UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Accelerating advanced preconditioning methods on hybrid architectures

Ernesto Dufrechou

Programa de Doctorado en Informática
PEDECIBA
Universidad de la República

Montevideo – Uruguay
Agosto de 2019

UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Accelerating advanced preconditioning methods on hybrid architectures

Ernesto Dufrechou

Tesis de Doctorado presentada al Programa de Doctorado en Informática del PEDECIBA, como parte de los requisitos necesarios para la obtención del título de Doctor en Informática.

Director de tesis:
D.Sc. Prof. Pablo Ezzatti

Codirector:
D.Sc. Prof. Enrique Quintana-Ortí

Director académico:
D.Sc. Prof. Pablo Ezzatti

Montevideo – Uruguay
Agosto de 2019

Dufrechou, Ernesto

Accelerating advanced preconditioning methods on hybrid architectures / Ernesto Dufrechou. - Montevideo: Universidad de la República, PEDECIBA, 2019.

xviii, 149 p. 29, 7cm.

Director de tesis:

Pablo Ezzatti

Codirector:

Enrique Quintana-Ortí

Director académico:

Pablo Ezzatti

Tesis de Doctorado – Universidad de la República, Programa de Doctorado en Informática, 2019.

Referencias bibliográficas: p. 101 – 149.

1. Sistemas lineales dispersos, 2. Precondicionadores, 3. Unidades de Procesamiento Gráfico (GPU), 4. Paralelismo de datos, 5. ILUPACK.

INTEGRANTES DEL TRIBUNAL DE DEFENSA DE TESIS

Dr. Hartwig Anzt

Dr. Manuel Ujaldón

Dr. José E. Moreira

Dr. Mauricio Delbracio

Dr. Héctor Cancela

Montevideo – Uruguay
Agosto de 2019

Agradecimentos

Obtener un doctorado en cualquier área de la ciencia es un proceso que involucra un considerable esfuerzo, una cantidad importante de dinero, y un sustento psicológico y afectivo mantenido durante varios años. Por lo tanto son muchos los que han contribuido a la finalización de esta tesis. Deseo expresar mi más sincero agradecimiento a todos los que han brindado su ayuda durante este tiempo, aunque por una cuestión práctica sólo haré algunos reconocimientos, y probablemente cometa alguna injusticia. Espero sepan disculpar.

Agradezco en primer lugar a Pablo Ezzatti y Enrique Quintana-Ortí, mis tutores, y a José Ignacio Aliaga, por el inmenso esfuerzo realizado, a veces dedicando las 24 horas del día, y algunas de la noche, a este trabajo de investigación.

A Mathias Bollhöfer agredezco el tiempo dedicado a la revisión de diversos trabajos, así como sus importantes aportes.

Por otro lado, es necesario el reconocimiento a las diversas entidades que han brindado apoyo financiero para realizar esta investigación. He realizado gran parte de esta tesis gracias a la beca de doctorado de la Comisión Académica de Posgrado de la Universidad de la República.También agradezco al Programa para el Desarrollo de las Ciencias Básicas (PEDECIBA) por su apoyo financiero y logístico. Además, muchas actividades relativas a esta tesis se han realizado gracias al apoyo de la Comisión Sectorial de Investigación Científica (CSIC) de Uruguay.

Agradezco al Centro Extremeño de Tecnologías Avanzadas (CETA-Ciemat) y a la Universidad Jaume I, especialmente a José Antonio Belloch y José Manuel Badía, por permitirme utilizar parte de su infraestructura en varias ocasiones.

Reconozco finalmente el dinero invertido en mí por el sistema educativo público Uruguayo, así como el trabajo abnegado de maestras, profesoras y profesores, durante toda mi vida, sin el cual esta y muchas otras investigaciones no hubieran sido posibles.

En cuanto a lo afectivo, deseo saludar cariñosamente a mis compañeros de la 048, Martín, Jimena, Ignacio, Danilo, Juan Pablo y Rodrigo, y a mis grandes amigos, Ruben, Albert y Nico. Por último, un especial abrazo, e infinito agradecimiento, a mis padres Julio y Hebe, a mi hermano Hugo y a Mariana, el amor de mi vida.

RESUMEN

Un gran número de problemas, en diversas áreas de la ciencia y la ingeniería, involucran la solución de sistemas dispersos de ecuaciones lineales de gran escala. En muchos de estos escenarios, son además un cuello de botella desde el punto de vista computacional, y por esa razón, su implementación eficiente ha motivado una cantidad enorme de trabajos científicos.

Por muchos años, los métodos directos basados en el proceso de la Eliminación Gaussiana han sido la herramienta de referencia para resolver dichos sistemas, pero la dimensión de los problemas abordados actualmente impone serios desafíos a la mayoría de estos algoritmos, considerando sus requerimientos de memoria, su tiempo de cómputo y la complejidad de su implementación.

Propulsados por los avances en las técnicas de precondicionado, los métodos iterativos se han vuelto más confiables, y por lo tanto emergen como alternativas a los métodos directos, ofreciendo soluciones de alta calidad a un menor costo computacional. Sin embargo, estos avances muchas veces son relativos a un problema específico, o dotan a los precondicionadores de una complejidad tal, que su aplicación en diversos problemas se vuelve poco práctica en términos de tiempo de ejecución y consumo de memoria.

Como respuesta a esta situación, es común la utilización de estrategias de Computación de Alto Desempeño, ya que el desarrollo sostenido de las plataformas de hardware permite la ejecución simultánea de cada vez más operaciones. Un claro ejemplo de esta evolución son las plataformas compuestas por procesadores multi-núcleo y aceleradoras de hardware como las Unidades de Procesamiento Gráfico (GPU). Particularmente, las GPU se han convertido en poderosos procesadores paralelos, capaces de integrar miles de núcleos a precios y consumo energético razonables. Por estas razones, las GPU son ahora una plataforma de hardware de gran importancia para la ciencia y la ingeniería, y su uso eficiente es crucial para alcanzar un buen desempeño en la mayoría de las aplicaciones.

Esta tesis se centra en el uso de GPUs para acelerar la solución de sistemas dispersos de ecuaciones lineales usando métodos iterativos precondicionados con técnicas modernas. En particular, se trabaja sobre ILUPACK, que ofrece implementaciones de los métodos iterativos más importantes, y presenta un interesante y moderno precondicionador de tipo ILU multinivel.

En este trabajo, se desarrollan versiones del precondicionador y de los métodos incluidos en el paquete, capaces de explotar el paralelismo de datos mediante el uso de GPUs sin afectar las propiedades numéricas del precondicionador. Además, se habilita y analiza el uso de las GPU en versiones paralelas existentes, basadas en paralelismo de tareas para plataformas de memoria compartida y distribuida. Los resultados obtenidos muestran una sensible mejora en el tiempo de ejecución de los métodos abordados, así como la posibilidad

de resolver problemas de gran escala de forma eficiente.

Palabras claves:
Sistemas lineales dispersos, Precondicionadores, Unidades de Procesamiento Gráfico (GPU), Paralelismo de datos, ILUPACK.

ABSTRACT

Many problems, in diverse areas of science and engineering, involve the solution of large-scale sparse systems of linear equations. In most of these scenarios, they are also a computational bottleneck, and therefore their efficient solution on parallel architectures has motivated a tremendous volume of research.

For many years, direct methods based on the well-known Gaussian Elimination procedure have been the default choice to address these problems, but the dimension of the problems being solved nowadays poses serious difficulties for most of these algorithms, regarding memory requirements, time to solution and complexity of implementation.

Driven by fairly recent advances in preconditioning techniques, iterative methods have become more reliable, and therefore have emerged as an appealing alternative, producing high quality solutions in many cases, and demanding a much smaller computational effort than their direct counterparts. These techniques, however, are often problem-dependent, or involve preconditioners of such complexity that their application in many problems becomes impractical from the point of view of execution time or memory consumption.

A means to alleviate this situation is the use of High Performance Computing (HPC) techniques. Scientific and domestic computing platforms have steadily evolved to enable the parallel execution of more and more operations. A clear example of this evolution is the widespread adoption of hybrid hardware platforms equipped with one or several multicore processors and compute accelerators such as Graphics Processing Units (GPUs). This type of platforms have become extremely powerful parallel architectures that integrate thousands of cores at a reasonable price and energy consumption. For these reasons, GPUs are now a hardware platform of paramount importance to science and engineering, and making an efficient use of them is crucial to achieve high performance in most applications.

This dissertation targets the use of GPUs to enhance the performance of the solution of sparse linear systems using iterative methods complemented with state-of-the-art preconditioned techniques. In particular, we study ILUPACK, a package for the solution of sparse linear systems via Krylov subspace methods that relies on a modern inverse-based multilevel ILU (incomplete LU) preconditioning technique.

We present new data-parallel versions of the preconditioner and the most important solvers contained in the package that significantly improve its performance without affecting its accuracy. Additionally we enhance existing task-parallel versions of ILUPACK for shared- and distributed-memory systems with the inclusion of GPU acceleration. The results obtained show a sensible reduction in the runtime of the methods, as well as the posibility

of addressing large-scale problems efficiently.

Keywords:
Sparse linear systems, Preconditioners, Grpahics Processing Units (GPU), Data-parallelism, ILUPACK.

Contents

CHAPTER 1

Introduction

Partial Differential Equations (PDEs) are, without doubt, one of the most important tools for modeling and understanding several aspects of our universe. There are countless examples of natural phenomena and human activities that can be analyzed as processes governed by PDEs, in areas that range from quantum mechanics to economics [25, 34].

Addressing this type of equations with an analytical approach is not possible in the general case. Thus, in order to solve such equations numerically, a typical strategy is to reduce the equations, which involve continuous quantities, to a set of equations with a finite number of unknowns. This sort of procedure is known as discretization. A natural consequence of the application of discretization techniques is the appearance of systems of linear equations in the methods used to solve PDEs. In most cases, since the entries in the coefficient matrix involved in such systems express local relations between discretized fragments of the domain, these matrices tend to be large and sparse, that is, they have very few nonzero entries.

The above elaboration should be enough to convince the reader that solving large and sparse linear systems efficiently is of great importance to science and engineering, but there are many other applications, such as circuit simulation or optimal control, that are not necessarily governed by PDEs but also rely on the solution of sparse linear systems.

The discussion would end up here if solving this sort of systems was a trivial task, but it is not. A number of current real-world applications (as for example three-dimensional PDEs) involve linear systems with millions of equations and unknowns. Direct solvers such as those based on Gaussian Elimination [57], which apply a sequence of matrix transformations to reach an equivalent but easier-to-solve system, once were the default choice to tackle this sort of problems due to their robustness. Unfortunately, today most of these algorithms fall short when solving large-scale problems because of their excessive memory requirements, impractical time to solution and complexity of implementation.

In these cases, iterative methods pose an appealing alternative. Although direct methods

are also iterative (for example they generally iterate over the rows and columns of the coefficient matrix), this type of algorithms earn their name because they work by iteratively improving an initial guess of the solution. Furthermore, they normally present smaller computational requirements than the direct counterparts.

Although the derivation of each method can be analyzed from different angles, most iterative methods can be interpreted as searching for the solution of the linear system inside a predefined subspace, by taking one step in a given search direction at each iteration of the solver. A large number of methods exist that differ from each other in how this subspace is defined, and the criteria under which this search direction and step size are chosen.

Many times, there are optimal choices for the above criteria, which can be derived analytically. In these cases, we say that the iterative method is optimal, and reaches the solution of the linear system (provided exact arithmetic is used) in at most as many steps as the dimension of the column-span of the matrix. Nevertheless, taking that many steps to arrive to the solution is often as costly as solving the linear system using a direct method. In general, it is desired that the iterative method rapidly improves the initial guess, with the purpose of finding an acceptable solution to the system in a small number of steps.

The process of converging rapidly to an acceptable solution is often impaired in practice by unavoidable rounding errors due to the use of finite-precision arithmetic. In many cases these errors can even prevent the method from reaching a solution at all. How floating point rounding errors affect the iterative method usually depends on the numerical properties of the problem being solved.

To remedy these shortcomings, preconditioning techniques are applied. In a broad sense, these techniques aim to transform the original linear system into an equivalent one that presents better numerical properties, so that an iterative method can be applied and converge faster to a solution. As an example, given $A \in \mathbb{R}^{n \times n}$ and $b \in \mathbb{R}^n$, consider the linear system

$$Ax = b.$$

A (left-)preconditioner for this system can be represented as a matrix M that is close to A^{-1} in some sense. This way, pre-multiplying the system on both sides to obtain

$$\hat{A}x = \hat{b} \equiv MAx = Mb$$

will hopefully be closer to the solution of the system, facilitating the convergence of an iterative solver.

The development of effective preconditioners is an active field of research in applied mathematics. There are many preconditioning techniques but, unfortunately, none of these are effective in all cases. One outstanding class of preconditioners, however, is based on Incomplete LU (ILU) factorizations.

ILU-type preconditioners are based on computing a LU factorization of the matrix where some entries are dropped during the process to maintain the sparsity in the factors, which will determine the memory footprint of the preconditioner, as well as the computational effort required to apply it in the context of an iterative method.

This family of preconditioners are known to achieve good results for important classes

of problems, such as those that arise from the discretization of elliptic PDEs. This property has motivated their massive use, and their inclusion in software packages. However, in order to make this technique applicable in a wider spectrum of problems, an active line of research is devoted to make ILUs more efficient and reliable.

Among the most recent advances in this field, ILUPACK (http://ilupack.tu-bs.de) stands out as a package for the solution of sparse linear systems via Krylov subspace methods that relies on an inverse-based multilevel ILU (incomplete LU) preconditioning technique for general as well as Hermitian positive definite/indefinite linear systems [31]. An outstanding characteristic of ILUPACK is its unique control of the growth in the magnitude of the inverse of the triangular factors during the approximate factorization process.

Unfortunately, the favorable numerical properties of ILUPACK's preconditioner in the context of an iterative solvers come at cost of expensive construction and application procedures, especially for large-scale sparse linear systems. This high computational cost motivated the development of variants of ILUPACK that can efficiently exploit High-Performance Computing (HPC) platforms. Previous efforts include parallel variants of ILUPACK's CG method [88], for symmetric positive definite (SPD) systems, on shared-memory and message-passing platforms [3, 4, 8]. These implementations showed remarkable results regarding their performance and scalability in many large-scale problems, but have the potential disadvantage of slightly modifying the original ILUPACK preconditioner to expose task-level parallelism, yielding distinct convergence rates (though not necessarily slower for the parallel versions). Additionally, the task-parallel variants usually require more floating-point arithmetic operations (flops) than the original ILUPACK, with the overhead cost rapidly growing with the degree of task-parallelism that is exposed [3]. This may imply a higher cost of the preconditioner in terms of storage and energy consumption.

For more than two decades now, scientific and domestic computing platforms have evolved to include multiple cores to enable the parallel execution of different operations, mitigating this way the physical limitations (mainly related with heat dissipation and errors in signal transmissions) that impair the increase in clock frequency and transistor integration dictated by Moore's law [94]. In this sense, hybrid hardware platforms equipped with one or several multi-core processors and compute accelerators have experienced an important evolution. In particular, Graphics Processing Units (GPUs) have developed into extremely powerful parallel architectures that integrate thousands of cores at a reasonable price and energy consumption. For these reasons, GPUs have become a tool of paramount importance in science and engineering, especially after the explosive advances in the field of machine learning and deep neural networks in recent years.

GPUs are now ubiquitous, making their efficient use crucial in order to obtain good performance on the most recent hardware for scientific computing. Even in sparse linear algebra where the computational intensity of the operations is generally low and does not allow to take full advantage of the computational power GPUs provide, the high memory bandwidth of these devices offers significant acceleration opportunities.

Previous to this dissertation, no parallel versions of ILUPACK existed aside the above-mentioned task-parallel implementations. It is therefore interesting to analyze the data-level parallelism in ILUPACK to develop new efficient parallel versions that do not suffer from

the limitations of the previous variants on the one hand, and to enhance the performance of these variants on the other. In this line, the use of hardware accelerators, and in particular of GPUs, is a valuable tool, and a clear path to explore.

1.1 Graphics Processing Units

The term GPU (Graphics Processing Unit) was popularized by NVIDIA in 1999, which advertised the GeForce 256 as "the world's first GPU" [77]. At that time, these devices were dedicated to efficiently process polygons, lighting and textures in order to display 3D images on the screen. As the processing of the geometry and the production of complex lighting and shading effects are the computationally most expensive stages of this graphics pipeline, the first GPU-architectures [83] included specialized hardware to compute them.

The operations available at these stages were configurable but not programmable. For instance, in the fixed-function pipeline, the programmer was able to control the position and color of the vertices and the point source of light, but not the lighting model that determined their interaction. For this reason, the first attempts to solving general-purpose problems with the GPUs implied a mapping of the solutions to the operations of the fixed graphics pipeline. Furthermore, this architecture presents a serious problem, since working with scenes that present complex geometries generates an overload on the vertex shader while under-utilizing the hardware of the pixel shader, while simple geometries with heavy lighting and texture effects overload the pixel shader and under-utilize the vertex shader. To overcome this issue, around 2007 NVIDIA proposed a new architecture of GPUs that replaced the vertex and pixel shaders by a generic stream processing unit capable of executing every step of the graphic pipeline. In conjunction with the hardware, NVIDIA introduced a programming framework for these devices, which allowed to use this new kind of multiprocessor to solve problems that were not related to rendering graphics in a more straightforward way, which was coined as General Purpose Computing on GPU (GPGPU). The conjunction of the hardware architecture with general purpose processors, the driver, a runtime, and C language extensions to program the GPU were released under the name of CUDA, which stands for Compute Unified Device Architecture.

More recently, the OpenCL standard (defined by the Khronos Group) has become broadly supported, although the CUDA community presents a more advanced adoption. Nowadays, the CUDA platform is designed to work with programming languages such as C, C++ and Fortran, including a great number of high performance libraries, application programming interfaces (APIs) and tools, which makes it easier for specialists in parallel programming to use GPU resources. Moreover, CUDA allows a remarkable portability, as most CUDA codes are backward compatible with most previous GPU architectures, and the same code can be executed in a large variety of devices, which range from mobile devices to HPC platforms, with very few or any adjustments.

A more detailed description of the particularities of the different NVIDIA GPU architectures employed in this work can be found in Appendix C.

1.2 Objectives

Motivated by the need of robust and efficient preconditioners to enhance the convergence of iterative linear system solvers in science and engineering applications, the main goal of this thesis is to advance the state-of-the-art in the efficient parallel implementations of modern preconditioning techniques. In particular, we are interested in the use of hardware accelerators to leverage the data-parallelism of iterative solvers working together with advanced incomplete-factorization methods.

As exposed in the opening paragraphs of this dissertation, ILUPACK is a prominent example of such solvers, but its remarkable numerical results come at the expense of a high complexity, which implies costly construction and application procedures. Previous efforts have provided task-parallel implementations of ILUPACK for shared-memory and message-passing platforms [8, 3, 4] but, despite showing good performance and scalability results, some shortcomings of these parallel variants exist.

In the first place, these variants of ILUPACK are limited to the solution of symmetric and positive-definite (SPD) linear systems, which means that there are no parallel versions of ILUPACK for non-symmetric or indefinite systems previous to this dissertation. Second, they slightly modify the preconditioner to exploit task-parallelism. This means that these modifications can impact the numerical properties and convergence of the preconditioner.

Therefore, to achieve our principal goal we have set the following specific objectives:

- **Enable the use of the GPU to accelerate ILUPACK's multilevel-preconditioner.**
  We introduce GPU computations to harness data-parallelism in the operations that compose the application of ILUPACK's preconditioner for different matrix types. Specifically, we provide implementations of the CG method for SPD systems, GMRES and BiCG for non-symmetric systems, and SQMR for symmetric indefinite systems [89]. Our new solvers preserve the number of floating-point operations and, in general, present the same accuracy and convergence rate of the original routines, within variations due to the use of finite precision.
- **Evaluation and improvement of the new GPU-aware solvers.**
  We evaluate the new data-parallel variants and identify the principal factors that limit their performance. We then deal with these bottlenecks through the design and implementation of enhanced variants. In the case of GMRES we develop an accelerated GPU-version of the modified-Gram Schmidt Re-Orthogonalization procedure (MGSRO) that amortizes the cost of host-device communication. In the case of BiCG, we leverage the task-parallelism intrinsic to the method to perform operations involving matrix A, such as SpMVs or the application of the preconditioner, with similar operations involving A^T. We develop versions of ILUPACK's BiCG for servers equipped with at least two GPUs, and extend these ideas to efficiently exploit single GPU systems.
- **Extend ILUPACK by adding new accelerated solvers.**
  We augment the family of Krylov subspace iterative methods for sparse general linear systems accelerated with an ILUPACK preconditioner with a data-parallel version of

the Bi-Conjugate Gradient Stabilized Method (BiCGStab). To accomplish this, we first develop a CPU version of the solver to then produce a variant that runs entirely on the graphics accelerator. For this reason, in the new solver we depart from the reverse-communication scheme followed by our existing ILUPACK implementations of GMRES and BiCG.

- **Enhance the task-parallel versions of ILUPACK by enabling the exploitation of data-parallelism.**
  We use the GPU to accelerate operations corresponding to the application of ILUPACK preconditioner within the task-parallel shared-memory and distributed-memory variants of this software package. We design a strategy that overcomes the reduction of data-parallelism caused by the partitioning of the workload into several tasks.

1.3 Structure of the document

The remainder of the document is organized in four chapters and three appendices. Chapter 2 provides the theoretical background and previous concepts that supports the rest of the dissertation. Specifically, it reviews some of the most important methods to solve sparse systems of linear equations, making special emphasis on iterative Krylov subspace methods. It then presents several widely-used preconditioning techniques, with particular attention to incomplete factorization methods, to later elaborate on the latest advances in this field. This is followed by an overview of ILUPACK, highlighting the implementation of its preconditioner, and the description of its task-parallel versions. The chapter is closed with a revision of related work, including some of the best known parallel software packages to solve sparse linear systems.

Chapter 3 presents our efforts to exploit the data-parallelism of ILUPACK using GPUs. The parallel variants presented in this chapter aim to accelerate ILUPACK without compromising its mathematical properties. The chapter starts with an analysis of the operations that compose the application process of the preconditioner and the elaboration of a general parallelization strategy. This strategy is employed to produce GPU-aware data-parallel versions of the preconditioner and solvers in ILUPACK for SPD, general (non-symmetric) and symmetric indefinite linear systems. This is followed by a numerical assessment of these baseline versions. The chapter also describes the extensions and enhancements that result from the previous evaluation. The first of these improvements focuses on leveraging the coarse-grain parallelism implicit in the Bi-Conjugate Gradient (BiCG) solver to take advantage of platforms with two hardware accelerators. This strategy is extended to provide an efficient version of the BiCG solver for single-GPU platforms. The second one aims to improve the performance of the Generalized Minimal Residual (GMRES) solver by offloading the expensive Grahm-Schmidt re-orthogonalization to the GPU. Finally, the family of solvers in ILUPACK is enlarged by adding an implementation of the Bi-Conjugate Gradient Stabilized (BiCGStab) solver. The chapter is closed with the experimental evaluation of the advanced variants.

Chapter 4 studies how to leverage the GPU to enhance the performance of the previous task-parallel variants of ILUPACK. Particularly, it commences by analyzing the combination

of task and data parallelism in shared-memory platforms equipped with several GPUs, to later leverage the computational power of GPUs distributed across several nodes of a cluster.

The main part of the dissertation concludes with Chapter 5, which offers some final remarks and the discussion of future directions in which this work can be extended.

Appendix A details about a line of research that we developed in parallel with the thesis, devoted to the study of a new approach for the solution of triangular linear systems in GPUs, and the construction of new routines to perform this operation following this strategy. Some of the resulting routines were used in the single-GPU version of BiCG of Chapter 3.

Another secondary line of research is related to study the energy efficiency of hardware and software. Some work on this line involving the energy aspects of ILUPACK are covered in Appendix B.

Finally, Appendix C gives details on the software and hardware platforms used in the experiments included in the dissertation.

CHAPTER 2

Systems of Linear Equations

This chapter is devoted to provide a shallow theoretical background that enables the reader to continue comfortably through the rest of the manuscript. It is not our purpose to go deep into mathematical details when they are not essential to the understanding of this thesis, and we will refer to the corresponding books and articles in those cases. The chapter starts by revisiting the solution of linear systems by Gaussian Elimination, making a special emphasis on the sparse case and direct methods such as the sparse LU decomposition. It will later switch to the solution of these problems via iterative methods, presenting the most common techniques. A section about preconditioners will follow, in which, after presenting some general concepts, we will focus on the ILU-based class, presenting the most important developments in this subject until arriving to the state-of-the-art "inverse based" multilevel ILU preconditioner which gives birth to ILUPACK, the software package on which the work of this thesis develops. The chapter finishes by a review of related work.

2.1 Direct methods

In this section, we revisit different approaches to compute the solution of a linear system $Ax = b$ with $A \in \mathbb{R}^{n \times n}$ by means of the so-called direct methods. This family of algorithms, which compute the exact solution of the problem (neglecting unavoidable floating-point rounding errors), are based on transforming the original problem so that is easier or even trivial to solve. For this reason, these methods usually involve some sort of matrix factorization. Specifically, many of these algorithms are derived from the well-known Gaussian Elimination procedure, which can be interpreted as the factorization of the matrix A into the product of a unit lower-triangular matrix L multiplied by an upper-triangular matrix U, which is known as the LU decomposition [57].

From the computational point of view, direct methods for dense matrices are characterized by having their memory requirements and operation counts completely determined

beforehand. Furthermore these methods and are, in general, robust from the numerical perspective, often with the help of pivoting techniques. Specifically, they generally present a computational cost of $O(n^3)$ flops (floating-point operations), and techniques to improve their *cache* efficiency as well as as their numerical stability are well-known since long ago.

For the sparse case, the performance of these methods is usually determined by the appearance of new nonzero elements during the transformation procedure, known as *fill-in*, which increases its cost and makes difficult the task of calculating it *a priori*. Therefore, the design of direct methods for sparse matrices is usually guided by the mitigation of this phenomenon, which is attempted by applying techniques such as re-orderings of the rows and columns of the sparse matrix, altering the order in which unknowns are eliminated, and the partitioning of the system, to identify dense submatrices that can be dealt with computationally efficient dense algebra kernels. However, it is not always possible to reduce the fill-in to satisfactory levels, especially in the non-SPDcases, where pivoting is required, which can lead to difficulties when solving reasonably large systems of equations.

Although this thesis is mostly concerned with the solution of large and sparse linear systems, which is precisely where direct methods fall short, some introduction about these algorithms is mandatory since, apart from their role in solving linear systems, they provide the framework for the development of several types of preconditioners.

2.1.1 Gaussian Elimination

In [98], Stewart referred to Gaussian Elimination (GE) as the most versatile of all matrix computations, as well as *the* algorithm that should be saved in case of a catastrophe menacing to destroy all numerical linear algebra methods. A proof of this versatility is that the method can be analyzed from various perspectives. The first approach that is usually taught in schools is that of GE as an algorithm to eliminate variables in a system of linear equations. This elimination process is then seen as the successive application of row operations to the coefficient matrix of the system. This, in turn, leads to the interpretation of the method as transforming a matrix into triangular form by multiplying it by a series of elementary lower triangular matrices. Finally, one arrives to regarding GE as the algorithm that performs the factorization of a matrix into a unit-lower triangular matrix L and an upper triangular matrix U, that is, the *LU decomposition*. Furthermore, when sparse matrices are considered, it is often useful to view GE as an algorithm over the adjacency graph of the matrix [1] .

Regarding its usefulness, the factorization $A = LU$ allows to solve the linear system $Ax = LUx = b$ by solving the triangular linear systems $Ly = b$ followed by $Ux = y$, which can be performed using simple forward and backward substitution. In turn, the linear system $A^T x = b$ can be solved by solving $U^T y = b$ followed by $L^T x = y$. As the computational effort required to factorize the matrix is much larger than that required to solve the two triangular linear systems, this decomposition is especially convenient when several systems involving the same coefficient matrix must be solved. In this case, the factors L and U can be reused while the matrix decomposition needs to be computed only once.

[1] This approach is specially useful for the development of matrix ordering techniques to reduce the fill-in of the matrix during the factorization. The theory behind these algorithms is often expressed in terms of the adjacency graph determined by the sparse matrix.

Apart from this good properties, variants of GE lead to other important algorithms, as the QR factorization if the elementary transformations to reach the triangular form are replaced by Householder reflections. It can also easily be adapted to exploit matrices of special structure, like in the Thomas algorithm for tridiagonal matrices [101], or in the factorization of band and Hessenberg matrices. Moreover, in the sparse case, GE also gives place to an entire family of preconditioners for the iterative solution of linear systems.

GE applies successive elementary transformations to cancel the elements below the diagonal of each column, one column at a time. Concretely, starting from matrix $A_0 = A$, and letting $A_{k-1} \in \mathbb{R}^{n\times n}$ be the result of applying $k-1$ of these elementary transformations, the result of the k-th transformation can be expressed as follows

$$A_k = L_k^{-1} A_{k-1}, \tag{2.1}$$

with

$$L_k^{-1} = I - \frac{1}{a_{kk}^{(k-1)}} \begin{pmatrix} 0 \\ a_{*k}^{(k-1)} \end{pmatrix} e_k^T \tag{2.2}$$

where e_k^T is the k-th canonical vector and where

$$a_{*k}^{(k-1)} = \left(a_{k+1,k}^{(k-1)}, \ldots, a_{n,k}^{(k-1)} \right)^T \tag{2.3}$$

is the $\mathbb{R}^{n-k}$ vector formed by the sub-diagonal elements in the k-th column of A_{k-1}.

It should be noted that, with L_k^{-1} defined as in (2.2), the new entry $a_{ik}^{(k)}$ (with $i > k$) will be calculated as

$$\begin{aligned} a_{ik}^{(k)} &= l_{ik}^{-1} a_{kk}^{(k-1)} + a_{ik}^{(k-1)} \\ &= -\tfrac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}} a_{kk}^{(k-1)} + a_{ik}^{(k-1)} \\ &= 0 \end{aligned} \tag{2.4}$$

and that

$$U = L_{n-1}^{-1} \times \cdots \times L_1^{-1} \times L_0^{-1} A \tag{2.5}$$

is upper triangular. Moreover,

$$L_k = I + \frac{1}{a_{kk}^{(k-1)}} \begin{pmatrix} 0 \\ a_{*k}^{(k-1)} \end{pmatrix} e_k^T, \tag{2.6}$$

therefore, $L = L_0 \times \cdots \times Ln - 1$ is unit lower triangular, yielding the desired decomposition

$$A = LU. \tag{2.7}$$

The expressions (2.2) and (2.3) allow to formulate an algorithm for GE known as the *right-looking* LU factorization, or KIJ variant, which is outlined in Algorithm 1. In this variant of GE, one column of L and one row of U are computed in each iteration, and the trailing $(n-k)\times(n-k)$ submatrix is modified using a rank-1 update.

It is evident that Algorithm 1 breaks down if $a_{kk}^{(k-1)}$ is zero or numerical difficulties arise in case it becomes very small. Hence, the LU factorization requires of a row pivoting strategy

Algorithm 1 Right-looking or KIJ variant of Gaussian Elimination.

Input: $A \in \mathbb{R}^{n \times n}$
Output: $L, U \in \mathbb{R}^{n \times n}$ triangular such as $A = LU$

```
1:  for k = 0 to n − 1 do
2:    for i = k + 1 to n − 1 do
3:      l_ik := a_ik / a_kk
4:    end for
5:    for j = k + 1 to n − 1 do
6:      u_kj := a_kj
7:    end for
8:    for i = k + 1 to n − 1 do
9:      for j = k + 1 to n − 1 do
10:       a_ij := a_ij − l_ik u_kj
11:     end for
12:   end for
13: end for
```

to ensure numerical stability [57]. In this case, the decomposition takes the form $PA = LU$, where $P \in \mathbb{R}^{n \times n}$ is a permutation matrix. In most implementations, the L and U factors are stored in the memory space of A (i.e. in-place strategy) to reduce the memory requirements, the main diagonal of L does not need to be stored since L is unit lower triangular, and P is implicitly stored as a permutation vector. We do not include pivoting in the outline of the algorithms for clarity sake.

2.1.2 The Sparse LU and the problem of fill-in

Assume that Algorithm 1 is applied to a sparse matrix. In this case, at each iteration k of the outer loop, the procedure will encounter the "difficulty" of having to update the lower $n - k$ rows of the matrix.

To explain why this is especially inconvenient, let us introduce the concept of *fill-in*. By looking at line 10 of Algorithm 1, one should note that, in the case A is sparse, most entries a_{ij} will be zero before this step and, if l_{ik} and u_{kj} are nonzero, a new nonzero will appear in a_{ij} after it. In this case, we can call a_{ik} and a_{kj} the "generating" entries of this nonzero.

This is quite a problem, as it implies that the storage and flop costs of the sparse LU factorization is not determined beforehand, and the harmful effects of this phenomenon will depend greatly on how the non-zeros are distributed among the sparse matrix, that is, its sparsity or nonzero pattern.

Now it is clear that updating the elements that lie in the lower $(n - k) \times (n - k)$ square of the matrix will imply writing fill-in elements in the vector that holds the $(n - k)$ last rows of the sparse matrix. A more adequate variant of the procedure is obtained by permuting the order of the loops in Algorithm 1, which leads to Algorithm 2, known as the IKJ or row-wise factorization.

The inner loop of the IKJ variant produces, at step i of the outer loop, the i-th row of L and the i-th row of U simultaneously. At step i, the previous $i - 1$ rows are accessed but not modified, while the $n - i$ lower rows are not accessed nor modified. This allows to handle

Algorithm 2 Row-wise or IKJ variant of Gaussian Elimination.
Input: A
Output: L, U triangular such as $A = LU$

```
1: for i = k + 1 to n − 1 do
2:   for k = 0 to n − 1 do
3:     l_ik := a_ik / a_kk
4:     for j = k + 1 to n − 1 do
5:       a_ij := a_ij − l_ik u_kj
6:     end for
7:   end for
8:   u_{i,i:n} := a_{i,i:n}
9: end for
```

the fill-in in a simpler data structure dedicated to hold only the current row.

Regarding the utilized data structures, the early codes for the LU factorization relied heavily on linked lists, which allow inserting new nonzero elements easily. Linked lists have been abandoned in more modern implementations, as static data structures are often preferred. In particular, a widely used technique consists of allocating an extra space so that these new non-zeros can be inserted, adjusting the structure accordingly as more fill-in is introduced.

In summary, a typical sparse direct solver will present four phases. First, as the effect of fill in will depend on the nonzero pattern, a pre-ordering of the rows and columns of the sparse matrix is applied aiming to obtain a matrix with a more convenient structure. Although the problem of finding the permutation that minimizes fill-in is NP-hard, there are many effective heuristics for this purpose. Second, a symbolic factorization is performed, which produces the factorization without caring for the numerical values. Third, the actual factors L and U are constructed in the numerical factorization phase. Finally, the forward and backward triangular systems are solved by substitution.

Some important references in this field are [53, 39].

2.2 Iterative methods

Gaussian Elimination can only take partial advantage of sparsity in the coefficient matrix. Although there are some exceptions regarding specific classes of matrices like banded or tridiagonal, the problem of fill-in makes it necessary to abandon this technique to solve the general sparse case for large-scale scenarios.

