODT-P step are described.

Step 1. Location of the 1st particle

In [99] it has been shown that the gradient of the cost function can be expressed as the sum of the simulated Bragg holograms between the illuminating field and the back propagated measured field. In particular, for the initial iteration, the illuminating

Algorithm 7 Algorithm of the NLODT-P model.

1: #Step 1. Location of the 1st particle (line 2)
2: Compute $PL(1)$
3: $It = 2$
4: **while** $It < iterMax$ **and** $Value < threshold$ **do**
5: **for** $i = 1, 2, ...$ *until* $i < N_h$ **do**
6: #Step 2. Update the refractive index field (line 7)
7: Update $n(r)$
8: #Step 3. Compute the updated gradient, $g(r)$ (lines 9 - 14)
9: $E_s^i(r) = Forward(E_r^i(r), n(r))$
10: $E_{r,n}^i(r) = E_s^i(r) + E_r^i(r)$
11: $E_c^i(r) = Filter(E_s^i(r), k_0^i, NA)$
12: $E_{m,n}^i(r) = Forward((E_c^i(r) - E_m^i(r))^*, n(r))$
13: $E_{m,n}^i(r) = E_{m,n}^i(r) + (E_c^i(r) - E_m^i(r))^*$
14: $g(r)^* = g(r)^* + (E_{m,n}^i(r), E_{r,n}^i(r))$
15: **end for**
16: #Step 4. Locate next particle (lines 17 and 18)
17: $g_{MF}(r) = Matched\_Filtering(g(r), sample)$
18: $[PL(It), Value] = max(abs(g_{MF}))$
19: $It = It + 1$
20: **end while**

field $E_r^i(r)$ and the back propagation of the measured field $E_m^i(r)$ are considered undisturbed. Thus the gradient is essentially the First Born Approximation of the scattering potential:

$$f(r)^* \approx \sum_i E_m^i(r)^* E_c^i(r) \tag{5.4}$$

where * represents the conjugate value. So, the particle location could be obtained from the location of the absolute value of the maximum of $f(r)^*$. Although a better performance of the model can be obtained if a matched-filtering is previously applied. This matched-filter can be obtained considering the (linear) ODT image from an isolated particle.

Step 2. Update the refractive index field, $n(r)$

From the a priori knowledge of the object in fluid velocimetry, we assume that the refractive index field can only take: (1) the refractive index of the seeding particles within a spherical region around any located particle; and (2) the fluid refractive index in the remaining voxels of the volume of interest. So, if a new particle is located the

Table 5.1: Notation used in this chapter

Abbreviation	Description
$E_r^i(r)$	Illumination field of $i-th$ hologram
$E_s^i(r)$	Scattering Field of $i-th$ hologram
$n(r)$	Estimated object index refraction (from the previously computed particle position)
$E_m^i(r)$	Measured field of $i-th$ hologram
$E_{m,n}^i(r)$	Updated measured field of $i-th$ hologram, given a $n(r)$
$g(r)$	Gradient of the cost function
$f(r)$	Scattering potential
$sample$	Image of an isolated particle
Vol	Number of voxels used for discretizing the volume to reconstruct
N_h	Number of holograms
$iterMax$	Maximum number of iterations of the model
PL	Particles Locations
$Value$	Maximum value of $g(r)$ modulus

refractive index around its position is updated.

Step 3. Compute the updated gradient, g(r)

The gradient $g(r)$ of the cost function provides the distribution that should be added to the estimated scattering potential $f(r)$ to minimize the cost function according to the classical Conjugated Gradient Optimization Method. However, our model does not need to solve the full image problem neither to update the scattering potential. Once an estimation of the scattering potential is available, a similar expression to Equation 5.4 can be obtained for the new gradient $g(r)$. However, the meanings of both interfering fields and its computational strategy have changed:

- Update illuminating field: $E_{r,n}^i(r)$. We need to take into account the presence of the particles already located. The forward solver computes the scattered field $E_s^i(r)$ due to the object described by $n(r)$ and the original illuminating field $E_r^i(r)$. The updated illuminating beam, $E_{r,n}^i(r)$ will be the sum of both fields $E_r^i(r) + E_s^i(r)$.

- Update measured field: $E_{m,n}^i(r)$. The back propagation of the measured field is computed in two stages: Firstly, the expected or computed measured field: $E_c^i(r)$

can be obtained by filtering the $E_s^i(r)$ to take into account the numerical aperture and the far field situation of the recording devices. Secondly, the difference between the measured field and the computed measured field $(E_m^i(r) - E_c^i(r))$ is back-propagated taking into account the diffraction introduced by the estimated refractive index $n(r)$. Let us remark that the role of the illuminating beam in this case will be the conjugated remaining field: $(E_m^i(r) - E_c^i(r))^*$.

As in the initial iteration, a separate contribution to the gradient $g(r)$ should be obtained for each hologram.

Step 4. Locate next particle and exit

As for the initial iteration, the most probable position of next particle is the absolute maximum of the matched-filtered gradient $(g_{MF}(r))$. The position is stored in the output variable PL, and the peak value $(Value)$, can be used to decide the end of the iterations. Experience shows that value will decrease in each iteration and a convenient value of *threshold* ensures the process stop. However, the criterion to select an appropriate value of the threshold is not addressed in this work. Thus, a maximum number of iteration is imposed.

In each iteration of the optimization problem, the Helmholtz equation (Forward solver in Algorithm 7) has to be solved. The Helmholtz equation is an example of a linear elliptic Partial Differential Equation (PDE), which has been extensively studied [129]. It can be numerically solved by means of an appropriate transformation based on Green's functions and a spatial discretization [73, 129], for example, Finite Element Method (FEM) [77, 79]. FEM discretizes the region of interest in small elements, assuming the function $E(r)$ can be approximated to a constant value in each of these elements. A regular mesh of elements is usually considered when the object shape is the unknown (inverse problems). So, the spatial derivatives of the Laplace operator can be discretized with a seven-point stencil in 3D.

Thus, when the discretization process is based on FEM and a spatial regular 3D mesh, the linear system of equations resulting from Equation 5.1 is described by a matrix with only seven non-zero diagonals. Therefore, FEM transforms the 3D Helmholtz equation into the linear system $(Ax = b)$, where the independent term, b, depends on the illumination field (E_r), the unknown vector, x, identifies the scattered field and the matrix A, related to the refractive index $(n(r))$, is sparse and exhibits a strong

regularity in both the pattern and the values of its non-zero elements and has a very large size which depends on the number of spatial discretization points or voxels into the volume (Vol) [116]. It is well known that the "pollution effect" limits the reliability of the FEM solution of the Helmholtz equation for high wave-numbers [30]. Thus, to avoid this instability effect, a high spatial discretization has to be used on the volume of interest. All these aspects mean that the Forward solver actually consists on the resolution of a large size linear system of equation composed by complex number (e.g. BCG-3DH method as described in Chapter 4).

5.3 Experimental validation of the model

In order to validate our model we have chosen an apparently simple particle distribution consisting on four $2\mu m$ particles recorded with three inline holograms. The illumination (and observation) direction for each hologram is chosen along x, y and z axis, respectively. We consider a typical coherent illumination provided by a He-Ne laser, with $\lambda = 0.633\mu m$ and that the holograms are recorded at far field with a $NA = 0.55$ microscope objective.

The volume of interest has been divided in $160 \times 160 \times 160$ voxels of one tenth of the wavelength. Multiple scattering presences will be due to two main reasons: (1) the particle cannot be considered as point source and (2) the scattering field diffracted by one particle will modify the illuminating beam that reaches the others. The first contribution to multiple scattering is apparent when an isolated particle recorded by the same optical system is considered. The linear ODT image, that is, the modulus of $f(r)$ obtained by Equation 5.4 has been represented in Figure 5.1. Left image shows the shape of the brightest region (larger than 0.6 the maximum value), meanwhile right image shows the central plane.

This particle image can be used as sample to compute a matched-filter and unravel the gradient for any other $2\mu m-$particle field.

One of the most difficult particle distributions to recover is shown in Figure 5.2, as for any of the holograms there is always one particle obscured by the others. The minimum distance between particles is $3\mu m$.