An iterative method for the solution of linear systems is one that, starting from an initial guess x_0 to the solution vector x, is capable of iteratively refining it until (under certain conditions) an approximation x_k that is acceptably close to x is reached, i.e. $\|Ax_k - b\|$ is relatively small for some vector norm. In order to avoid the difficulties that make GE impractical, such approximation must be performed such that the sparsity of the matrix is preserved throughout the process. This is achieved by exploiting a matrix primitive that greatly benefits from sparsity, concretely, the matrix-vector product.

The intuitive concept behind these methods is that one should be able to improve an

approximate solution x_k by finding an appropriate linear combination of b, x_k and Ax_k. If one is to use all the information available in the expression $Ax = b$, it is reasonable to set the initial guess x_0 to some multiple of b, which implies that

$$x_0 \in \text{span}\{b\}. \tag{2.8}$$

Then, following the previous idea, we should replace the current solution x_0 with a linear combination of itself with Ax_0. In this case the current solution x_1 belongs to the subspace $\text{span}\{b, Ab\}$. After k iterations, it is clear that the current iterate will belong to

$$\mathcal{K}_k(A, b) = \text{span}\{b, Ab, A^2b, \ldots, A^{k-1}b\}. \tag{2.9}$$

This is called the Krylov subspace of dimension k for A and b.

The challenge of this sort of procedure lies on how to construct these linear combinations such that an acceptable approximation is reached in just a few iterations, and this is what gives place to a myriad of different methods. In fact, although there are optimal methods that are able to find the actual solution of the linear system in exact arithmetic, the convergence of the iterations will be determined by the spectral properties of the matrix, and the use of finite precision arithmetic can complicate things even further.

Fortunately, when the properties of the matrix A are not perfect, one can still multiply the coefficient matrix by a certain matrix M^{-1} and solve

$$M^{-1}Ax = M^{-1}b \tag{2.10}$$

instead of $Ax = b$. Now the approximate solution x_k will belong to the subspace $\mathcal{K}_k(M^{-1}A, x_0)$. Matrix M is known as a preconditioner as its purpose is to improve the "condition number" of the iteration matrix, a number closely related with the way a matrix amplifies the numerical errors due to finite precision when, for instance, performing a matrix-vector product. Broadly speaking, M^{-1} is chosen so that it resembles A^{-1} in some way, but this choice is nothing but trivial and is the subject of active research. Moreover, if inverting M or solving a linear system with M is required, such inversion or system should be easier to solve than the original one, or the whole exercise would be pointless.

In the remainder of the chapter we will go over the derivation of some of the most widely used iterative methods. Once again, it is not the purpose of this document to go deep into the mathematical analysis of the methods but to offer a shallow introduction, since this is enough to understand the rest of the thesis and the interested reader can refer to the vast material on the subject by Greenbaum [59], Saad [89], Meurant [46] or Stewart [98] to name a few only. Later, we will focus on preconditioners and present some advances in a particular class, called incomplete factorizations, which motivate this work.

2.2.1 Fixed-point iterations

The simplest form of iterative method to solve a linear system $Ax = b$ are the fixed-point iterations, also known as relaxation or splitting methods. Given an approximate solution x_k,

these methods aim to improve it by modifying a single or a few components of the current solution at each iteration such that one or more components of the residual $r_{k+1} = b - Ax_{k+1}$ approach zero.

Consider, for example, the Jacobi iteration. Zeroing out a component i of the residual imposes

$$(b - Ax)^{(i)} = 0, \tag{2.11}$$

where $y^{(i)}$ denotes the i-th component of vector y. Expanding the matrix-vector product yields

$$b^{(i)} = \sum_{j=0, j\neq i}^{n} a_{ij}x^{(j)} + a_{ii}x^{(i)}, \tag{2.12}$$

which motivates the following iteration,

$$x_{k+1}^{(i)} = \frac{1}{a_{ii}}(b^{(i)} - \sum_{j=0, j\neq i}^{n} a_{ij}x_k^{(j)}). \tag{2.13}$$

Here each component is updated independently, using information from the previous step, and therefore the updates can be performed in parallel. If expressed in matrix form, Equation (2.13) becomes

$$x_{k+1} = D^{-1}(b - (L + U)x_k), \tag{2.14}$$

where D is a diagonal matrix whose diagonal is equal to that of A, and L and U are the strict lower and upper triangles of A. Adding and subtracting x_k from the previous expression obtains

$$\begin{aligned} x_{k+1} &= D^{-1}(b - (L + U)x_k) + x_k - x_k \\ &= x_k + D^{-1}(b - (L + U + D)x_k) \\ &= x_k + D^{-1}(b - Ax_k). \end{aligned} \tag{2.15}$$

Similar derivations can be done for the Gauss-Seidel and SOR methods. The main difference between the Jacobi and Gauss-Seidel methods is that, while in the former the updates the components of the current iterate x_k are performed independently, in the latter they are performed in ascending or descending numerical order, using the updated values to calculate the following ones. This is equivalent to solving a triangular system in each iteration. The SOR method is in turn similar to Gauss-Seidel but incorporates a relaxation parameter ω to the diagonal.

In general, if x_k is an approximate solution to the linear system, we can express all the previous methods as an iteration of the form

$$x_{k+1} = x_k + M^{-1}(b - Ax_k). \tag{2.16}$$

Equation (2.16) is sometimes referred as "simple iteration" [59], and can be interpreted as correcting the current iterate by adding a transformation of the current residual. It is easy to see that, if M is taken to be the diagonal of A, the method is the Jacobi iteration. If, instead, M is equal to the lower triangle of the matrix A, the iteration is named as the Gauss-Seidel method. Finally, for $M = \omega^{-1}D - L$, where D is the diagonal of A, L is

the strict lower triangle, and ω is a relaxation parameter, the iteration is the Successive Over-Relaxation (SOR).

Rearranging Equation (2.16) as

$$x_{k+1} = M^{-1}b + (I - M^{-1}A)x_k = c + \hat{B}x_k. \tag{2.17}$$

and taking $x_0 = 0$, it follows that

$$x_{k+1} \in x_0 + \text{span}\{c, \hat{B}c, \hat{B}^2c, \ldots, \hat{B}^{k-1}c\} = \mathcal{K}_k, \tag{2.18}$$

which is the Krylov subspace spanned by c and $\hat{B}$.

It is relatively easy to prove that a necessary and sufficient condition for the convergence of iteration (2.16) to the solution $x = A^{-1}b$ of the linear system (assuming exact arithmetic) is that the spectral radius $\rho(\hat{B})$ is smaller than 1. In practise, the convergence of these simple iterative methods can be very slow (or never attained) due to floating point rounding errors, especially for ill-conditioned problems. In consequence, it is rare to utilize these methods to solve large scale linear systems. Nevertheless, these simple iterations are computationally cheap and they can be combined with more advanced methods and result quite useful in some contexts.

2.2.2 Krylov subspace solvers

Considering the previous discussion, it is reasonable to ask if there is a better way of choosing the iterates x_k from the Krylov subspace generated by c and $\hat{B}$.

Let $\mathcal{K}_k$ be the subspace for the k-th iterate, where $k-1$ is its dimension. In order to find an approximate solution, one starts from an initial guess which, of course, belongs to $\mathcal{K}_0$. Then, successive attempts to improve this initial solution should be able to expand this subspace in dimension, and some constraints should be imposed on them so that they result in good approximations to the solution of the system. One way of describing these constraints is as orthogonality conditions. In other words, a typical means of finding good approximations is forcing the residual vector $b - Ax$ to be orthogonal to k linearly independent vectors. This set of vectors form a basis to another subspace $\mathcal{L}$, of dimension k, which is called the subspace of constraints or left subspace [89]. This general idea, known as the Petrov-Galerkin conditions, is common to all the solvers that will be presented next.

Considering again the iteration (2.16), one can say that it is static in the sense that the information that will be employed to compute an approximate solution x_{k+1} is the same regardless of the iteration. An attempt to improve this algorithm would be to introduce a dynamically calculated parameter α_k in order to optimize how far away from the current solution the next one should be.

Without loosing generality, assume that the system $Ax = b$ denotes now the preconditioned linear system of (2.10). Then, introducing this new parameter yields

$$x_{k+1} = x_k + \alpha_k(b - Ax_k). \tag{2.19}$$

In order to determine the value of α_k, it should be noted that the residual follows the relation

$$r_{k+1} = r_k - \alpha_k A r_k. \tag{2.20}$$

Thus, it is natural to choose α_k so that r_{k+1} is minimized. Then, it is easy to prove that this happens when the vectors $r_k - \alpha_k A r_k$ and $\alpha_k A r_k$ are orthogonal, which yields

$$\alpha_k = \frac{\langle r_k, r_k \rangle}{\langle A r_k, r_k \rangle}. \tag{2.21}$$

With this choice of α_k, the expression in (2.20) can be seen as subtracting from r_k its orthogonal projection onto the direction Ar_k.

When the matrix A is Hermitian and positive definite, the above choice of α_k combined with formula (2.19) gives place to the so called *method of steepest descent*. This method belongs to the family of *gradient descent* methods, and earns its name from looking at the linear system $Ax = b$ as the optimization problem of finding the solution x that minimizes

$$f(x) = x^T A x - 2x^T b, \tag{2.22}$$

which is a quadratic form that determines a paraboloid with $x = A^{-1}b$ as its minimum. For a given point x_k, the vector $-\nabla_x f(x_k) = b - Ax_k$ points in the direction where f decreases more quickly, i.e. its direction of *steepest descent*. Hence, the method can be viewed as iteratively taking a step of size α_k in the direction of steepest descent so that the next residual is minimized. In general, one could write the previous method as

$$x_{k+1} = x_k + \alpha_k p_k, \tag{2.23}$$

where $p_k = r_k$ and $\alpha_k = \frac{\langle r_k, r_k \rangle}{\langle A r_k, r_k \rangle}$, and devise different iterations by replacing the direction p_k and the step size α_k by other expressions.

Method of the Conjugate Gradients (CG)

Although the *steepest descent* algorithm chooses r_{k+1} so that it is orthogonal to r_k, it is not necessarily orthogonal to the other $k-1$ residuals or, equivalently, the error $e_{k+1} = x_{x+1} - x$ is not A-orthogonal (orthogonal with respect to the inner product $\langle u, v \rangle_A = \langle u, Av \rangle$) to p_{k-1}. This causes the method to often take steps in previously explored directions. This form of approaching x is, of course, not optimal.

To improve this strategy, one approach is to look for n linearly independent search directions $\{p_0, \ldots, p_n\}$, and take one step in each one so that, in step n, the exact solution x is reached. A convenient choice is to require the search directions p_k to be A-orthogonal with each other. This way if, like in *steepest descent*, one also requires e_{k+1} to be A-orthogonal to p_k, the following expression for the step α_k can be derived

$$\alpha_k = \frac{\langle p_k, r_k \rangle}{\langle p_k, A p_k \rangle}. \tag{2.24}$$

This is referred in the seminal article by Hestenes and Stiefel [63] as the *method of conjugate*

directions.

The description of the method of conjugate directions is not complete in the sense that it does not define how the set of conjugate (or A-orthogonal) vectors $\{p_0, \ldots, p_n\}$ should be computed. There is, in fact, more than one way of finding such vectors, and some of these strategies characterize particular methods themselves.

As the Gram-Schmidt procedure can be utilized to form an orthonormal basis of a given subspace, a variation of this algorithm, called *conjugate Gram-Schmidt*, is one straightforward approach to form the set of conjugate directions. Starting from an initial set of linearly independent vectors $\{u_0, \ldots, u_n\}$, and setting $p_0 = u_0$, the rest of the A-orthogonal directions p_k are calculated as follows

$$p_k = u_k - \sum_{j=0}^{k-1} \frac{\langle u_k, Ap_j \rangle}{\langle p_j, Ap_j \rangle}. \tag{2.25}$$

This simply means taking a vector u_k and subtracting each component of such vector that is not A-orthogonal to the $k-1$ conjugate directions previously computed.

The main disadvantages of this procedure are that, in order to compute the direction p_k, the other $k-1$ directions have to be kept in memory, and it takes $O(n^3)$ operations to compute the whole set. In fact, when the vectors $\{u_0, \ldots, u_n\}$ are taken to be the columns of the identity matrix of size $n \times n$, this process is equivalent to Gaussian Elimination.

The method of *conjugate gradients* [63] (CG) is a variant of the method of conjugate directions that elegantly eliminates the problems of the conjugate Gram-Schmidt procedure by taking the vectors $\{u_0, \ldots, u_n\}$ that are A-orthogonal to the residuals $\{r_0, \ldots, r_n\}$. Using that the residual r_k is orthogonal to all other $k-1$ previous residuals, and that they follow the recurrence

$$r_{i+1} = r_k + \alpha_k A p_k, \tag{2.26}$$

it can be derived from (2.25) that the recurrence that determines the search directions is given by

$$p_{i+1} = r_k + \frac{\langle r_k, r_k \rangle}{\langle r_{k-1}, r_{k-1} \rangle} p_k. \tag{2.27}$$

As it is a variant of the conjugate directions method, the CG method finds the exact solution of the linear system in, at most, n iterations. Moreover, it can be shown that, in a given step k, the method minimizes the A-norm of the error e_k over the affine subspace spanned by the search directions $e_0 + \{p_0, \ldots, p_k\}$. In other words, the error is as small as possible (under the A-norm) given the search direction visited so far. A proof of this result can be found in [59].

Putting it all together, one possible formulation of the Conjugate Gradient method is that on Algorithm 3. This formulation includes the application of left-preconditioning with the matrix M. Of course, the matrix-vector product $z_{k+1} := M^{-1} r_{k+1}$ can be replaced by the solution of a linear system.

Algorithm 3 Preconditioned Conjugate Gradient

Input: $A \in \mathbb{R}^{n\times n}, M \in \mathbb{R}^{n\times n}, b \in \mathbb{R}^n$
Output: $x \in \mathbb{R}^n$

$x_0 := 0$
$r_0 := b - Ax_0$
$z_0 := M^{-1}r_0$
$p_0 := z_0$
$\rho_0 := r_0^T z_0$
$\tau_0 :=\| r_0 \|_2$
$k := 0$
while $(\tau_k > \tau_{\max})$ **do**
  $w_k := Ap_k$
  $\alpha_k := \rho_k / p_k^T w_k$
  $x_{k+1} := x_k + \alpha_k p_k$
  $r_{k+1} := r_k - \alpha_k w_k$
  $z_{k+1} := M^{-1} r_{k+1}$
  $\rho_{k+1} := r_{k+1}^T z_{k+1}$
  $\beta_k := \rho_{k+1}/\rho_k$
  $p_{k+1} := z_{k+1} + \beta_k p_k$
  $\tau_{k+1} := \|r_{k+1}\|$
  $k := k + 1$
end while

The Bi-Conjugate Gradients method (BiCG)

When the matrix A is indefinite or non-symmetric, Algorithm 3 will fail in case it finds a direction p_k for which the denominator $\langle p_k, Ap_k\rangle = 0$. In this sense, the Bi-conjugate Gradient (BiCG) algorithm can be considered as a generalization of the CG method designed to avoid this breakdown situation.

The method was first derived by Lanczos [69] in 1952 as a variation of the two-sided Lanczos algorithm to compute the eigenvalues of a non-symmetric matrix A. In a broad sense, it is based on maintaining two parallel recurrences, one for matrix A and the other for A^T, and imposing bi-conjugacy and bi-orthogonality conditions between the vectors of each recurrence. As a result, apart from solving the system $Ax = b$, the algorithm is also able to solve the *dual* linear system $A^T x^* = b$. For simplicity, the formulation of the method given in Algorithm 4 leaves out the solution of this the solution of this *dual* system.

Without going into excessive detail, given two initial vectors r_0 and $\hat{r}_0$, and letting $p_0 = r_0$ and $\hat{p}_0 = \hat{r}_0$, the recurrences of the method for step k are given by

$$r_{k+1} = r_k - \alpha_k A p_k, \tag{2.28}$$

$$\hat{r}_{k+1} = \hat{r}_k - \alpha_k A^T \hat{p}_k, \tag{2.29}$$

with

$$\alpha_k = \frac{\langle \hat{r}_k, r_k\rangle}{\langle \hat{p}_k, Ap_k\rangle}, \tag{2.30}$$

and

$$p_{k+1} = r_k - \beta_k p_k, \tag{2.31}$$

$$\hat{p}_{k+1} = \hat{r}_k - \beta_k \hat{p}_k, \tag{2.32}$$

where

$$\beta_k = \frac{\langle \hat{r}_{k+1}, r_{k+1} \rangle}{\langle \hat{r}_k, r_k \rangle}. \tag{2.33}$$

The choice of the scalar α_k forces the bi-orthogonality conditions

$$\langle \hat{r}_{k+1}, r_k \rangle = \langle r_{k+1}, \hat{r}_{k+1} \rangle = 0, \tag{2.34}$$

while β_k is chosen so that

$$\langle \hat{p}_{k+1}, A p_k \rangle = \langle p_{k+1}, A \hat{p}_k \rangle = 0. \tag{2.35}$$

As in the case of the CG, a nice property of the algorithm is that the above bi-orthogonality and bi-conjugacy conditions hold for any pairs of vectors $(\hat{p}_i, p_j)$ and $(\hat{r}_i, r_j)$ such that $i \neq j$, without explicitly enforcing them. In turn, as the vectors p_{k+1} are nothing more than linear combinations of the vectors r_{k+1} to r_0, the previous orthogonality conditions also imply that

$$\langle \hat{p}_i, r_j \rangle = \langle p_i, \hat{r}_j \rangle = 0. \tag{2.36}$$

A more detailed derivation of the algorithm, as well as proofs of the aforementioned properties can be found in [48].

Algorithm 4 Algorithmic formulation of the preconditioned BiCG method.

Input: $A \in \mathbb{R}^{n \times n}, M \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n$
Output: $x \in \mathbb{R}^n$

Initialize $x_0, r_0, q_0, p_0, s_0, \rho_0, \tau_0; k := 0$
while $(\tau_k > \tau_{\max})$ **do**
  $\alpha_k := \rho_k / (q_k^T A p_k)$
  $x_k := x_k + \alpha_k p_k$
  $r_k := r_k - \alpha_k A p_k$
  $t_k := M^{-1} r_k$
  $z_k := M^{-T} A^T q_k$
  $s_{k+1} := s_k - \alpha_k z_k$
  $\rho_{k+1} := (s_{k+1}^T r_k) / \rho_k$
  $p_{k+1} := t_k + \rho_{k+1} p_k$
  $q_{k+1} := s_{k+1} - \rho_{k+1} q_k$
  $\tau_{k+1} := \| r_k \|_2$
  $k := k + 1$
end while

Generalized Minimal Residual method (GMRES)

The Generalized Minimum Residual Method (GMRES) is a projection method for non-symmetric and indefinite matrices that first constructs an orthogonal basis for the subspace

$\mathcal{K}(A, r_0)$, and then extracts the approximate solution from this subspace so that the norm of the residual is minimized.

In its most common formulation, this orthogonal basis is formed by the modified Gram-Schmidt process that is outlined in Algorithm 5, which is usually referred to as Arnoldi iteration [20].

Algorithm 5 Arnoldi's iteration

Input: $A \in \mathbb{R}^{n\times n}, q_0 \in \mathbb{R}^n$
Output: $\{q_0, \ldots, q_k\}$ orthonormal vectors

1: **for** $j = 0$ **to** k **do**
2: $\quad \tilde{q}_{j+1} := Aq_j$
3: $\quad$ **for** $i = 0$ **to** j **do**
4: $\quad\quad h_{ij} := \langle \tilde{q}_{j+1}, q_i \rangle$
5: $\quad\quad \tilde{q}_{j+1} := \tilde{q}_{j+1} - h_{ij} q_i$
6: $\quad$ **end for**
7: $\quad h_{j+1,j} := \|\tilde{q}_{j+1}\|$
8: $\quad q_{j+1} := \tilde{q}_{j+1}/h_{j+1,j}$
9: **end for**

If Q_k is the matrix with columns that correspond to the k orthogonal Arnoldi vectors q_i, the algorithm can be expressed in matrix form as

$$AQ_k = Q_{k+1} H_{k+1,k}, \tag{2.37}$$

where $H_{k+1,k}$ is a Hessenberg matrix (a matrix with all the elements below the first sub-diagonal set to zero) that contains the h_{ij} coefficients of Algorithm 5.

Once this set of vectors is constructed, GMRES will compute the approximate solution x_k such that is has the form

$$x_k = x_0 + Q_k y_k, \tag{2.38}$$

for some vector y_k. In other words, the iterate x_k will be computed as the initial guess plus some linear combination of the $\{q_0, \ldots, q_{k-1}\}$ Arnoldi vectors.

To obtain the approximation of the form 2.38 that minimizes the 2-norm of the residual $\|r_0 - AQ_k y_k\|$, GMRES solves the following least squares problem

$$\begin{aligned} y_k &= \operatorname{argmin}_y \|r_0 - AQ_{k+1} y\| \\ &= \operatorname{argmin}_y \|r_0 - AQ_k H_{k+1,k} y\| \\ &= \operatorname{argmin}_y \|Q_{k+1}(\beta e_1 - H_{k+1,k} y)\| \\ &= \operatorname{argmin}_y \|\beta e_1 - H_{k+1,k} y\|, \end{aligned} \tag{2.39}$$

where $\beta = \|r_0\|$ and e_1 is the first column of the identity matrix of size $k+1$. The second step of the previous equality takes into account that $Q_{k+1}e_1$, the first column of matrix Q_{k+1}, is equal to $r_0/\|r_0\|$.

The solution to the above least-squares problem can be obtained inexpensively by computing the QR factorization of $H_{k+1,k}$ into a unitary matrix F_k and an upper-triangular matrix R_k. Once this factorization is obtained, the solution y for (2.39) is given by the

linear system

$$\tilde{R}y = \beta(F_k e_1), \tag{2.40}$$

where $\tilde{R}$ denotes the $k \times k$ top-left submatrix of R_k, and $F_k e_1$ are the top k entries of the first column of F. This factorization is inexpensive, as it can be obtained by applying plane rotations in order to annihilate the first sub-diagonal of $H_{k+1,k}$. Furthermore, provided that the QR factorization of $H_{k+1,k}$ is available, it is possible to compute the factorization of $H_{k+2,k+1}$ with little overhead. Hence, the above process can be performed in a progressive manner, and this allows to obtain the residual norm inexpensively, at each step of GMRES, in order to check for convergence. For details we refer to [59, 89].

The GMRES algorithm becomes impractical when k is large, given that it is necessary to store and operate with the whole set of q_i vectors. One commonly-used strategy consists of simply restarting GMRES after m iterations, where m is a user-defined parameter, and use the current iterate as the initial guess for the restart. This is referred to as GMRES(m), which is outlined in Algorithm 6.

Without going into details about the convergence properties of full and restarted GMRES, it is safe to say that small values of m will cause a slow convergence, and that large values of m can lead to considerable runtimes and memory requirements, so a trade-off between these factors must be considered. Moreover, the restarted GMRES algorithm can stagnate, i.e. cease to improve the approximation, when the matrix is not positive definite. This drawback can be, in part, overcome by preconditioning techniques. The references [59, 89] contain convergence analysis of these methods, so the interested reader should consult them for further details.

Algorithm 6 Algorithmic formulation of the preconditioned GMRES (m) method. The threshold $\tau_{\max}$ is an upper bound on the relative residual for the computed approximation to the solution.

Input: $A \in \mathbb{R}^{n\times n}, M \in \mathbb{R}^{n\times n}, b \in \mathbb{R}^n, m$ (integer)
Output: $x \in \mathbb{R}^n$

Initialize $x_0, r_0, q_0, p_0, \ldots$
$k := 0$, $\beta := \|r_0\|$, $\xi_1 := (1, 0, \ldots, 0)^T$
while $(k < m)$ **do**
  $z_{k+1} := M^{-1}(Aq_k)$
  $[H(:, k+1), q_{k+1}] := \text{MGSO}(Q_k, z_{k+1})$
  $k := k + 1$
end while
$y_k := \arg\min_y \|\beta\xi_1 - H_k y\|$
$x_k := x_0 + Q_k y_k$
$\tau_k := \|b - Ax_k\|$
if $\tau_k > \tau_{\max}$ **then**
  $x_0 := x_k$
  restart GMRES
end if

Algorithm 7 Algorithmic formulation of MGSO. The method performs a re-orthogonalization if the cosine between the two vectors is greater than τ, which is set to 0.99 in our case.

Input: $Q_k \in \mathbb{R}^{n\times n}, z_k \in \mathbb{R}^n$
Output: $q_{k+1} \in \mathbb{R}^n, H_{:,k+1} \in \mathbb{R}^n$

Initialize $\omega := \|z_k\|^2, \tau := 0.99, i := 1$
while $(i <= k)$ **do**
  $H(i, k+1) := q_i^T z_k$
  $z_k := z_k - H(i, k+1) q_i$
  if $(|H(i, k+1)|^2 > \omega\tau)$ **then**
    $\beta := q_i^T z_k$
    $H(i, k+1) := H(i, k+1) + \beta$
    $z_k := z_k - \beta q_i$
  end if
  $\omega := \omega - |H(i, k+1)|^2$
  if $(\omega < 0)$ **then**
    $\omega := 0$
  end if
  $i := i + 1$
end while
$q_{k+1} := z_k / \|z_k\|$
$H_{k,k+1} := \|z_k\|$

Simplified symmetric Quasi-Minimal Residual method (SQMR)

Just as GMRES takes the matrix A into a Hessenberg matrix by means of an Arnoldi iteration, QMR applies a double-sided or non-symmetric Lanczos process on the original matrix that transforms it into a tridiagonal (also Hessenberg) matrix. This transformation can be described in matrix form by the following recurrence

$$AV_k = V_{k+1} T_{k+1,k}, \tag{2.41}$$

where V_k are the k Lanczos basis vectors and T_{k+1} are the coefficients of the three-term recurrence of the Lanczos iteration. Once this basis V_k is obtained, the same strategy is applied, and the iterate x_k is taken to be of the form

$$x_k = x_0 + V_k y. \tag{2.42}$$

The main difference with GMRES is that the Lanczos process does not generate an orthonormal basis. By taking $v_0 = r_0 / \|r_0\|$ and performing the same derivation as in GMRES to get y such that the residual $r_0 - AV_{k+1}y$ is minimal, one obtains

$$\begin{aligned} y_k &= \text{argmin}_y \|r_0 - AV_{k+1}y\| \\ &= \text{argmin}_y \|r_0 - AV_k T_{k+1,k} y\| \\ &= \text{argmin}_y \|V_{k+1}(\beta e_1 - T_{k+1,k} y)\| \end{aligned} \tag{2.43}$$

,

where $\beta = \|r_0\|$ and e_1 is the first column of the identity matrix. Unlike in GMRES, it is not possible to eliminate V_{k+1} from (2.43) so that the above least-squares problem can be solved easily. However, it is reasonable to consider that if y minimizes $\|\beta e_1 - T_{k+1,k}y\|$, this will be good enough to converge properly to the solution of the system. Hence, the method earns its qualification of "quasi-minimal" instead of minimal, and $\|\beta e_1 - T_{k+1,k}y\|$ is called the "quasi-residual" norm.

Now, a result due to Freund and Zha [49] states that the Lanczos process can be simplified when one can find a matrix P such that

$$A^T P = PA. \tag{2.44}$$

In this case, the following relation can be established between the right Lanczos vectors w_k and the left Lanczos vectors v_k

$$w_k = \frac{Pv_k}{\|v_k\|}. \tag{2.45}$$

If the matrix-vector product Pv_k is easy to compute, as in the case when P is sparse, this relation allows to obtain the Lanczos basis with almost half the storage and computing effort.

Consider next a linear system with a symmetric indefinite matrix $A \in \mathbb{R}^{n\times n}$ and a preconditioner M such that

$$M = M_L M_R = M_R^T M_L^T = M^T, \tag{2.46}$$

so that the preconditioned system is given by

$$\hat{A}\hat{x} = \hat{b} \tag{2.47}$$

where

$$\hat{A} = M_L^{-1} A M_R^{-1}, \quad \hat{b} = M_L^{-1}, \quad \text{and} f \quad \hat{x} = M_R x \tag{2.48}$$

Here it is not necessary that M_L and M_R are each others transpose, and left or right-preconditioned variants can be obtained by simply setting $M_R = I$ or $M_L = I$, respectively.

Choosing $P = M_L^T M_R^{-1}$ one gets

$$\begin{aligned} \hat{A}^T P &= M_R^{-T} A M_L^{-T} (M_L^T M_R^{-1}) \\ &= (M_L^{-1} M_R^T)^{-1} (M_L^{-1} M_R^T) M_R^{-T} A M_R^{-1} \\ &= M_R^{-T} M_L M_L^{-1} A M_R^{-1} \\ &= M_L^T M_R^{-1} M_L^{-1} A M_R^{-1} \\ &= P\hat{A}, \end{aligned} \tag{2.49}$$

so the simplified symmetric QMR is obtained by applying QMR on the preconditioned system (2.48). The matrix P does not need to be computed explicitly, as matrices M_L and M_R appear in the algorithm only when solving the two linear systems corresponding to the application of the preconditioner M.

One formulation of the method is offered in Algorithm 8. As in the case of GMRES, the

least squares problem is solved by applying plane rotations and solving the linear system, which can be done progressively as the Lanczos vectors and coefficients become available. In the algorithm, the coefficients ν_k and c_k are related with the Givens rotations applied to the last column of $T_{k+1,k}$. The formulation presented is based on that derived in [50], which exploits the close relation between BiCG and SQMR iterates and residuals. In fact, here r_k denotes the residual of BiCG and not the residual obtained by computing $b - Ax_k$ with the current iterate. Moreover, $z_k = M_R^{-1} M_L^{-1} r_k = M_{-1} r_k$ is the preconditioned BiCG residual.

Algorithm 8 Simplified Quasi-Minimal Residual (SQMR).

Input: $A \in \mathbb{R}^{n \times n}, M = M_L M_R \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n$
Output: $x \in \mathbb{R}^n$

Initialize $x_0, r_0 = b - Ax_0, t = M_L^{-1} r_0, \gamma_0 = \|t\|, \ldots$
$v_0 = M_R^{-1} t, \nu_0 = 0, \rho_0 = \langle r_0, v_0 \rangle$
for $k = 0$ **to** $n - 1$ **do**
  $t = (Av_k)$
  $\sigma_k = \langle v_k, t \rangle$
  if $\sigma_k = 0$ **then**
    stop
  end if
  $\alpha_k = \rho_k / \sigma_k$
  $r_{k+1} = r_k - \alpha_k t$
  $t = M_L^{-1} r_{k+1}$
  $\nu_{k+1} = \|t\| / \gamma_k$
  $c_{k+1} = 1/\sqrt{1 + \nu_k^2}$
  $\gamma_{k+1} = \gamma_k \nu_{k+1} c_{k+1}$
  $d_{k+1} = c_{k+1}^2 \nu_k^2 d_k$
  $x_{k+1} = x_k + d_k$
  if x_k has converged **then**
    stop
  end if
  $z_{k+1} = M_R^{-1} t$
  $\rho_{k+1} = \langle r_{k+1}, z_{k+1} \rangle$
  $\beta_{k+1} = \rho_{k+1} / \rho_k$
  $v_{k+1} = z_{k+1} + \beta_{k+1} v_k$
end for

The Bi-Conjugate Gradient Stabilized (BiCGStab)

When A is not symmetric, the simplification of the Lanczos process can not be made, and thus the BiCG and the QMR methods imply recursions with both A and A^T. To avoid using the transpose and to improve the convergence of BiCG at comparable computational cost, Sonneveld introduced the Conjugate Gradient Squared (CGS) algorithm [97]. The method is based in noting that the residual vector at step k can be expressed as

$$r_k = \phi_k(A) r_0, \tag{2.50}$$

where ϕ_k is a certain polynomial of degree k satisfying the constraint $\phi_k(0) = 1$. Similarly, the conjugate-direction polynomial $\pi_k(t)$ is given by

$$p_k = \pi_k(A)r_0, \tag{2.51}$$

in which π_j is a polynomial of degree j. The directions s_k and q_k of Algorithm 4 are defined through the same recurrences as r_k and p_k in which A is replaced by A^T and, as a result,

$$s_k = \phi_j(A^T)s_0, q_k = \pi_j(A^T)q_0. \tag{2.52}$$

Similarly, it can be shown that the scalar α_k is given by

$$\alpha_k = \frac{\langle \phi_k(A)r_0, \phi_k(A^T)s_0 \rangle}{\langle A\pi_k(A)r_0, \pi_k(A^T)s_0 \rangle} = \frac{\langle \phi_k^2(A)r_0, s_0 \rangle}{\langle A\pi_k^2(A)r_0, s_0 \rangle}, \tag{2.53}$$

which indicates that, if it is possible to get a recursion for the vectors $\phi_k^2(A)r_0$ and $\pi_k^2(A)r_0$, then computing α_k and, similarly, β_k causes no problem. Hence, the idea of seeking an algorithm which would give a sequence of iterates whose residual norms r'_k satisfy

$$r'_k = \phi_k^2(A)r_0. \tag{2.54}$$

As CGS is based on squaring the residual polynomial, it is susceptible to the accumulation of rounding errors, which is a disadvantage in comparison to BiCG.

A variant of the CGS algorithm that aims to solve these problems is the Bi-conjugate Gradient Stabilized (BiCGStab) algorithm. The main difference to CGS is that, instead of obtaining a residual vector of the form r'_k defined in (2.54), BiCGStab produces residual vectors of the form

$$r'_k = \psi_k(A)\phi_k(A)r_0, \tag{2.55}$$

where $\psi_k(t)$ is a polynomial that aims to "stabilize" the convergence behavior of the original algorithm. Specifically, $\psi_k(t)$ is defined by

$$\psi_{k+1}(t) = (1 - \omega_k t)\psi_k(t), \tag{2.56}$$

in which the scalar ω_k can be selected such that it achieves a steepest descent step in the residual direction obtained before multiplying the residual vector by $(I - \omega_k A)$. In other words, ω_k is chosen to minimize the 2-norm of the vector $(I - \omega_k A)\psi_k(A)\phi_{k+1}(A)r_0$. Algorithm 9 outlines one of the possible formulations of the procedure. A complete derivation of the algorithm can be found in [59, 89, 104].

2.3 Preconditioners

In general, iterative solvers exhibit a lack of robustness that severely limits its applicability in industrial contexts [89]. Preconditioners were introduced in Section 2.2 as a transformation of the original system into an alternative one that has the same solution, performed with

Algorithm 9 Algorithmic formulation of the preconditioned BiCGStab method.

Input: $A \in \mathbb{R}^{n\times n}, M \in \mathbb{R}^{n\times n}, b \in \mathbb{R}^n$
Output: $x \in \mathbb{R}^n$

Initialize $x_0, r_0, q_0, p_0, \ldots$
$k := 0$, $\beta := \|r_0\|, \xi_1 := (1, 0, \ldots, 0)^T$
while $(\tau_k > \tau_{\max})$ **do**
  $\rho_{k+1} = (\hat{r}_0, r_k)$
  $\beta = (\rho_{k+1}/\rho_k)(\alpha/\omega_k)$
  $p_{k+1} = r_k + \beta(p_k - \omega_k v_k)$
  $v_{k+1} = M^{-1}Ap_{k+1}$
  $\alpha = \rho_{k+1}/(\hat{r}_0, v_{k+1})$
  $s = r_k - \alpha v_{k+1}$
  $t = M^{-1}As$
  $\omega_{k+1} = (t, s)/(t, t)$
  $x_{k+1} = x_k + \alpha p_k - \omega_{k+1}s$
  $r_{k+1} = s - \omega_{k+1}t$
  $\tau_{k+1} :=\| r_{k+1} \|_2$
  $k := k + 1$
end while

the purpose of improving its condition number, which is the rate between the largest and smallest eigenvalue of the coefficient matrix of the system. Roughly speaking, when this number is large, the unavoidable rounding errors due to the use finite precision can prevent iterative methods from converging to an acceptable solution of the system. This modification can be defined in a number of ways, and it is usually represented by a matrix M which, in general, should meet some minimal requirements.

In the basic form of an iterative method introduced in Equation (2.16), one possibility is to apply the transformation to the system before starting the solver. In many cases, this is not convenient since $M^{-1}A$ is much less sparse than A and M, which increments the storage and computation cost of the solver. The alternative is to either perform a matrix-vector product

$$z_k = M^{-1}r_k \tag{2.57}$$

or to solve the linear system

$$Mz_k = r_k \tag{2.58}$$

in each iteration k, where r_k is the residual computed in the previous step. Therefore, the first requirement any preconditioner M should meet is that the linear system (2.58) is easy to solve or, at least, easy compared to the effort of solving $Ax = b$ with a direct method. Of course, M should also be non-singular so that (2.58) has a unique solution.

The second requirement is more complex and it is related with the quality of the preconditioner. It is evident that any of the aforementioned preconditioned iterations converge in one iteration if the preconditioner M is set to be the matrix A itself. It is also evident that this is not practical since the effort would be equivalent to that of computing A^{-1}. However, this leads to the intuitive idea that iterative methods can converge faster if the preconditioner M resembles A in some sense. The exact sense in which M should approximate A

depends on the iterative method to be used and it is not always clear. For example, when we described the fixed-point iterations, it was mentioned that their convergence is related with the largest eigenvalue of $I - M^{-1}A$, so a preconditioner M such that $\rho(I - M^{-1}A) \ll 1$ is desirable in this case. For other Krylov subspace methods, there are results that characterize the effectiveness of the preconditioner in terms of its eigenvalue distribution or that of the preconditioned matrix, but this is much better understood for Hermitian matrices than it is for non-Hermitian or indefinite ones [59].

There are three ways of applying a preconditioner. Left-preconditioning was already presented in Equation (2.10). Similarly, right-preconditioning leads to the following system

$$AM^{-1}u = b, x = M^{-1}u. \tag{2.59}$$

Here, a change of variables $u = Mx$ is made, and the system is solved for the unknown u, so finding the solution x of the original system will imply solving $Mx = u$ at the end. Often, the preconditioner can be expressed in factorized form $M = M_L M_R$, where typically M_L and M_R are triangular matrices. In this case, the preconditioner can be applied as follows

$$M_L^{-1}AM_R^{-1}u = M_L^{-1}b, x = M_R^{-1}u. \tag{2.60}$$

Splitting the preconditioner in such a way is convenient in order to preserve symmetry when the original matrix shows this property.

Strictly speaking, a preconditioner is any form of internal solver that aims to accelerate the convergence of an iterative method. In some cases, preconditioners can be derived from the intrinsic properties of the original physical phenomena that the linear system or PDE is looking to solve. Most times however, preconditioners are constructed departing from the original coefficient matrix, as useful physical properties are rarely available in general. In this sense, preconditioner techniques can range from simply scaling all rows of the linear system to make the diagonal elements equal to one, to a preconditioner M that reflects a complicated mapping involving FFT transforms, integral calculations, and additional linear system solutions [89].

One of the most popular ways of constructing a preconditioner is by means of a factorization of the original matrix A, as the Cholesky or LU factorization. When A is sparse, the triangular factors L an U are, in general, much more dense than the original matrix. This leads to the so-called incomplete factorizations, which are decompositions of the form $A = \tilde{L}\tilde{U} - R$, where $\tilde{L}$ and $\tilde{U}$ are similar to L and U but present a much sparser nonzero structure, and R is the residual or error of the factorization. Their importance and wide range of applications motivates that we dedicate the rest of this section to present some of their most popular variants, as well as the latest advances in order to enhance their robustness.

2.3.1 Incomplete LU-based preconditioners

Let $A = LU$ be the LU factorization of matrix A. Now consider the left-preconditioned system

$$M^{-1}Ax = M^{-1}b \tag{2.61}$$

where $M = LU$. If an iterative method is applied to solve the system using M as preconditioner, each iteration of the solver will imply the solution of two triangular linear systems (one with L and the other with U as coefficient matrices) in order to obtain the preconditioned residual. It should be easy to see that the above is equivalent to solve the system directly using the LU factorization, since any solver would converge immediately after the first residual is evaluated.

In the sparse case, the main drawback of the LU factorization is, as argued earlier, the fill-in generated in the L and U factors during Gaussian Elimination. When the dimension of the matrices is large, the necessary amount of memory required to store the factorization, and the number of floating point operations implied by the triangular system solution, turn this strategy highly impractical. However, it is reasonable to think that not all the fill-in entries are equally important, and that by dropping some of the fill-in values during the factorization process, one can find a factorization $\tilde{L}\tilde{U} \approx A$, where $\tilde{L}$ and $\tilde{U}$ are much sparser than L and U, that still serves as a good preconditioner for the iterative method.

Unfortunately, it is not trivial to determine which fill-in entries are important and should be kept in the L and U factors in order to attain a good preconditioner, and which of them are disposable and will not affect considerably the quality of the approximation. As a consequence, there are many flavours of ILU factorization, which differ mainly in their "dropping policy".

The "dropping policy" is the criteria that is applied during the update of an entry during Gaussian Elimination, to decide if it should be kept in the L or U factors or if it should be replaced by a 0. To make this decision, there are two factors that can be taken into consideration: the position of the nonzero entry and its numerical value.

Positional ILUs discard a nonzero entry generated in a given stage of GE if its position in the sparse matrix lies inside a target zero pattern. This is sometimes referred to as ILU_P. In the simplest case, the target zero pattern P is chosen prior to the factorization, and any fill-in entry that falls inside that pattern is discarded. It is also possible to adapt this pattern as the factorization proceeds using some criteria. In this case, the resulting factors will depend on the order in which the computations are made in the factorization.

ILU0

A paramount example of the positional type of ILU is the ILU factorization with no fill-in, frequently denoted as ILU(0) or ILU0. In this widely-applied incomplete factorization, the $\tilde{L}$ and $\tilde{U}$ factors present the same sparsity pattern as the lower and upper parts of the matrix A respectively.

The ILU0 factorization algorithm can be directly derived from the ILU_P algorithm by simply choosing P to be the zero pattern of A. However, the usefulness of this preconditioner can be better appreciated by analyzing the particular case of systems derived from the

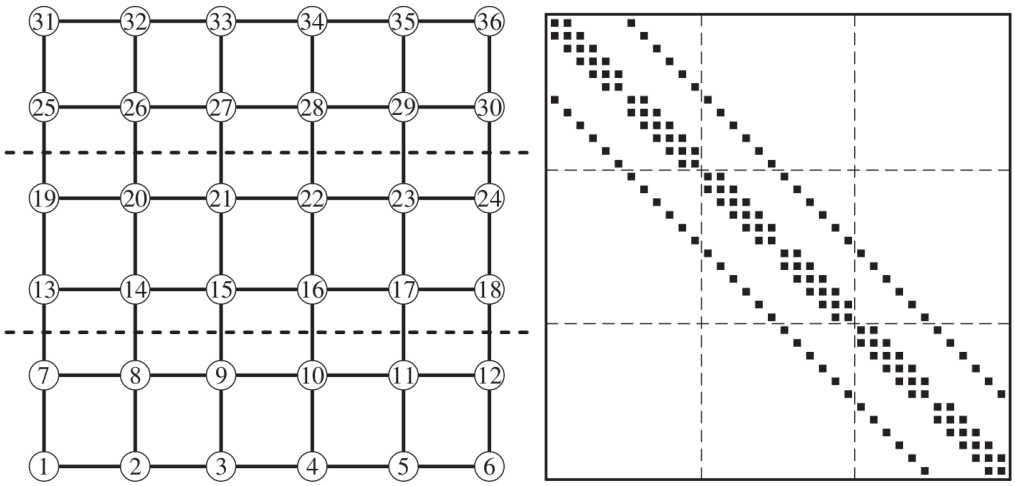


Figure 2.1: Regular 2D grid and sparsity structure of the stiffness matrix. Extracted from [89]

discretization of elliptic PDEs on regular grids. Considering (for simplicity) the case of regular 2D grids, it can be noticed that in these type of problems the stiffness matrix presents a structure that consists of the main diagonal, two adjacent diagonals and two other diagonals that are at a distance from the main one that is equal to the width of the mesh. This is illustrated in Figure 2.1.

In general, the product of any lower and upper triangular matrices $\tilde{L}$ and $\tilde{U}$ such that these two matrices present the same sparsity pattern, respectively, of the lower and upper parts of A, will not have the same sparsity pattern than that of A. Specifically, two diagonals of nonzero entries will appear in the product at the positions $n_x - 1$ and $-n_x + 1$. However, if this fill-in diagonals are ignored, it is possible to find matrices $\tilde{L}$ and $\tilde{U}$ such that

$$\tilde{L}\tilde{U}_{ij} = A_{ij} \quad \text{where} \quad A_{ij} \neq 0. \tag{2.62}$$

The product of those factors would produce a matrix $\tilde{A}$ that differs from A only in these two extra fill-in diagonals, so it can be expected that $\tilde{A}$ reasonably resembles A, and therefore turns to be a good preconditioner [89]. By analyzing the algorithm derived from ILU$_P$, where we replace P by the zero pattern of A, it can be observed that the above property holds. Therefore, the ILU factorizations with no fill-in, or ILU0, can be defined broadly as those factorizations such that the $\tilde{L}$ and $\tilde{U}$ factors have the same sparsity pattern as the lower and upper parts of A, respectively, and such that Equation (2.62) holds. However, it should be noted that this factorization is not unique, since, for example, it does not impose restrictions on the magnitude of the fill-in elements generated by the product.

From the above discussion, it is natural to think that this type of preconditioners are especially well suited to solve elliptic PDEs. In fact, there are cases where, if the stiffness matrix is written as

$$A = L_A + D_A + U_A, \tag{2.63}$$

where L_A, and U_A are, respectively, the strict-lower and upper part of A, and D_A is the

main diagonal, it is possible to find an ILU0 preconditioner M such that

$$M = (L_A + D)D^{-1}(D + U_A). \tag{2.64}$$

Here, D is a diagonal matrix that is derived from the terms L_A and U_A by imposing that M meets the ILU criteria of Equation (2.62). The convenience of this approach is evident in the case of a 5- (or 7)-point stencil, as then it is only necessary to store a single extra diagonal in order to compute the preconditioner, and the entries of this diagonal can be calculated from a simple recurrence. During the iterative solver, the application of the preconditioner to the residual r consists in solving the systems $(L_A + D)y = r$, $D^{-1}z = y$ and $(D + U_A)x = z$, which can also be expressed in terms of simple relations between the coefficients of A and D.

ILU(l)

When moving away from elliptic PDEs and considering unstructured matrices, the ILU0 approach presents higher instability and often does not lead to acceptable preconditioners. An obvious strategy of coping with this limitation is by allowing certain degrees of fill-in in the $\tilde{L}$ and $\tilde{U}$ factors, looking for a trade-off between the accuracy of the preconditioner and the storage requirements to hold the factors, versus the floating point operations required to solve the resulting triangular linear systems.

This alternative leads to a version of ILU_P where the target pattern is either updated during the GE process, or is determined prior to the numerical factorization by a symbolic factorization phase. In this variant, a "level of fill" is assigned to each position of the sparse matrix so that, an element is dropped if its level of fill is greater than a predefined threshold l.

The following definition and explanation of the level of fill will take into account the row-wise version of Gaussian Elimination. Nevertheless, all the concepts can be adapted to the remaining variants of GE.

Recall Section 2.1.2 where the problem of fill-in was introduced. There we characterized a fill-in entry a_{ij}, created during the inner loop of the row-wise or IKJ variant of GE, as being generated by other two "generating entries" a_{ik} and a_{kj}. Most of the times, it is reasonable to expect that these new non-zeros will be smaller in magnitude than their generating entries.

To justify this assumption, a simple model is frequently used in literature [89], in which a size $\delta_k < 1$ is assigned to any element whose level of fill is k. The update of the size of an element during GE can be derived directly from line 5 of Algorithm 2, and follows

$$size(a_{ij}) = \delta^{lev_{ij}} - \delta^{lev_{ik}} \times \delta^{lev_{kj}} = \delta^{lev_{ij}} - \delta^{lev_{ik}+lev_{kj}}. \tag{2.65}$$

The updated size of the entry will then depend on the relation between the exponents lev_{ij} and $lev_{ik} + lev_{kj}$ but, as $\delta < 1$, the model assumes that this size will be close to either $\delta^{lev_{ij}}$

or $\delta^{lev_{ik}+lev_{kj}}$, so it defines the updated level of fill as

$$lev_{ij} = \min(lev_{ij}, lev_{ik} + lev_{kj}). \tag{2.66}$$

Now, if initial levels of fill are assigned to each matrix position such that

$$lev_{ij} = \begin{cases} 1 & \text{if } a_{ij} \neq 0 \\ \infty & \text{if } a_{ij} = 0 \end{cases} \tag{2.67}$$

there are two considerations that should be made. The first one is that, if an entry a_{ij} is initially a nonzero, it will remain assigned to level 1 during the entire GE process. The second one is that, if a nonzero entry is generated during GE by generating entries a_{ik} and a_{kj}, it will be assigned to level $lev_{ik} + lev_{kj}$ and, the higher the level of the generating entries, the smaller the size of the new nonzero, so the level of fill could give an idea of the importance of each fill in entry generated during GE.

In order to be consistent with the definition of ILU0, the initial level assigned to each entry is shifted by -1, and the update of the level of fill becomes

$$lev_{ij} = \min(lev_{ij}, lev_{ik} + lev_{kj} + 1). \tag{2.68}$$

This is just a notation issue and does not change any of the aforementioned concepts.

Using the ILU(l) factorization as preconditioner can be very effective in accelerating the convergence of iterative methods for many problems. However, it has some important drawbacks. For example, one major problem is that, in general, it is impossible to predict the storage required to hold an ILU(l) factorization. In this sense, the codes that obtain this type of ILU are often composed of an initial symbolic factorization phase, in which the storage requirements are calculated, an a numerical factorization phase, which computes the actual ILU. In relation to this, allowing only a few levels of fill in during the factorization often results in factors very close to L and U. In this sense, there is a "maximum level of fill", which is defined as the lowest level of fill l for which the sparsity pattern of the incomplete factors is equal to that of L and U.

An additional important issue is that the level of fill-in may not always be a good indicator of the size of the elements that are being dropped. This can cause the dropping of large elements which may severely harm the accuracy factorization, probably leading to an ineffective preconditioner.

MILU

It has been observed that when ILU techniques are applied to solve elliptic PDEs, the asymptotic convergence rate of iterative solvers, as the mesh size becomes smaller, is only marginally better than an approach with no preconditioner [42]. In order to improve this situation, the Modified-ILU (MILU) aims to compensate the effect of the discarded entries by adding a term to the update of the diagonal entry of the $\tilde{U}$ factor during a given step of the factorization. This term can be problem dependent. For example, in problems derived

from second order PDEs, it is recommended to hold a term ch^2, where $c \neq 0$ is a constant and h is the mesh size. In general, a popular strategy is to add all the elements that have been dropped during the update of a given row to the diagonal entry (of U) corresponding to that row.

MILU has proven to be very effective in problems such as elliptic PDEs but, unfortunately, standard ILUs often outperform their MILU counterparts. According to Van der Vorst [103] this is related to a higher sensitivity of MILU to round-off errors. This has motivated intermediate versions by introducing relaxation parameters.

ILUT

When dropping an entry, positional ILU strategies make assumptions on the importance of the elements being dropped based on their position on the matrix. It turns out that in many cases these assumptions are inaccurate, and lead to difficulties when addressing real problems. To remedy this situation, it sounds reasonable to take a look at the magnitude of a fill-in entry before taking the decision of dropping it. This gives place to the so called threshold-based ILUs, or ILUT.

This type of ILUs is based on the same variants of GE as ILU(l). The difference is that now "dropping rules" are introduced at some point of the algorithm, which set an entry to zero if it meets certain criteria. For example, a simple strategy would be to drop all entries smaller than certain threshold τ. Instead, one could be more clever and drop the entry if its magnitude is smaller that a given fraction of the norm of the row it belongs to. Other rules can be devised and, in fact, more than one rule can be applied. What is usually known as ILU(p,τ) in the literature applies two dropping rules. The first one drops any entry of row i whose magnitude is less than τ_i, which is equal to τ multiplied by the norm of row i in the original matrix A. The second rule sets a limit to the number of non-zeros that can be kept in one row, dropping an element if it is not among the p largest values (in magnitude) of the row. While the first rule aims to save computing effort during the triangular solves in the preconditioner application by discarding entries that would (a priori) not contribute significantly to the result, the second rule keeps the storage space needed to hold the preconditioner within bounds.

The discarding criteria based on relative tolerance are, in general, more reliable than the criteria based exclusively on the absolute tolerance, and they usually yield a good solution. A common disadvantage for these criteria is the selection of a satisfactory value for τ. For this selection, it is common to start from a representative subset of the linear systems to be solved in the context of a specific application and, through experimentation, find a good value for the systems which arise from the same application. In most cases, selecting values of τ in the interval $[10^{-4}, 10^{-2}]$ yields good results, though the optimum value depends on the concrete problem [87].

Crout's algorithm and ILUC

The ILU variants that have been described so far are mostly based on the so-called IKJ variant of Gaussian Elimination. This variant is attractive from a computational point of

view because it proceeds row-wise, and it is therefore especially well-suited for row-oriented sparse storage formats as the widespread CSR[2]. Nevertheless, as it is done with IKJ-based ILUs, one can derive ILU factorization algorithms from any other variant. Although in exact arithmetic all variants would produce the same factorizations, each variant relies on different kernels and the computation patterns may lead to specialized techniques and storage formats for certain computer architectures. In this sense, these variants present considerable practical differences.

The Crout variant, which can be seen as a combination of the IKJ variant to compute $\tilde{L}$ and a transposed version to compute $\tilde{U}$, aims to solve one of the main drawbacks of the IKJ-based ILU. During the update stage, in order to update or introduce a fill-in entry, the structure that contains the factorization has to be accessed in increasing column order. This implies a search through the vector that holds the current row, which is being dynamically modified by the fill-in process. There are some strategies to perform this search efficiently, as line searches or using data structures such as binary trees. However, this issue remains an important problem when high accuracy is required in the factorization and considerable fill-in has to be allowed.

Another important advantage of this version of the ILU factorization, which will be analyzed in the following section, is that it is well suited to the implementation of inverse-based dropping strategies [27], which are an interesting breakthrough from the theoretical perspective, and have shown to be effective in many scenarios.

Algorithm 10 Crout's variant of the LU factorization.
Input: A
Output: L, U

1: **for** $k = 1$ **to** n **do**
2: **for** $i = 1$ **to** $k - 1$ and if $a_{ki} \neq 0$ **do**
3: $a_{k,k:n} = a_{k,k:n} - a_{ki} \times a_{i,k:n}$
4: **end for**
5: **for** $i = 1$ **to** $k - 1$ and if $a_{ki} \neq 0$ **do**
6: $a_{k+1:n,k} = a_{k+1:n,k} - a_{ik} \times a_{k+1:n,i}$
7: **end for**
8: $a_{k+1:n,k} = a_{k+1:n,k}/a_{kk}$
9: **end for**

2.3.2 Inverse-based ILUs

One of the major problems of the approaches described previously is that there is not a direct relationship between the magnitude of an entry and the impact that dropping it will have on the preconditioned system. In order to understand such impact, one has to inspect how preconditioners are used. Suppose A as been factored as $A = LDU$ where $L \in \mathbb{R}^{n\times n}$ is unit lower triangular, $U \in \mathbb{R}^{n\times n}$ is unit upper triangular and $D \in \mathbb{R}^{n\times n}$ is a diagonal matrix. This factorization is closely related to those presented previously, but is more adequate to expose some important properties. In fact, if the LU factorization $A = \hat{L}\hat{U}$ exists, then

[2]From Compressed Sparse Row format [39]

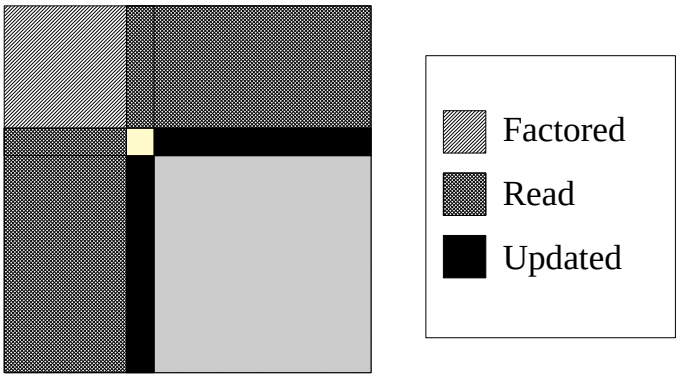


Figure 2.2: Computation pattern of Crout's LU factorization for a given step k.

$L = \hat{L}$ and $DU = \hat{U}$, so $D = diag(\hat{U})$.

To construct this factorization, an option consists in considering the following partition

$$\begin{bmatrix} \beta & d \\ c & E \end{bmatrix} \in \mathbb{R}^{n \times n}, \tag{2.69}$$

where $\beta \in \mathbb{R}$, which determines the dimension of the remaining blocks in 2.69. To factorize this matrix, at step k one has to exploit the following relation

$$\begin{bmatrix} \beta & d \\ c & E \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ l & I \end{bmatrix} \begin{bmatrix} \delta & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} 1 & u \\ 0 & I \end{bmatrix}, \tag{2.70}$$

where $S = E - l\delta u \in \mathbb{R}^{n-k,n-k}$ is called the Schur complement. The process is completed by recursively applying the same factorization to S, obtaining the exact $A = LDU$ factorization at step n.

In the context of preconditioners, it is not necessary to compute the exact factorization so, as explained previously with other versions of ILU, the common strategy consists in dropping elements of vectors l and u, and using the sparsified versions of these vectors to compute an approximate Schur complement. The approach suggested by Bollhöfer in [27], is to construct this block as

$$S = E - \tilde{l}d - (c - \tilde{l}\beta)\tilde{u}, \tag{2.71}$$

where $\tilde{l}$ and $\tilde{u}$ are the sparsified versions of l and u, respectively. The procedure to obtain an incomplete factorization using this strategy is described in Algorithm 11. Pivoting is not included for simplicity.

The justification for this choice lies in that this expression for S can be obtained from the lower-right block of $\tilde{L}^{-1}A\tilde{U}^{-1}$, which can be useful in the following sense. When we apply LDU as a preconditioner in a Krylov subspace or some other iterative method, the matrix for the preconditioned system will be $L^{-1}AU^{-1}D^{-1}$. This means that, when we

Algorithm 11 ILDU.

Input: A
Output: L, D, U

Set $L = U = D = I$, $S = A$.
for $i = 1$ **to** $n - 1$ **do**
  $d_{ii} = s_{ii}$
  $c = S_{i+1:n,i}$
  $d = S_{i,i+1:n}$
  $p = c/d_{ii}$
  $q = d/d_{ii}$
  Apply dropping rule to p and q
  $L_{i+1:n,i} = p$
  $U_{i,i+1:n} = q$
  $\hat{S} = S_{i+1:n,i+1:n}$
  $\hat{S} = \hat{S} - pd - (c - p\beta)q$
end for
$d_{nn} = s_{nn}$

drop an entry from the L or U factors, even if it is small, we do not know the impact of that dropping on the preconditioned system as the inverse factors L^{-1} and U^{-1} are, generally, not available. It is therefore interesting to devise a way in which the inverse factors can be monitored during the process, so the choice of the expression for the Schur complement sounds reasonable.

The computation of L^{-1} and U^{-1} during the factorization is, of course, out of the question. Instead, one alternative is to obtain information about the approximate inverses of the factors. Incorporating this information in the dropping rules of the ILU could, in principle, be expected to yield good results. In this sense, important developments about the relation between ILU factorizations and approximate inverses were introduced by Bollhöfer and Saad at the beginning of this century [30].

Approximate Inverse factorizations (AINV) are based on constructing the incomplete factors of A^{-1} explicitly. If no dropping is applied, the idea is finding unit upper triangular matrices $W \in \mathbb{R}^{n\times n}$ and $Z \in \mathbb{R}^{n\times n}$, and a diagonal matrix $D \in \mathbb{R}^{n\times n}$, so that $W^T AZ = D$ or, equivalently, $A^{-1} = ZD^{-1}W^T$. Such a factorization can be obtained, for example, by means of a bi-orthogonalization process in which $W^T A$ and $Z^T A^T$ are transformed into triangular matrices one row/column at a time. In a similar way as it is done with ILUs, dropping can be applied to W and Z during the factorization, obtaining an incomplete factorization $A^{-1} \approx ZD^{-1}W^T$. This process is described in Algorithm 12.

Although Algorithm 12 presents some difficulties to encode an efficient implementation for sparse matrices, like the use of rank-1 updates, it shows that it should be possible to obtain an ILDU factorization such that $L^{-1} \approx W^T$ and $U^{-1} \approx Z$ by including information about the approximate inverse factors. In this sense, consider a modification of the ILDU algorithm, supplementing it with a progressive inversion process, in which the inverses of the L and U factors are calculated on the fly. At iteration $i-1$, the factor $\tilde{U}_{i-1}$ should take

Algorithm 12 Right-looking variant of AINV.
Input: A
Output: W, D, Z

$p = c = (0, \dots, 0) \in \mathbb{R}^n$, $Z = W = I_n$.
for $i = 1$ **to** n **do**
  $p_i = e_i^T A Z e_i^T$
  $q_i = e_i^T W^T A e_i^T$
  for $j = i + 1$ **to** n **do**
    $p_j = e_j^T A Z e_i / p_i$
    $q_j = e_i^T W^T A e_j / q_i$
    $W_{1:i,j} = W_{1:i,j} - W_{1:i,i} p_j$
    $Z_{1:i,j} = Z_{1:i,j} - Z_{1:i,i} q_j$
  end for
  for all $k \leq i, l > i$: drop w_{kl} if $|w_{kl}| \leq \tau$ and drop z_{kl} if $|z_{kl}| \leq \tau$
  $d_{ii} = p_i$
end for

the form

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ O & 1 & O \\ O & O & I \end{bmatrix} = \begin{bmatrix} A & b & C \\ O & v & w^T \\ O & O & Z \end{bmatrix}.$$

The i-th step is responsible for calculating the entries of vector w^T and adding them to U_{i-1} to obtain U_i. If $\tilde{q} = (0, \dots, 0, \tilde{q}_{i+1}, \dots, \tilde{q}_n)$ is the row vector updated in the i-th iteration of Algorithm 11, then

$$\tilde{U}_i = \tilde{U}_{i-1} + e_i \tilde{q}. \tag{2.72}$$

Given the structure of both $\tilde{U}_i$ and $\tilde{q}$, the following relation between $\tilde{U}_i$ and $\tilde{U}_{i-1}$ holds

$$\tilde{U}_i = (I + e_i \tilde{q}) \tilde{U}_{i-1}. \tag{2.73}$$

Note that $\tilde{q} = e_i(\tilde{U}_{i-1} - I)$ and $e_i \tilde{q} = e_i \tilde{q} \tilde{U}_{i-1}$, so that Equation (2.73) simply implies that the new matrix $\tilde{U}_i$ is obtained by replacing the entries from $i+1$ to n with the corresponding entries of $\tilde{q}$. Inverting (2.73) and operating accordingly we obtain the following expression for the partial inverse factor U_i^{-1}

$$\tilde{U}_i^{-1} = \tilde{U}_{i-1}^{-1} (I + e_i \tilde{q})^{-1} = \tilde{U}_{i-1}^{-1} (I - e_i \tilde{q}). \tag{2.74}$$

A similar derivation can be done for $\tilde{L}$.

Equation (2.74) and its counterpart for $\tilde{L}$ specify how to compute the (approximate) inverse factors as the factorization advances. In [30] the authors demonstrated that, if the above inverse factors are computed during the ILU and the Schur complement is constructed adequately, this ILU preconditioner is essentially equivalent to the AINV algorithm, so the computed inverse factors are equivalent to W and Z. Moreover, they state that, if the dropping criteria is modified such that an entry l_{ji} is dropped only when $|l_{ji}| \cdot \max(1 \cup |w_{ki}| : k = 1, \dots, i-1) \leq \tau$, an entry w_{kl} is dropped if $|w_{kl}| \leq \tau$, and the (modified) Schur

complement is set to be the lower $(n-i)\times(n-i)$ block of W^TA, then for any $k>l$

$$|(I-WL^T)_{lk}|\leq 2\tau(k-l). \tag{2.75}$$

This result is important because it introduces a dropping rule for the factor L that is based on the entries of the approximate inverse W, and sets a bound for the error obtained in the incomplete factorization based on the approximate inverse. An analogous result can be stated for the U and Z factors.

It can be expected that dropping small entries in the approximate inverse factors W and Z has a much lower impact on the preconditioned system W^TAZ than dropping small entries of L and U. In fact, there are numerical results [29], which demonstrate that approximate inverse preconditioners behave better than ILUs in highly ill-conditioned scenarios. The problem with this result is that it is not practical to compute the approximate inverse factors together with the ILU factorization, so in order to apply the previous dropping rule, some compromise should be made.

As the rule depends on the maximum entry in magnitude of the i-th column of W, which is equivalent to the i-th row of L^{-1}, one option is to devise an estimate for $\left\|e_i^TL^{-1}\right\|_\infty$ (and $\left\|U^{-1}e_i\right\|_\infty$) that can be computed inexpensively and avoids the computation of the i-th column of W and Z. Such estimates can be derived from a condition estimator for upper triangular matrices [36, 63]. This estimate is based on exploiting that $\left\|e_i^TL^{-1}b\right\|_\infty\leq\left\|e_iL^{-1}\right\|_\infty\|b\|_\infty$ to find a vector b such that $\|b\|_\infty=1$ and that, for each step $i=1,\ldots,n$,

$$\left\|e_i^TL^{-1}b\right\|_\infty\approx\left\|e_iL^{-1}\right\|_\infty. \tag{2.76}$$

The approach described by [28] consists in solving the linear system $Lx=b$ and taking $|x_i|\approx e_i^TL^{-1}b$ as an estimate, choosing the i-th entry of vector b dynamically in each step, so that it (hopefully) maximizes $|x_i|$. A common choice is to set the entries of the vector to ±1 as the solution via forward substitution advances, depending on which value maximizes $|x_i|$. The procedure described in Algorithm 13 also seeks to prevent, at step i, the growth of the x_k entries such that $k>i$.

2.3.3 Multilevel inverse-based ILUs

In the previous sections we presented several strategies that have been developed with the aim of improving the stability or the robustness of ILU factorizations. Such strategies mostly rely on different criteria to decide which fill-in elements can be safely discarded during the factorization procedure. However, for simplicity, pivoting, which is the main strategy to enhance the stability of factorizations in full matrix algebra, has not been discussed yet for the sparse incomplete factorization case.

In principle, pivoting can be incorporated to any of the ILU techniques mentioned in this chapter with the aim of improving its numerical stability. A typical example of this strategy is the ILU variant known as ILUTP, which essentially consists in applying partial column pivoting to ILUT [89]. Unfortunately, although this strategy can indeed improve ILUT preconditioners, it is prone to failures as a result of zero rows appearing during the

Algorithm 13 Condition estimator for L^{-1}.

Input: L, $b \in (\pm 1, \ldots, \pm 1)^T$
Output: x such that $|x_i| \approx e_i^T L^{-1} b$

$v = x = (0, \ldots, 0)^T \in \mathbb{R}^n$.
for $i = 1$ **to** n **do**
  $\mu^+ = 1 - v_i$
  $\mu^- = -1 - v_i$
  $v_+ = v_{i+1:n} + L_{i+1:n,i}\mu^+$
  $v_- = v_{i+1:n} + L_{i+1:n,i}\mu^-$
  if $|\mu^+| + \|v_+\|_\infty > |\mu^-| + \|v_-\|_\infty$ **then**
    $x_i = \mu^+$
    $v_{i+1:n} = v^+$
  else
    $x_i = \mu^-$
    $v_{i+1:n} = v^-$
  end if
end for

factorization due to the combination of permutations and dropping. If significant fill-in is allowed, ILUTP can produce accurate preconditioners, but at a high computational cost.

In [44], pivoting is avoided during the factorization by reordering the rows of the matrix previously in order to improve its diagonal dominance. The ordering strategy aims to bring large pivots as close to the diagonal as possible, and is combined with a static post-ordering to reduce fill-in. This type of reordering, combined with ILU preconditioners, have yield good results in several scenarios [23, 95].

A different strategy to improve the stability of ILUs consists in applying a reordering to partition the rows/columns of the matrix into two sets. When solving or factorizing a linear system with such a partition, the first set of unknowns or rows can be dealt with immediately, while the solution or factorization of the second set will use the result of the first one. These sets are often classified "coarse" and "fine" because of the resemblance of this strategy with Algebraic Multigrid methods.

As an example, one can split a linear system like

$$Ax = b \rightarrow \begin{bmatrix} B & F \\ E & C \end{bmatrix} \begin{bmatrix} u \\ y \end{bmatrix} x = \begin{bmatrix} f \\ g \end{bmatrix} \tag{2.77}$$

seeking that the block B is easy to invert, as in the case of a diagonal matrix, or that it is diagonally dominant so it can be factorized safely without pivoting. In the first case, the unknown u is easy to express in terms of y as B can be inverted trivially, so the system can be expressed in terms of the Schur complement $S_C = C - EB^{-1}F$. In the second case, a partial factorization of A can be computed by first factorizing $B = LU$ and then completing the procedure by factorizing the Schur complement $S_C = C - EU^{-1}L^{-1}F$. Moreover, a similar procedure can be recursively applied to factorize S_C in what is often called a multilevel procedure.

The factorization of a given level l will be of the form

$$P_l A_l Q_l^T = \begin{bmatrix} B_l & F_l \\ E_l & C_l \end{bmatrix} \approx \begin{bmatrix} L_l & 0 \\ E_l U_l^{-1} & I \end{bmatrix} \begin{bmatrix} U_l & L_l^{-1} F_l \\ 0 & S_C \end{bmatrix}. \tag{2.78}$$

Each level requires the construction of the L_l and U_l factors as well as the blocks $E_l U_l^{-1}$ and $L_l^{-1} F_l$, which can be done with any of the previously presented ILUT techniques, and the construction of the Schur complement S_C. In this process, the Schur complement can suffer substantial fill-in, which should be dealt with by applying some technique to drop small elements of L_l, U_l, $E_l U_l^{-1}$, $L_l^{-1} F_l$ and S_C itself. Once the previous blocks are obtained, the process is repeated with $A_{l+1} = S_C$ until the Schur complement of that level is small or dense enough to be handled by a direct solver or a maximum number of levels is reached. It is worth noticing that, since the matrices EU^{-1} and $L^{-1}F$ are only used to calculate the Schur complement corresponding to the matrix that will be factorized in the next level, they can be discarded once such Schur complement is available. All operations involving these two matrices or their approximate versions can be computed using L_l, U_l, E_l, F_l.