The particle distribution of solving a linear ODT image problem is shown in Figure 5.3. The scattering potential obtained from Equation 5.4, which coincides with the

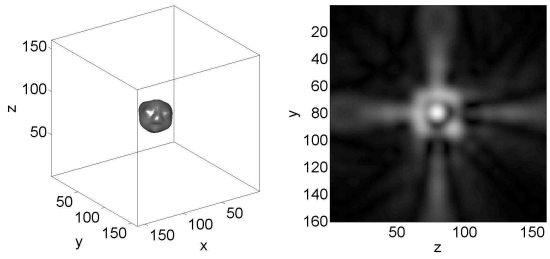


Figure 5.1: Matched-Filtering sample image of an isolated particle: 3D view of the surface draw by the voxels above 0.6 of the maximum value and 2D view at the central plane.

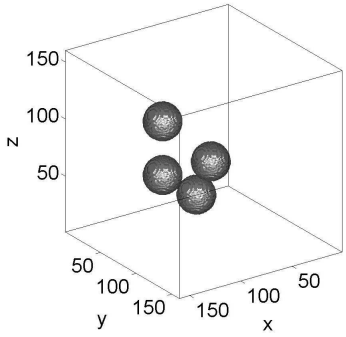


Figure 5.2: 3D view of the particle distribution problem.

first gradient, is shown on the left, and the filtered gradient with the matched filter obtained from Figure 5.2 on the right.

Top row pictures in Figure 5.3 show the shape of the voxels with higher values (above 0.6 the maximum value). To better illustrate the blurred image, the modulus of the gradient at the planes centered on the particle positions are shown below: plane $z = 57$ pixels (middle row) and plane 104 pixels (bottom row).

Although the expected resolution of the system should be $\lambda/(2NA) = 9$ pixels. The position of the particles cannot be recovered from the linear ODT image (left row). However, taking into account that particles cannot be considered point sourced and computing the matched-filtered gradient, we can clearly identify the position of four peaks. The error of the particle positions are below 2.5 pixels, that means $0.16\mu m$. As expected, NLODT-P can also solve this particle distribution problem with similar error results. For this case, the use of a matched-filter was enough to unravel the linear ODT image.

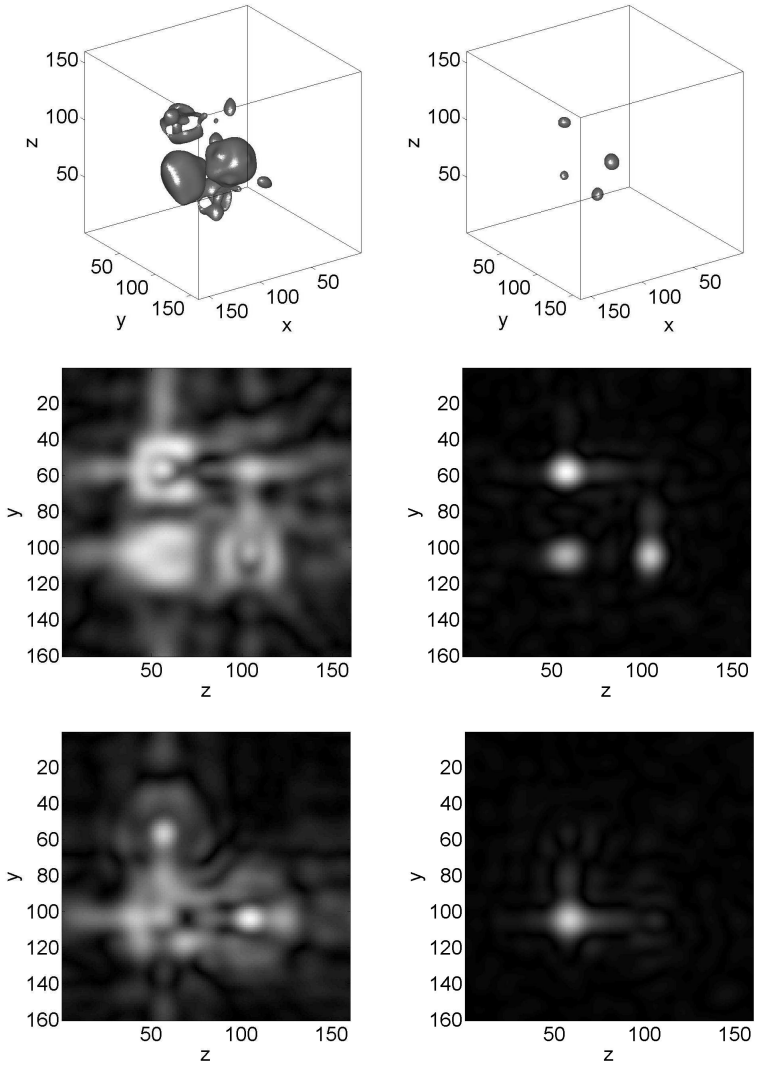


Figure 5.3: Gradient (left) and filtered gradient (right) computed at the initial iteration: 3D view of the surface at 0.6 the maximum value (top) and 2D views at $z = 57$ pixels (middle) and at $z = 104$ pixels (bottom).

However, for real fluid velocimetry application, the number of particles will increase to the order of one thousand particles. In that case, the effect of the multiple scattering between particles will be more significant. In addition, a limited optical access can make compulsory the iterative optimization. To illustrate the problem let consider the case when the minimum distance between particles is $2\mu m$ (close enough), so multiple scattering is mainly due to its proximity (see Figure 5.4). We considered two configura-

tions: (1) a full-optical access with three in-line hologram as in the previous experiment and (2) a slightly more realistic set-up, in which we illuminate as before, but we only have one camera. The observation direction has been chosen pointing along the direction $\widetilde{k}_{obs} = 1, 1, 1$. That means the camera will look to the particle distribution of Figure 5.4 roughly as the reader.

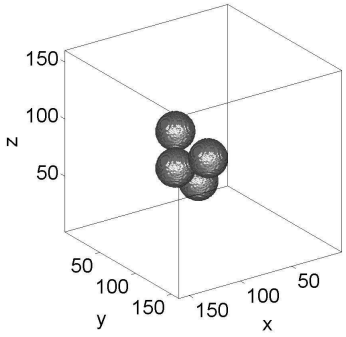


Figure 5.4: 3D view of the particle distribution problem.

As expected, the linear ODT image does not solve the four particles. The matched-filtered scattering potential is shown in Figure 5.5. For the full optical access configuration (Figure 5.5 left) does resolve the four particles. The particle image at the bottom corner is significantly smaller than the others. That particle image cannot be recovered from the one-camera configuration (Figure 5.5 right). Only the other three particles can be envisaged, even for any other selection of the iso-valued surface.

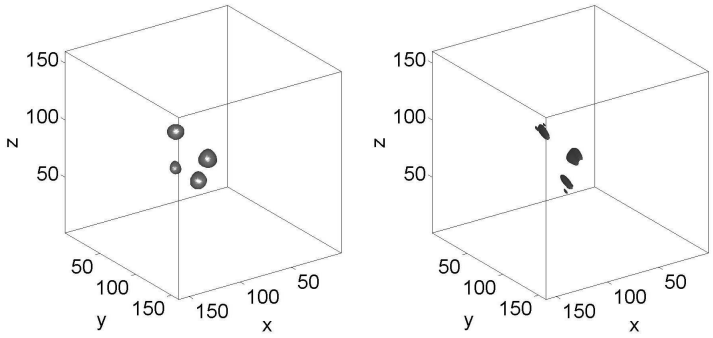


Figure 5.5: Linear ODT image after computing the corresponding Match-filter of the particle distribution of Figure 5.4 for a three-in-line hologram configuration (left) and three illumination and one observation direction configuration (right).

Meanwhile NLODT-P finds the particle position for both configurations, with a position error roughly of 4.6 pixels ($0.3\mu m$). Further work using more realistic problems with larger number of particles, is needed. It has also to be considered the non-linear ODT performance when there is a particle size distribution and some background noise is present. From these small numerical experiments, it can be derived that the performance of NLODT-P is clearly advantageous compared to the linear approach. In the next section it will be shown that the combination of MATLAB with GPUs for solving the Forward problem makes feasible the study of such kind problems and, subsequently, the NLODT-P for fluid velocimetry applications.