The effectiveness of this approach naturally lies on the quality of the permutation chosen to obtain the fine and coarse sets. In [90], Saad proposes several strategies to obtain a block B that is as diagonally dominant as possible. They differ from the aforementioned strategies in that Saad's approach is dynamic rather than based on static orderings. Moreover, in [31] Bollhöfer and Saad proposed another dynamic ordering scheme based on controlling the growth of the inverse L_l^{-1} and U_l^{-1}. This consists in using the condition estimator presented in Algorithm 13 to detect rows or columns that would push the norm of the inverse factors above a certain threshold, defined as a parameter of the algorithm. If such row or column is detected, a diagonal pivoting is performed such that the current row and column is pushed to the bottom-right corner of the matrix. This can be combined with previous static ordering to generate an initial splitting like the above, where B is likely to have good properties. In summary, one has a possibly non-symmetric permutation $\hat{P}$ and $\hat{Q}$ such that

$$\hat{P}^T A \hat{Q} = \tilde{P}^T (P^T A Q) \tilde{Q} = \tilde{P}^T \begin{bmatrix} B & F \\ E & C \end{bmatrix} \tilde{Q} = \begin{bmatrix} B_{11} & B_{12} & F_1 \\ B_{21} & B_{22} & F_2 \\ E_1 & E_2 & C \end{bmatrix} = \begin{bmatrix} B_{11} & \hat{F} \\ \hat{E} & C \end{bmatrix}. \tag{2.79}$$

The factorization is later performed on the smaller B_{11} leading block.

Many static orderings can be chosen to obtain an initial matrix with reasonably good properties. Techniques such as reverse Cuthill-McKee (RCM) [38], Approximate Minimum Degree (AMD) [16] or Nested Dissection (ND) [51] can be used to reduce the fill-in in the subsequent factorization. More modern algorithms, like MC64 [44], aim to improve diagonal dominance.

Although any form of the Gaussian Elimination procedure could be adapted to yield a partial ILU factorization like the above, the Crout algorithm seems well suited to obtain an efficient implementation, since in this variant it is possible to compute the k-th column of L and k-th row of U at step k. This makes it easy to incorporate the aforementioned diagonal pivoting strategy. A graphical description of this process is offered in Figure 2.3

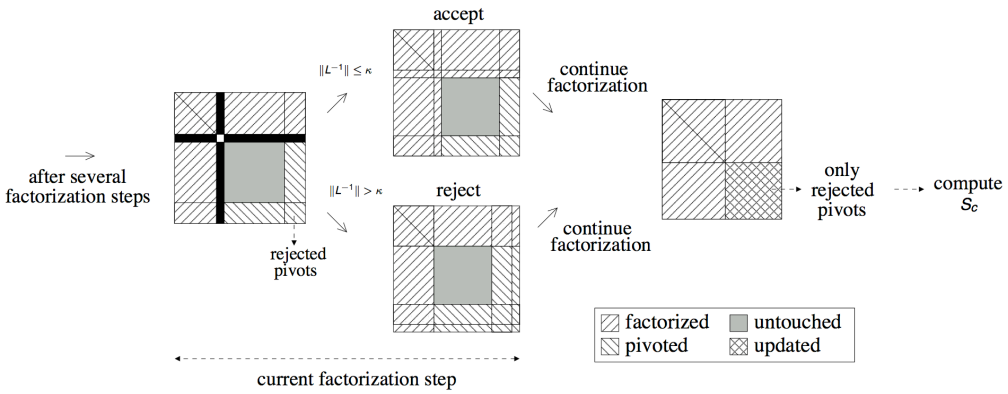


Figure 2.3: A step of the Crout variant of the multilevel ILU factorization. Extracted from [8].

2.4 ILUPACK

Consider the linear system $Ax = b$, where the $n \times n$ coefficient matrix A is large and sparse, and both the right-hand side vector b and the sought-after solution x contain n elements. ILUPACK [26] provides software routines to calculate an inverse-based multilevel ILU preconditioner M, of dimension $n \times n$, which can be applied to accelerate the convergence of Krylov subspace iterative solvers. The package includes numerical methods for different matrix types, precision, and arithmetic, covering Hermitian positive definite/indefinite and general real and complex matrices. When using an iterative solver enhanced with an ILUPACK preconditioner, the most demanding task from the computational point of view is the application of the preconditioner, which occurs (at least once) per iteration of the solver.

The implementation of all solvers in ILUPACK follows a reverse communication approach, in which the backbone of the method is performed by a serial routine that is repeatedly called. This routine is re-entered at different points, and sets flags before exiting so that operations such as SpMV, the application of the preconditioner, and convergence checks can be then performed by external routines implemented by the user. This is aligned with the decision adopted by ILUPACK to employ SPARSKIT[3] as the backbone of the solvers.

2.4.1 ILUPACK non-symmetric preconditioner

Let us focus on the real case, where $A, M \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$. The computation of ILUPACK's preconditioner proceeds following three steps:

1. Initially, a pre-processing stage scales A by a diagonal matrix $\tilde{D} \in \mathbb{R}^{n \times n}$ and reorders the result by a permutation $\tilde{P} \in \mathbb{R}^{n \times n}$: $\hat{A} = \tilde{P}^T \tilde{D} A \tilde{D} \tilde{P}$.
2. An incomplete factorization next computes $\hat{A} \approx LDU$, where $L, U^T \in \mathbb{R}^{n \times n}$ are unit lower triangular factors and $D \in \mathbb{R}^{n \times n}$ is (block) diagonal. In some detail, $\hat{A}$ is

[3] Available at http://www-users.cs.umn.edu/~saad/software/SPARSKIT/.

processed in this stage to obtain the partial ILU factorization:

$$\begin{aligned} \hat{P}^T \hat{A} \hat{P} \equiv \begin{pmatrix} B & F \\ G & C \end{pmatrix} &= LDU + E \\ &= \begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & S_c \end{pmatrix} \begin{pmatrix} U_B & U_F \\ 0 & I \end{pmatrix} + E. \end{aligned} \tag{2.80}$$

Here, $\hat{P} \in \mathbb{R}^{n \times n}$ is a permutation matrix, $\|L^{-1}\|, \|U^{-1}\| \lessapprox \kappa$, with κ a user-predefined threshold, E contains the elements "dropped" during the ILU factorization, and S_C represents the approximate Schur complement assembled from the "rejected" rows and columns.

3. The process is then restarted with $A = S_c$, (until S_c is void or "dense enough" to be handled by a dense solver,) yielding a multilevel approach.

At level l, the multilevel preconditioner can be recursively expressed as

$$M_l \approx \tilde{D}^{-1} \tilde{P} \hat{P} \begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} \begin{pmatrix} U_B & U_F \\ 0 & I \end{pmatrix} \hat{P}^T \tilde{P}^T \tilde{D}^{-1}, \tag{2.81}$$

where L_B, D_B and U_B are blocks of the factors of the multilevel LDU preconditioner (with L_B, U_B^T unit lower triangular and D_B diagonal); and M_{l+1} stands for the preconditioner computed at level $l+1$.

For the review of this operation, we consider its application at level l, for example, to compute $z := M_l^{-1} r$. This requires solving the system of linear equations:

$$\begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} \begin{pmatrix} U_B & U_F \\ 0 & I \end{pmatrix} \hat{P}^T \tilde{P}^T \tilde{D}^{-1} z = \hat{P}^T \tilde{P}^T \tilde{D} r. \tag{2.82}$$

Breaking down (2.82), we first recognize two transformations to the residual vector, $\hat{r} := \hat{P}^T \tilde{P}^T (\tilde{D} r)$, before the following block system is defined:

$$\begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} \begin{pmatrix} U_B & U_F \\ 0 & I \end{pmatrix} w = \hat{r}. \tag{2.83}$$

This is then solved for $w (= \hat{P}^T \tilde{P}^T \tilde{D}^{-1} z)$ in three steps,

$$\begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} y = \hat{r}, \quad \begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} x = y, \quad \begin{pmatrix} U_B & U_F \\ 0 & I \end{pmatrix} w = x, \tag{2.84}$$

where the recursion is defined in the second one. In turn, the expressions in (2.84) also need to be solved in two steps. Assuming y and $\hat{r}$ are split conformally with the factors, for the expression on the left of (2.84) we have

$$\begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} y_B \\ y_C \end{pmatrix} = \begin{pmatrix} \hat{r}_B \\ \hat{r}_C \end{pmatrix} \Rightarrow L_B y_B = \hat{r}_B \ , \ y_C := \hat{r}_C - L_G y_B. \tag{2.85}$$

Partitioning the vectors as earlier, the expression in the middle of (2.84) involves the

diagonal-matrix multiplication and the effective recursion:

$$\begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} \begin{pmatrix} x_B \\ x_C \end{pmatrix} = \begin{pmatrix} y_B \\ y_C \end{pmatrix} \;\Rightarrow\; x_B := D_B^{-1} y_B \;,\; x_C := M_{l+1}^{-1} y_C. \tag{2.86}$$

In the recursion base step, M_{l+1} is void and only x_B has to be computed. Finally, after an analogous partitioning, the expression on the right of (2.84) can be reformulated as

$$\begin{pmatrix} U_B & U_F \\ 0 & I \end{pmatrix} \begin{pmatrix} w_B \\ w_C \end{pmatrix} = \begin{pmatrix} x_B \\ x_C \end{pmatrix} \;\Rightarrow\; w_C := x_C \;,\; U_B w_B = x_B - U_F w_C, \tag{2.87}$$

where z is simply obtained from $z := \tilde{D}(\tilde{P}(\hat{P}w))$.

To save memory, ILUPACK discards the off-diagonal blocks L_G and U_F once the level of the preconditioner is calculated, keeping only the rectangular matrices G and F of (2.77), which are often much sparser. Thus, (2.85) is changed as:

$$L_G = G U_B^{-1} D_B^{-1} \quad \Rightarrow \quad y_C := \hat{r}_C - G U_B^{-1} D_B^{-1} y_B = \hat{r}_C - G U_B^{-1} D_B^{-1} L_B^{-1} \hat{r}_B, \tag{2.88}$$

while the expressions related to (2.87) are modified as

$$U_F = D_B^{-1} L_B^{-1} F \quad \Rightarrow \quad U_B w_B = D_B^{-1} y_B - D_B^{-1} L_B^{-1} F w_C. \tag{2.89}$$

Operating with care, the final expressions are thus obtained,

$$L_B D_B U_B s_B = \hat{r}_B, L_B D_B U_B \hat{s}_B = F w_C \;\Rightarrow\; y_C := \hat{r}_C - G s_B,\; w_B := s_B - \hat{s}_B. \tag{2.90}$$

To summarize the previous description of the method, the application of the preconditioner requires, at each level, two sparse matrix-vector products (SpMV), solving two linear systems with coefficient matrix of the form LDU, and a few vector kernels.

In case the matrix A is SPD, ILUPACK is capable of obtaining a SPD preconditioner by means of the Crout variant of the incomplete Cholesky (IC) factorization. This yields the approximation $A \approx L\Sigma L^T$, with $L \in \mathbb{R}^{n\times n}$ sparse lower triangular and $\Sigma \in \mathbb{R}^{n\times n}$ diagonal. As in the non-symmetric case, a scaling and a reordering (defined respectively by $P, D \in \mathbb{R}^{n\times n}$) are applied to A in order to improve the numerical properties as well as reduce the fill-in in L. The IC factorization operates on $\hat{A} = P^T DADP$ and it is then performed obtaining

$$\hat{P}^T \hat{A} \hat{P} \equiv \begin{bmatrix} B & F^T \\ F & C \end{bmatrix} = \begin{bmatrix} L_B & 0 \\ L_F & I \end{bmatrix} \begin{bmatrix} D_B & 0 \\ 0 & S_c \end{bmatrix} \begin{bmatrix} L_B^T & L_F^T \\ 0 & I \end{bmatrix} + E. \tag{2.91}$$

Here, $\|L_B^{-1}\| \lessapprox \kappa$ and E contains the elements dropped during the IC factorization. Restarting the process with $A = S_c$, we obtain a multilevel approach.

Then, the application of the preconditioner in the PCG algorithm, for a given level l, is

derived from (2.91) as

$$M_l = D^{-1} P \hat{P} \begin{bmatrix} L_B & 0 \\ L_F & I \end{bmatrix} \begin{bmatrix} D_B & 0 \\ 0 & M_{l+1} \end{bmatrix} \begin{bmatrix} L_B^T & L_F^T \\ 0 & I \end{bmatrix} \hat{P}^T P^T D^{-1}, \tag{2.92}$$

where $M_0 = M$.

Operating properly on the vectors,

$$\hat{P}^T P^T D^{-1} z = \hat{z} = \begin{bmatrix} \hat{z}_B \\ \hat{z}_C \end{bmatrix}, \; \hat{P}^T P^T D r = \hat{r} = \begin{bmatrix} \hat{r}_B \\ \hat{r}_C \end{bmatrix}, \tag{2.93}$$

and applying $L_F = F L_B^{-T} D_B^{-1}$ –derived from (2.91)–, we can expose the following computations to be performed at each level of the preconditioner:

$$\begin{array}{ll} & \hat{r} := \hat{P}^T P^T D r, \quad \text{Solve } L_B D_B L_B^T s_B = \hat{r}_B \text{ for } s_B, \\ & t_B := F s_B, \qquad y_C := \hat{r}_B - t_B, \\ \text{Recursive step:} & \text{Solve } M_{l+1} \hat{z}_C = y_C \text{ for } \hat{z}_C, \\ & \hat{t}_B := F^T \hat{z}_C, \qquad \text{Solve } L_B D_B L_B^T \hat{s}_B = \hat{t}_B \text{ for } \hat{s}_B, \\ & \hat{z}_B := s_B - \hat{s}_B, \quad z := D P \hat{P} \hat{z}. \end{array} \tag{2.94}$$

2.4.2 Task-parallel ILUPACK

In this section, we summarize the main ideas underlying the task-parallel version of ILUPACK. A more detailed explanation can be found in [3].

Computation of the preconditioner.

The task-parallel version of this procedure employs Nested Dissection (ND) [89] to extract parallelism. To illustrate this, consider a partitioning, defined by a permutation $\bar{P} \in \mathbb{R}^{n \times n}$, such that

$$\bar{P}^T A \bar{P} = \left[\begin{array}{cc|c} A_{00} & 0 & A_{02} \\ 0 & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right]. \tag{2.95}$$

Computing a partial IC factorizations of the two leading blocks, A_{00} and A_{11}, yields the following partial approximation of $\bar{P}^T A \bar{P}$

$$\left[\begin{array}{cc|c} L_{00} & 0 & 0 \\ 0 & L_{11} & 0 \\ \hline L_{20} & L_{21} & \mathrm{I} \end{array}\right] \left[\begin{array}{cc|c} D_{00} & 0 & 0 \\ 0 & D_{11} & 0 \\ \hline 0 & 0 & S_{22} \end{array}\right] \left[\begin{array}{cc|c} L_{00}^T & 0 & L_{20}^T \\ 0 & L_{11}^T & L_{21}^T \\ \hline 0 & 0 & \mathrm{I} \end{array}\right] + E_{01},$$

where

$$S_{22} = A_{22} - (L_{20} D_{00} L_{20}^T) - (L_{21} D_{11} L_{21}^T) + E_2, \tag{2.96}$$

is the approximate Schur complement. By recursively proceeding in the same manner with S_{22}, the IC factorization of $\bar{P}^T A \bar{P}$ is eventually completed.

The block structure in (2.95) allows the permuted matrix to be decoupled into two

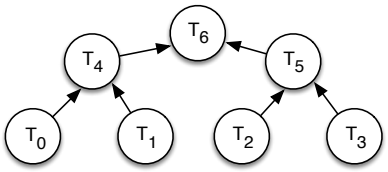


Figure 2.4: Dependency tree of the diagonal blocks. Task T_j is associated with block A_{jj}. The leaf tasks are associated with the sub-graphs of the leading block of the ND, while inner tasks are associated to separators.

submatrices, so that the IC factorizations of the leading block of both submatrices can be processed concurrently, with

$$A_{22} = A_{22}^0 + A_{22}^1 \,, \begin{cases} \left[\begin{array}{c|c} A_{00} & A_{02} \\ \hline A_{20} & A_{22}^0 \end{array}\right] = \left[\begin{array}{c|c} L_{00} & 0 \\ \hline L_{20} & \mathrm{I} \end{array}\right] \left[\begin{array}{c|c} D_{00} & 0 \\ \hline 0 & S_{22}^0 \end{array}\right] \left[\begin{array}{c|c} L_{00}^T & L_{20}^T \\ \hline 0 & \mathrm{I} \end{array}\right] + E_0 \\ \\ \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22}^1 \end{array}\right] = \left[\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & \mathrm{I} \end{array}\right] \left[\begin{array}{c|c} D_{11} & 0 \\ \hline 0 & S_{22}^1 \end{array}\right] \left[\begin{array}{c|c} L_{11}^T & L_{21}^T \\ \hline 0 & \mathrm{I} \end{array}\right] + E_1, \end{cases} \tag{2.97}$$

and

$$S_{22}^0 = A_{22}^0 - \left(L_{20} D_{00} L_{20}^T\right) + E_2^0; \quad S_{22}^1 = A_{22}^1 - \left(L_{21} D_{11} L_{21}^T\right) + E_2^1.$$

Once the two systems are computed, S_{22} can be constructed given that

$$E_2 \approx E_2^0 + E_2^1 \;\rightarrow\; S_{22} \approx S_{22}^0 + S_{22}^1. \tag{2.98}$$

To further increase the amount of task-parallelism, one could find a permutation analogous to $\bar{P}$ for the two leading blocks following the ND scheme. For example, a block structure similar to (2.95) would yield the following decoupled matrices:

$$\left[\begin{array}{cccc|cc|c} A_{00} & 0 & 0 & 0 & A_{04} & 0 & A_{06} \\ 0 & A_{11} & 0 & 0 & A_{14} & 0 & A_{16} \\ 0 & 0 & A_{22} & 0 & 0 & A_{25} & A_{26} \\ 0 & 0 & 0 & A_{33} & 0 & A_{35} & A_{36} \\ \hline A_{40} & A_{41} & 0 & 0 & A_{44} & 0 & A_{46} \\ 0 & 0 & A_{52} & A_{53} & 0 & A_{55} & A_{56} \\ \hline A_{60} & A_{61} & A_{62} & A_{63} & A_{64} & A_{65} & A_{66} \end{array}\right] \begin{array}{l} \bar{A}_{00} = \left[\begin{array}{c|cc} A_{00} & A_{04} & A_{06} \\ \hline A_{40} & A_{44}^0 & A_{46}^0 \\ A_{60} & A_{64}^0 & A_{66}^0 \end{array}\right] \bar{A}_{11} = \left[\begin{array}{c|cc} A_{11} & A_{14} & A_{16} \\ \hline A_{41} & A_{44}^1 & A_{46}^1 \\ A_{61} & A_{64}^1 & A_{66}^1 \end{array}\right] \\ \rightarrow \\ \bar{A}_{22} = \left[\begin{array}{c|cc} A_{22} & A_{25} & A_{26} \\ \hline A_{52} & A_{55}^2 & A_{56}^2 \\ A_{62} & A_{65}^2 & A_{66}^2 \end{array}\right] \bar{A}_{33} = \left[\begin{array}{c|cc} A_{33} & A_{35} & A_{36} \\ \hline A_{53} & A_{55}^3 & A_{56}^3 \\ A_{63} & A_{65}^3 & A_{66}^3 \end{array}\right] \end{array} \tag{2.99}$$

Figure 2.4 illustrates the dependency tree for the factorization of the diagonal blocks in (2.99). The edges of the preconditioner *directed acyclic graph* (DAG) define the dependencies between the diagonal blocks (tasks), which dictate the order in which these blocks of the matrix have to be processed.

In summary, the task-parallel version of ILUPACK partitions the original matrix into a number of decoupled blocks, and then delivers a partial IC factorization during the computation of (2.97), with some differences with respect to the sequential procedure. The main change is that the computation is restricted to the leading block, and therefore the rejected pivots are moved to the bottom-right corner of the leading block; see Figure 2.5.

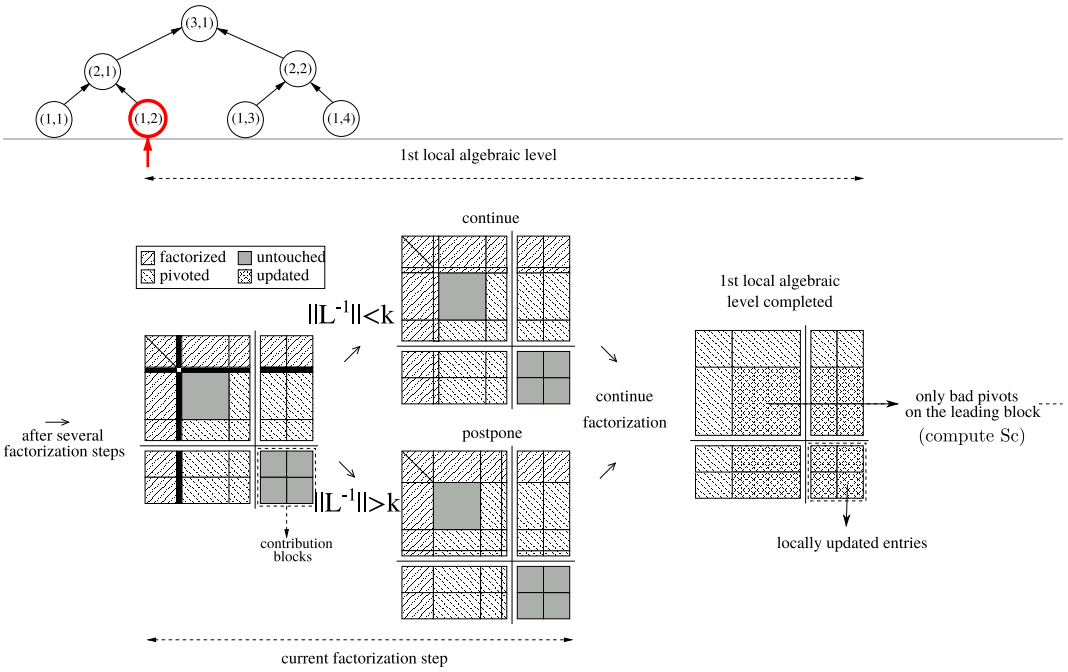


Figure 2.5: A step of the Crout variant of the parallel preconditioner computations. Extracted from [8].

Although the recursive definition of the preconditioner, shown in (2.92), is still valid in the task-parallel case, some recursion steps are now related to the edges of the corresponding preconditioner DAG. Therefore different DAGs involve distinct recursion steps yielding distinct preconditioners, which nonetheless exhibit close numerical properties to that obtained with the sequential ILUPACK [3].

Application of the preconditioner.

As the definition of the recursion is maintained, the operations to apply the preconditioner, in (2.94), remain valid. However, to complete the recursion step in the task parallel case, the DAG has to be crossed two times per solve $z_{k+1} := M^{-1}r_{k+1}$ at each iteration of the PCG: once from bottom to top and a second time from top to bottom (with dependencies/arrows reversed in the DAG).

2.5 Related work

Many research works have reported important benefits for the solution of sparse linear algebra problems on many-core platforms. Unfortunately, many of these efforts address only non-preconditioned versions of the methods, where the sparse matrix-vector product (SpMV) is the main bottleneck.

A few early works proposed GPU implementations of well-known iterative solvers before CUDA even existed. Among these pioneers, we can highlight the efforts by Rumpf and Strzodka [86] on the CG iteration for linear systems arising in finite element methods; Bolz

et al's [32] acceleration of multigrid problems leveraging the graphics pipeline; and the studies of the CG method by Goodnight et al. [58], as well as Krüger et al. [68].

With the start of the CUDA era, Buatois et al. [33] implemented the CG method in conjunction with a Jacobi preconditioner, using the block compressed sparse row (BCSR) format. Later, Bell and Garland [21] addressed SpMV, including several sparse storage formats, that became the basis for the development of the CUSP library [22].

A parallel CG solver with preconditioning for the Poisson equation optimized for multi-GPU architectures was presented by Ament et al. [15]. Sudan et al. [99] introduced GPU kernels for SpMV and the block-ILU preconditioned GMRES in flow simulations, showing promising speed-ups. In parallel, Gupta completed a master thesis [60] implementing a deflated preconditioned CG for Bubbly Flow.

Naumov [75] produced a solver for triangular sparse linear systems in a GPU, one of the major kernels for preconditioned variants of iterative methods such as CG or GMRES. Later, the same author extended the proposal to compute incomplete LU and Cholesky factorizations with no fill-in (ILU0) in a GPU [74]. The performance of the aforementioned algorithms strongly depends on the sparsity pattern of the coefficient matrix.

Li and Saad [70] studied the GPU implementation of SpMV with different sparse formats to develop data-parallel versions of CG and GMRES. Taking into account that the performance of the triangular solve for CG and GMRES, preconditioned with IC (Incomplete Cholesky) and ILU respectively, was rather low on the GPU, a hybrid approach was proposed in which the CPU is leveraged to solve the triangular systems.

Recently, He et. al. [61] presented a hybrid CPU-GPU implementation of the GMRES method preconditioned with an ILU-threshold preconditioner to control the fill-in of the factors. In the same work, the authors also propose a new algorithm to compute SpMV in the GPU.

Some of these ideas are currently implemented in libraries and frameworks such as CUSPARSE, CUSP, CULA, PARALUTION and MAGMA-sparse. To our best knowledge, no GPU implementations of multi-level preconditioners such as that underlying ILUPACK have been developed, with the exception of our previous efforts, and in some aspects the proposal by Li and Saad.

2.5.1 Other software packages

The considerable research dedicated to the solution of linear systems by means of iterative methods has given birth to a number of software tools, most of them designed to exploit massively-parallel HPC infrastructures. Next, we present an overview of the most relevant examples.

ILU++

ILU++ is a software package, written entirely in C++ that comprises a set of iterative linear systems solvers complemented with modern ILU preconditioners for the solution of sparse linear systems via iterative methods. The package includes different permutation and scaling techniques, pivoting strategies and dropping rules. In particular ILU++ bundles the inverse-

based multilevel preconditioners provided by ILUPACK, together with dual-threshold ILUs developed by Saad [87] and weighted dropping strategies for ILUTP studied by Mayer in [73]. It is currently available on GitHub[4]

ViennaCL

The Vienna Computing Library (ViennaCL) is a scientific computing library written in C++ that provides a set of routines for the solution of large sparse systems of equations by means of iterative methods. It uses either a host based computing back-end, an OpenCL computing back-end, or CUDA to enable the execution on parallel architectures such as multi-core CPUs, GPUs and MIC platforms (as Xeon Phi processors). The package includes implementations of the Conjugate Gradient (CG), Stabilized BiConjugate Gradient (BiCGStab), and Generalized Minimum Residual (GMRES) methods.

As for preconditioners, ViennaCL offers implementations of classical ILU0/IC0 and ILUT/ICT, as well as a variant of ILU0/IC0 recently proposed by Chow and Patel [35] that is more suitable for the execution on massively parallel devices than the standard approach. It also offers Block-ILU preconditioners, simple Jacobi and Row-Scaling preconditioners, and Algebraic Multi-Grid preconditioners.

The library strongly focuses on providing compatibility with the widest range of computing platforms, as opposed to being specifically tailored to the hardware solutions of a particular vendor.

pARMs

pARMS is a library of parallel solvers for distributed sparse linear systems of equations that extends the sequential library ARMS. It provides Krylov subspace solvers, preconditioned using a domain decomposition strategy based on a Recursive Multi-level ILU factorization. The library includes many of the standard domain-decomposition type iterative solvers in a single framework. For example, the standard Schwartz procedures, as well as a number of Schur complement techniques are included. ILUPACK implements its inverse-based multilevel preconditioner using ARMS framework.

PETSc

PETSc, which stands for Portable Extensible Toolkit for Scientific Computation, is a highly complete suite of data structures and routines for the solution of scientific applications related to partial differential equations. It supports MPI, and GPUs through CUDA or OpenCL, as well as hybrid MPI-GPU parallelism. Among its features, PETSc includes a number of Krylov space solvers for linear systems (CG, GMRES, BiCGStab, CGS, and others), as well as standard ILU, Jacobi, and AMG preconditioners.

Hypre

Developed at Lawrence Livermore National Laboratory, HYPRE is a library that offers a comprehensive suite of scalable solvers for large-scale scientific simulation, including parallel

[4] https://github.com/CognitionGuidedSurgery/ILUpp

multigrid methods for both structured and unstructured grid problems. The HYPRE library is highly portable and supports a number of languages.

Hypre contains several families of preconditioner algorithms focused on the scalable solution of very large sparse linear systems. For instance, it includes "grey-box" algorithms, such as structured multigrid, that use additional information to solve certain classes of problems more efficiently than general-purpose libraries.

It also offers a suite of common iterative methods, as the most commonly used Krylov-based iterative methods complemented with scalable preconditioners, such as ILU, AMG and AINV.

HiPS

HIPS (Hierarchical Iterative Parallel Solver) is a parallel solver for sparse linear systems developed by Jérémie Gaidamour and Pascal Hénon in the INRIA team-project "Scalaplix", in collaboration with Yousef Saad from the University of Minnesota. It uses a domain decomposition approach based on a Hierarchical Interface Decomposition (HID), based on building a decomposition of the adjacency graph of the system into a set of small subdomains with overlap [64]. This is similar to the application of nested dissection orderings used by ILUPACK to expose parallelism and generate a task tree. Using this partition of the problem, HIPS provides a solver that combines direct and iterative strategies to solve different parts. Specifically, it uses a direct factorization to solve the leading block, while using ILU-preconditioned iterative solvers for the Schur complement.

Trilinos

The Trilinos Project is an effort of the Sandia National Laboratory (USA), to provide a suite of algorithms for the solution of large-scale, complex multi-physics engineering and scientific problems.

Trilinos includes a wide range of iterative and direct solvers, preconditioners, high-level interfaces, and eigen-solvers, bundled in different software packages:

- **AztecOO: Preconditioners and Krylov subspace methods**
  Object oriented suite of standard Krylov iterative methods and preconditioners such as SOR, ILU, polynomial, and domain decomposition methods.
- **ShyLU: Hybrid iterative/direct Schur complement solver**
  ShyLU is an MPI and threaded framework designed to solve medium-size problems and to be used as a subdomain solver or smoother for very large problems within an iterative scheme. It can also be used as a black-box solver. Like HIPS, it uses a hybrid direct/iterative approach based on Schur complements.
- **Teko: Block preconditioning framework**
  A package for development and implementation of block preconditioners that includes support for manipulation and setup of block operators. It also features a small number of generic block preconditioners, including block Jacobi, and block Gauss-Seidel. For the Navier-Stokes equation, Teko has implementations of SIMPLE, PCD and LSC.

- **ML: smoothed aggregation algebraic multigrid**
  This package contains several parallel multigrid schemes for preconditioning or solving large sparse linear systems of equations arising from elliptic PDE discretizations.

Watson Sparse Matrix Package (WSMP)

WSMP is a software package, developed by the IBM T.J. Watson Research Center, targeted at the solution of large-scale sparse linear systems. It contains common Krylov subspace solvers as CG, GMRES, TFQMR, and BiCGStab. Accompanying these solvers, the package currently supports preconditioners as Jacobi, Gauss-Seidel, and Incomplete Cholesky/LDL^T preconditioners for SPD matrices. It supports Jacobi, Gauss-Seidel, and Incomplete LU factorization based preconditioners for general matrices. For the incomplete Cholesky preconditioner, the package allows the user to set the drop tolerance and fill factor. An automatic tuning mechanism is also provided.

MAGMA Sparse

MAGMA is an open-source project developed by Innovative Computing Laboratory (ICL), University of Tennessee, Knoxville, USA,

that includes a collection of state-of-the-art GPU accelerated routines for linear algebra, desinged to exploit heterogeneous CPU-GPU architectures, providing an interface similar to that of standar libraries like LAPACK and BLAS.

In addition to the extensive collection of dense linear algebra kernels, MAGMA includes a package to handle sparse computations. The package supports the most common sparse matrix formats, as CSR, ELL, and also supports the MAGMA-specific format SELL-P, and implements a comprehensive set of iterative linear solvers, eigensolvers, and preconditioners. Some examples are the CG, BiCG, GMRES, BiCGStab, with ILU0, ILUT, and Incomplete Sparse Approximate Inverse preconditioners.

CHAPTER 3

Enabling GPU computing in sequential ILUPACK

The preconditioner on which ILUPACK is based effectively reduces the number of iterations of common Krylov subspace solvers. Its convergence properties have been studied both with theoretical and empirical approaches, where it has proven to be superior to other types of ILU preconditioners in many moderate to large-scale scenarios. Unfortunately, these appealing properties come at the price of a highly complex construction process, and a multilevel structure that implies a computationally demanding application. The considerable increment in the cost of most solvers when using ILUPACK instead of simpler preconditioners severely limits the applicability and usefulness of the package. In other words, it is often preferable to perform many lightweight iterations with a simple ILU preconditioner than only a few ones with ILUPACK. Of course there are some cases in which simpler preconditioners do not allow the convergence of the solvers or achieve convergence in an impractical number of iterations, but this range of application is narrower than desired.

One alternative is then to boost the performance ILUPACK by means of HPC and parallel computing techniques. In this sense, there are two main strategies that can be followed, which are the exploitation of task-parallelism on the one hand, and data-parallelism on the other. The first strategy consists in dividing the work into several tasks such that two or more tasks can be executed concurrently in different computational units on the same or different sets of data. The second applies when the same function or operation can be applied to multiple data elements in parallel.

Previous efforts on the parallelization of ILUPACK have focused on the exploitation of task-parallelism. The intrinsic sequential structure of most Krylov subspace solvers and the multilevel preconditioner itself, however, severely constrains the execution schedule of the different tasks involved. To overcome this limitation, the task-parallel ILUPACK versions proposed in [3] and [4], respectively designed for shared-memory systems and for distributed-memory platforms, leverage the parallelism extracted by a Nested Dissection ordering of the coefficient matrix to divide both the construction and the application of the preconditioner

into tasks that can be executed concurrently on different processors. This strategy will be explained in more detail in the next chapter of this thesis, but at this point it is important to mention the two main drawbacks of this approach. First, the strategy makes some mathematical simplifications, and increases the amount of floating point operations necessary to construct and apply the preconditioner in order to expose additional concurrency. This results in a different preconditioner than that calculated by the sequential ILUPACK, which could mean that some of its beneficial numerical properties no longer apply. The second drawback is that the formulation of the task-parallel ILUPACK relies on the coefficient matrix being symmetric and positive-definite (SPD), and an extension of this strategy to general matrices is not trivial to derive.

This chapter investigates the exploitation of data-parallelism to accelerate the execution of ILUPACK by utilizing massively parallel processors, such as GPUs. Unlike the previous parallel versions, instead of trading floating-point operations (flops) for increased levels of task-parallelism, the rationale is to exploit the data-parallelism intrinsic to the major numerical operations in ILUPACK, off-loading them to the hardware accelerator, where they are performed via highly-tuned data-parallel kernels. This allows to accelerate ILUPACK without compromising its mathematical properties.

The chapter begins by describing the general strategy, which is presented in the context of accelerating the execution of SPD systems. The same strategy is applied later to deliver a baseline data-parallel version of ILUPACK's solvers for general (non-symmetric) and symmetric indefinite linear systems on GPUs. This is immediately followed by an experimental evaluation of these baseline versions.

Later, the chapter describes the extensions and enhancements to our baseline developments, which tackle three important problems of the baseline data-parallel versions of ILUPACK for sparse general linear systems. This is again accompanied by an experimental evaluation.

The major contributions of this chapter are the following:

- We introduce data-parallel GPU versions of the ILUPACK preconditioner and four of the most relevant sparse solvers bundled in the package: CG for SPD systems, GMRES and BiCG for general systems, and SQMR for symmetric indefinite systems. Our new solvers maintain the number of floating-point operations and, although minor differences appear in some cases[1], they also preserve the accuracy and convergence rate of the original routines.
- Our experimental analysis compares the performance advantages of the different methods using a number of real problems, in particular, from the SuiteSparse collection (formerly known as University of Florida Matrix Collection or UFMC [40]).
- The results show that the baseline versions of GPU-enabled solvers can efficiently exploit the hardware resources of state-of-the-art GPU platforms, especially for moderate and large problems, with speed-up values of up to 3×.
- In the case of the GMRES solver, we identify that, for many sparse linear systems, a key constraint to attain higher speed-ups with respect to the sequential ILUPACK

[1] Related with the use of floating-point arithmetic.

is the computation of the (modified) Gram-Schmidt re-orthogonalization (MGSO) on the CPU. In order to deal with this bottleneck, we design and integrate into GMRES an accelerated GPU-version of MGSO that amortizes the cost of host-device communication.

- The BiCG solver operates on two simultaneous recurrences, one involving A and the second one $\bar{A} = A^T$, requiring a sparse matrix-vector (SpMV) product per iteration on each matrix. While these SpMV operations can be computed in the GPU, via the appropriate invocations to Nvidia's cuSparse library for sparse linear algebra [100], in our experiments we noticed that the SpMV operating on the transpose matrix was considerably slower. To deal with this, we take advantage of the independence between the recurrences involving A and A^T, to execute the operations of each one in a different GPU, synchronizing when necessary. In particular, we leverage the duplicate memory capacity of the dual-GPU system to store a transposed copy of $\bar{A}$ (and the preconditioner) in one of the accelerators, so that we can rely on the faster version of SpMV in cuSparse that involves a non-transposed operand for both recurrences.
- Extending the work summarized in the previous point, we study the application of several techniques in order to further exploit the task and data-parallelism of the BiCG in hardware platforms equipped with a single GPU. In this sense, we explore the use of GPU streams and concurrent CPU and GPU computations, on the premise that, in single-GPU contexts, the use of the multi-core CPU could partially compensate the absence of a second GPU. We also evaluate the use of a recent synchronization-free strategy for the solution of sparse triangular linear systems, proposed in [45] and summarized in Appendix A of this document. Although this technique does not always improve the runtime attained by cuSparse for the solution of triangular systems involved in ILUPACK, it can favour the overlapping of operations between CPU and GPU.
- We augment the family of Krylov subspace iterative methods for sparse general linear systems accelerated with an ILUPACK preconditioner with a data-parallel version of the Bi-Conjugate Gradient Stabilized Method (BiCGStab). To accomplish this, we first develop a CPU version of the solver to then produce a variant that runs entirely on the graphics accelerator.

3.1 Platforms and test cases

The experiments conducted to test the performance of the developments presented in this chapter were performed on different hardware platforms using several test cases. Since some of the contributions to be presented in the following sections were motivated by experimental results extracted from the baseline versions, the evaluation of the baseline is presented before the enhancements motivated by these results. In order to facilitate the reading, in this section we compile all the platforms and test cases that will appear later in the remaining of the chapter.

3.1.1 Hardware and software platforms

Next we describe the hardware platforms and software used in this chapter.

Bach

Bach is a server equipped with an Intel i7-2600 CPU (4 cores at 3.4 GHz), 8 MB of L3 cache and 8 GB of RAM. The GPU in this platform is an Nvidia S2070, of the Fermi generation, with 448 cores at 1.15 GHz and 5 GB of GDDR5 RAM. The CUDA Toolkit, which includes the compiler and accelerated libraries such as cuBlas and cuSparse, is version 4.1. The C and Fortran compiler employed for the CPU codes was gcc v4.4.6, and the operating system on the server is CentOS Rel. 6.2.

Mozart

This platform features a low-end Intel i3-3220 CPU (2 cores at 3.3 GHz), with 3 MB of L3 cache and 16 GB of RAM. Although the CPU in Bach is clearly more advanced, the performance of both CPUs in single-thread codes is comparable. The server includes a Nvidia K20 GPU of the Kepler architecture. It contains 2,496 cores at 0.71 and 6 GB of GDDR5 RAM. The version of the CUDA Toolkit in this case is 5.0. C and Fortran compiler is gcc v4.4.7. The operative system is CentOS Rel. 6.4.

Beethoven

Beethoven presents an Intel i7-4770 processor (4 cores at 3.40 GHz) and 16 GB of DDR3 RAM (26 GB/s of bandwidth), connected to an NVIDIA Tesla K40 GPU, with 2,880 cuda cores at 0.75 GHz, and 12 GB of DDR5 RAM (288 GB/s bw). NVIDIA CUBLAS/CUSPARSE 6.5 was employed in the experimentation. For the CPU codes we used gcc v4.9.2 with `-O4` as an optimization flag.

Brahms

This platform has an Intel(R) Xeon(R) CPU E5-2620 v2 (6 cores at 2.10GHz), and 128 GB of DDR3 RAM memory. The platform also contains two NVIDIA "Kepler" K40m GPUs, each with 2,880 CUDA cores and 12 GB of GDDR5 RAM. The CPU codes were compiled with the Intel(R) Parallel Studio 2016 (update 3) with the `-O3` flag set. The GPU compiler and the CUSPARSE library were those in version 6.5 of the CUDA Toolkit.

3.1.2 Test cases

SuiteSparse test cases

We selected a set of matrices from the SuiteSparse matrix collection that comprises three medium to large-scale SPD matrices, five large-scale non-symmetric matrices with dimension $n > 1,000,000$, and six symmetric indefinite ones with $n > 100,000$. The dimension, number of non-zeros and sparsity ratio of these matrices is displayed in Table 3.1.

	Matrix	n	nnz	nnz/n
SPD	ldoor	952,203	23,737,339	24,93
	thermal2	1,228,045	4,904,179	3,99
	G3_circuit	1,585,478	4,623,152	2,92
Symmetric Indefinite	darcy003	389,874	2,097,566	5.38
	F1	343,791	26,837,113	78.06
	mario002	389,874	2,097,566	5.38
	cbig	345,241	2,340,859	6.78
	nlpkkt120	3,542,400	95,117,792	26.85
	nlpkkt160	8,345,600	225,422,112	27.01
Non-symmetric	cage14	1,505,785	27,130,349	18.02
	memchip	2,707,524	13,343,948	4.93
	Freescale1	3,428,755	17,052,626	4.97
	rajat31	4,690,002	20,316,253	4.33
	cage15	5,154,859	99,199,551	19.24

Table 3.1: Matrices from the SuiteSparse collection used in the experiments.

SPD and Symmetric Indefinite PDE

We considered the Laplacian equation $\Delta u = f$ in a 3D unit cube $\Omega = [0,1]^3$ with Dirichlet boundary conditions $u = g$ on $\delta\Omega$. The discretization consists in a uniform mesh of size $h = \frac{1}{N+1}$ obtained from a seven-point stencil. The resulting SPD linear system $Au = b$ has a sparse SPD coefficient matrix with seven nonzero elements per row, and $n = N^3$ unknowns. The problem is set to generate five benchmark SPD linear systems of order $n \approx$ 1M, 1.9M, 3.3M, 8M and 16M. In the experiments related with the SQMR solver, a random set of eigenvalues of these matrices were modified by changing their sign, so that the resulting problem becomes indefinite. We display the dimension and number of non-zeros of each problem instance in Table 3.2.

Matrix	Dimension n	Non-zeros nnz	nnz/n
A50	125,000	492,500	3.98
A159	4,019,679	16,002,873	3.98
A171	5,000,211	19,913,121	3.98
A200	8,000,000	31,880,000	3.99
A252	16,003,008	63,821,520	3.99
A318	32,157,432	128,326,356	3.99
A400	64,000,000	255,520,000	3.99
G3_Circuit	1,585,478	7,660,826	4.83

Table 3.2: Matrices employed in the experimental evaluation.

Non-symmetric Convection-Diffusion Problems (CDP)

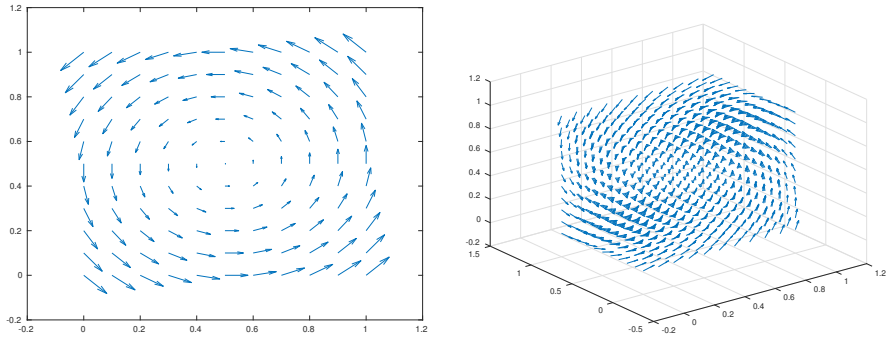
We considered the PDE $\varepsilon\Delta u + b * u = f$ in Ω, where $\Omega = [0,1]^3$. For this example, we use homogeneous Dirichlet boundary conditions, i.e. $u = 0$ on $\partial\Omega$. The diffusion coefficient ε is

set to 1, and the convective functions $b(x, y, z)$ are given by:

$$\begin{array}{ll} \text{conv. in } x\text{-direction:} & [1, 0, 0], \\ \text{diagonal convection:} & \frac{1}{\sqrt{3}}[1, 1, 1], \\ \text{circular convection:} & [\frac{1}{2} - z, x - \frac{1}{2}, \frac{1}{2} - y]. \end{array}$$

The domain is discretized with a uniform mesh of size $h = \frac{1}{N+1}$ resulting in a linear system of size N^3. For the experiments we set $N = 200$. For the diffusion part $-\varepsilon\Delta u$ we use a seven-point-stencil. The convective part $b * u$ is discretized using up-wind differences.

We display the circular convective function in 2D/3D for illustration.



3.2 Baseline Data-Parallel variants of ILUPACK

In this section we describe our general strategy to obtain GPU-accelerated versions of the Krylov space solvers bundled in ILUPACK. We focus our efforts mainly on the application of the multilevel preconditioner, since it is the main computational bottleneck in most cases.

ILUPACK supports several matrix types, and provides a wide range of solvers and preconditioners to cover their specificities. For example, for SPD matrices ILUPACK includes an implementation of the CG method (see Section 2.2.2), and a preconditioner based on an LDL^T, where the L matrix is unit-lower triangular and D is diagonal. In the case of symmetric indefinite matrices, the solver provided by ILUPACK is the Symmetric Quasi-Minimal Residual –SQMR– method, and the preconditioner is again of the form LDL^T, but with D being block diagonal due to the application of the Bunch-Kaufman pivoting strategy [65]. For the non-symmetric case, the package presents implementations of the BiCG and GMRES algorithms, while the preconditioner is of the form LDU, with L unit-lower triangular, U unit-upper triangular, and D diagonal.

Each of these preconditioners is computed by an specific routine and is stored in a slightly different sparse matrix representation. Nevertheless, the construction of the different variants of the ILUPACK preconditioner, and thus the application of each variant, follow a procedure similar to that described in Section 2.4.

ILUPACK's multilevel preconditioner is stored as a linked list of structures that contain the information computed at each level. Concretely, a level contains pointers to the submatrices that form the ILU factorization: the B submatrix that comprises the LDU factored upper left block and the G, F rectangular matrices, along with the diagonal scaling and per-

mutation vectors that correspond to $\tilde{D}$, $\tilde{P}$, and $\hat{P}$; see Section 2.4. The case of symmetric matrices is analogous, and the differences reside in that only the lower triangular factor L and the (block-)diagonal D of the LDL^T factorization are stored explicitly, and that G is not stored because it is equal to F^T.

The computational cost required to apply the preconditioner is dominated by the sparse triangular system solves (SpTrSV) and SpMVs. Nvidia cuSparse [100] library provides efficient GPU implementations of these two kernels that support several common sparse matrix formats. Therefore, it is convenient to rely on this library. The rest of the operations are mainly vector scalings and re-orderings, which gain certain importance only for highly sparse matrices of large dimension, and are accelerated in our codes via *ad-hoc* CUDA kernels.

In the following we provide some details about the work performed in order to enable the use of the GPU in each of the main types of operations: LDU systems, matrix-vector products and vector operations.

3.2.1 Solution of LDL^T and LDU linear systems

Given the modified-CSR (MCSR) storage layout adopted by ILUPACK for the LDL^T and LDU factors, and the native format handled by the cuSparse library (CSR), a layout reorganization is necessary before the corresponding kernel can be invoked. In the current implementation, this process is performed by the CPU, during the calculation of the preconditioner.

In the case of SPD systems, after the transformation, the LDL^T factors are transferred and stored in the GPU as a matrix $\hat{L} = LD^{1/2}$ in (plain) CSR format. Since cuSparse provides a triangular system solver for matrices in this format, a system of the form $LDL^T x = b$ is tackled by first solving $\hat{L}y = b$ for y, and then $\hat{L}^T x = y$ for x. This is performed via two consecutive calls to routine `cusparseDcsrsv_solve`, with the appropriate arguments to operate with the triangular coefficient matrix $\hat{L}$ or its transpose $\hat{L}^T$. The analysis phase required by the CUSPARSE solver, which gathers information about the data dependencies and aggregates the rows of the triangular matrix into levels, is executed only once for each level of the preconditioner, and it runs asynchronously with respect to the host CPU.

In the symmetric-indefinite case, ILUPACK's MCSR representation stores the inverse of the symmetric block-diagonal matrix D, using $2nB$ floating point numbers (with nB denoting the size of the leading block), while the rest of the structure contains $\hat{L} \in \mathbb{R}^{nB \times nB}$, that is, the strictly lower triangle of L, in CSR format. ILUPACK then solves a system of the form $(\hat{L}D^{-1} + I_{nB})D(\hat{L}D^{-1} + I_{nB})^T$. Those columns that correspond to 2×2 pivots are stored interleaved, since in $\hat{L}D^{-1}$ they have the same nonzero pattern and this is exploited by the serial CPU solver.

To solve these systems using cuSparse, we split their structure into a symmetric tridiagonal matrix D^{-1}, which we store as two vectors of size nB, and then form matrix $\tilde{L} = (\hat{L}D^{-1} + I_{nB})$ explicitly. After this transform, at each level, we solve linear systems with coefficient matrix of the form $\tilde{L}D^{-1}\tilde{L}^T$. Here, as the inverse of D is available, this involves a tridiagonal matrix-vector product, which is performed in the GPU by means

of a simple *ad-hoc* CUDA kernel.

In the case of ILUPACK's non-symmetric preconditioner, the leading block B is stored in MCSR format with the L and U factors kept in an interlaced manner. Concretely, the vector containing the diagonal entries in the MCSR structure holds the inverses of the diagonal elements of U, while the CSR part of the structure holds each column of the strict-lower triangle of L (as it is unit-diagonal) followed by the corresponding row of U. The integer vectors of the CSR part are adapted accordingly.

In order to use cuSparse, we thus need to split each B submatrix into separate L and U factors, stored by rows in the conventional CSR format. As in the previous cases, this transform is done only once, during the calculation of each level of the preconditioner, and occurs entirely in the CPU. After that, the L and U factors, in CSR format, are transferred to the GPU, where the triangular systems involved in the preconditioner application are solved via two consecutive calls to `cusparseDcsrsv_solve`.

3.2.2 Matrix-vector products

In order to compute the SpMV operations involved in the application of the preconditioner on the GPU, G and F are also transferred to the device during the construction phase. As these matrices are stored by ILUPACK in CSR format, no reorganization is needed prior to the invocation of the cuSparse kernel for SpMV.

However, in the SPD case, the products that need to be computed during the application of the preconditioner, at each level, are of the form $x := Fv$ and $x := F^T v$. As both the matrix and its transpose are involved in the calculations, two different strategies can be applied. In particular, the cuSparse kernel for this operation, `cusparseDcsrmv`, includes an argument (switch) that allows to multiply the vector with the transpose of the matrix passed to the function. This makes it possible to store only F in the GPU, but compute both forms of the product by setting the appropriate value for the transpose switch. Unfortunately, the implementation of SpMV in cuSparse delivers considerable low performance when operating with the transposed matrix.

For these reasons we allocated both F and F^T in the GPU, using a cuSparse routine to transpose the matrix, which was done only once for each level of the preconditioner. The memory overhead incurred for storing the transpose explicitly is unavoidable in the non-symmetric case, but we find this performance-storage trade-off acceptable.

A similar situation occurs in the case of BiCG solver, as the application of the preconditioner involves G and F as well as their respective transposes. In this case, storing both transposes can exceed significantly the memory requirements of the non-symmetric versions of ILUPACK, as two additional matrices need to be stored, unlike in the symmetric case, where only one extra matrix is needed. For this reason, our approach in this case only keeps G and F in the accelerator, and operates with the transposed/non-transposed matrices by setting the appropriate value for the transpose switch.

3.2.3 Vector operations

As mentioned earlier, the solution of the triangular systems and the matrix-vector multiplications that appear at each level of the preconditioner application are the most time-consuming operations in most problems. On the other hand, although the remaining vector operations involved in the application of the preconditioner are not computationally-intensive, if they are performed on the CPU, it is necessary to transfer their results to the GPU when they are needed as well as to retrieve the results of the triangular system solves and SpMV to the CPU. Furthermore, as the "recursive" application the preconditioner consists of a strictly ordered sequence of steps (where concurrency is extracted from the operations that compose each one of these steps), no overlapping between data transfers and calculations is possible.

For this reason, we off-load the entire preconditioner application to the GPU. This implies that the residual r_{k+1} is transferred to the GPU before the operations commences, and then all levels of the preconditioner are processed in the accelerator to obtain $z_{k+1} := M^{-1}r_{k+1}$. After the operation is completed, the preconditioned residual z_{k+1} is retrieved back to the CPU. To make this possible, we implemented three additional GPU kernels:

- The *diagonal scaling kernel* multiplies each entry of an input vector by the corresponding entry of a scaling vector. This is equivalent to multiplying a diagonal matrix –as scaling vector– by the input vector.
- The *ordering kernel* reorganizes an input vector v_{in} by applying a vector permutation p, and produces an output vector v_{out}, with entries $v_{out}[i] := v_{in}[p(i)]$.
- The third kernel simply implements the subtraction $c := a - b$ where a, b and c are vectors.

By performing these operations in the GPU we can avoid some data transfers and reduce the runtime significantly, especially for those cases where the size of the vectors is rather large compared with the number of non-zeros of the triangular factors.

These three kernels appear in every variant of the preconditioner application routine.

3.2.4 Parallelization of SpMV and other kernels

In addition to the parallelization of the preconditioner, we further enhanced the solvers in ILUPACK by off-loading the SpMV required by the solver to the GPU. For this purpose, it is necessary to store A in the GPU. In our implementation this matrix is transferred to the GPU memory before the iterative solve commences, and resides there until completion. The matrix is stored in CSR format and the SpMV is performed via the implementation of this kernel in cuSparse. The transposed SpMV of the BiCG is performed using the transposed variant of cuSparse`csrMv` routine.

In general, the level-1 Blas operations of the solvers contribute little to the overall computational cost, which is dominated completely by the application of the preconditioner. Therefore, these operations are performed in the CPU in our baseline versions.

3.2.5 Experimental evaluation of baseline parallel versions

Next we present the experimental evaluation of our baseline GPU-aware ILUPACK solvers. We start by analyzing the results for the SPD case, to then evaluate the solvers for indefinite and non-symmetric systems.

As more modern GPUs became available during the development of these variants, the evaluation uses a variety of platforms. The evaluation of the SPD variant was performed in platforms BACH and MOZART, while the rest of the solvers were tested in platform BEETHOVEN (see Section 3.1).

All the results in this section were obtained using IEEE double-precision arithmetic. In all cases, the total runtime includes the cost of transferring the matrix to the GPU.

Evaluation of CG

Table 3.3 compares the original CPU-based implementation of ILUPACK against our GPU-aware version BASE_PCG, using the SPD cases of the benchmark collection in BACH and MOZART hardware platforms. In particular, in the results we detail the number of iterations needed for convergence (label "#iter"); the execution times spent during the preconditioner application in the solution of triangular systems with coefficient matrix of the form LDL^T, the SPMV operations, the CPU-GPU data transfers and the total preconditioner application time (labels "LDL^T time", "SPMV time", "Transfer time" and "Preconditioner time", respectively); the total solver time (label "Total PCG time"); and the relative residual

$$\mathcal{R}(x^*) := \frac{||b - Ax^*||_2}{||x^*||_2}, \tag{3.1}$$

where x^* stands for the computed solution.

The results in the table show that our GPU-based version and the original CPU version required the same number of iterations to converge in all experiments. Additionally, the differences between the attained residual errors are very small, of a magnitude that can be easily explained by the unavoidable floating-point errors. These results confirm that both data-parallel GPU-based versions preserve the numerical properties of the original ILUPACK solver.

From the performance point of view, the new GPU-based version exhibits an important reduction of the preconditioner application time with respect to the original ILUPACK solver for all test cases and both hardware platforms. The results in Table 3.3 report that BASE_PCG outperforms the CPU implementation of ILUPACK, in factors varying from 1.25× to 5.84×, depending on the case/platform, and that these speed-ups tend to increase with the size of the problem. This responds to the fact that, for small problems, it is difficult to fully exploit the cores of the GPU and hide the important latency of memory operations. In this sense, the platform equipped with a "Fermi" GPU offers lower execution times than the "Kepler"-based server only for the LDOOR case (smallest matrix). This is due to the "Fermi" architecture (448 CUDA cores at 1.15 GHz) being more appropriate when the problem features a low degree of concurrency. In all other cases, though, the "Kepler" processor delivers higher performance.

Table 3.3: Experimental results for the baseline GPU-enabled implementation of ILUPACK's CG method (BASE_PCG) in MOZART and BACH. Times are in milliseconds.

Platform	Matrix	Device	#iter	Time					$\mathcal{R}(x^*)$
				LDL^T	SPMV	Transfer	Preconditioner	Total PCG	
MOZART	ldoor	CPU	200	101,713	8,646	-	114,548	136,040	4.845E-07
		GPU	200	74,805	1,191	1,074	77,291	96,880	4.963E-07
	thermal2	CPU	186	18,149	5,871	-	30,097	40,060	2.397E-08
		GPU	186	7,525	687	1,263	9,792	19,850	2.392E-08
	G3_Circuit	CPU	75	8,364	3,120	-	14,844	18,290	2.674E-06
		GPU	75	3,603	385	645	4,787	8,200	2.674E-06
	A126	CPU	44	15,122	3,442	-	21,056	23,660	5.143E-09
		GPU	44	10,359	444	469	11,381	14,010	5.143E-09
	A159	CPU	52	37,097	8,616	-	51,973	58,220	2.751E-09
		GPU	52	17,351	1,063	1,094	19,751	26,080	2.751E-09
	A200	CPU	76	70,275	30,370	-	124,123	142,880	5.618E-09
		GPU	76	20,688	3,532	3,172	28,276	47,020	5.618E-09
	A252	CPU	338	383,617	98,296	-	652,209	815,950	5.721E-08
		GPU	338	72,262	5,763	28,240	111,754	279,780	5.721E-08
BACH	ldoor	CPU	200	90,199	7,637	-	101,381	119,800	4.845E-07
		GPU	200	74,147	1,373	552	76,337	88,240	4.842E-07
	thermal2	CPU	186	16,021	5,126	-	26,617	34,730	2.397E-08
		GPU	186	10,948	987	658	13,024	21,450	2.302E-08
	G3_Circuit	CPU	75	7,138	2,725	-	12,762	15,340	2.674E-06
		GPU	75	4,610	546	339	5,704	8,380	2.675E-06
	A126	CPU	44	13,509	2,972	-	18,688	20,680	5.143E-09
		GPU	44	10,720	541	247	11,673	13,760	5.143E-09
	A159	CPU	52	32,648	7,424	-	45,671	50,410	2.751E-09
		GPU	52	19,814	1,234	575	22,022	26,010	2.751E-09
	A200	CPU	76	60,513	25,626	-	106,540	120,830	5.618E-09
		GPU	76	32,245	4,263	1,674	39,536	54,670	5.618E-09
	A252	CPU	338	264,697	74,812	-	476,384	602,160	5.721E-08
		GPU	338	106,944	6,527	14,795	136,918	266,690	5.721E-08

Evaluation of SQMR

The results of the evaluation of the GPU-aware symmetric-indefinite variant of ILUPACK (BASE_SQMR) are summarized in Table 3.4. The first part contains the experiments with the modified symmetric PDE of scalable size presented in Section 3.1, and illustrates a performance advantage of the GPU solvers that grows with the size of the instances. For this set of matrices, the acceleration of the SPMV is about 3×, and the preconditioning stage is improved by around 4× for the larger test cases, but the unaccelerated stages of the solver represent more than 10% of the total runtime and keep the global speedups below 3×. In most of the test cases extracted from the SuiteSparse collection, shown in the second part of the table, the preconditioner is not able to converge for $\tau > 0.01$. Decreasing the value of τ introduces a large amount of fill-in in the preconditioner, reducing the degree of data-parallelism. Figure 3.1 shows the sparsity pattern of the L factor that corresponds to each level of the preconditioner for the problem *c-big*. The plots illustrate that the fill-in in the L factor increases dramatically from the first to the second level of the factorization. This has two main effects. On the one hand, the new non-zero elements are likely to generate data dependencies between the rows of the L factor during the solution of the triangular linear systems, which severely harms the performance of CUSPARSE's level-based solver. On the other hand, as the dimension of the systems grow, the important memory requirements turn increasingly difficult to store the necessary matrices in the GPU, and thus the symmetric instances presented in the middle section of the table are all of intermediate size. The little improvement obtained for this set of matrices is mostly due to the speedup of the SPMV.

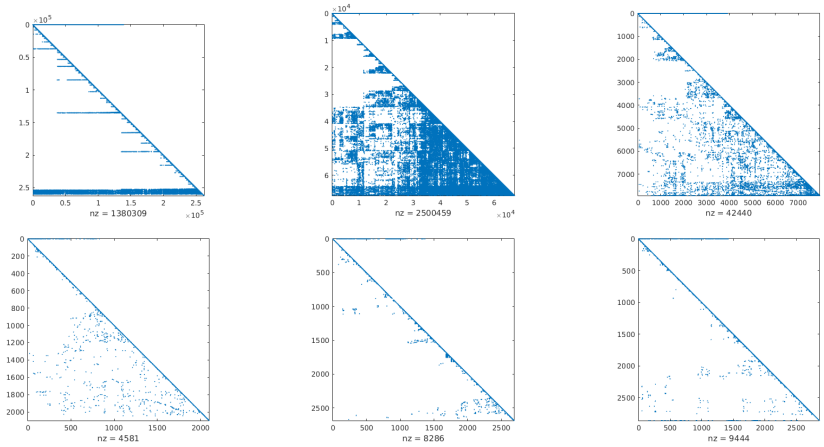


Figure 3.1: L factor of each level of the LDL multilevel factorization of matrix *c-big*. Levels increase from left to right and from top to bottom.

The largest symmetric indefinite instance we were able to test was a non-linear programming problem from the SuiteSparse collection, also studied in [96]. The results for this benchmark are closer to those obtained in the symmetric indefinite PDE. There are four instances of this problem, which vary in size. When $\tau = 0.01$, the fill-in of the preconditioner allows only the three smaller instances to be executed in the GPU.

To close this discussion, we note that, contrary to the results obtained in the previous experiments, there are discrepancies in the number of iterations as well as residual errors for some of the tested instances. These could be caused by floating point errors and the use of Buch-Kaufman pivoting, but further tests are required.

Evaluation of GMRES and BiCG

For the evaluation of our GPU implementations for non-symmetric matrices, we first applied the BiCG and GMRES methods to the SPD matrices associated with the Laplacian PDE, treating them as if they were non-symmetric. The test instances in this benchmark can be scaled up arbitrarily while preserving certain pattern in the non-zeros structure of the factorization. The results in Table 3.5 show an important improvement in the performance of the two GPU-accelerated solvers, though this is more notorious for BiCG. The reason is that the only stages of the solver that are off-loaded to the accelerator are the application of the preconditioner and SpMV. The first one occurs twice per iteration for BiCG (as the transposed preconditioner also has to be applied), but only once for GMRES. If we consider the stages that involve the preconditioner, the acceleration factor reaches up to 6× for the transposed preconditioner in the largest test case. This is not surprising, given the memory-bound nature of the problem and that the GPU has a memory bandwidth only 11× higher than the CPU. The matrices of this set are SPD and well-conditioned, allowing us to use a drop tolerance factor $\tau = 0.1$ and still converge in a few iterations. This arguably high value of τ produces a sparser preconditioner, exposing a larger volume of data-parallelism that is exploited by the GPU kernels.

To expose the performance of the non-symmetric solvers, we repeated the evaluation with a set of large non-symmetric problems from the SuiteSparse collection. Table 3.5 reports

Table 3.4: Experimental results for the baseline GPU-enabled implementation of ILUPACK's SQMR method (BASE_SQMR) in platform BEETHOVEN. Times are in seconds.

Matrix	Device	#Iters.	Time				$\mathcal{R}(x^*)$	Speed-up
			SPMV	M^{-1}	Rem.	Total		Total
A050	CPU	40	0.04	0.29	0.03	0.37	7.90E-09	
	GPU	40	0.02	0.28	0.04	0.34	7.90E-09	1.09
A100	CPU	72	0.79	5.00	0.71	6.51	3.00E-08	
	GPU	72	0.26	1.72	0.77	2.76	3.00E-08	2.35
A159	CPU	114	5.03	34.78	4.65	44.49	4.10E-08	
	GPU	114	1.67	9.32	4.88	15.89	4.10E-08	2.79
A200	CPU	137	12.27	85.07	11.19	108.56	3.00E-08	
	GPU	137	4.07	21.30	12.16	37.56	3.00E-08	2.89
A252	CPU	170	31.37	201.91	27.33	260.66	6.60E-08	
	GPU	170	9.81	51.89	28.59	90.35	4.50E-08	2.88
darcy003	CPU	88	0.68	2.73	0.26	3.68	4.00E-08	
	GPU	88	0.12	2.02	0.33	2.48	3.60E-08	1.48
F1	CPU	477	23.03	33.90	1.35	58.29	1.30E-07	
	GPU	477	1.74	38.72	1.55	42.01	1.40E-07	1.39
c-big	CPU	22	0.11	0.87	0.06	1.06	1.10E-09	
	GPU	22	0.12	0.72	0.09	0.93	1.10E-09	1.13
mario002	CPU	88	0.63	2.58	0.26	3.48	4.00E-08	
	GPU	88	0.13	2.03	0.34	2.51	3.90E-08	1.38
nlpkkt120	CPU	187	24.35	73.02	6.34	103.73	1.40E-06	
	GPU	176	3.67	24.24	6.86	34.78	4.40E-06	2.98
nlpkkt160	CPU	252	82.92	252.22	21.77	356.95	4.40E-06	
	GPU	252	12.34	73.26	22.51	108.14	4.50E-06	3.30

fair acceleration factors for the GPU versions of the solvers. In some detail, the speed-up values for the application of the preconditioner are quite similar to those attained for the Laplace matrices set, but the improvement experienced by the SPMV kernel is larger in all cases.

Next, we tested our solver on the non-symmetric CDP cases (see Section 3.1). In these experiments, the parallel versions outperform the serial CPU solvers, with speed-ups in the order of 2×, while the acceleration of the preconditioner is almost 3×.

Comparing both solvers, the acceleration attained by GMRES is always lower than that observed for BiCG. This can be easily explained by noting that, in BiCG, the GPU-accelerated stages (preconditioner application and SPMV) take most of the execution time. For GMRES, the time of the unaccelerated stages is more important, to the extent that, in some cases, the unaccelerated steps of the GPU versions consume a higher fraction of the time than the accelerated ones. A detailed analysis revealed that, in these cases, the bottleneck of the GMRES method is the modified Gram-Schmidt re-orthogonalization. Regarding the quality of the computed solution x^*, the GPU-enabled solvers converge in the same number of iterations and present the same final relative residual error (calculated according to 3.1) as the original version of ILUPACK.

3.3 Enhanced data-parallel variants

The experimental evaluation of our baseline data-parallel solvers reveals opportunities for further improvement. The most clear case is that of GMRES, where after accelerating the application of the preconditioner, the rest of the operations of the solver become the most

Table 3.5: Experimental results for the baseline GPU-enabled implementation of ILUPACK's BiCG and GMRES methods (Base_GMRES and Base_BICG) in platform Beethoven. Times are in seconds.