5.4 GPU-Based implementation of the model

The computational cost of the advanced numerical model described in this chapter is very high. So, it is essential to apply High Performance Computing techniques to its implementation. We have chosen GPU computing as the tool for accelerating this model because GPUs offer desktop massive parallelization that can accelerate this kind of computations. CUDA is a parallel computing platform and programming model that enables to increase the performance by harnessing the power of GPUs [8]. Scientists usually describe their models by means of a higher-level abstraction programming such as MATLAB. So, it is necessary to assemble both languages (MATLAB and CUDA) to have the ease of use of MATLAB and the acceleration of GPU computing. Concretely, a model can be implemented in MATLAB, and GPU computing can be used in the procedure/s with the highest computational cost. To be more precise, in our model 95% of the total runtime is devoted to the resolution of the large linear system of equations involved in the Forward solver. The resolution of this linear system is closely related to the numerical method used. However, even using one of the most efficient solver for these types of systems, the resolution of this large linear system of equations takes a long time and has large memory requirements because of the large dimension of A ($Ax = b$).

The rest of this section is devoted to the description of the computational resources, the numerical methods and the computing optimizations used to implement the NLODT-P model.

5.4.1 Software resources

Our implementation of the model shown in Algorithm 7 is based on the exploitation of both, the MATLAB framework and GPU computing by means of MEX-files routines (see Section 1.1.2.4). As aforementioned, our starting point has been a MATLAB implementation of NLODT-P, for which the procedure with the highest computational cost has been accelerated using a GPU device. Our NLODT-P model relies on numerical calculations, specifically linear system solvers and vector operations. An approach to facilitate GPU programming is based on the use of basic routines or libraries for computing the most used operations in scientific and practical applications. Table 5.2 shows several packages that provide options for carrying out the calculations on GPUs in combination with the MATLAB environment. In this context, it is necessary to develop a specific library to solve large, complex and sparse systems of equations by the integration of MEX-files and CUDA interface. This way, MEX-files routines [11] combined with MATLAB have been developed for accelerating the most computational cost procedures of the model (Forward solver).

Table 5.2: Comparison of features of available CUDA-based numerical linear algebra packages that can be combined with the MATLAB environment.

Package	Open-Source	Sparse		System Solver
		Real	Complex	
Jacket [13]	No	No	No	Yes
GPUmat [21]	Yes	No	No	No
Ad hoc routines based on MEX-files [11]	Yes	Yes	Yes	Yes

5.4.2 Numerical methods and memory optimizations for computing Forward

As aforementioned, our effort is devoted to accelerating accelerate the Forward solver, used in the resolution of the NLODT-P model, by means of HPC techniques such as GPUs platforms (using CUDA interface). The Forward solver is in charge of obtaining a solution for the 3D Helmholtz PDE. The large system of equations obtained from the discretization of the 3D Helmholtz PDE can be solved using different solvers. In general, the Krylov subspace methods based on Lanczos biorthogonalization are effective for

solving complex systems in terms of convergence properties and memory requirements [27, 142, 143]. Examples of this are the Biconjugate Gradient method (BCG) and the Biconjugate Gradient Stabilized method (BCGSTAB). BCG and its variant BCGSTAB are the most widely used iterative solvers for complex systems [127]. These methods are based on a one-term recurrence which consists of a set of vector and matrix operations. In this thesis, in previous chapters, the BCG method and more specifically the BCG-3DH algorithm for solving the Helmholtz equation have been extensively analyzed and several parallel implementation have been discussed and evaluated.

We have developed an approach to accelerate the resolution of the Helmholtz equation using the BCG solver by exploiting the regularities of matrix A and the GPU computing. From a computational point of view, BCG has a high memory requirement because of the storage of a large sparse matrix (A) [112]. It is necessary to use strategies for the reduction of memory resources runtime because the ODT model has double precision complex numbers. Bearing in mind that the sparse matrix exhibits several regularities, a specific storage format, which stores the minimal information to define the sparse matrix, is considered. In this way, both, memory requirements and BCG runtime are considerably reduced since this new format requires less memory accesses to read sparse matrix elements. The format which takes advantage of the regularities of A was called "Compressed Regular Format (CRF)" and was defined in Chapter 4, Section 4.4. CRF optimally minimizes the amount of data needed to store the sparse matrix. CRF significantly reduces the memory requirements with respect to the COO format in a factor of 10. Table 5.3 shows the memory requirements in GB to store the sparse matrix A using the COO format (set by default by MATLAB) and our proposed CRF.

Table 5.3: Memory requirements (GB) for storing A using two formats: COO and CRF.

Vol	COO	CRF
200^3	1.25	0.12
220^3	1.66	0.16
240^3	2.16	0.21
260^3	2.75	0.26
280^3	3.43	0.33

5.5 Computational experiments

This section analyses the computational performance of our NLODT-P model with different dimensions of the volume. In order to show the evolution of the runtime of our NLODT-P model, several tests (with different dimensions of Vol) have been carried out. Four particles of 0.5μ diameter have been located in the tested volumes. The remaining parameters of the example are: $N_h = 3$, $iterMax = 4$ and the values of Vol range from 200^3 to 280^3 voxels. For the evaluation, a CPU ($2\times$ 4 Intel Xeon E5620 cores, 48 GB RAM, 2.4 GHz clock speed and under Linux) and a GPU device NVIDIA Tesla M2090 have been considered. The NVIDIA GPU, which is CUDA enabled, is used in parallelizing the Forward solver. The main characteristics of the GPU device are shown in Chapter 1, Section 1.3.

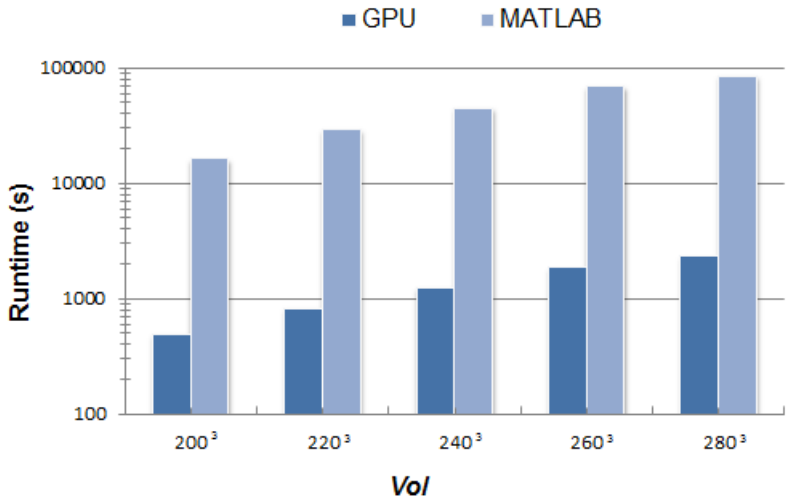


Figure 5.6: Evolution of the runtime of NLODT-P using different values for Vol (from 200^3 to 280^3). The remaining parameters of the example are: $N_h = 3$ and $iterMax = 4$.

As it is well known, there can still be significant differences in performance between a program written in MATLAB and one written in a lower-level language, say C. In this work, the model has been developed and tested for several examples using (1) MATLAB framework; and (2) MATLAB framework and CUDA programming. It means that one GPU device is called from MATLAB for accelerating the two Forward solvers involved for each iteration. Preliminary experiments have shown that both approaches of the model obtain the same accurate results for the three-dimensional location of particles.

Moreover, the use of MATLAB and GPU computing has strongly reduced the runtime. Figure 5.6 shows the runtime (in seconds) of the execution of NLODT-P using both approaches (only MATLAB framework and CUDA-MATLAB combination). For the GPU implementation of NLODT-P, the runtime is approximately reduced by a factor of $\approx 40\times$ with respect to the initial MATLAB version.

5.6 Conclusions

In this chapter we have detailed a three dimensional non-linear ODT model to locate particles, as part of a fluid velocimetry technique. We have presented some numerical experiments to illustrate the accuracy, reliability and effectiveness of the Non-Linear ODT model compared to the Linear ODT one. In particular, we have considered some circumstances when an image post-processing, such as a matched filter, could be enough to unravel the linear ODT image, and when an iterative optimization such as NLODT-P will be compulsory. We have implemented the model using MATLAB combined with GPU computing by means of MEX-files. Acceleration factors $\approx 40\times$ with respect to the approach that only uses MATLAB framework have been obtained. Additionally, the use of a specific format to store the large sparse matrix (A), involved in the BCG method, has reduced the memory requirements until $\approx 10\times$ with respect to the traditional format for specifying sparse matrices in MATLAB (coordinate list (COO)).