Solver	Matrix	Device	#Iters.	Time					$\mathcal{R}(x^*)$	Speed-up		
				SpMV	M^{-1}	M^{-T}	Rem.	Total		M^{-1}	M^{-T}	Total
BiCG	A050	CPU	16	0.010	0.016	0.017	0.008	0.05	1.80E-09	-	-	-
		GPU	16	0.009	0.012	0.010	0.008	0.04	1.80E-09	1.35	1.62	1.31
	A100	CPU	14	0.07	0.18	0.21	0.08	0.54	5.80E-09	-	-	-
		GPU	14	0.06	0.05	0.05	0.08	0.24	5.80E-09	3.25	4.60	2.25
	A159	CPU	14	0.29	0.67	0.63	0.31	1.90	4.90E-09	-	-	-
		GPU	14	0.24	0.20	0.18	0.32	0.93	4.90E-09	3.43	3.79	2.04
	A200	CPU	14	0.60	1.35	1.25	0.62	3.82	5.30E-09	-	-	-
		GPU	14	0.47	0.39	0.35	0.63	1.84	5.30E-09	3.45	3.78	2.08
	A252	CPU	14	1.15	3.40	4.13	1.23	9.92	5.70E-09	-	-	-
		GPU	14	0.93	0.77	0.69	1.22	3.62	5.80E-09	4.39	6.27	2.74
	cage14	CPU	12	0.60	0.72	0.75	0.13	2.20	2.70E-09	-	-	-
		GPU	12	0.21	0.19	0.16	0.10	0.68	2.70E-09	3.78	4.69	3.24
	Freescale1	CPU	292	15.24	29.52	45.98	5.71	96.44	1.00E-03	-	-	-
		GPU	292	6.92	5.04	10.17	4.82	26.96	1.00E-03	5.86	4.52	3.58
	rajat31	CPU	8	0.48	0.93	0.86	0.29	2.55	1.40E-06	-	-	-
		GPU	8	0.16	0.19	0.29	0.22	0.88	1.40E-06	4.89	2.97	2.90
	cage15	CPU	12	2.25	2.83	3.28	0.43	8.78	5.50E-09	-	-	-
		GPU	12	0.86	0.60	0.51	0.35	2.33	5.50E-09	4.72	6.43	3.77
	CDP/circ	CPU	286	18.61	82.22	107.58	11.27	219.68	1.20E-07	-	-	-
		GPU	286	14.18	18.86	42.58	11.35	86.97	1.20E-07	4.36	2.53	2.53
	CDP/diag	CPU	298	19.37	78.44	81.87	11.59	191.27	2.00E-07	-	-	-
		GPU	298	14.48	19.47	44.24	11.58	89.77	2.00E-07	4.03	1.85	2.13
	CDP/u-vec	CPU	316	20.50	83.05	86.71	12.29	202.55	4.10E-08	-	-	-
		GPU	316	15.46	20.59	46.85	12.42	95.33	4.10E-08	4.03	1.85	2.12
GMRES	A050	CPU	9	0.006	0.019	-	0.018	0.043	9.20E-10	-	-	-
		GPU	9	0.005	0.013	-	0.017	0.035	9.20E-10	1.43	-	1.23
	A100	CPU	8	0.04	0.22	-	0.15	0.42	4.60E-09	-	-	-
		GPU	8	0.03	0.06	-	0.15	0.24	4.60E-09	3.60	-	1.75
	A159	CPU	8	0.16	0.97	-	0.66	1.79	4.10E-09	-	-	-
		GPU	8	0.10	0.23	-	0.66	0.99	4.10E-09	4.22	-	1.81
	A200	CPU	8	0.33	1.90	-	1.34	3.57	4.00E-09	-	-	-
		GPU	8	0.20	0.45	-	1.34	1.99	4.00E-09	4.22	-	1.79
	A252	CPU	8	0.63	3.13	-	2.58	6.34	3.90E-09	-	-	-
		GPU	8	0.41	0.89	-	2.59	3.88	3.90E-09	3.52	-	1.63
	cage14	CPU	7	0.36	0.84	-	0.26	1.46	2.40E-09	-	-	-
		GPU	7	0.05	0.22	-	0.21	0.49	2.40E-09	3.82	-	2.98
	Freescale1	CPU	46	2.70	9.32	-	6.19	18.21	6.30E-03	-	-	-
		GPU	46	0.54	1.60	-	5.18	7.33	6.30E-03	5.83	-	2.89
	rajat31	CPU	4	0.27	0.93	-	0.53	1.74	3.60E-07	-	-	
		GPU	4	0.06	0.19	-	0.41	0.67	3.60E-07	4.89	-	2.60
	cage15	CPU	7	1.31	3.30	-	0.91	5.52	4.80E-09	-	-	-
		GPU	7	0.18	0.70	-	0.71	1.61	4.80E-09	4.65	-	3.43
	CDP/circ	CPU	203	12.84	116.44	-	63.32	192.61	1.40E-06	-	-	-
		GPU	203	5.65	26.75	-	62.97	95.37	1.40E-06	2.27	-	2.02
	CDP/diag	CPU	241	15.25	127.66	-	75.89	218.81	1.60E-06	-	-	-
		GPU	241	6.73	31.59	-	76.21	114.52	1.60E-06	2.27	-	1.91
	CDP/u-vec	CPU	251	15.69	131.47	-	79.67	226.85	1.40E-06	-	-	-
		GPU	251	7.00	32.70	-	79.91	119.61	1.40E-06	2.24	-	1.90

Operation	kernel	Computed in...
$A \rightarrow M$ Initialize $x_0, r_0, q_0, p_0, s_0, \rho_0, \tau_0; k := 0$ **while** $(\tau_k > \tau_{\max})$	Compute preconditioner	
$\alpha_k := \rho_k / (q_k^T A p_k)$ $x_k := x_k + \alpha_k p_k$ $r_k := r_k - \alpha_k A p_k$ $t_k := M^{-1} r_k$	SpMV + DOT product AXPY AXPY Apply preconditioner	GPU A
$z_k := M^{-T} A^T q_k$	SpMV + apply prec.	GPU B
synchronization $s_{k+1} := s_k - \alpha_k z_k$ $\rho_{k+1} := (s_{k+1}^T r_k) / \rho_k$ $p_{k+1} := t_k + \rho_{k+1} p_k$ $q_{k+1} := s_{k+1} - \rho_{k+1} q_k$ $\tau_{k+1} :=\| r_k \|_2$ $k := k + 1$ **end while**	 AXPY DOT product AXPY AXPY DOT product	CPU

Figure 3.2: Algorithmic formulation of the preconditioned BiCG method. The steps have been re-arranged so that the two sequences that compose the method can be isolated and executed in different devices.

time-consuming stage. In the case of BiCG, the application of the transposed preconditioner is often much slower than that of the non-transposed variant. This is related to the use of cuSparse transposed operations, and especially the transposed version of the matrix-vector product which, in general, is much slower than the non-transposed variant.

In this section we analyze the principal bottlenecks that arise after accelerating the preconditioner application of sequential ILUPACK. Then we propose new versions of the corresponding solvers in order to overcome these new limitations. Finally we perform an experimental assessment of the advanced parallel variants.

3.3.1 Coarse-grain parallel version of BiCG for dual-GPU systems

Our initial data-parallel version of the BiCG method off-loaded the SpMV (with A and A^T) as well as the application of the preconditioner (and its transpose) to the GPU [6]. An important issue that severely constrained the performance of this GPU-version of BiCG was the need to operate with A^T, F^T, G^T in one of the recurrences of the iterative solver. As highlighted previously, our implementation stored only A, F and G in the GPU memory and relied on a parameter of the `csrMv` routine from cuSparse to perform the transposed SpMV. Unfortunately, the implementation of this kernel in cuSparse offers much lower performance when invoked with a transposed matrix operand.

The left-hand side column of Figure 3.2 offers an algorithmic description of BiCG, and Table 3.6 reviews the execution time of the main operations performed inside BiCG, using the baseline accelerated data-parallel version from Section 3.2. This evaluation shows that the calls to SpMV that operate with A^T are, on average, 2–3× slower than those working with the non-transposed matrix, with one special case for which it becomes almost 7× slower. In addition, the application of the transposed preconditioner sometimes takes more than twice the time of its non-transposed counterpart.

A possibility to deal with the slower SpMV kernel implemented in cuSparse is to

Table 3.6: Evaluation of BiCG for selected cases in platform BRAHMS. The SPMV and the application of the preconditioner both proceed in a single GPU while other minor operations are performed by the CPU. Times are in seconds.

Matrix	#Iter.	SPMV with A	SPMV with A^T	Appl. M^{-1}	Appl. M^{-T}	Remaining operations	Total time
A200	10	0.28	0.57	0.37	0.45	0.74	2.26
A252	10	0.54	1.22	0.71	0.88	1.47	4.50
Freescale1	292	3.14	10.66	5.38	11.64	5.78	36.52
cage15	12	0.27	1.84	0.63	0.71	0.83	3.88
diagonal	298	8.91	24.43	21.24	48.77	15.63	118.75
unit-vector	316	8.04	24.47	21.74	50.22	15.21	119.43

maintain explicit copies of the transposed operands in the GPU memory and operate directly with them. However, this solution is not satisfactory as, for some large-scale problems, the amount of memory in the accelerator may be insufficient.

Contrary to most Krylov-based iterative linear solvers, in which there exist strict data dependencies that serialize the sequence of kernels appearing in the iteration, BiCG is composed of two quasi-independent recurrences, one with A and a second with A^T; see the left-hand side column of Figure 3.2. Moreover, in the preconditioned version of the method, there is no data dependence between the application of the transposed and non-transposed preconditioner inside the iteration, exposing coarse-grain parallelism at the recurrence-level.

To improve the performance of BiCG, we rearranged the operations in the BiCG method so that these two sequences are isolated and executed in a server equipped with two discrete graphics accelerators. This enables the exploitation of coarse-grain task-parallelism in conjunction with the data-parallelism that is leveraged inside each accelerator.

The specific operations off-loaded to each device in our dual-GPU implementation of the BiCG method are outlined in Figure 3.2. This partitioning of the workload executes a single SPMV and the application of one of the preconditioners in each GPU. As these are the most computationally-demanding steps of the solver, this distribution of the workload can be expected to be fairly well-balanced, despite the fact that one of the devices also computes a dot product and two AXPY operations.

In a system equipped with two discrete accelerators, each GPU (A and B) has its own separate memory. Thus, to avoid wasting memory, and simultaneously exploit the faster version of CUSPARSESPMV, we maintain the non-transposed operands in GPU A and the transposed ones in GPU B. With this approach we do not use additional memory, but exploit the best storage format in each device. However, it is now necessary to transfer the transposed data to the second GPU, which adds some overhead.

In the previous implementation, the preconditioner was copied to the GPU asynchronously, as it was calculated by the incomplete factorization routine, overlapping the transferences with the calculation of the subsequent levels of the preconditioner. As the BiCG method is the only one that makes use of the transposed matrix and preconditioner, it makes no sense to perform the transfer of the these transposed operands by default. For that reason we report this communication cost as an overhead of BiCG. Nevertheless, a significant part of this cost can be hidden if the transference is performed asynchronously and

overlapped with the calculation of the preconditioner, as it is the case of the non-transposed data.

Regarding the concurrent execution in both devices, we leverage asynchronous memory copies between the CPU and the GPU so that the accelerated section of the solver proceeds asynchronously with respect to the host. To accomplish this, we previously register the memory area used by the solver as non-pageable so that the operations corresponding to different devices overlap even though they are launched serially.

In summary, by using both devices, we can exploit the increased computational power provided by the second GPU in addition to the much faster execution of the SpMV when invoked with non-transposed operands.

3.3.2 Concurrent BiCG for single GPU platforms

The results for the baseline GPU variant of BiCG indicate that, when there is only one GPU available, using cuSparse to accelerate the most data-parallel stages of BiCG is, in general, convenient. As mentioned in previous sections, at least two approaches can be derived. On the one hand, storing A^T explicitly in the accelerator enables the use of the "fast" version of the cuSparse SpMV routine, which can be more than 7× faster than its transposed counterpart. On the other hand, the memory overhead implied by storing A^T can be excessively high. Furthermore, the transposition of the matrix and the memory allocation that is required takes considerable time. Therefore, the selection of the best strategy should consider the memory limitations of the target platform, the size of the coefficient matrix, the number of iterations performed by the subsequent iterative solver, and the impact of the SpMV with A^T on the overall performance. In this context we are interested in leveraging the task-parallelism present in BiCG to improve performance when storing A^T is not an option. Therefore, to perform $A^T v$ on the GPU, we will use the transposed variant of the SpMV routine in cuSparse.

Departing again from Base_BICG version of Section 3.2, but trying to keep the advantages offered by the task-parallel version, perhaps the most straightforward strategy consists in exploiting the concurrent GPU execution offered by *streams*. The first of the accelerated versions we propose performs the operations that corresponded to GPU A in Figure 3.2 in the first stream, while the operations corresponding to GPU B are performed in the second stream. No operations are left to the default stream, and we use the cuBlas *device pointer mode* [81] to avoid unnecessary synchronizations due to scalar parameters being transferred to and from the GPU.

As the resources of the GPU must be shared between all streams, the overlapping of kernels in different streams can be modest in those cases where one of the kernels fully utilizes the accelerator. In such scenario, the performance of the solver will be almost identical to the variant that uses only one stream. Given the important difference between the performance of the transposed and non-transposed SpMV routines, one possible alternative is to leverage the multi-core CPU to perform the transposed operations (those corresponding to GPU B in Figure 3.2), while using cuSparse and GPU A for the non-transposed part. However, as the results in Table 3.5 indicate that we can still obtain an important

acceleration for the transposed preconditioner on the GPU, we will only evaluate the use of the CPU for the transposed SpMV, using cuSparse to compute the application of the transposed preconditioner. This should allow the overlapping of an important part of the non-transposed SpMV and the application of the non-transposed preconditioner, with the transposed SpMV, while the eventual overlapping with the transposed preconditioner will be due to the use of streams.

Enhancing task-parallelism

One of the main bottlenecks of the application of ILUPACK multilevel ILU preconditioner is the solution of four triangular linear systems in each level, which can be derived from Equation (2.90). The solution of these operations involves routine `cusparseDcsrsv_solve` in cuSparse, whose implementation is based on the so-called level-set strategy [17], and appears described in [76]. In a broad sense, it consists in performing an analysis of the triangular sparse matrix to determine sets of independent rows called *levels*. The triangular solver will then launch a GPU kernel to process each one of these levels, processing the rows that belong to each level in parallel. The number of levels that derive from the analysis can vary greatly according to the sparsity pattern of each triangular matrix, so that for matrices of considerable size it is usually in the order of hundreds or even a few thousands. In such cases, the overhead due to launching the kernels that correspond to each level can become significant [47]. In the context of a CPU-GPU concurrent execution, wasting one core on launching kernels instead of doing useful computations can have a more significant impact on the overall performance.

Recent research has aimed to reduce the synchronization overhead, as well as replacing the costly analysis phase by a less computationally-demanding process, based on a self-scheduled strategy that effectively avoids the synchronization with the CPU [71]. In [45] we followed these ideas to develop a synchronization-free GPU routine to solve triangular linear systems for matrices in CSR format. The cost of launching the kernels involved in this routine is completely negligible, so that solving the triangular linear systems that correspond to the application of the non-transposed with this strategy should enable a better overlapping with the transposed SpMV in the GPU. We next provide a brief description of our routine.

CSR synchronization-free sparse triangular solver

Our self-scheduled procedure to compute the solution of a (lower) triangular sparse linear system proceeds row-wise, assigning a warp to each unknown. To manage the dependencies between unknowns, each warp must busy-wait until the entries in the solution vector that are necessary to process that row have their final values. To keep track of this, a *ready* vector of booleans, with an entry for each required unknown, indicates if the corresponding row has been processed or not.

Before a warp can start processing its assigned row, it iteratively polls the corresponding entries of the *ready* vector until all of them have been set to one by the corresponding warps. The values in the local variable of each thread of the warp are then reduced by a warp-voting

primitive that returns one if the polled value is nonzero for all the active threads in the warp, or zero otherwise.

This process will likely cause a deadlock if the warps that produce the values needed by the current warp are not executed on the multiprocessor because all active warps are caught executing a busy-waiting. We rely on the fact that accessing the *ready* vector on global memory causes a *warp stall* and, therefore, the warp scheduler of the Streaming Multiprocessor (SM) activates another warp, so eventually all the dependencies of the warp get fulfilled.

Once this occurs, each thread of the warp multiplies a nonzero value of the current row by the appropriate entry of the solution vector, accumulating the result in a register. If there are more than 32 nonzero elements in the row, the warp moves back to busy-waiting for the solution of the unknowns that correspond to the next 32 nonzero elements of the row. The cycle is repeated until all nonzero entries in the row have been processed.

Finally, the registers that accumulate the products performed by each thread of the warp are reduced using warp shuffle operations, and the first thread of the warp is responsible for updating the solution and *ready* vector accordingly.

A more detailed explanation of this process, accompanied by experimental results, can be found in Appendix A.

3.3.3 GMRES with Accelerated Data-Parallel MGSO

In past work, we accelerated the implementations of BiCG and GMRES in ILUPACK by off-loading the application of the preconditioner and the SpMV, appearing at each iteration, to the graphics co-processor. These were, in principle, the most computationally-expensive operations in both methods. However, for GMRES, once we accelerated these two tasks, the cost of other operations of the solver became important, especially for some problems of large dimension showing slow convergence rate. A quick analysis identified that, for these cases, most of the execution time was spent on the orthogonalization required by the restarted GMRES (see Algorithm 6). In particular, the cost of this step is proportional to the matrix dimension, and grows with each iteration until a restart occurs.

Table 3.7 reports the cost of the main operations comprised by the solver, when executing the sequential implementation in ILUPACK on the CPU and the baseline data-parallel version introduced in Section 3.2. The cases in the table correspond to a few examples featuring an expensive re-orthogonalization stage. For the CPU version, the application of the preconditioner is the most time-consuming operation of the sequential solver implemented in ILUPACK, taking more than 50% of the execution time for all problem instances in the table. Once we off-load the SpMV as well as the application of the preconditioner to the accelerator, the remaining stages of the solver gain importance though. Specifically, the MGSO applied as part of each iteration of the GMRES then represents between 24% and 62% of the total execution time.

Table 3.7: Evaluation of GMRES for selected cases in platform BRAHMS. We display the execution time (in seconds) of each stage of the GMRES solver and their cost (in %) relative to the total time. In the GPU variant, the SPMV and the application of the preconditioner (identified with the label "Appl. M^{-1}") both proceed in the GPU, while MGSO and other minor operations are performed by the CPU.

			SPMV		Appl. M^{-1}		MGSO		Total
Matrix	Device	#Iter.	Time	%	Time	%	Time	%	time
A200	CPU	6	0.67	15.58	2.18	51.08	0.57	13.45	4.27
	GPU	6	0.47	17.24	0.58	21.50	0.73	26.93	2.70
A252	CPU	6	1.30	15.78	4.42	53.53	1.13	13.68	8.25
	GPU	6	0.54	13.69	0.97	24.32	1.13	28.38	3.97
Freescale1	CPU	46	2.82	13.14	11.83	55.22	6.03	28.15	21.43
	GPU	46	0.99	10.82	1.97	21.56	5.69	62.16	9.15
cage15	CPU	7	1.32	23.75	3.32	59.90	0.48	8.57	5.54
	GPU	7	0.28	14.17	0.78	39.61	0.47	24.06	1.97
circular	CPU	203	30.05	9.47	210.92	66.51	72.15	22.75	317.12
	GPU	203	10.19	9.02	31.09	27.52	68.38	60.54	112.95
unit-vector	CPU	251	37.14	9.42	260.51	66.11	91.43	23.20	394.08
	GPU	251	12.60	8.92	38.07	26.95	86.64	61.34	141.23

Accelerated data-parallel version of MGSO

The implementation of MGSO integrated into ILUPACK's routine for GMRES is described in Algorithm 7. For our enhanced data-parallel implementation of this procedure, we designed a hybrid version that leverages the best type of architecture for each operation while, at the same time, limiting the data transferences.

In particular, we rely on the CUBLAS library to perform the dot products (O1 and O3) and vector updates (O2, O4 and O5) on the GPU. Since we already off-loaded the SPMV appearing in GMRES to the accelerator, the basis vectors of MGSO reside in the GPU. The output of the process is the current basis vector, orthogonalized with respect to the remaining vectors, and the coefficients of the current row of the Hessenberg matrix. This matrix is small, and the following application of rotations and triangular solve expose little parallelism, so it is natural to keep it in the CPU memory. The coefficients of the matrix are calculated serially via vector products with the basis vectors, so the GPU computes n flops for each coefficient that is transferred back to the CPU. A similar approach was studied in [61].

3.3.4 BiCGStab

The BiCGStab method (see Section 2.2.2) is one of the most widespread iterative solvers for general linear systems [89] for which, unfortunately, there is no support in the current distribution of ILUPACK.

As stated in Section 2.4, the solvers in ILUPACK are implemented following a reverse communication strategy. However, as BiCGStab can be efficiently implemented in the GPU via calls to CUBLAS, CUSPARSE and our data-parallel version of ILUPACK's preconditioner, it is natural to encapsulate the method in one monolithic function that receives as inputs,

Table 3.8: Experimental results for the enhanced coarse-grain data-parallel version of BiCG for dual-GPU servers (ADV_BICG_2GPU) in platform BRAHMS.

Matrix	Device/ routine	SPMV A, A^T M^{-1}, M^{-T}	Copies A^T, M^{-T}	Remaining operations	Total Time	Speed-up ADV_BICG_2GPU with respect to...
A200	CPU	7.01	–	0.75	7.77	1.92
	BASE_BICG	1.61	–	0.74	2.21	0.54
	ADV_BICG_2GPU	0.45	2.93	0.48	4.04	–
A252	CPU	11.06	–	1.21	12.29	1.50
	BASE_BICG	3.31	–	1.44	4.48	0.55
	ADV_BICG_2GPU	0.88	5.98	0.89	8.14	–
cage14	CPU	2.42	–	0.15	2.59	1.40
	BASE_BICG	0.99	–	0.24	1.13	0.61
	ADV_BICG_2GPU	0.31	1.34	0.10	1.85	–
Freescale1	CPU	90.94	–	5.82	96.78	7.41
	BASE_BICG	30.84	–	5.79	36.56	2.80
	ADV_BICG_2GPU	7.20	1.33	4.42	13.05	–
rajat31	CPU	2.56	–	0.29	2.87	1.28
	BASE_BICG	1.15	–	0.38	1.45	0.64
	ADV_BICG_2GPU	0.26	1.61	0.23	2.23	–
cage15	CPU	9.29	–	0.46	9.77	2.52
	BASE_BICG	3.43	–	0.83	3.87	0.60
	ADV_BICG_2GPU	1.04	4.64	0.33	6.36	–
circular	CPU	306.95	–	13.59	320.55	8.35
	BASE_BICG	93.33	–	13.30	106.40	2.77
	ADV_BICG_2GPU	23.71	4.04	10.35	38.38	–
diagonal	CPU	390.99	–	17.71	408.72	10.25
	BASE_BICG	98.77	–	14.37	112.90	2.83
	ADV_BICG_2GPU	24.65	4.01	10.94	39.85	–
unit-vector	CPU	415.08	–	18.95	434.04	10.37
	BASE_BICG	105.42	–	15.18	120.37	2.88
	ADV_BICG_2GPU	25.99	4.01	11.73	41.84	–

among other parameters, the right-hand side vector and the initial guess, and produces the approximate solution to the system in response.

Our CPU implementation of BiCGStab relies on a subset of the BLAS routines distributed with ILUPACK, though a different implementation of BLAS can be used without any major modification. On the other hand, our implementation of BiCGStab follows the ideas in [74] to off-load the entire solver (i.e., all kernels) to the GPU.

3.3.5 Experimental evaluation of advanced variants

Next we present the performance analysis of our advanced parallel variants. The platforms and test cases used for the experiments are the same as those used in the evaluation of the baseline versions.

Experimental evaluation of the dual-GPU version of BiCG

Table 3.8 reports the execution time of the coarse-grain data-parallel version of BiCG for dual-GPU servers (ADV_BICG_2GPU). In this table, we separate the operations of each iteration of BiCG into three groups/stages, and assess the relative cost corresponding to each of them as well as the total time. The first stage aggregates the operations that were accelerated by the two GPU versions of the solver: SPMV and application of the preconditioner. The second stage measures the overhead of the initialization due to the copy of the transposed matrix and preconditioner to the second GPU; therefore, it is only visible for the new dual-GPU variant. The third stage comprises the remaining operations, mostly

dot products and vector updates, which are performed in the CPU in all our implementations of BiCG.

The results show that the modifications produce a sensible reduction of the execution time for the cases that take a higher number of iterations to converge. This is necessary to compensate the additional cost of copying and transferring the preconditioner and the coefficient matrix to the second GPU. If we consider only the first stage, the acceleration achieved by the dual-GPU variant with respect to the CPU version is of up to 16×. Nevertheless, the cost of initialization and the unaccelerated parts of the solver significantly affect the performance. Overall, despite this costly initialization, for some of the problem instances the execution time of the iterative solve is reduced by a factor in the range 7–10× with respect to the sequential version.

On the other hand, it is especially remarkable that with our strategy the speed-up associated to doubling the many-core devices overcomes the linear evolution for the largest cases. Note that the dual-GPU version of BiCG outperforms the original GPU variant by a factor of up to 2.88× for the whole method yielding a super-linear speed-up.

Evaluation of the single-GPU variants of BiCG

We next perform the analysis of the experimental results obtained from the execution of the single-GPU variants discussed in Section 3.3.2. We use the same set of matrices and hardware platform employed for the evaluation of the dual-GPU proposal.

We include four variants in the evaluation:

- CPU_BICG_MKL performs all the computations on the multi-core processor. The two SPMVs of the BiCG are performed using the multi-threaded of the MKL library (version 2017, update 3), while the triangular solvers are computed with an optimized sequential code. This variant does not exploit task-parallelism.
- ADV_BICG_2STR computes the operations corresponding to the GPU A and GPU B blocks in Figure 3.2) in one device, assigning one GPU stream to each block. The GPU computation of the SPMVs and sparse triangular linear systems are performed using CUSPARSE library, the vector operations of BiCG are computed using CUBLAS, and the less important vector operations inside the preconditioner application are implemented using *ad-hoc* GPU kernels.
- ADV_BICG_HYB_CUSP employs the multi-core CPU to perform the transposed SPMV of BiCG using the MKL library. The rest of the computations are performed in the GPU using CUSPARSE and CUBLAS libraries, as in ADV_BICG_2STR.
- ADV_BICG_HYB_SF replaces the triangular solver of the routine that applies the non-transposed preconditioner by our new synchronization-free routine. It employs the MKL library to perform the transposed SPMV of BiCG, and CUSPARSE to compute the main operations of the transposed preconditioner.

The results obtained for the four variants described are displayed in Table 3.9. We only present the data corresponding to the accelerated part of BiCG because it is on these computations that the four variants differ from each other. The runtimes of the remaining stages are similar to those in Table 3.8. A discrepancy in the number of iterations performed

by each variant is obtained for the *diagonal* problem, which can be attributed to the effect of floating-point rounding errors.

It can be observed that the proposed task-parallel variants are able to significantly accelerate the involved section of BiCG iteration, with speed-ups that reach 12× relative to the data-parallel multi-core version. In most of the cases, the improvement is mainly due to the GPU acceleration of the application of the preconditioner, which can be deduced from the speed-ups corresponding to version Adv_BICG_2str. The usage of GPU streams, however, has little effect on the overall performance, as almost no overlapping of operations is observed in the timelines extracted with Nvidia Visual Profiler. This can be observed for the *A200* and *cage15* sparse matrices in Figures 3.3 and 3.4 respectively.

Regarding the hybrid GPU-CPU variants, the results show that off-loading the transposed SpMV to the multi-core can yield important benefits. In our experiments, the improvements with respect to the Adv_BICG_2str variant ranges from 8% to almost 65%.

A number of factors influence the relation between the performance of the Adv_BICG_2str and Adv_BICG_Hyb_Cusp variants. For example, as can be observed in Figure 3.4 for matrix *cage15*, a great fraction of the performance improvement is due to the reduction of the time taken by the transposed SpMV. In Adv_BICG_2str, this takes more than half of the execution time of the iteration, and the multi-core implementation in Adv_BICG_Hyb_Cusp is able to both reduce this time in half and overlap of part of the application of the preconditioner. A different situation is observed for matrix *A200*, where the cuSparse routine for the transposed SpMV outperforms the MKL counterpart. In this case though, the difference between the runtime of both routines is less critical, and a performance improvement is obtained regardless, due to the almost perfect overlapping of the application of the preconditioner with the transposed SpMV.

The results also show that the use of our synchronization-free routine contributes with an additional performance improvement by enabling a higher degree of overlapping between operations. This will depend mostly on the number of level-sets of the incomplete factors, as additional levels imply having to launch more kernels, and augment the corresponding overhead. For instance, the analysis of the triangular factors generated for matrix *A200* yields only 40 levels, whereas for *cage15* it generates 616. Thus, it is not surprising that the difference between the runtimes of Adv_BICG_Hyb_Cusp and Adv_BICG_Hyb_SF for matrix *A200* is minimal, while for *cage15* the performance gain is relevant.

Table 3.9: Aggregated runtime (in seconds) and acceleration of the parallelized part of BiCG in platform BRAHMS.

Matrix	Routine	# It.	SpMV App. Prec.	Speedup vs. CPU_BICG_MKL
A200	CPU_BICG_MKL	12	9.05	
	ADV_BICG_2STR	12	1.20	7.51
	ADV_BICG_HYB_CUSP	12	1.04	8.67
	ADV_BICG_HYB_SF	12	1.04	8.74
A252	CPU_BICG_MKL	12	12.52	
	ADV_BICG_2STR	12	2.49	5.02
	ADV_BICG_HYB_CUSP	12	2.05	6.09
	ADV_BICG_HYB_SF	12	2.04	6.14
cage14	CPU_BICG_MKL	14	2.15	
	ADV_BICG_2STR	14	0.78	2.77
	ADV_BICG_HYB_CUSP	14	0.27	7.88
	ADV_BICG_HYB_SF	14	0.22	9.85
Freescale1	CPU_BICG_MKL	442	145.09	
	ADV_BICG_2STR	442	37.75	3.84
	ADV_BICG_HYB_CUSP	442	32.02	4.53
	ADV_BICG_HYB_SF	442	28.41	5.11
rajat31	CPU_BICG_MKL	12	3.49	
	ADV_BICG_2STR	12	1.27	2.75
	ADV_BICG_HYB_CUSP	12	0.85	4.10
	ADV_BICG_HYB_SF	12	0.83	4.21
cage15	CPU_BICG_MKL	16	10.16	
	ADV_BICG_2STR	16	3.09	3.29
	ADV_BICG_HYB_CUSP	16	1.17	8.67
	ADV_BICG_HYB_SF	16	0.85	12.02
diagonal	CPU_BICG_MKL	170	296.24	
	ADV_BICG_2STR	190	107.91	2.75
	ADV_BICG_HYB_CUSP	192	97.45	3.04
	ADV_BICG_HYB_SF	176	79.97	3.70
circular	CPU_BICG_MKL	158	262.67	
	ADV_BICG_2STR	158	86.56	3.03
	ADV_BICG_HYB_CUSP	158	78.89	3.33
	ADV_BICG_HYB_SF	158	72.23	3.64
unit-vector	CPU_BICG_MKL	170	303.07	
	ADV_BICG_2STR	170	97.91	3.10
	ADV_BICG_HYB_CUSP	170	89.65	3.38
	ADV_BICG_HYB_SF	170	80.27	3.78

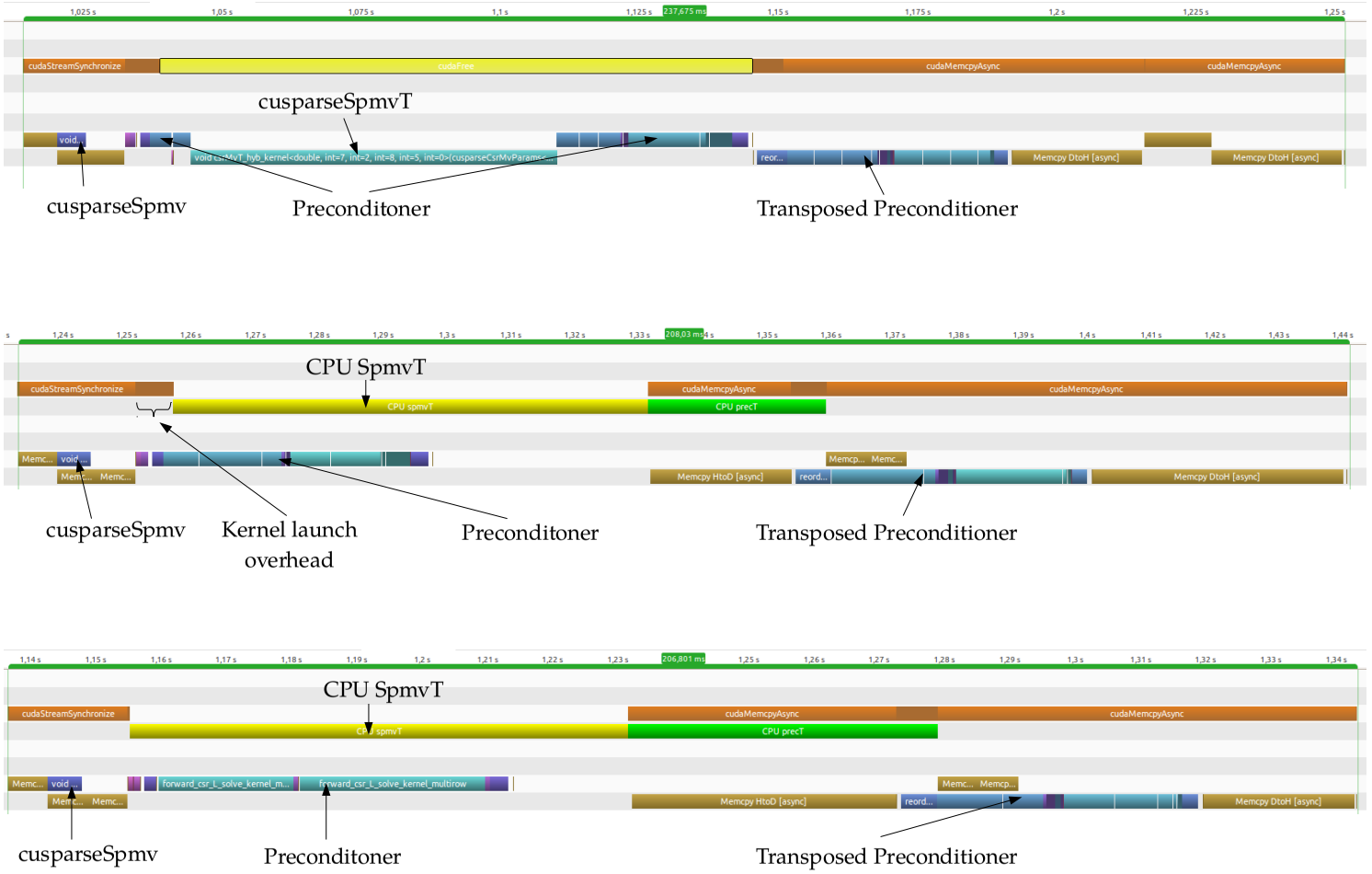


Figure 3.3: Output from NVIDIA Visual Profiler for matrix *A200*. The timeline is magnified to show a single iteration of the solver. The upper timeline corresponds to ADV_BICG_2STR, the middle to ADV_BICG_HYB_CUSP, and the bottom to ADV_BICG_HYB_SF. The time spans of the iterations from top-down are 237.675ms, 208.03ms and 206.801ms respectively.

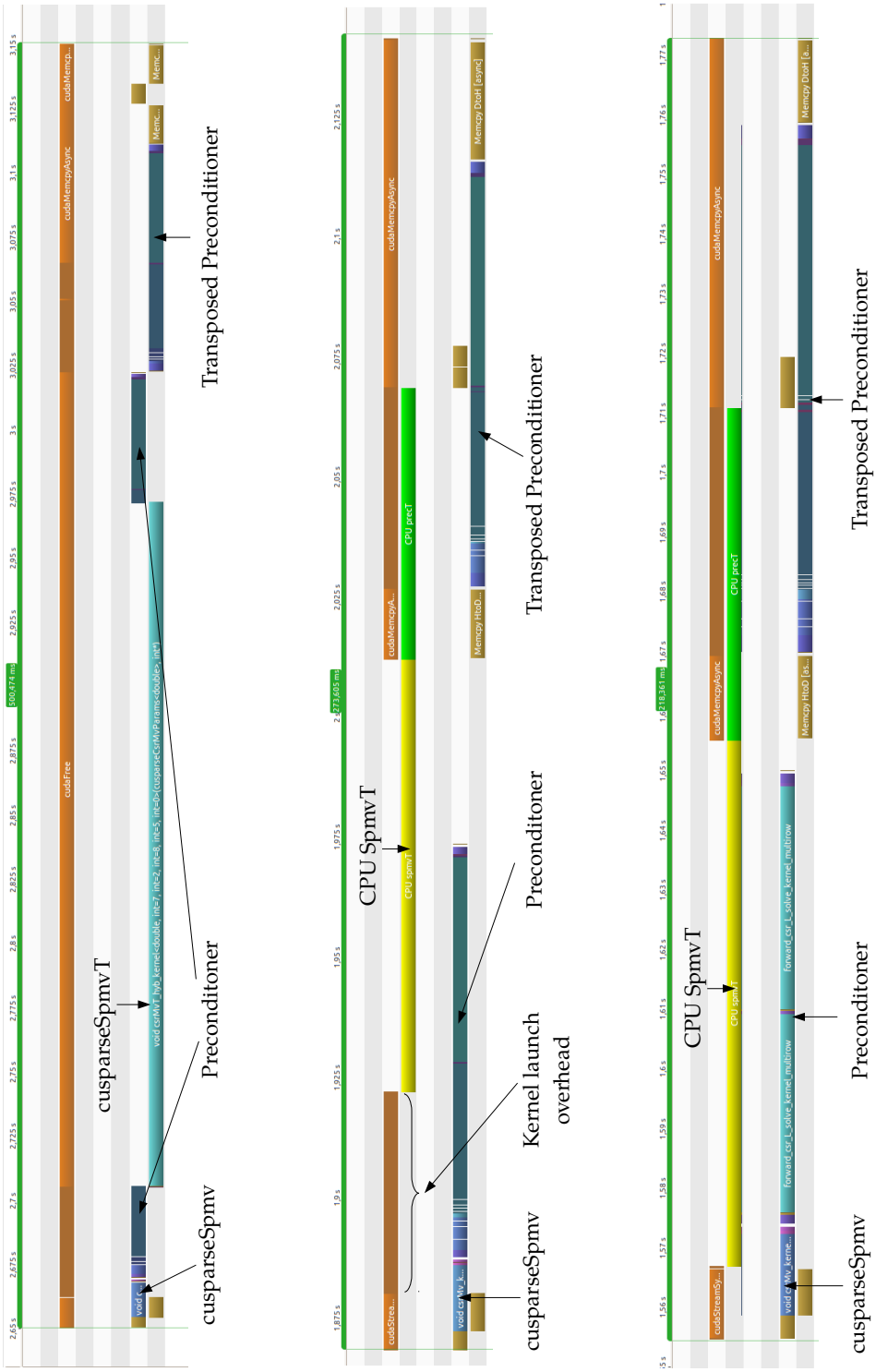


Figure 3.4: Output from NVIDIA Visual Profiler for matrix *cage15*. The timeline is magnified to show a single iteration of the solver. The upper timeline corresponds to ADV_BICG_2STR, the middle to ADV_BICG_HYB_CUSP, and the bottom to ADV_BICG_HYB_SF. The time spans of the iterations from top-down are 500.474ms, 273.205ms and 218.361ms respectively.

Table 3.10: Experimental results for the enhanced data-parallel version of MGSO and the GMRES method (ADV_GRMES) in platform BRAHMS. Times are in seconds.

Matrix	#Iter.	Time CPU		Time GPU		Speed-up GPU	
		MGSO	Total	MGSO	Total	MGSO	Total
A200	6	0.57	4.27	0.17	1.61	3.45	2.65
A252	6	1.13	8.25	0.33	3.60	3.41	2.29
cage14	7	0.14	1.54	0.04	0.50	3.73	3.10
Freescale1	46	6.03	21.43	0.97	4.42	6.24	4.85
rajat31	4	0.22	2.23	0.05	0.76	4.14	2.93
cage15	7	0.48	5.54	0.13	1.63	3.77	3.39
circular	203	72.15	317.12	10.14	54.66	7.12	5.80
diagonal	241	86.63	377.17	13.15	68.56	6.59	5.50
unit-vector	252	91.43	394.08	13.20	68.68	6.93	5.74

Experimental evaluation GMRES with accelerated data-parallel MGSO

Table 3.10 compares the execution time of the GPU-accelerated MGSO routine against its CPU counterpart, showing the effect on the execution time of the entire solver. The results demonstrate that the accelerated version of the routine achieves a notable reduction of its execution time, reaching speed-ups between 3× and 7× over the CPU version. Moreover, combining this with the previous enhancements, we obtain speed-ups of up to 5.8× for the entire solver.

An additional observation from the data in Table 3.10 is that the factor which most affects the acceleration is the number of iterations of the solver. This is due to the transference of the basis vectors from the device to the host that is necessary in order to perform the last step of the GMRES on the CPU. The two factors that determine the cost of MGSO at a given iteration k of GMRES are the number of vectors involved in the orthogonalization and their dimension, which is equivalent to k modulo the restart parameter of GMRES. Since one of these transfers occur every time MGSO is invoked, the ratio between the volume of transfers and the amount of computation in MGSO improves with each iteration until the restart. As a consequence, the impact of the transference overhead is greater in those cases that converge in a small number of steps and do not reach the restart point of GMRES, which we set in our experiments to 30 iterations. This communication can be avoided if the last step of GMRES is performed on the accelerator.

Evaluation of GPU-based BiCGStab

Table 3.11 compares our CPU and GPU variants of BiCGStab. In addition to the total execution time of the solver, we present the average time (and speed-up) per iteration, as in some cases we obtain slightly different values between the GPU and CPU versions. The discrepancies are small and occur for those cases with higher condition number, which are more prone to floating-point rounding errors.

The execution times observed for BiCGStab highlight the benefits of including this method in the suite of CPU solvers supported by ILUPACK, as in general it attains better convergence rates and delivers lower execution times than the CPU versions of GMRES and

Table 3.11: Experimental results for the new data-parallel version of BiCGStab. Times are in seconds.

Matrix	Device	#Iter.	Total time	Avg. time per iter.	Avg. speed-up per iter. with respect to...	Relative residual
A200	CPU	3	4.27	1.423	8.21	1.40E-11
	GPU	3	0.52	0.173	–	1.40E-11
A252	CPU	3	7.16	2.386	7.02	1.30E-11
	GPU	3	1.02	0.340	–	1.30E-11
cage14	CPU	3	1.93	0.643	5.36	3.40E-10
	GPU	3	0.36	0.120	–	3.40E-10
Freescale1	CPU	92	54.21	0.589	9.07	2.30E-03
	GPU	87	5.65	0.064	–	2.20E-03
rajat31	CPU	2	1.62	0.810	5.40	7.80E-09
	GPU	2	0.30	0.150	–	7.80E-09
cage15	CPU	3	7.28	2.426	6.07	5.40E-10
	GPU	3	1.20	0.400	–	5.40E-10
circular	CPU	115	239.84	2.085	8.07	6.90E-08
	GPU	100	25.84	0.258	–	3.90E-08
diagonal	CPU	111	281.80	2.538	10.09	6.60E-08
	GPU	114	28.68	0.251	–	5.20E-08
unit-vector	CPU	115	240.07	2.087	8.22	4.70E-08
	GPU	108	27.42	0.253	–	4.00E-08

BiCG.

Regarding the use of the GPU, the acceleration factor for the solver iteration varies between 4× and 10×, with the exact speed-up depending on characteristics of the problem such as the dimension of the problem, the sparsity pattern of the coefficient matrix, and the sparsity of the incomplete factors produced by the multilevel ILU factorization underlying ILUPACK.

CHAPTER 4

Design of a Task-Parallel version of ILUPACK for Graphics Processors

The data-parallel variants presented in Chapter 3 are confined to execute in one compute node, equipped with a multi-core CPU and one or two GPUs. This limitation operates in two senses. First, compute nodes or servers equipped with more than two GPUs are increasingly more common, in part driven by the computational requirements of machine-learning. Second, being restricted to the memory subsystem of one node, including the memory of the GPUs present in that node, constrains the size of the problems that can be tackled with our current strategy.

A third drawback of our previous developments, is that their potential to exploit the cores of modern CPUs is fairly limited as well. This is mainly a consecuence of the characteristics of the problem at hand. Except for the BiCG method, the rest of the studied iterative solvers are based on a sequence of basic matrix and vector computations that allow little overlapping between them. Moreover, although data-parallelism can be exploited in these operations, the performance gain will be strongly limited by the memory throughput of the CPU. It is thus safe to assume that no radical performance boost can be extracted from the exploitation of the multi-core CPU without significant effort.

In the final sections of Chapter 2 one such effort is described, based on a modification of both the preconditioner and the PCG method of ILUPACK, which is capable of exposing task-level parallelism. We call this software Task-Parallel ILUPACK, and the underlying ideas are detailed in [3] and [4], where two different implementations, one for shared-memory and the other for distributed-memory platforms, are presented and analyzed.

Although as mentioned before, the Task-Parallel ILUPACK is restricted to the SPD case, and might have some numerical disadvantages with respect to the sequential and data-parallel versions, these factors are compensated by its excellent performance and scalability, which makes this variant useful in large-scale SPD scenarios.

Encouraged by the results presented in Chapter 3, which suggest that sensible performance improvements can be attained by the use of GPUs, and that this only exerts a mild effect on the accuracy attained by the preconditioner, we are now iteresting in studying how this data-parallelism can be harnessed in the Task-Parallel ILUPACK.

We start by evaluating the combination of task-parallelism and co-processor data-parallelism in shared-memory platforms equipped with GPUs. Specifically, we leverage the computational power of one GPU (via the exploitation of data-level parallelism) to accelerate the operations that compose the application of the multilevel preconditioner of each individual task in the shared-memory (task-parallel) variant of ILUPACK. Later, we address the distributed-memory variant of ILUPACK, and leverage the computational power of GPUs distributed across several nodes of a cluster, to accelerate the computations of the tasks that reside in these nodes. We show how this strategy allows the addressing of large-scale problems.

The major contributions of this chapter are the following:

- We extend the task-parallel version of ILUPACK for shared-memory machines exploiting the data-parallelism of the operations that compose the application of the multilevel preconditioner. We do this by offloading the SpMV and the solution of triangular linear systems appearing in the leaf tasks, along with some minor vector operations, to the hardware accelerator.
- We introduce a new strategy to reduce the effect of data-parallelism loss inside the individual tasks that is implied by the matrix partitioning. This technique is based on a threshold which determines if a given algebraic level of the preconditioner presents enough granularity to take advantage of the GPU.
- The experimental evaluation shows that the threshold-based strategy is able to execute each operation in the most convenient device while mantaining a moderate communication cost, outperforming the original multicore version for all the tested instances.
- We enable the use of multiple GPUs to accelerate the execution of the leaf tasks in the distributed-memory variant of ILUPACK.
- To handle the mentioned loss of parallelism due to the partitioning of the problem, we propose an enhancement to our threshold strategy, which takes advantage of the devices even in the smaller algebraic levels of the preconditioner. This is achieved by performing the highly serial triangular linear system solves on the CPU while offloading the parallel SpMV and vector operations to the device.
- The results obtained on 4 nodes equipped with 2 GPUs each show that the execution time of the all-CPU variant can be improved by a factor of up to 2× by assigning the tasks to the GPUs.
- The chapter shows scaling properties that suggest that it is possible to address large-scale problems efficiently.

4.1 GPU acceleration of the shared-memory variant of ILUPACK

In this section we present our strategy to introduce GPU acceleration in the multi-core version of ILUPACK [7]. This implementation of the Task-parallel ILUPACK uses OpenMP parallel regions to delimit the different tasks, while the handling of the dependendencies and scheduling of the tasks is performed by a middleware layer added to ILUPACK. The implementation is compatible with version 2 of the OpenMP standard, and the task-managing features introduced in version 3.0 and later extended in version 4 are not exploited.

We analyze two different approaches to introduce GPU computations in this version of ILUPACK. The first one entirely off–loads the leaf tasks of the preconditioner application phase to the GPU, while the second one uses a threshold to exploit the GPU only when there is enough work to take advantage of the accelerator.

Our solution is designed for multi-core platforms equipped with one GPU, using different streams to enqueue work that belongs to different tasks, but the idea is easily extensible to a shared-memory hardware connected to multi-GPU contexts.

4.1.1 All leafs in GPU, ShMem_GPU_All

The task-parallel version of ILUPACK is based on a Nested Dissection (ND) ordering, resulting in a *task tree* where only leaf tasks perform an important amount of work. In particular, the inner tasks correspond to the separator sub-graphs in the ND process, and hence have much less work than their leaf counterparts. For this reason we only consider leaf tasks from here on.

The leaf tasks are independent from each other and can be executed concurrently provided sufficient threads are available. Therefore, we associate each of these tasks with a different GPU stream. Also, each task has its own data structures, both in CPU and GPU memory, containing the part of the multilevel preconditioner relevant to it, together with private CPU and GPU buffers. At the beginning of the application, these buffers are allocated, and our GPU-enabled versions make this memory non-pageable in order to perform asynchronous memory transferences between the CPU and the GPU.

For the ShMem_GPU_All version of the preconditioner application, the computation on each node of the DAG is based on the data-parallel version presented in [5]. It proceeds as described in Section 2.4.1, with the difference that, in this case, the forward and backward substitution are separated and distributed among the levels of the task-tree. Therefore, entering or leaving the recursive step in Equation (2.92) sometimes implies moving to a different level in the task tree hierarchy. In these cases, the residual r_{k+1} has to be transferred to the GPU at the beginning of the forward substitution phase, and the intermediate result has to be retrieved back into the CPU buffers before entering the recursive step. Once the inner tasks compute the recursive steps, the backward substitution proceeds from top to bottom until finally reaching the leaf tasks again, where the z_{k+1} vector has to be transferred to the GPU. There the last steps of the calculation of the preconditioned residual $z_{k+1} := M^{-1}r_{k+1}$ are performed. Upon completion, the preconditioned residual z_{k+1} is retrieved

back into the CPU.

As in the GPU-aware solvers presented in Chapter 3, we rely on the cuSparse library to execute the most computationally-demanding kernels, while the low cost operations (diagonal scalings, vector permutations, and vector updates) are performed by *ad-hoc* kernels.

This version aims to accelerate the computations involved by the leaf tasks while incurring a low communication cost, relying on the results obtained for the GPU acceleration of the serial version, and the streaming capabilities offered by the new GPU architectures. Unfortunately, this version has serious drawbacks. The division of the work in various leaf tasks reduces the size of each independent linear system, and the multilevel ILU-factorization of the preconditioner produces levels of even smaller dimension. This can have a strong negative impact on the performance of massively parallel codes [67], and specifically on the cuSparse library kernels. It should be noted that the amount of data-parallelism available in the sparse triangular linear systems is severely reduced, leading to a poor performance of the whole solver. Additionally, the work assigned to the CPU in this variant is minor, impeding the concurrent use of both devices.

4.1.2 Threshold based version, ShMem_Thres

In order to deal with the disadvantages of the previous version, we propose a threshold-based strategy that computes the algebraic levels in the GPU until certain granularity, and executes the remaining levels in the CPU. This aims to produce two effects. On one hand, allowing the smaller and highly data-dependent levels to be computed on the CPU while the initial levels, of larger dimension and higher data-parallelism, run on the GPU, implies that each operation is performed in the most convenient device. On the other hand, this strategy also improves the concurrent execution in both devices, increasing the overlap of the CPU and GPU sections of the code.

Regarding data transfer, in this approach the working buffer has to be brought to the CPU memory at some point of the forward substitution phase, and it has to be transferred back to the GPU before the backward substitution of the upper triangular system ends. Moreover, these transfers are synchronous with respect to the current task or GPU stream, since the application of one algebraic level of the multilevel preconditioner cannot commence until the results from the previous level are available.

In this variant we determine the threshold value experimentally. Our on-going work aims to identify the best algorithmic threshold from a model capturing the algorithm's performance.

4.2 Overcoming memory capacity constraints

The work presented in the previous sections exploits the data-parallelism present in the operations performed by each leaf task in order to accelerate the execution of the task-parallel multi-core version of ILUPACK (DistMem_CPU). Unfortunately, the number of accelerators that can be integrated into one compute node is relatively small. For this reason, using the GPUs attached to only one node strongly constrains the memory and

compute thrust available to solve the systems, and thus the size of the problems that can be tackled. In this section we present the key features of our new GPU-enabled variant of the distributed-memory ILUPACK, which makes use of massively-parallel architectures residing in the nodes of a cluster. As before, we focus on the stage that corresponds to the application of the preconditioner during the iteration of the PCG method.

This parallelization of ILUPACK builds upon the task-parallel version presented in [4], following the main aspects described in Section 2.4.2. To execute in distributed-memory environments, this version of ILUPACK maps each leaf task of the DAG to a different MPI rank. This correspondence between tasks and ranks is statically defined by the root node during the initialization steps, and is maintained for all subsequent operations. Then, the root node distributes the data required for the parallel construction of the preconditioner to the corresponding tasks. In our case, the initialization steps also include the creation of a CUDA Stream for each leaf task. If there are many devices available in the compute node, the streams, and consequently the tasks resident in that node, are mapped to different devices in a round-robin fashion, based on their *task_id*.

As in the case of the task-parallel multi-core variant in the previous section, the application of the preconditioner consists of forward and backward substitution processes that are now divided across the levels of the task-tree. Once the tree is traversed from bottom to top, the backward substitution proceeds from top to bottom, until finally reaching the leaf tasks again. Since the information of the preconditioner is spread across the task-tree, traversing this structure requires a communication operation, and sometimes presents data dependencies with the next or previous levels in the task-tree hierarchy. From the point of view of the execution on a GPU, besides storing the residual r_{k+1} in GPU memory at the beginning of the forward substitution phase, now it is necessary to transfer the intermediate results of each task back to the CPU buffers before entering the recursive step.

As earlier, the partitioning strategy implies that only the tasks lying on the bottom level of the tree perform a significant amount of work. Moreover, the multilevel structure of ILUPACK's preconditioner partitions the workload further, severely limiting the amount of data-parallelism. These are the main motivations underlying the threshold introduced for the multi-core variant. However, not all the operations involved in the application of the preconditioner inside a leaf task are equally affected by these constraints to parallelism. For example, the sparse triangular system solves that appear in the application of the preconditioner tend to present considerably more data dependencies between their equations, while the sparse matrix-vector products and vector operations present a degree of parallelism similar to the non-partitioned case. In response to this, we propose a variation of the threshold strategy where, instead of migrating the computation of the smaller levels entirely to the CPU, we do this only for the triangular solves corresponding to such levels. This way we can take a certain advantage of the data-parallelism present in the sparse matrix-vector products and the vector operations, even in the smaller levels of each leaf. We call this variant DistMem_Thres.

Both approaches imply that some sections of the working buffer have to be transferred between the CPU and GPU memories during the forward and backward substitution operations. Moreover, these transfers are synchronous with respect to the current task or GPU

stream, since the application of one algebraic level of the multilevel preconditioner cannot commence until the results from the previous level are available. In our first approach, once the first level is processed during the forward substitution, the data to be read by all the operations of the following levels is sent to the CPU memory. In the backward substitution phase, the data to be read by the first level needs to be transferred to the GPU. In our second approach, only the data needed by the triangular solves of the smaller algebraic levels is transferred. Although this scatters the communications into many small transfers, an scenario which is not desirable in the GPU context for bandwidth and latency reasons, the amount of data to be passed is almost the same, and the number of data movements does not increase significantly.

4.3 Numerical evaluation

In this section we summarize the experiments carried out to evaluate the performance of our GPU-aware implementations of the task-parallel version of ILUPACK for shared and distributed memory systems. With this purpose, we first compare the two GPU-accelerated variants of ILUPACK designed for shared-memory platforms with the original multi-core version.

Next, we assess the benefits of utilizing multiple GPUs in a distributed-memory setting using the new GPU-enabled version of ILUPACK's MPI implementation for clusters. With this aim, we first compare our proposal with the original distributed version and conduct both performance and scalability analyses. All experiments reported in this section were obtained using IEEE double-precision arithmetic.

4.3.1 Platforms and test cases

Multi-core GPU server (Brahms)

The multi-core GPU server utilized for the experiments is Brahms, presented in Section 3.1.

The CPU code was compiled with the Intel(R) Parallel Studio 2016 (update 3), which partially supports version 4.5 of the OpenMP specification (only for C++). However, none of the new features of versions 3 and 4 is used in our code, and the compiler is fully compliant with OpenMP v2.×.

Distributed memory platform (Falla)

The performance evaluation of the distributed-memory variant was carried out using 4 nodes of a cluster installed in the CETA-CIEMAT[1] center (Trujillo, Spain). Each node is equipped with a 12-core Intel(R) Xeon(R) E5-2680 v3 processor (2.50GHz), 64 GB of DDR3 RAM memory, and 2 Tesla K40 GPU with 2,880 CUDA Cores and 12 GB of GDDR5 RAM each.

We used the OpenMPI v10.3.1 implementation of the MPI standard and `gcc` 4.8.5 with the `-O3` flag to compile the CPU code. The GPU compiler and the CUSPARSE library

[1] Centro Extremeño de Tecnologías Avanzadas

correspond to version 7.5 of the CUDA Toolkit.

Test cases

The benchmark utilized for the experimental evaluation is the SPD case of scalable size presented in Section 3.1.2.

Additionally, we tested our distributed-memory proposal on a circuit simulation problem from the SuiteSparse[2] collection (G3_Circuit), also described in Section 3.1.2 in order to evaluate our strategy on a matrix with a different sparsity pattern than those offered by the Laplace benchmark.

4.3.2 Evaluation of the shared-memory variant

Each test instance was executed using 2 and 4 CPU threads with $f = 2$ and $f = 4$ respectively. The parameter f is related with the construction of the task tree. The algorithm that forms this tree relies on an heuristic estimation of the computational cost of each leaf task and divides a leaf into two whenever its correspondent sub-graph has more edges than the number of edges of the whole graph divided by f. The parameter f is chosen so that, in general, there are more leaf tasks than processors. In [4, 7] the authors recommend choosing a value between p and $2p$, where p is the number of processors.

Table 4.1: Number of leaf tasks and average structure of each algebraic level of the preconditioner using $f = 2$ and $f = 4$. To represent the structure of the levels, the average dimension, the number of non-zeros and the rate of non-zeros per row is presented, together with the respective standard deviations.

Matrix	# th. / f	# leaves	level	avg. n	$\sigma(n)$	avg. nnz	$\sigma(nnz)$	$\frac{nnz}{n}$	$\sigma(\frac{nnz}{n})$
A159	2	3	0	1,006,831	345,798	6,193,794	2,183,862	6.1	0.1
			1	317,362	113,151	9,682,114	3,486,401	30.5	0.2
			2	2,875	736	10,099	2,014	3.6	0.6
	4	6	0	502,108	159,044	3,116,629	1,005,408	6.2	0.1
			1	156,048	50,647	4,685,500	1,537,905	30.0	0.2
			2	1,251	437	4,095	1,754	3.2	0.4
A171	2	2	0	1,881,030	16,604	11,421,390	123,868	6.1	0.1
			1	598,152	1,384	18,490,583	154,695	30.9	0.2
			2	6,304	984	23,444	7,247	3.7	0.6
	4	4	0	937,998	6,011	5,764,461	71,397	6.1	0.1
			1	294,702	1,310	8,967,985	72,180	30.4	0.2
			2	2,845	506	10,885	3,768	3.7	0.7
A200	2	3	0	2,003,212	795,192	12,207,556	4,592,834	6.1	0.1
			1	636,895	253,696	19,665,756	8,089,987	30.8	0.4
			2	6,466	3,316	23,189	12,912	3.5	0.2
	4	7	0	856,365	186,595	5,283,746	1,141,907	6.2	0.1
			1	268,523	595,53	8,155,375	1,842,559	30.3	0.2
			2	2,449	525	8,552	2,032	3.5	0.4
A252	2	3	0	4,004,955	1,694,044	24,271,087	9,856,575	6.1	0.1
			1	1,283,180	543,882	39,965,828	17,408,294	31.0	0.3
			2	14,762	7,162	57,168	28,744	3.8	0.1
	4	6	0	1,998,470	494,294	12,196,071	3,070,313	6.1	0.1
			1	635,612	159,942	19,603,140	4,936,718	30.8	0.1
			2	6,523	1,429	23,807	5,758	3.6	0.3

Table 4.1 summarizes the structure of the multilevel preconditioner and the linear systems

[2] http://faculty.cse.tamu.edu/davis/suitesparse.html

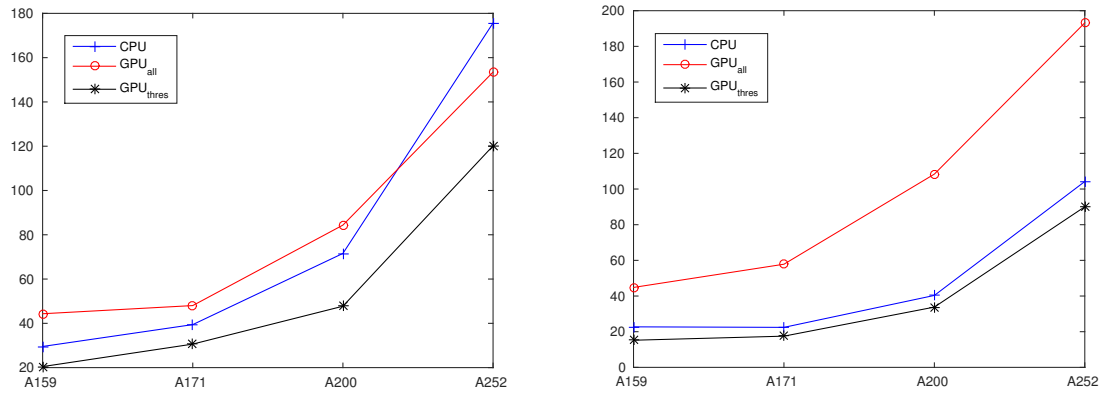


Figure 4.1: Execution time (in seconds) of preconditioner application for the three task parallel variants, using two (left) and four (right) CPU threads. CPU version is the blue line with crosses. ShMem_GPU_All version is the red line with circles. ShMem_Thres is the black line with stars.

corresponding to leaf tasks that were generated using the aforementioned parameters. For each tested matrix, the table presents the number of leaf tasks that resulted from the task tree construction for $f = 2$ and $f = 4$, and next to it shows the average dimension of the algebraic levels of the corresponding multilevel preconditioner, the average number of nonzeros, and the average row density of the levels, with their respective standard deviations. It can be easily observed that a higher value of f results in more leaf tasks of smaller dimension. Regarding the algebraic levels of the factorization, the table shows how the average dimension of the sub-matrices decreases from one level to the next. It is important to notice how, in the second algebraic level, the submatrices already become about one third smaller in dimension, and contains five times more nonzero elements per row. In other words, the sub-problems become dramatically smaller and less sparse with each level of the factorization, causing that, in this case, only the first algebraic level is attractive for GPU acceleration.

Table 4.2 shows the results obtained for the original shared-memory version and the two GPU-enabled ones for the matrices of the Laplace problem. The table reports the total runtime of PCG, as well as the time spent on the preconditioner application stage and the SpMV are presented. The table also shows the number of iterations taken to converge to the desired residual tolerance, and the final relative residual error attained, which is calculated as in (3.1)

First, it should be noted that from the perspective of accuracy, there are no significant differences between the task-parallel CPU variant and the GPU-enabled ones. Specifically, the three versions reach the same number of iterations and final relative residual error for each case.

From the perspective of performance it can be observed that, on the one hand, ShMem_GPU_All only outperforms ShMem_CPU for the largest matrix (A252) and in the context of 2 CPU threads. This result was expected, as the GPU requires large volumes of computations to truly leverage the device and hide the overhead due to memory transfer. On the other hand, ShMem_Thres is able to accelerate the multi-core counterpart for all covered

Table 4.2: Runtime (in seconds) of the three task-parallel shared-memory variants in platform Brahms.

# threads	Matrix	Version	Iters.	tot. Spmv	tot. Prec	tot. PCG	$\mathcal{R}(x^*)$
2	A159	ShMem_CPU	88	2.30	29.55	32.86	1.39E-008
		ShMem_GPU_All			44.33	47.46	
		ShMem_Thres			20.46	23.83	
	A171	ShMem_CPU	97	3.07	39.43	43.87	1.52E-008
		ShMem_GPU_All			48.02	52.36	
		ShMem_Thres			30.62	35.19	
	A200	ShMem_CPU	107	5.83	71.58	79.98	2.45E-008
		ShMem_GPU_All			84.37	92.61	
		ShMem_Thres			47.73	56.26	
	A252	ShMem_CPU	131	13.86	175.66	195.67	3.23E-008
		ShMem_GPU_All			153.48	173.62	
		ShMem_Thres			120.19	140.50	
4	A159	ShMem_CPU	88	1.30	22.72	24.55	9.96E-009
		ShMem_GPU_All			44.82	46.40	
		ShMem_Thres			15.21	17.15	
	A171	ShMem_CPU	95	1.58	22.43	24.76	2.20E-008
		ShMem_GPU_All			57.84	59.78	
		ShMem_Thres			17.50	19.87	
	A200	ShMem_CPU	108	3.13	40.34	45.03	1.06E-008
		ShMem_GPU_All			108.37	112.41	
		ShMem_Thres			33.80	38.60	
	A252	ShMem_CPU	130	8.25	104.21	116.37	2.16E-008
		ShMem_GPU_All			193.19	204.74	
		ShMem_Thres			90.05	104.60	

cases; see Figure 4.1. This result reveals the potential benefit that arises from overlapping computations on both devices. Hence, even in cases where the matrices present modest dimensions, this version outperforms the highly tuned multi-core version. Additionally, the benefits related with the use of the GPU are similar for all matrices of each configuration, though the percentage of improvement is a bit higher for the smaller cases. This behavior is not typical for GPU-based solvers and one possible explanation is that the smaller cases are near the optimal point (from the threshold perspective) while the largest cases are almost able to compute 2 levels in GPU. This can be confirmed in Table 4.3, were we add a variant that computes the first 2 levels on the accelerator. As the multilevel factorization generates only 3 levels, with the third one very small with respect to the other two, it is not surprising that the runtimes of this version are almost equivalent to those of ShMem_GPU_All. The table shows how the penalty of computing the second level in the GPU decreases as the problem dimension grows.

Finally, ShMem_Thres also offers higher performance improvements for the 2-threads case than for its 4-threads counterpart.

4.3.3 Evaluation of the distributed-memory variant

Our proposal aims to exploit the data-parallelism present inside the operations comprised by each task of the tree. It is reasonable to think that the initial reorganization of the preconditioner into a tree of smaller multilevel matrices reduces the degree of data-parallelism, severely constraining it. In order to obtain a reference of the maximum speed-up that we

Table 4.3: Runtime (in seconds) of ShMem_Thres, adjusting the threshold to compute 1 and 2 levels in the GPU, in platform Brahms.

# threads	Matrix	ShMem_Thres (1 lev)	ShMem_Thres (2 lev)	ShMem_GPU_All
2	A159	23.83	43.79	44.33
	A171	35.19	47.52	48.02
	A200	56.26	84.16	84.37
	A252	140.50	153.79	153.48
4	A159	17.15	44.54	44.82
	A171	19.87	57.21	57.84
	A200	38.60	108.70	108.37
	A252	104.60	185.72	193.19

Table 4.4: Runtime (in seconds) of the serial CPU version of ILUPACK's PCG and its corresponding GPU-accelerated version. The speed-up in the table can serve as an upper bound of the maximum possible improvement when leveraging the GPUs in the distributed memory variant in platform Falla.

Matrix	Iterations	CPU	GPU	Speedup	$\mathcal{R}(x^*)$
A50	37	0.54	1.09	0.50	7.45E-10
A159	89	50.46	20.32	2.48	8.33E-09
A171	94	66.61	24.80	2.69	8.99E-09
A200	110	125.91	41.22	3.05	1.33E-08
A252	135	311.30	94.73	3.29	1.73E-08
G3_circuit	149	20.90	9.12	2.28	3.15E-06

can expect by including the GPU in the distributed-memory version, Table 4.4 reports the speed-up experienced when comparing the execution of the serial CPU version and the corresponding GPU-accelerated version in the target platform. In the table, we report the total runtime of PCG and the residual error achieved in each case, which is calculated according to Equation (3.1).

Table 4.5 shows the results obtained for the original distributed-memory version and the GPU-enabled code (single node) for the different evaluated test instances. The experiments were run with 2, 4 and 8 tasks that are distributed among the four nodes in a round-robin fashion. For the instances consisting of 2 and 4 tasks, each task resides in a different node and uses a different GPU. For the instance with 8 tasks there are two tasks per compute node, but each task runs on a distinct GPU.

The number of iterations and residual errors obtained for the CPU and GPU versions differ slightly, especially in the case with 8 tasks. These discrepancies are due to the distinct numerical properties of the task-parallel preconditioners and do not represent a significant variation in the accuracy. The results for the A400 benchmark case with 2 tasks were not added because the per-task memory requirements of this test case are greater than the 64GB of RAM that are available in each compute node. This illustrates one of the main benefits of our novel approach. Our previous efforts addressed the inclusion of GPUs in the shared-memory version of the task-parallel ILUPACK. Obviously, using only one compute node strongly limited the number of devices, as well as the amount of GPU memory available and, therefore, the size of the problems that could be tackled with our previous parallel version of ILUPACK.

Table 4.5: Runtime (in seconds) of version DistMem_Thres in platform Falla. The experiments were performed setting ILUPACK to use 2, 4, and 8 tasks. In the executions with 2 and 4 tasks, each one resides in a different node with one GPU. In the case of the execution with 8 tasks, there are 2 tasks per compute node, but each one uses a different GPU.

#Tasks	Matrix	Version	#It.	PCG	$\mathcal{R}(x^*)$	Accel.
2	A159	CPU	84	36.03	4.2e-7	1.70
		GPU	84	21.23	4.2e-7	
	A171	CPU	93	50.20	5.5e-7	1.73
		GPU	93	29.02	5.5e-7	
	A200	CPU	107	94.31	7.2e-7	1.84
		GPU	107	51.33	7.2e-7	
	A252	CPU	125	228.60	8.2e-7	1.85
		GPU	125	123.44	8.2e-7	
	A318	CPU	153	817.84	1.3e-6	2.15
		GPU	153	381.14	1.3e-6	
	A400	CPU	-	-	-	-
		GPU	-	-	-	-
	G3_Circuit	CPU	116	13.39	2.1e-05	1.78
		GPU	116	7.49	2.1e-05	
4	A159	CPU	83	18.49	4.3e-7	1.33
		GPU	83	13.92	3.5e-7	
	A171	CPU	86	26.24	4.4e-7	1.52
		GPU	85	17.24	3.6e-7	
	A200	CPU	97	43.32	5.8e-7	1.44
		GPU	97	29.99	5.8e-7	
	A252	CPU	125	113.56	8.1e-7	1.72
		GPU	123	66.03	9.3e-7	
	A318	CPU	155	389.55	1.1e-6	1.94
		GPU	155	200.72	9.9e-7	
	A400	CPU	192	983.49	1.4e-6	2.00
		GPU	192	491.70	1.6e-6	
	G3_Circuit	CPU	139	7.45	2.7e-05	1.26
		GPU	139	5.87	2.7e-05	
8	A159	CPU	68	8.33	2.9e-7	0.84
		GPU	67	9.96	3.3e-7	
	A171	CPU	74	11.12	3.0e-7	0.89
		GPU	70	12.44	2.7e-7	
	A200	CPU	83	19.97	5.3e-7	1.05
		GPU	77	18.99	4.3e-7	
	A252	CPU	101	49.23	7.6e-7	1.41
		GPU	98	34.85	8.3e-7	
	A318	CPU	131	130.44	9.5e-7	1.63
		GPU	122	80.14	1.0e-6	
	A400	CPU	158	341.94	1.4e-6	1.83
		GPU	149	186.70	1.4e-6	
	G3_Circuit	CPU	136	3.82	2.4e-05	0.87
		GPU	136	4.36	2.4e-05	

Table 4.6: An estimation of the memory overhead due to the partitioning for two of the benchmark cases. *Sum* represents the sum of the dimension of the leaves, *Overhead* is the difference between *Sum* and the original dimension of the coefficient matrix, while *Ratio* is the ration between *Overhead* and the original dimension of the coefficient matrix.