As consequence of this work we can say the NLODT-P is able to improve the accuracy of linear ODT methods to locate particles into a fluid and the High Performance Computing is essential to develop and apply this approach. Our future work will be focused on the extension of the NLODT-P model to experimental data of practical interest, it means that larger volumes will have to be computed. For this purpose, an additional effort based on the integration of the distributed resolution of the Helmholtz equation, described in Chapter 4, in combination with MATLAB will be carried out. Thus, the resolution of larger problem sizes of practical interest will be possible.

6

Contributions and future work

In this thesis, several computational issues related to the resolution of linear systems of equations which come from the discretization of physical models described by means of Partial Differential Equations (PDEs) have been discussed. We start from the algebraic description of the model associated with the physical phenomena and its contributions focus on the design of techniques and computational models that allow the resolution of these linear systems of equations. To be more specific, it deals with the study of models which require a high level of discretization inn which the matrix A involved in the system is usually very large and sparse (very low percentage of non-zero elements). One of the major contributions of this thesis is the implementation of routines to solve sparse linear systems of equations using High Performance Computing (HPC). More concretely, routines which require the exploitation of Graphics Processing Units (GPUs) and multi-GPU clusters have been implemented.

In order to compute the Sparse Matrix Matrix product (SpMM) on GPU platforms, a strategy called FastSpMM has been proposed. It has been evaluated and compared to another algorithm described in literature, i.e., the CUSPARSE library (supplied by NVIDIA). Results have shown that FastSpMM outperforms the existing approaches because it combines the use of the ELLPACK-R storage format with the exploitation of the high ratio computation/memory access of the SpMM operation and the overlapping of CPU-GPU communications/computations by CUDA streaming computation. Future work in this research line will consist of extending FastSpMM to matrices with complex elements, broadening the kinds of platforms which can be exploited by FastSpMM. Moreover, we are particularly interested in the reduction of the Processing Time

through improving the memory management. In order to achieve this goal, FastSpMM will be re-written according to the GPU programming tool CudaDMA. Additionally, the implementation of the multi-GPU version of FastSpMM is also a future line to work on.

In addition, a BCG implementation to solve complex and/or nonsymmetric linear systems of equations on GPUs has been carried out ($CuBCG_{ET}$). After an extensive experimental evaluation using two sets of representative test matrices, it can be concluded that $CuBCG_{ET}$ clearly achieves better performance that other approaches in the literature. This is mainly due to the fact that the kernel ELLR-T can be better adapted to a wide variety of sparse matrix patterns. The analysis has shown that despite the fact that the inner products in the BCG algorithm represent a small percentage of the total workload, their poor performance on GPUs has a strong impact on the performance of the BCG.

In this research line a methodology has also been developed for expressing every iteration of BCG by the fusion of algebraic operations in single kernel (see Section 1.2.1.1). Preliminary experimental evaluations of this methodology have shown the relevant improvements on the performance of BCG on GPUs [137]. An extensive evaluation of this methodology applied to other Krylov methods (CG and BCGSTAB) will be part of future work.

In this thesis a solver has also been developed for the 3D Helmholtz equation which combines the exploitation of the high regularity of the matrices involved in the numerical methods, and the massive parallelism supplied by heterogeneous architecture of modern multi-GPU clusters. This parallel approach (called Fast-Helmholtz) not only makes it possible to obtain a faster solution, but also to solve larger size instances. According to this study's experimental results, several aspects can still be improved, so future work will be focused on further optimizations. The difficulty of having the CPU and GPU workload well-balanced is an important part of our current and future research work. Our goal will consist of determining the automatic workload distribution among CPUs and GPUs. Furthermore, another future work will be focused on the development of a hybrid version MPI-OpenMP on the heterogeneous multi-GPU cluster. Apart from that, in order to increase the flexibility of the implementation, we also plan to use rCUDA. This virtualization framework allows the execution of GPU-

accelerated applications within virtual machines (VMs), enabling the sharing of a pool of centralized GPU resources that reside externally to the compute nodes.

Regarding the contribution of this thesis to the resolution of large and sparse linear systems of equation, there are several research issues that we believe are worth exploring in the future. One of them is the development of new strategies for the most widely used routines in the resolution of real problems. We plan to create an efficient open source library capable of improving the performance of other existing approaches. Moreover, we are interested in the performance evaluation of some of these routines on many-core architectures such as the new GPU generations and Xeon Phi coprocessors.

Chapter 5 of this thesis has been focused on the resolution of a real physical problem of the Optical Diffractional Tomography (ODT) based on holographic information. It allows the extraction of the shape of objects with high accuracy and with a non-damaging technique. To be more specific, a three dimensional non-linear ODT model (named NLODT-P) to locate particles, as part of a fluid velocimetry technique, has been implemented using MATLAB interface in combination with GPU devices. Our implementation based on HPC has proven to be compulsory for the resolution of the 3D reconstruction based on non-linear ODT problems. With a long term perspective, we plan to solve real problems of HPIV in aneurysms models where locating hundreds of particles will be needed.

Glossary

This glossary provides a list of the nomenclature and definitions of acronyms used in the thesis.

API: Application Programming Interfaces

ASIC: Application-Specific Integrated Circuit

BLAS: Basic Linear Algebra Subprograms

BCG: Biconjugate Gradient

BCG-3DH: BiConjugate Gradient method solving the 3D Helmholtz equation

BCGSTAB: BiConjugate Gradient Stabilized

BS: Block Size

CG: Conjugate Gradient

CGM: Conjugate Gradient Optimization Method

CGS: Conjugate Gradient Squared method

COO: COOrdinate storage format

CPU: Central processing Unit

CRF: Compressed Regular Format

CRS: Compressed Row Storage

CUBLAS: Optimized BLAS for NVIDIA based GPU cards

CUDA: Compute Unified Device Architecture

CULA: NVIDIA CUDA Linear Algebra library

CUSP: NVIDIA CUDA sparse linear algebra and graph computations library

CUSPARSE: NVIDIA CUDA Sparse Matrix library

DRAM GDDR: DRAM Graphics Double Data Rate

DP: Double Precision

DSP: Digital Signal Processor

FGMRES: Flexible Generalized Minimal Residual method

FPGA: Field-Programmable Gate Array

GMRES: Generalized Minimal Residual method

GPU: Graphics processing Unit

HPC: High Performance Computing

HPIV: Holographic Particle Image Velocimetry

ILP: Instruction Level Parallelism

ISA: Instruction Set Architecture

LAPACK: Linear Algebra PACKage

MAGMA: Linear Algebra library for heterogeneous GPU-based architectures

MATLAB: High-level language and interactive environment for computation

MEX-file: MEX stands for MATLAB Executable

MIMD: Multiple Instructions Multiple Data

MISD: Multiple Instructions Single Data

MKL: Math Kernel Library

MPI: Message Passing Interface

MPMD: Multiple Programs Multiple Data

MUMPS: MUltifrontal Massively Parallel sparse direct Solver

NA: Numerical Aperture

NUMA: Non-Uniform Memory Access systems

ODT: Optical Diffraction Tomography

OpenCL: Open Computing Language

PDE: Partial Differential Equation

PETSc: Portable, Extensible Toolkit for Scientific computation

Pthreads: Posix Threads

QCG: Quadratic Conjugate Gradient

QMR: Quasi-minimal Residue

SAXPY: Single-Precision A · X Plus Y

SIMD: Single Instruction Multiple Data

SISD: Single Instruction Single Data

SIMT: Single Instruction, Multiple Threads

SM-SIMD: Shared Memory SIMD

SM-MIMD: Shared Memory MIMD

SM: Streaming Multiprocessor

SP: Single Precission

SpMM: Sparse Matrix Matrix product

SpMV: Sparse Matrix Vector Product

SuperLU: Sparse Direct Solver

TFLOPS: Tera Floating Point Operations per Second

TFQMR: Transpose-free QMR

UMA: Uniform Memory Access systems

Bibliography

[1] Intel math kernel library (reference manual) (630813-060us), 2013. URL: http://download-software.intel.com/sites/products/documentation/doclib/mkl_sa/111/mklman.pdf.

[2] Engadget Site, accessed 15 april 2014. URL: http://www.engadget.com/2013/03/19/nvidia-roadmap-volta-gpu/.

[3] rCUDA Site, accessed 2 April 2014. URL: http://www.rcuda.net/.

[4] CUBLAS user guide (du-06702-001_v5.5), accessed 21 april 2014. URL: http://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf.

[5] CUDA CUSPARSE library (du-06709-001_v6.0), accessed 21 april 2014. URL: http://docs.nvidia.com/cuda/pdf/CUSPARSE_Library.pdf.

[6] MATLAB R2014, accessed 21 april 2014. URL: http://www.mathworks.com/help/pdf_doc/matlab/getstart.pdf.

[7] Next generation CUDA architecture. Fermi Architecture, accessed 21 april 2014. URL: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[8] NVIDIA CUDA TOOLKIT V5.5 (RN-06722-001_v5.5.), accessed 21 april 2014. URL: http://docs.nvidia.com/cuda/pdf/CUDA_Toolkit_Release_Notes.pdf.

[9] NVIDIA Site, accessed 21 april 2014. URL: http://www.nvidia.co.uk/object/geforce-gtx-750-feb18-2014-uk.html.

[10] OpenCL. Introduction and overview, accessed 21 april 2014. URL: http://www.khronos.org/assets/uploads/developers/library/overview/OpenCL-Overview-Jun10.pdf.

[11] MATLAB with Mex Files. Mathworks, accessed 21 january 2014. URL: http://www.mathworks.com/help/matlab/matlab_external/c-c-source-mex-files.html.

[12] PETSc Users Manual. Revision 3.3, accessed 29 april 2014. URL: http://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf.

[13] ArrayFire for CUDA and OpenCL, accessed 29 march 2014. URL: http://arrayfire.com/arrayfire/.

[14] CUDA C Best Practices Guide, accessed 29 march 2014. URL: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#abstract.

[15] CULA Site, accessed 29 march 2014. URL: http://www.culatools.com/.

[16] CUSP Site, accessed 29 march 2014. URL: http://code.google.com/p/cusp-library/.

[17] Intel C++ Compiler XE 13.1 User and Reference Guides, accessed 29 march 2014. URL: https://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/.

[18] MAGMA Site, accessed 29 march 2014. URL: http://icl.cs.utk.edu/magma/software/index.html.

[19] Mathematica edition: Version 9.0, accessed 29 march 2014. URL: http://www.wolfram.com/mathematica/.

[20] TOP 500 Supercomputing Site, accessed 29 march 2014. URL: http://www.top500.org/.

[21] GPUmat: GPU Toolbox for MATLAB, accessed 3 march 2014. URL: http://sourceforge.net/projects/gpumat/.

[22] P.R. Amestoy, A. Buttari, I.S. Duff, A. Guermouche, J.Y. L'Excellent, and B. Uçar. MUMPS. In David Padua, editor, *Encyclopedia of Parallel Computing*. Springer, 2011.

[23] E. Anderson et al. *LAPACK Users' guide (Third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[24] D. Andrade, B.B. Fraguela, J. Brodman, and D. Padua. Task-parallel versus data-parallel library-based programming in multicore systems. In Didier El Baz, François Spies, and Tom Gross, editors, *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 101–110. IEEE Computer Society, Feb 2009.

[25] H. Anzt, M. Castillo, J.C. Fernández, V. Heuveline, F.D. Igual, R. Mayo, and E. Quintana-Ort. Optimization of power consumption in the iterative solution of sparse linear systems on graphics processors. *Computer Science - Research and Development*, 27(4):299–307, 2012.

[26] M.P. Arroyo and K.D. Hinsch. Recent Developments of PIV towards 3D Measurements. In *Particle Image Velocimetry*, volume 112 of *Topics in Applied Physics*, pages 127–154. Springer Berlin Heidelberg, 2008.

[27] O. Axelsson and A. Kucherov. Real valued iterative methods for solving complex symmetric linear systems. *Numerical Linear Algebra with Applications*, 7(4):197–218, 2000.

[28] H. Ayasso, B. Duchêne, and A. Mohammad-Djafari. Bayesian inversion for optical diffraction tomography. *J. Mod. Opt.*, 57(9):765–776, 2010.

[29] I. Babuska, U. Banerjee, and J.E. Osborn. Generalized Finite Element Methods-Main ideas, Results and Perspective. *International Journal of Computational Methods*, 01(01):67–103, 2004.

[30] I. Babuska and S. Sauter. Is the pollution effect of the fem avoidable for the helmholtz equation considering high wave numbers? *SIAM Review*, 42(3):451–484, 2000.

[31] R. Barrett, M.W. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*, volume 43. SIAM, Philadelphia, PA, 1994.

[32] A. Basermann, B. Reichel, and C. Schelthoff. Preconditioned CG methods for sparse matrices on massively parallel machines. *Parallel Computing*, 23(3):381 – 398, 1997.

[33] M. M. Baskaran and R. Bordawekar. *Sparse Matrix-Vector Multiplication Toolkit for Graphics Processing Units*. Ohio, USA, April, 2009. URL: http://www.alphaworks.ibm.com/tech/spmv4gpu.

[34] M.M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs. *IBM Research Report RC24704, IBM*, April 2009.

[35] M. Bauer, H. Cook, and B. Khailany. CudaDMA: Optimizing GPU memory bandwidth via warp specialization. In *Proc. of Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC '11)*, pages 12:1–12:11. ACM, New York, NY, USA, Seattle, Washington, November 12–18 2011.

[36] K. Belkebir and A. Sentenac. High-resolution optical diffraction microscopy. *J. Opt. Soc. Am. A*, 20(7):1223–1229, Jul 2003.

[37] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM. doi:10.1145/1654059.1654078.

[38] R.H. Bisseling. *Parallel Scientific Computation*. Oxford Univ. Press, 2004.

[39] OpenMP Architecture Review Board. OpenMP Application Program Interface (version 4.0). Technical report, July 2013.

[40] C. Bordage. Parallelization on Heterogeneous Multicore and Multi-GPU Systems of the Fast Multipole Method for the Helmholtz Equation Using a Runtime System. In *ADVCOMP 2012, The Sixth International Conference on Advanced Engineering Computing and Applications in Sciences*, pages 90–95, 2012.

[41] S.C. Brenner and L.R. Scott. *The mathematical theory of finite element methods*. Texts in applied mathematics. Springer-Verlag, New York, 1994.

[42] I.N. Bronshtein and K.A. Semendyayev. *Handbook of Mathematics (3rd Ed.)*. Springer-Verlag, London, UK, 1997.

[43] L. Buatois, G. Caumon, and B. Lévy. Concurrent number cruncher - A GPU implementation of a general sparse linear solver. *International Journal of Parallel Emergent and Distributed Systems*, 24(3):205–223, 2009.

[44] D.R. Butenhof. *Programming with POSIX Threads*. Professional Computing Series. Addison-Wesley, 1997.

[45] D. Castaño-Diez, D. Moser, A. Schoenegger, S. Pruggnaller, and A.S. Frangakis. Performance evaluation of image processing algorithms on the GPU. *J. Struct. Biol.*, 164(1):153–60, 2008.

[46] S. Chattopadhyay. *Compiler Design*. PHI Learning, 2005.

[47] P.C. Chaumet, A. Sentenac, K. Belkebir, G. Maire, and H. Giovannini. Improving the resolution of grating-assisted optical diffraction tomography using a priori information in the reconstruction procedure. *J. Mod. Optic.*, 57(9):798–808, 2010.

[48] 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 '10)*, pages 115–126. ACM, NY, USA, 2010.

[49] J.M. Coupland and N.A. Halliwell. Particle image velocimetry: Three-dimensional fluid velocity measurements using holographic recording and optical correlation. *Applied optics*, 31(8):1005–1007, 1992.

[50] R. Dautray and J.L. Lions. *Mathematical analysis and numerical methods for science and technology*. Springer-Verlag, Berlin, 1990.

[51] T.A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

[52] H. Diao. *Fourier Analysis of Iterative Methods for the Helmholtz problem*. Master Thesis at Delft University of Technology. Faculty of Electrical Engineering, Mathematics and Computer Science. 2012.

[53] J. Dongarra. *Freely Available Software for Lineal Algebra on the Web*. http://www.netlib.org, March 2014.

[54] D. Donno, A. Esposito, G. Monti, and L. Tarricone. Iterative solution of linear systems in electromagnetics (and not only): Experiences with CUDA. In *Euro-Par 2010 Parallel Processing Workshops*, volume 6586 of *LNCS*, pages 329–337. Springer, 2011.

[55] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Inc., New York, USA, 1986.

[56] A. Dziekonski, A. Lamecki, and M. Mrozowski. A memory efficient and fast sparse matrix vector product on a GPU. *Progress In Electromagnetics Research*, 116:49–63, 2011.

[57] D.L. Eager, J. Zahorjan, and E.D. Lazowska. Speedup versus efficiency in parallel systems. *Computers, IEEE Transactions on*, 38(3):408–423, 1989.

[58] J.W. Eaton, D. Bateman, and S. Hauberg. *GNU Octave Manual Version 3*. Network Theory Ltd., 2008.

[59] H.C. Elman and D.P. O'Leary. Efficient Iterative Solution of the Three-dimensional Helmholtz Equation. *J. Comput. Phys.*, 142(1):163–181, 1998.

[60] G.E. Elsinga. *Tomographic particle image velocimetry and its application to turbulent boundary layers*. PhD thesis, Delft University of Technology, June 2008.

[61] K. Eriksson, D. Estep, P. Hansbo, and C. Johnson. *Computational Differential Equations*. Cambridge University Press, 1996.

[62] Y. Erlangga. Advances in iterative methods and preconditioners for the Helmholtz equation. *Archives of Computational Methods in Engineering*, 15(1):37–66, 2008.

[63] O. Ernst and G.H. Golub. A domain decomposition approach to solving the Helmholtz equation with a radiation boundary condition. In *Domain Decomposition in Science and Engineering*, pages 177–192. American Mathematical Society, 1994.

[64] V. Faber and T. Manteuffel. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM Journal on Numerical Analysis*, 21(2):352–362, 1984.

[65] M.J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 21(9):948–960, 1972.

[66] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, Boston, MA, USA, 1995.

[67] R.W. Freund. A transpose-free quasi-minimal residual algorithm for non-hermitian linear systems. *SIAM J. Sci. Comput.*, 14(2):470–482, 1993.

[68] A. Gaikwad and I.M. Toke. Parallel iterative linear solvers on GPU: A financial engineering case. In *Proc. of the 18th Euromicro Conf. on Parallel, Distributed and Network-based Processing*, pages 607–614. IEEE Computer Society, 2010.

[69] N. Garcia. Parallel power flow solutions using a biconjugate gradient algorithm and a newton method: A GPU-based approach. In *Power and Energy Society General Meeting, 2010 IEEE*, pages 1 –4. july 2010.

[70] G.H. Golub and C.F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.

[71] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris. Performance evaluation of the sparse matrix-vector multiplication on modern architectures. *J Supercomput.*, 50(1):36–77, 2009.

[72] A. Greenbaum. *Iterative methods for solving linear systems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[73] N.A. Gumerov and R. Duraiswami. *Fast Multipole Methods For The Helmholtz Equation In Three Dimensions.* Electronics & Electrical. Elsevier Science & Technology Books, 2004.

[74] M. Harris. Fast fluid dynamics simulation on the GPU. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05. ACM, New York, NY, USA, 2005.

[75] J.L. Hennessy and D.A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.).* Morgan Kaufmann, 2012.

[76] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49:409–436, 1952.

[77] U. Banerjee I. Babuska and J.E. Osborn. Generalized Finite Element Methods-Main ideas, Results and Perspective. *Journal of Computational Methods*, 1(01):765–776, 153156.

[78] F. Ihlenburg and I. Babuska. Solution of Helmholtz problems by knowledge-based FEM. *Computer Assisted Mechanics and Engineering Sciences*, 4:397–416, 1997.

[79] F. Ihlenburg and I. Babuška. Finite element solution of the Helmholtz equation with high wave number Part I: The h-version of the FEM. *Computers & Mathematics with Applications*, 30(9):9–37, November 1995.

[80] D.A. Jacobsen, J.C. Thibault, and I. Senocak. An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters. In *48th AIAA Aerospace Sciences Meeting and Exhibit*, volume 16, 2010.

[81] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming.* Morgan Kaufmann Pub. Inc., San Francisco, CA, USA, 1st edition, 2013.

[82] M.C. Junger and D. Feit. *Sound, Structures, And Their Interaction, Second Edition.* The MIT press, 1986.

[83] P. Kalinová. *Solving Large Sparse Systems of Linear Equations on GPU*. Master Thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Computer Graphics and Interaction. 2011.

[84] K. Kim, K.S. Kim, H. Park, J.C. Ye, and Y. Park. Real-time visualization of 3-D dynamic microscopic objects using optical diffraction tomography. *Opt. Express*, 21(26):32269–32278, Dec 2013.

[85] C. Kincaid. *Numerical Analysis*. Brooks/Cole, 1993.

[86] D.R. Kincaid, T.C. Oppe, and D.M. Young. *ITPACKV 2D User's Guide*, 1989. URL: http://rene.ma.utexas.edu/CNA/ITPACK/manuals/userv2d/.

[87] H. Knibbe, C.W. Oosterlee, and C. Vuik. 3D Helmholtz Krylov Solver Preconditioned by a Shifted Laplace Multigrid Method on Multi-GPUs. In *Numerical Mathematics and Advanced Applications 2011*, pages 653–661. Springer, 2013.

[88] W.D. Koek, N. Bhattacharya, and J.J.M. Braat. Influence of virtual images on the signal-to-noise ratio in digital in-line particle holography. *Opt. Express*, (13):2578–2589, 2005.

[89] A.V. Kononov, C.D. Riyanti, S.W. de Leeuw, C.W. Oosterlee, and C. Vuik. Numerical performance of a parallel solution method for a heterogeneous 2D Helmholtz equation. *Computing and Visualization in Science*, 11(3):139–146, April 2008.

[90] J. Kurzak, W. Alvaro, and J. Dongarra. Optimizing matrix multiplication for a short-vector SIMD architecture - CELL processor. *Parallel Computing*, 35(3):138 – 150, 2009.

[91] C. Lanczos. Solution of systems of linear equations by minimized iterations. *J. Res. Natl. Bur. Stand*, 49:33–53, 1952.

[92] E. Larsson. A domain decomposition method for the helmholtz equation in a multilayer domain. *SIAM Journal on Scientific Computing*, 20:1713–1731, 1999.

[93] A.L. Lastovetsky. Special issue on heterogeneity in parallel and distributed computing. *Journal of Parallel and Distributed Computing.*, 73(12), 2013.

[94] V. Lauer. New approach to optical diffraction tomography yielding a vector equation of diffraction tomography and a novel tomographic microscope. *Journal of Microscopy*, 205(2):165–176, 2002.

[95] V.W. Lee et al. Debunking the 100X GPU vs. CPU Myth: An evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 38(3):451–460, June 2010.

[96] L. Li, W. Xue, R. Ranjan, and Z. Jin. A scalable Helmholtz solver in GRAPES over large-scale multicore cluster. *Concurrency and Computation: Practice and Experience*, 25(12):1722–1737, 2013.

[97] X.S. Li, J.W. Demmel, J.R. Gilbert, L. Grigori, M. Shao, and I. Yamazaki. SuperLU Users' Guide. Technical report, Berkeley, CA, USA, 2011. URL: http://crd.lbl.gov/~xiaoye/SuperLU/superlu_ug.pdf.

[98] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[99] J. Lobera and J. M. Coupland. Multiple Scattering in HPIV: Use of ODT Analysis Techniques. *Photon*, 2006.

[100] J. Lobera and J.M. Coupland. Optical diffraction tomography in fluid velocimetry: the use of a priori information. *Measurement Science and Technology*, 19(7):074013, 2008.

[101] M.M. Made. Incomplete factorization-based preconditionings for solving the Helmholtz equation. *Internat. J. Numer. Methods Eng*, 50(5):1077–1101, 2001.

[102] G. Maire et al. Experimental Demonstration of Quantitative Imaging beyond Abbe's Limit with Optical Diffraction Tomography. *Phys. Rev. Lett.*, 102:213905, May 2009.

[103] A.D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb. Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs. In *Proc. of the 2011 Int. Conf. on Parallel Processing*, ICPP '11, pages 176–185, Washington, DC, USA, 2011. IEEE Computer Society.

[104] M. Marcus and H. Minc. *Introduction to linear algebra*. Dover Publications, New York, 1988. Reprint. Originally published: New York : Macmillan, 1965.

[105] J. Mellor-Crummey and J. Garvin. Optimizing Sparse Matrix-Vector Product Computations Using Unroll and Jam. *Int. Journal of High Performance Computing Applications.*, 18(2):225–236, 2004.

[106] A. Monakov, A. Lokhmotov, and A. Avetisyan. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *Proc. of HiPEAC 2010, LNCS 5952*, pages 111–125. Pisa, Italy, January 25–27 2010.

[107] J. Nickolls and W.J. Dally. The GPU Computing Era. *IEEE Micro*, 30(2):56–69, March 2010.

[108] A.T. Ogielski and W. Aiello. Sparse matrix computations on parallel processor arrays. *SIAM J. Sci. Comput*, 14:519–530, 1992.

[109] G. Ortega, I. García, and E.M. Garzón. A Hybrid Approach for Solving the 3D Helmholtz Equation on Heterogeneous Platforms. In *Euro-Par 2013: Parallel Processing Workshops*, volume 8374 of *LNCS*, pages 198–207. Springer Berlin Heidelberg, 2014.

[110] G. Ortega, E.M. Garzón, F. Vázquez, and I. García. Aceleración del método del gradiente biconjugado para matrices dispersas en GPUs. In *Proc. of the I Minisimposium de Ciencias Experimentales*. Almería, Spain, November 2011.

[111] G. Ortega, E.M. Garzón, F. Vázquez, and I. García. Evaluación del método del gradiente biconjugado para matrices dispersas en GPUs. In *Proc. of the XXII Jornadas de Paralelismo (JP2011)*, volume II, pages 135–140. La Laguna, Spain, September 2011.

[112] G. Ortega, E.M. Garzón, F. Vázquez, and I. García. Exploiting the regularity of differential operators to accelerate solutions of PDEs on GPUs. In *11th Int. Conf. on Computational and Mathematical Methods in Science and Engineering.CMMSE*, pages 908–917, Benidorm. Spain, June 26–30 2011.

[113] G. Ortega, E.M. Garzón, F. Vázquez, and I. García. The BiConjugate gradient method on GPUs. *J Supercomput.*, 64:49–58, 2013.

[114] G. Ortega, J. Lobera, M.P. Arroyo, I. García, and E.M. Garzón. High Performance Computing for Optical Diffraction Tomography. In *Proc. of The 2012 International Conference on High Performance Computing & Simulation (HPCS 2012)*, pages 195–201. IEEE Computer Society, Madrid, Spain, July 2 - 6 2012.

[115] G. Ortega, J. Lobera, M.P. Arroyo, I. García, and E.M. Garzón. Exploration of a HPC approach for coherent tomography. In *Proc. of the 2013 International Conference on Mathematical Methods in Science and Engineering (CMMSE)*, pages 1109–1113. June 2013.

[116] G. Ortega, J. Lobera, M.P. Arroyo, I. García, and E.M. Garzón. Parallel resolution of the 3D Helmholtz Equation based on multi-GPU clusters. *Concurrency and Computation: Practice and Experience*, 2014. doi:10.1002/cpe.3212.

[117] G. Ortega, J.A. Martínez, E.M. Garzón, A. Plaza, and I. García. ADITHE: An approach to optimise iterative computation on heterogeneous multiprocessors. In *Proc. of the XX Jornadas de Paralelismo*, pages 111–116. La Coruña, Spain, September 2009.

[118] G. Ortega, F. Vázquez, I. García, and E.M. Garzón. FastSpMM: An efficient library for sparse matrix matrix product on GPUs. *The Computer Journal*, 2013. doi:10.1093/comjnl/bxt038.

[119] K. Otto and E. Larsson. Iterative solution of the Helmholtz Equation by a second-order method. *SIAM Journal on Matrix Analysis and Applications*, 21:209–229, 1999.

[120] C.C. Paige and M.A. Saunders. Solution of Sparse Indefinite Systems of Linear Equations. *SIAM Journal on Numerical Analysis*, 12(4):617–629, 1975.

[121] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2012.

[122] J.A. Piedra-Fernández, G. Ortega, J.Z. Wang, and M. Cantón-Garbín. Fuzzy content-based image retrieval for oceanic remote sensing. *IEEE Trans. Geosci. Remote Sens.*, 52(9):5422–5431, Sept 2014.

[123] J. Popovic and O. Runborg. Analysis of a fast method for solving the high frequency Helmholtz equation in one dimension. *BIT Numerical Mathematics*, 51(3):721–755, 2011.

[124] M.J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[125] S. Ristov, M. Gusev, L. Djinevski, and S. Arsenovski. Performance impact of reconfigurable L1 cache on GPU devices. In M. Ganzha, L.A. Maciaszek, and M. Paprzycki, editors, *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*, pages 507–510, 2013.

[126] Y. Saad. Krylov subspace methods on parallel computers. In M. Papadrakakis, editor, *Solving Large-Scale Problems in Mechanics: Parallel and Distributed Computer Applications*. J. Wiley, 1996.

[127] Y. Saad. *Iterative Methods for Sparse Linear Systems, Second Edition*. SIAM, April 2003.

[128] Y. Saad and M.H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, 1986.

[129] M.N.O. Sadiku. *Numerical Techniques in Electromagnetics. Second Edition*. CRC Press, Boca Raton, 2001.

[130] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[131] J. Sheng, E. Malkiel, and J. Katz. Single Beam Two-Views Holographic Particle Image Velocimetry. *Appl. Opt.*, 42(2):235–250, Jan 2003.

[132] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[133] J. Soria and C. Atkinson. Towards 3C-3D digital holographic fluid velocity vector field measurement-tomographic digital holographic PIV (Tomo-HPIV). *Measurement Science and Technology*, 19(7):074002, 2008.

[134] F. Soulez, L. Denis, É. Thiébaut, C. Fournier, and C. Goepfert. Inverse problem approach in particle digital holography: out-of-field particle detection made possible. *J. Opt. Soc. Am. A*, 24(12):3708–3716, 2007.

[135] G. Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, 1993.

[136] S. Tabik. *Parallel computing of partial differential equations-based applications*. PhD thesis, Dpt. Computer Architecture. University of Almería, 2008.

[137] S. Tabik, G. Ortega, and E.M. Garzón. Performance Evaluation of Kernel Fusion BLAS Routines on the GPU: Iterative Solvers as Case Study. *J Supercomput.*, 2014. doi:10.1007/s11227-014-1102-4.

[138] S. Toledo. Improving the memory-system performance of sparse matrix-vector multiplication. *IBM J. Res. Dev.*, 41(6):711–725, 1997.

[139] Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos. Understanding the Impact of CUDA tuning techniques for Fermi. In *Proc. of The 2011 International Conference on High Performance Computing & Simulation (HPCS 2011)*, pages 631–639. IEEE Compute Society, Istanbul, Turkey, July 4-8 2011, 2011.

[140] A.J. van Der Steen. Overview of Recent Supercomputers. Technical report, EuroBen Foundation, SURFsara, The Netherlands, 2013. URL: https://surfsara.nl/sites/default/files/SURFsara_recent_supercomputers_2013.pdf.

[141] H.A. van der Vorst. BI-CGSTAB: A Fast and Smoothly Converging Variant of BI-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comput.*, 13(2):631–644, March 1992.

[142] H.A. van der Vorst. *Iterative Krylov methods for large linear systems*. Cambridge monographs on applied and computational mathematics. Cambridge University Press, Cambridge, UK, New York, 2003.

[143] H.A. van der Vorst and J.B.M. Melissen. A Petrov-Galerkin type method for solving Axk=b, where A is symmetric complex. *Magnetics, IEEE Transactions on*, 26(2):706–708, 1990.

[144] R.S. Varga. *Matrix Iterative Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1962.

[145] F. Vázquez, J.J. Fernández, and E.M. Garzón. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience*, 23:815–826, 2011.

[146] F. Vázquez, J.J. Fernández, and E.M. Garzón. Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach. *Parallel Computing*, 38(8):408–420, 2012.

[147] F. Vázquez, E.M. Garzón, and J.J. Fernández. A matrix approach to tomographic reconstruction and its implementation on GPUs. *J. Struct. Biol*, 170(1):146 – 151, 2010.

[148] F. Vázquez, E.M. Garzón, and J.J. Fernández. Matrix implementation of simultaneous iterative reconstruction technique (SIRT) on GPUs. *The Computer Journal*, 54(11):1861–1868, 2011.

[149] F. Vázquez, E.M. Garzón, J.A. Martínez, and J.J. Fernández. Accelerating sparse matrix vector product with GPUs. In *9th Int. Conf. on Computational and Mathematical Methods in Science and Engineering. CMMSE*, pages 1081–1092, Gijón. Spain, July 1–3 2009.

[150] F. Vázquez, G. Ortega, J.J. Fernández, I. García, and E.M. Garzón. Fast Sparse Matrix Matrix Product Based on ELLR-T and GPU Computing. In *Proc. of the IEEE 10th Int. Symposium on Parallel and Distributed Processing with Applications (ISPA '12)*, pages 669–674. IEEE Computer Society, Madrid, Spain, July 10–13 2012.

[151] F. Vázquez, G. Ortega, J.J. Fernández, and E.M. Garzón. Evaluating the sparse matrix vector product on multi-GPU. In *Proc. of the 10th Int. Conf. on Computational and Mathematical Methods in Science and Engineering. CMMSE*, pages 961–972. Almería. Spain, 2010.

[152] F. Vázquez, G. Ortega, J.J. Fernández, and E.M. Garzón. Improving the performance of the sparse matrix vector product with GPUs. In *Proc. IEEE Int. Conf.*

Computer and Information Technology (CIT 2010), pages 1146–1151, Bradford, UK, June 29–July 1 2010. IEEE Press.

[153] V. Voevodin. The problem of non-self-adjoint generalization of the Conjugate Gradient method is closed. *Comput. Maths. and Math. Phys.*, 23:143–144, 1983.

[154] H.F. Walker. Implementation of the GMRES method using Householder Transformations. *SIAM J. Sci. Stat. Comput.*, 9(1):152–163, January 1988.

[155] J.B. White and J.J. Dongarra. Overlapping Computation and Communication for Advection on Hybrid Parallel Computers. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 59–67. IEEE, 2011.

[156] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.

[157] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[158] E. Wolf. Three-Dimensional structure determination of semi-transparent objects from holographic data. *Opt. Commun.*, 1(4):153–156, 1969.

[159] L. Xiaoyes and J.W. Demmel. SuperLU DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29:110–140, 2003.

[160] D.M. Young. *Iterative Solution of Large Linear Systems.* Academic Press Inc., 1971.

[161] T. Zhang et al. Full-polarized tomographic diffraction microscopy achieves a resolution about one-fourth of the wavelength. *Phys. Rev. Lett.*, 111:243904, Dec 2013.

Publications arising from this thesis

The research work carried out for the present thesis resulted in a number of publications. This appendix lists them along with their respective quality indicators and sorted by their year of publication (oldest first) within each category.

Publications in International Journals

[113] G. Ortega, E.M. Garzón, F. Vázquez, and I. García. The BiConjugate gradient method on GPUs. *J Supercomput.*, 64:49–58, 2013

Impact factor JCR 2012: 0.917.

[118] G. Ortega, F. Vázquez, I. García, and E.M. Garzón. FastSpMM: An efficient library for sparse matrix matrix product on GPUs. *The Computer Journal*, 2013. doi:10.1093/comjnl/bxt038

Impact factor JCR 2012: 0.755.

[116] G. Ortega, J. Lobera, M.P. Arroyo, I. García, and E.M. Garzón. Parallel resolution of the 3D Helmholtz Equation based on multi-GPU clusters. *Concurrency and Computation: Practice and Experience*, 2014. doi:10.1002/cpe.3212

Impact factor JCR 2012: 0.845.

[137] S. Tabik, G. Ortega, and E.M. Garzón. Performance Evaluation of Kernel Fusion BLAS Routines on the GPU: Iterative Solvers as Case Study. *J Supercomput.*, 2014. doi:10.1007/s11227-014-1102-4

Impact factor JCR 2012: 0.917.

Publications in Proceedings of International Conferences with DOI

[152] F. Vázquez, G. Ortega, J.J. Fernández, and E.M. Garzón. Improving the performance of the sparse matrix vector product with GPUs. In *Proc. IEEE Int. Conf. Computer and Information Technology (CIT 2010)*, pages 1146–1151, Bradford, UK, June 29–July 1 2010. IEEE Press

[114] G. Ortega, J. Lobera, M.P. Arroyo, I. García, and E.M. Garzón. High Performance Computing for Optical Diffraction Tomography. In *Proc. of The 2012 International Conference on High Performance Computing & Simulation (HPCS 2012)*, pages 195–201. IEEE Computer Society, Madrid, Spain, July 2 - 6 2012

[150] F. Vázquez, G. Ortega, J.J. Fernández, I. García, and E.M. Garzón. Fast Sparse Matrix Matrix Product Based on ELLR-T and GPU Computing. In *Proc. of the IEEE 10th Int. Symposium on Parallel and Distributed Processing with Applications (ISPA '12)*, pages 669–674. IEEE Computer Society, Madrid, Spain, July 10–13 2012

[109] G. Ortega, I. García, and E.M. Garzón. A Hybrid Approach for Solving the 3D Helmholtz Equation on Heterogeneous Platforms. In *Euro-Par 2013: Parallel Processing Workshops*, volume 8374 of *LNCS*, pages 198–207. Springer Berlin Heidelberg, 2014

Publications in other International Conferences

[151] F. Vázquez, G. Ortega, J.J. Fernández, and E.M. Garzón. Evaluating the sparse matrix vector product on multi-GPU. In *Proc. of the 10th Int. Conf. on Computational and Mathematical Methods in Science and Engineering. CMMSE*, pages 961–972. Almería. Spain, 2010

[112] G. Ortega, E.M. Garzón, F. Vázquez, and I. García. Exploiting the regularity of differential operators to accelerate solutions of PDEs on GPUs. In *11th Int.*

Conf. on Computational and Mathematical Methods in Science and Engineering.CMMSE, pages 908–917, Benidorm. Spain, June 26–30 2011

[115] G. Ortega, J. Lobera, M.P. Arroyo, I. García, and E.M. Garzón. Exploration of a HPC approach for coherent tomography. In *Proc. of the 2013 International Conference on Mathematical Methods in Science and Engineering (CMMSE)*, pages 1109–1113. June 2013

Publications in National Conferences

[117] G. Ortega, J.A. Martínez, E.M. Garzón, A. Plaza, and I. García. ADITHE: An approach to optimise iterative computation on heterogeneous multiprocessors. In *Proc. of the XX Jornadas de Paralelismo*, pages 111–116. La Coruña, Spain, September 2009

[110] G. Ortega, E.M. Garzón, F. Vázquez, and I. García. Aceleración del método del gradiente biconjugado para matrices dispersas en GPUs. In *Proc. of the I Minisimposium de Ciencias Experimentales*. Almería, Spain, November 2011

[111] G. Ortega, E.M. Garzón, F. Vázquez, and I. García. Evaluación del método del gradiente biconjugado para matrices dispersas en GPUs. In *Proc. of the XXII Jornadas de Paralelismo (JP2011)*, volume II, pages 135–140. La Laguna, Spain, September 2011

Other publications produced during the elaboration of this thesis

The research effort invested during the time span in which this thesis was elaborated produced an additional publication as the result of other research lines not included in the present dissertation. Those lines were image processing, pattern recognition and image retrieval. This appendix lists it along with its respective quality indicator.

Publications in International Journals

[122] J.A. Piedra-Fernández, G. Ortega, J.Z. Wang, and M. Cantón-Garbín. Fuzzy content-based image retrieval for oceanic remote sensing. *IEEE Trans. Geosci. Remote Sens.*, 52(9):5422–5431, Sept 2014

Impact factor JCR 2012: 3.467