Matrix		Number of leaves				
		2	4	8	16	32
A050	*Sum*	127,518	134,821	152,201	191,130	274,540
	Overhead	2,518	9,821	27,201	66,130	149,540
	Ratio	2.01	7.86	21.76	52.90	119.63
A159	*Sum*	4,047,709	4,121,004	4,298,050	4,715,372	5,591,299
	Overhead	28,030	101,325	278,371	695,693	1,571,620
	Ratio	0.70	2.52	6.93	17.31	39.10

At this point we note that there is a certain memory overhead caused by the initial partitioning of the matrix, as some blocks have to be shared across several processes. This can be appreciated in the expression in (2.99). In order to assess the importance of this overhead, we evaluated the difference between the sum of the dimensions of the sub-matrices corresponding to the leaves of the task tree and the original dimension of the coefficient matrix. This distance was computed for two of the benchmark matrices, one small and the other of moderate size, generating partitions of 2, 4, 8, 16 and 32 leaf tasks. Table 4.6 shows that this overhead increases with the granularity of the partition, and the ratio between this overhead and the original dimension of the matrix is much smaller for the larger matrix. This means that, in order to improve the memory efficiency, one should partition the matrix only when there is not enough memory in one node to perform the factorization and solution. Moreover, this scheme is suitable to address large problems, where the relative memory cost of the partitioning is less important.

Regarding the execution times, the GPU-enabled variant outperforms the CPU version with an equal number of processes, except for the smaller cases of the benchmark when using 8 tasks. Utilizing the GPU offers a 2× speed-up for the largest case. Moreover, the advantage of using the GPUs tends to increase with the dimension of the problem.

The low performance of the smaller test cases can be easily explained, as smaller workloads make it more difficult to fully occupy the GPU resources and strongly increase the relative cost associated to CPU-GPU communications. Additionally, partitioning the work into many tasks constrains the data-parallelism and undermines the performance of some GPU kernels.

Considering the previous discussion, it is obvious that our solution cannot attain *strong scalability* [62] (i.e., a linear growth of the speed-up when augmenting the number of compute resources for a problem of fixed size). However, as it can be appreciated in Figure 4.2, the results show that we are close to attaining *weak scalability* (i.e., maintaining the speed-up when augmenting the compute resources in the same proportion as the problem size). For example, if we take the number of nonzero values of the matrices as a rough estimation of the problem size, the execution of the A159 problem in 2 processors/GPUs, A252 in 4 processors/GPUs, and A400 in 8 processors/GPUs all present similar acceleration factors. Furthermore, when analyzing the speed-ups in Table 4.5, it should be taken into account

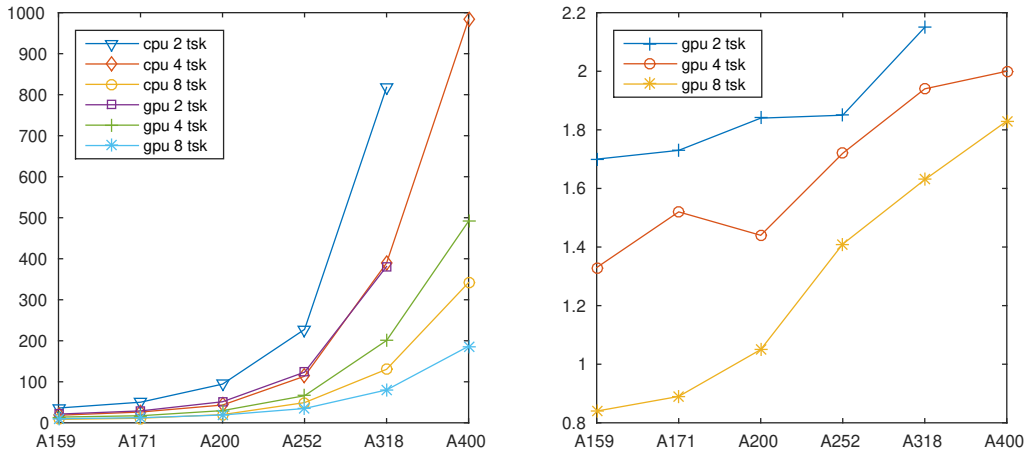


Figure 4.2: Execution time (in seconds) of DistMem_Thres for each of the executed test cases (left) and acceleration obtained by the inclusion of the GPU over DistMem_CPU (right) in platform Falla.

that only (part of) the application of preconditioner in the leaf tasks is accelerated using GPUs, while the rest of the PCG operations, though being potentially parallelizable, at this moment belong to the unaccelerated part of the program.

CHAPTER 5

Conclusions

This chapter closes the thesis by offering some final comments about the work described previously. It also presents an outline of the publications related to this dissertation, and is closed with a discussion of several potential lines of future work.

5.1 Closing remarks

The undeniable relevance of sparse linear systems in many areas of science and engineering, as well as the rapid scaling in the size of the problems we are witnessing nowadays, motivate the research for robust and efficient iterative solvers and preconditioners, and significant advances in this subject have led to the development of several software packages. For these packages to be successful, the correct utilization of modern computational platforms is mandatory to obtain acceptable performance, and failing to do so can become an important disadvantage against simpler but more established techniques. In this sense, hardware accelerators such as GPUs have become an ubiquitous and powerful parallel architecture, and making an efficient use of these devices is of utmost importance.

In this thesis we have made a comprehensive study of ILUPACK, a package that bundles several of the most widely used Krylov subspace solvers with a sophisticated multilevel "inverse-based" ILU preconditioner, in order to enable its efficient execution in platforms equipped with hardware accelerators. This study involves the development of several GPU-aware implementations of its preconditioner and most important solvers.

Prior to this dissertation, there were no variants of ILUPACK capable of exploiting data-parallelism. Previous task-parallel implementations of ILUPACK [3, 4, 8] have shown good performance and scalability in many scenarios, and remain a valuable tool to this day, but also face some limitations.

In the first place, the task-parallel variants of ILUPACK are constrained to the solution of SPD linear systems, and the generalization of this approach to the general case is a

daunting endeavour from a mathematical perspective. In this sense, we have developed the only existing parallel implementations of ILUPACK for non-symmetric and indefinite systems.

Second, the previous versions rely on a slight modification of the preconditioner, necessary to extract task-parallelism, that can impact the convergence properties of the preconditioner. Our new data-parallel implementations are able to improve the performance of the original preconditioner without significantly affecting its numerical properties.

In addition to our data-parallel variants of the sequential version of ILUPACK, we studied the exploitation of GPU-enabled platforms for the two task-parallel implementations available at the moment. Specifically, we enabled GPU computations for the shared-memory and distributed-memory implementations of ILUPACK.

Next, we highlight the most relevant details about each of these main contributions.

5.1.1 Development of GPU-aware variants of ILUPACK

We have provided a fully functional data-parallel version of ILUPACK, accelerated by means of GPUs, that preserves the numerical properties of the sequential solver. For this purpose, we developed data-parallel implementations of ILUPACK's preconditioner for all its supported matrix types, covering four solvers: CG for SPD systems, GMRES and BiCG for general systems, and SQMR for symmetric indefinite ones. All our data-parallel solvers offload the most computationally-demanding operations to the graphics accelerators, where they are carried out via *ad-hoc* GPU kernels.

Our results, using NVIDIA GPUs from Fermi and Kepler generations and a collection of examples from the Suite Sparse matrix collection, a convection-diffusion problem, and the Laplace equation, show speed-ups with respect to the original CPU version that are around $2\times$ in many cases, reaching an improvement of $5.8\times$ for one of the SPD cases and $3.7\times$ for one non-symmetric problem instance.

These are the first implementations of this software package capable of leveraging data-parallelism. In the case of the non-symmetric and indefinite solvers, they are the only parallel versions of ILUPACK available, so these speed-ups correspond to fair acceleration factors. Furthermore, these values are mostly determined by the scarce parallel efficiency of the triangular solvers in cuSPARSE, and are similar to the results found in the literature for this type of operation.

The contributions summarized in this section are reflected in the following publications:

- Aliaga, J. I., Bollhöfer, M., Dufrechou, E., Ezzatti, P., and Quintana-Ortí, E. S. (2014). Leveraging data-parallelism in ILUPACK using graphics processors. In *IEEE 13th International Symposium on Parallel and Distributed Computing, ISPDC 2014, Marseille, France, June 24-27, 2014*, pages 119–126
- Aliaga, J. I., Bollhöfer, M., Dufrechou, E., Ezzatti, P., and Quintana-Ortí, E. S. (2016a). A data-parallel ILUPACK for sparse general and symmetric indefinite linear systems. In *Euro-Par 2016: Parallel Processing Workshops - Euro-Par 2016 International Workshops, Grenoble, France, August 24-26, 2016, Revised Selected Papers*, pages 121–133

5.1.2 Enhanced parallel variant of GMRES

The analysis of our baseline implementation of the GMRES method revealed that, after the acceleration effort using GPUs, the performance bottleneck in the GMRES is shifted so that the (modified) Gram-Schmidt re-orthogonalization (MGSO) becomes the critical task to achieve further gains in many cases.

In this context, we designed and developed a new approach for the massively parallel version of GMRES method that integrates a GPU-based version of the MGSO method. Our results report considerable speed-ups with respect to the sequential solvers integrated in ILUPACK, with speed-up values between 2.3 and 5.8× considering the execution time of the complete procedure.

The contributions in this section are summarized in:

- Aliaga, J. I., Dufrechou, E., Ezzatti, P., and Quintana-Ortí, E. S. Accelerating a preconditioned GMRES method in massively parallel processors *Proceedings of the 18th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2018* July 9-14, 2018.
- Aliaga, J. I., Dufrechou, E., Ezzatti, P., and Quintana-Ortí, E. S. An efficient GPU version of the preconditioned GMRES method *Accepted for publication at the Journal of Supercomputing*, 2018.

5.1.3 Task-data-parallel variants of BiCG

We have revisited and extended our first GPU-aware version of BiCG to offer a variant of this solver capable of efficiently exploiting the parallel processing power of dual-GPU platforms. Taking advantage of the inherent task-parallelism of the BiCG method, we perform the most computationally demanding part of the iteration using both devices in parallel. Furthermore, the extended memory capacity allows us to use the most adequate data representation in each device, avoiding the use of slow transposed cuSparse routines.

The experiments reveal that our dual-GPU variant of the application of the preconditioner achieves speedups of up to 10× with respect to the original CPU version. These values are much higher if only the accelerated part of the solver is taken into account. Regarding the comparison with the previous GPU version, our dual-GPU variant of BiCG delivers super-linear speed-ups for the computation of the whole method, in the sense that we obtain more than two times the speedup by duplicating the computational resources.

In addition, we have extended the work analyzing the benefits offered by the exploitation of task-parallelism, when only a single GPU is available. In this context, we evaluated the use of GPU streams in order to increase the concurrency, and the use of the multi-core CPU to allow further overlapping of independent operations. Finally, we took advantage of a recently proposed synchronization-free solver for sparse triangular linear systems to enhance the use of the multi-core CPU and unleash higher levels of concurrency, since this type of methods eliminates the CPU overhead due to the launching of kernels that cuSparse routines imply. The experimental evaluation, performed over the same hardware platform and with the same test cases, shows that we can reach fair runtime reductions in single-GPU platforms, despite of the processing and memory limitations.

The contributions in this section led to the following publications:

- Aliaga, J. I., Bollhöfer, M., Dufrechou, E., Ezzatti, P., and Quintana-Ortí, E. S. (2018). Extending ILUPACK with a task-parallel version of BiCG for dual-GPU servers. In *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM@PPoPP 2018, February 25, 2018, Vienna, Austria*, pages 71–78
- Aliaga J. I., Dufrechou E., Ezzatti P., Quintana-Ortí E. S. Accelerating the task/data-parallel version of ILUPACK's BiCG in multi-CPU/GPU configurations. *Under peer-review at the Journal of Parallel Computing.*

5.1.4 GPU variant of BiCGStab

We have proposed and evaluated a data-parallel implementation of BiCGStab method, a well-known iterative method, enhanced with the inverse-based multilevel preconditioner of ILUPACK. Our results report considerable speed-ups with respect to our baseline massively parallel solvers.

These contributions are reflected in the following publication:

- Aliaga J. I., Bollhöfer M., Dufrechou E., Ezzatti P., Quintana-Ortí E. S. (2018) Extending ILUPACK with a GPU version of the BiCGStab method, *XLIV Latin American Computing Conference CLEI 2018*, Sao Paulo, Brazil.

5.1.5 GPU version of ILUPACK for shared-memory platforms

We have extended the task-parallel version of ILUPACK designed for shared-memory platforms so that leaf tasks can exploit the data-parallelism of the operations that compose the application of the multilevel preconditioner, which are the SpMV and the solution of triangular linear systems, along with some minor vector operations. We presented two different GPU versions, one that computes the entire leafs in the accelerator (ShMem_GPU_All) and an alternative that employs a threshold to determine if a given algebraic level of the preconditioner presents coarse enough granularity to take advantage of the GPU (ShMem_Thres). Both variants are executed on a single GPU, assigning a GPU stream to each independent leaf task.

The experimental evaluation shows that the division of the workload into smaller tasks difficults the extraction of enough data-parallelism to fully occupy the hardware accelerator, and this results in poor performance for ShMem_GPU_All. However, ShMem_Thres is able to execute each operation in the most convenient device while maintaining a moderate communication cost, outperforming the original multi-core version for all the tested instances.

These contributions are reflected in the following publication:

- Aliaga, J. I., Dufrechou, E., Ezzatti, P., and Quintana-Ortí, E. S. (2016b). Design of a task-parallel version of ILUPACK for graphics processors. In *High Performance Computing - Third Latin American Conference, CARLA 2016, Mexico City, Mexico, August 29 - September 2, 2016, Revised Selected Papers*, pages 91–103

5.1.6 GPU version of ILUPACK for distributed-memory platforms

The distributed-memory version of ILUPACK allows to tackle very large sparse systems of linear equations efficiently. We have extended this tool to leverage the GPUs present in the compute nodes of a cluster, exploiting the data-parallelism relative to the operations performed during the application of the multilevel preconditioner. Our GPU-enabled variant seeks to take advantage of the devices, even in the smaller algebraic levels of the preconditioner, by off-loading the highly parallel SpMV and vector operations to the device while performing the more serial triangular linear system solves on the CPU.

The experiments were obtained using 4 nodes equipped with 2 GPUs each, assigning one core and one GPU to each independent task. The results of this evaluation show that the execution time of the all-CPU variant can be improved by a factor of up to 2× by assigning the tasks to the GPUs. They also demonstrate that partitioning the work in many tasks constrains the data-parallelism of each task, so that the smaller test cases encounter difficulties to exploit the full capabilities of the accelerator. Nevertheless, the advantage of using the GPU tends to increase with the size of the test case. Although it is clear that we cannot attain *strong scalability*, the results suggests that we reach *weak scaling*, making possible to address larger problems via linearly increasing the amount of hardware resources.

These contributions are reflected in the following publication:

- Aliaga, J. I., Dufrechou, E., Ezzatti, P., and Quintana-Ortí, E. S. (2017b). Overcoming memory-capacity constraints in the use of ILUPACK on graphics processors. In *29th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2017, Campinas, Brazil, October 17-20, 2017*, pages 41–48

Jetson

Also in the line of energy efficiency, we adapted and evaluated the performance of our data-parallel version of ILUPACK for SPD systems in a low power Jetson TX1 platform, equipped with a Tegra GPU and low power ARM processors. Although the use of these devices implies an important performance compromise, the results suggest that they can be convenient in cases where the size of the problem does not allow to fully exploit regular GPUs, considering their low power consumption.

The contributions relative to this topic are summarized in the following articles:

- Aliaga, J. I., Dufrechou, E., Ezzatti, P., and Quintana-Ortí, E. S. (2017a). Evaluating the NVIDIA tegra processor as a low-power alternative for sparse GPU computations. In *High Performance Computing - 4th Latin American Conference, CARLA 2017, Buenos Aires, Argentina, and Colonia del Sacramento, Uruguay, September 20-22, 2017, Revised Selected Papers*, pages 111–122

5.2 Open lines of research

In this thesis we have focused in enhancing ILUPACK with the capacity of executing efficiently in platforms equipped with massively parallel processors such as GPUs. We have

explored several lines of work that led to efficient implementations of the most important solvers in ILUPACK, but during the process we have detected some directions in which this work can be extended.

Next we provide some details about these aspects:

- The computation of ILUPACK preconditioner is a complex and time consuming process and its parallel execution has not been studied for non-SPD problems. The main reasons for this are that the computation of the preconditioner is performed only once for each matrix, and it is therefore more interesting to accelerate its application, which is likely to be performed even hundreds of times for each matrix. However, there are use cases in which the computation time of the preconditioner completely overshadows the reduction in the iteration count of the solvers. The study of this computation process and its parallelization is an interesting line of future work.
- The solution of sparse triangular linear systems is one of the most time consuming stages in the application of the ILUPACK preconditioner. We currently rely on the cuSparse library for the execution of this operation, but this implementation is generic, and targeted to standard CSR matrices. Our recent advances on a synchronization-free approach to solve sparse triangular linear systems on the GPU, in combination with the preliminary results obtained for the inclusion of these solvers into ILUPACK, motivate the future development of GPU synchronization-free *LDU*-system solvers specially designed for ILUPACK data types and characteristic sparsity patterns.
- In the context of our acceleration of the task-parallel variants of the ILUPACK preconditioner we encountered that it is sometimes convenient to offload part of the computations to the CPU, since some small operations in the higher numbered levels on the multilevel structure cannot fully exploit the execution parallelism offered by the GPU. Currently, we defined an empirical threshold to offload such operations, but the development of an automatic mechanism to determine the most convenient device is an interesting line of future research.
- The preliminary results obtained for the execution of ILUPACK in low power platforms, such as the Jetson TX1, opens some lines that could be followed in the future. First, this kind of devices continue to develop, and new generations introduce important improvements regarding memory bandwidth, computing power, the unified memory system, and the energy performance aspects. It is therefore interesting to test our data-parallel solvers on these new architectures to assess how these advances can benefit their performance. Second, recent initiatives have aimed to construct small clusters prototypes of these devices, showing promising energy and performance results. This motivates us to consider the execution and analysis of the GPU-aware distributed variant of ILUPACK in such contexts.
- Driven by the outstanding development of machine learning in the last years, GPUs have adapted to this reality by introducing several architectural novelties and radical changes in the programming and execution models. It is therefore interesting to study the exploitation of these new features, like the Tensor Cores or the group synchroniza-

tion mechanisms introduced in CUDA 9.0. These developments are too recent to be covered in this thesis, so they will be addressed as part of future work.

Appendices

APPENDIX A

Solution of sparse triangular linear systems

Many Numerical Linear Algebra methods, as the direct solution of sparse linear systems, the application of preconditioners based on incomplete factorizations, and least squares problems, require the solution of sparse triangular linear systems as one of their most important building blocks [57]. This is one of the reasons that explain the special attention devoted to this kernel over the years, and the efforts towards developing efficient implementations for almost all competitive hardware platforms.

This operation poses serious challenges regarding its parallel execution as, in general, the elimination of one unknown depends on the previous elimination of others. Additionally, the triangular structure of the matrices is prone to create load imbalance issues. Although in the case of dense systems, these limitations can be partially overcome by rearranging the operations of the solver into a sequence of smaller triangular systems and dense matrix-vector multiplications [24], in the sparse case this solution does not offer any benefit in most cases. However, the sparsity of the matrices often allows different unknowns to be eliminated in parallel (e.g. consider the extreme case of a diagonal matrix).

The parallel algorithms for the solution of this operation can be classified in two main contrasting categories. On the one hand we can find two-stage methods, which rely on a preprocessing stage that analyzes the data dependencies to determine a scheduling for the elimination of the unknowns that reveals as much parallelism as possible. On the other hand, one-stage methods, based on a *self-scheduled* pool of tasks, on which some of the tasks have to wait until the data necessary to perform their computations is made available by other tasks[1]. Both paradigms are a good option for some instances and not so good for others, hence making impossible to offer a general result that determines which is the best method. This decision problem is one of our lines of ongoing work and some preliminary concepts are presented in [47].

[1] Diagonal inverse-based methods [13, 12, 84, 14], and iterative approaches [19, 105, 102], are two alternative categories can be considered. These are interesting strategies but face important difficulties or are not general enough to be widely applicable.

Graphics Processors have evolved to become the top parallel device architecture, so the development of algorithms that are able to exploit their benefits is of the utmost importance. To this day, the most widespread GPU implementation of sparse triangular solvers is the one distributed with the NVIDIA cuSPARSE library[2], which belongs to the first class of methods. Although the implementation is highly efficient, and has been improved and adapted to new NVIDIA architectures over the years, it has two main disadvantages. First, the analysis phase, which determines the execution schedule, is very expensive. Second, this strategy often pays the cost of constant synchronizations with the CPU [47].

As an alternative, a synchronization-free method, was recently proposed by W. Liu et al. [71]. The method is based on a dynamically-scheduled strategy and involves only a light-weight analysis of the matrix before the solution phase. However, it has the potential disadvantage of making an extensive use of GPU atomic operations. Additionally, it targets sparse triangular matrices stored in the CSC format, which is not as ubiquitous as CSR, since the latter offers a much higher performance in other common operations like the Sparse Matrix-Vector product [57].

This appendix offers details about a new GPU algorithm, presented in [45], for the solution of sparse triangular linear systems in which the coefficient matrix is stored in the CSR format. The algorithm belongs to the second class of methods, i.e. it does not involve a pre-processing stage, and it is also synchronization-free.

We also present a similar strategy to compute the *depth* of each node of graph represented by the sparse matrix. This information can be later used to calculate the same level sets than those computed by cuSPARSE. Moreover, we design a routine that, if provided with the sparse matrix coefficients and a right hand side vector, is able to combine the computation of the level sets and the solution of a linear system implying little extra work. In contrast with the routines offered by cuSPARSE, both our analyzer and our solver do not need to synchronize with the CPU until the end of the computations.

A.1 Solution of sparse triangular linear systems

The usual approach to solve a linear system of the form

$$Lx = b \tag{A.1}$$

where $L \in \mathbb{R}^{n \times n}$ is a (lower) sparse triangular matrix, $b \in \mathbb{R}^n$ is the right hand side vector, and $x \in \mathbb{R}^n$ is the sought-after solution, is called *forward-substitution*. It consists on substituting the value of the solved unknowns on the next equations, i.e. multiplying the coefficients of these rows by their corresponding unknowns and subtracting the obtained values to the right hand side, before dividing by the diagonal element in order to solve the new unknowns. This clearly implies some serialization, since an equation cannot be solved before all the unknowns on which it depends have been substituted by their actual values.

Figures A.1 and A.2 present serial versions of the algorithm for solving a sparse lower triangular linear system $Lx = b$ where the matrix L is stored in CSR and CSC sparse storage

[2]Available at https://developer.nvidia.com/cuda-downloads, as part of the CUDA Toolkit.

formats respectively. The procedure differs in order to ensure data locality in the access to the sparse matrix.

Input: row_ptr, col_idx, val, b
Output: x
$x = b$
for $i = 0$ **to** $n - 1$ **do**
  for $j = row\_ptr[i]$ **to** $row\_ptr[i + 1] - 2$ **do**
    $x[i] = x[i] - val[j] \times x[col\_idx[j]]$
  end for
  $x[i] = x[i]/val[row\_ptr[i + 1] - 1]$
end for

Figure A.1: Serial solution of sparse lower triangular systems for matrices stored in the CSR format. The vector *val* stores the nonzero values of L by row, while row_ptr stores the indices that correspond to the beginning of each row in vector *val*, and col_idx stores the column index of each element in the original matrix.

Input: col_ptr, row_idx, val, b
Output: x
$x = b$
for $i = 0$ **to** $n - 1$ **do**
  $x[i] = x[i]/val[col\_ptr[i]]$
  for $j = col\_ptr[i] + 1$ **to** $col\_ptr[i + 1] - 1$ **do**
    $x[row\_idx[j]] = x[row\_idx[j]] - val[j] * x[i]$
  end for
end for

Figure A.2: Serial solution of sparse lower triangular systems for matrices stored in the CSC format. The vector *val* stores the nonzero values of L by column, while col_ptr stores the indices that correspond to the beginning of each column in vector *val*, and row_idx stores the row index of each element in the original matrix.

A.1.1 The *level-set* strategy

Although *forward-substitution* might seem strictly sequential at first glance, when most of the coefficients of the matrix are zeros, each equation depends only on a small set of the previous unknowns, which correspond to the nonzero entries of the related row. Therefore, there is a good chance that many rows do not depend of each other and can be solved in parallel once all their other dependencies have been fulfilled.

The idea of *level-set* scheduling was introduced by Anderson and Saad [18] with the aim of finding an execution schedule for the SpTrSV that exposes as much parallelism as possible. The strategy consists in viewing the sparse matrix as a Directed Acyclic Graph (DAG) that represents the dependencies of each unknown. A nonzero element in l_{ij} means that unknown i depends on the value of unknown j, so there is an edge in the DAG from node j to node i. By means of renumbering the nodes of this DAG, they can be organized into an ordered list of *levels*, in which the nodes in one level depend only on the nodes of the previous levels; see Figure A.3. This means that all nodes (equations) in one level can be solved in parallel given that the previous levels have already been computed.

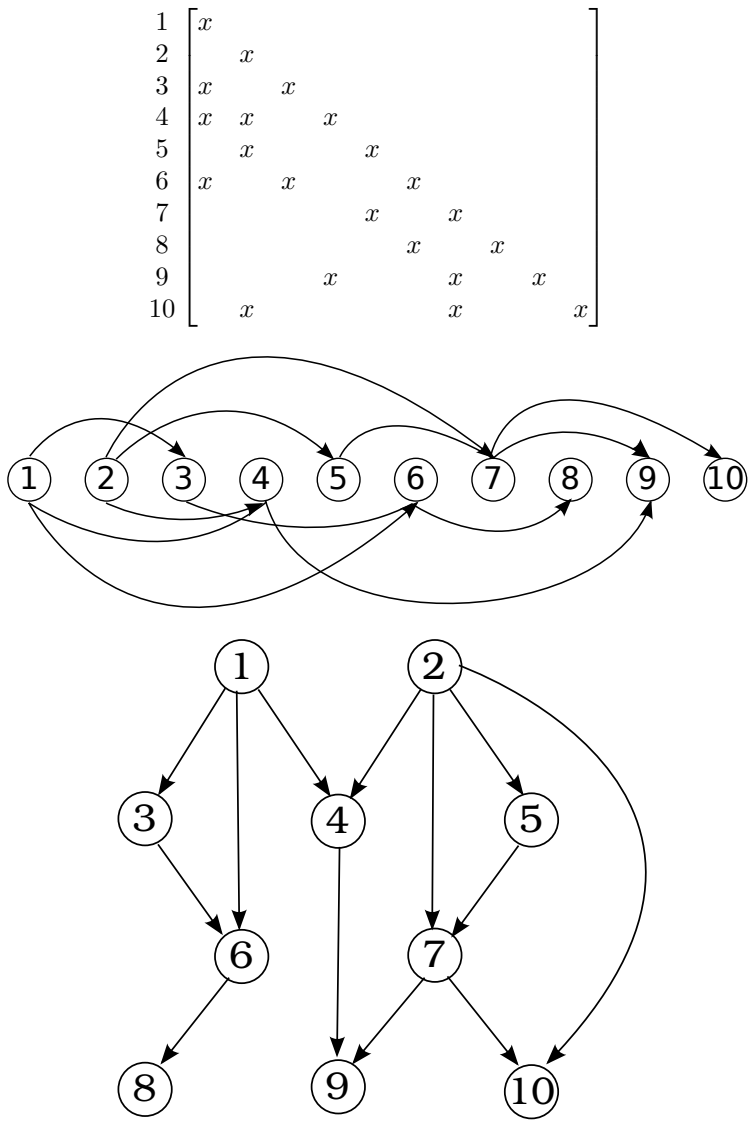


Figure A.3: Nonzero pattern of lower triangular sparse matrix (top), its DAG (center), and its level-set reordering (bottom).

A commonly used algorithm to compute the *level sets* in a DAG finds a *topological ordering* of the graph using a variation of Kahn's algorithm [66]. It consists on successively finding all the nodes that have no incoming edges (root nodes), adding them to the ordered list, and removing their outgoing edges from the graph. In a given iteration, the root nodes are the nodes that can be processed in parallel, given that they have no dependencies, and hence form the current level. Removing their outgoing edges ensures that the root nodes of one iteration are the nodes that depended only on the root nodes of the previous iteration. After all nodes have been processed, this procedure retrieves an ordered list of nodes *iorder*, where those that belong to the same level are grouped together, and a list of pointers *ilevels* that indicates the starting position of each level in *iorder*.

In [76] Naumov presented a GPU implementation of this approach. The implementation is based on three GPU kernels that are executed iteratively until all the nodes of the graph

have been processed. The FIND_ROOTS kernel loads a buffer with the list of root nodes in the current graph. Later, the ANALYZE kernel processes the list of root nodes and the current graph, removing the dependencies of the current root nodes. To optimize the process, it also produces a list of candidates to be root nodes in the next iteration, so that the FIND_ROOTS kernel only needs to process this list. Naumov utilizes two buffers for storing root nodes. One of them stores the root nodes that need to be processed in the current iteration, while the other is used to write the root nodes of the next iteration. To avoid copying data between these two arrays, Naumov simply flips the pointers to the arrays at the end of each iteration. To the best of our knowledge, the implementation distributed with the CUSPARSE library follows these ideas.

Another approach that could be used to perform the analysis phase of the parallel SPTRSV is based on computing the *depth* of each node. In [106], Wing and Huang define the *depth* of a node v_i as the maximum distance from an initial node to v_i. In the case of DAGs that represent task dependencies, the *depth* of a node i represents the minimum step at which task i will have its dependencies fulfilled. With this definition, a *level* of the DAG can be seen as a group of nodes that share the same *depth*. As stated by Anderson and Saad in [18], the *depth* of each node can be calculated in one sweep through the adjacency matrix L following

$$depth(i) = \begin{cases} 1 & \text{if } l_{ij} = 0 \ \forall j < i \\ \max_{j<i}\{1 + depth(j) : l_{ij} \neq 0\} & \text{otherwise} \end{cases}$$

Once the *depth* of each node is calculated, the *iorder* vector can be computed in $O(n)$ time, where n is the dimension of the matrix. One possible way of calculating such an ordering could be the following:

- Given an array *idepth* of length n containing the *depth* of each node, obtain the maximum, which is equal to the total number of levels.
- Allocate the vector *ilevels* and initialize it such that *ilevels*(i) contains the number of nodes in level i.
- Performing a *scan* operation on this vector will yield the starting position of each level in the final *iorder* array, which is the final content of *ilevels*.
- Maintaining an *offset* variable for each level, assign each node j to the *iorder* array the following way

  $$iorder(ilevels(idepth(j)) + offset(idepth(j))) = j$$

  incrementing the offset by 1 afterwards.

A.2 Related work

In this section we offer a revision of the state of the art in the parallel solution of sparse triangular linear systems. Specifically, we describe several efforts grouped around the two

main paradigms that have prevailed overtime and present well known GPU implementations, namely, level-set and synchronization-free approaches.

A.2.1 Level-set based methods

The foundation for this kind of algorithms was established by the work of O. Wing [106]. In this study, the author focuses on using the DAG representation to perform the analysis of the data dependencies between operations that compose the resolution of the sparse triangular systems resulting from matrix factorizations.

Later, seeking to improve the computational performance of methods corresponding to the family of the preconditioned conjugate gradients on different parallel platforms, Saad and Schultz [91] studied the solution of sparse triangular linear systems that arise in Incomplete LU (ILU) preconditioners. The main contributions of the work are the analysis of the *wavefront* parallelism pattern and the benefits of applying a *red-black* re-ordering of the unknowns prior to the incomplete factorization. The idea behind the *wavefront* consists on that the unknowns corresponding to the *wavefront* can be solved in parallel, and the advance of the *wavefront* implies that all previous unknowns have been processed and the data generated is accessible.

The parallelization of ILU preconditioners is also addressed by Saltz in [93]. This time, the author focuses on problems derived from 2D grids, directly mapping the wavefront parallelism pattern to the grid. He also uses the DAG, but in this case each node represents one row/column of the corresponding matrix. Later, Anderson and Saad [17] presented a study of different strategies to solve triangular linear systems. One of the proposed methods is a reordering scheme for the sparse matrix which they call *level scheduling* after the way nodes are organized in the adjacency graph.

Regarding the use of GPUs to perform the SpTrSV, Naumov [76] presented a GPU implementation following the approach described in Section A.1.1. The routine is highly optimized and distributed as part of the cuSparse library. According to the author, ILU preconditioned iterative solvers using this implementation of the SpTrSV, obtain an average speedup values of 2× over multi-threaded MKL counterparts. However, the results obtained are too fluctuating and strongly depend on the sparse-pattern of the matrix.

A.2.2 Dynamically scheduled algorithms

The article by George et al. [52] was one of the first works that proposed a self-scheduled approach to solve triangular sparse linear systems on shared memory processors. The procedure is based on a pool of tasks that wait until their data dependencies have been resolved to commence their execution. Later, Saltz [92] extended the analysis of asynchronous algorithms in order to consider runtimes. Finally, Rothberg [85] improved this new paradigm by making a better use of cache levels, working with an incomplete Cholesky preconditioned iterative method.

In the general purpose massively parallel computing context, Liu et al. [71] recently presented an asynchronous method able to take advantage of the computational power offered by GPUs. There are practically no significant research efforts applying this strategy

on hardware accelerators other than this work.

The authors propose the novel GPU procedure as a means to overcome two main drawbacks of the level-set based cuSparse implementation. On the one hand, they seek to lower the pre-processing cost associated with this approach, presenting a very lightweight pre-processing stage and, on the other hand, they intend to avoid the excessive synchronization with the CPU, which harms the performance of cuSparse routine in many cases, by using a synchronization-free strategy.

The *analysis phase* consists in calculating the number of dependencies that each row has. This value is equivalent to the number of non zeros (nnz) in the row minus one. As the implementation targets matrices in CSC format, this data is collected in the GPU by a parallel histogram-like procedure, and stored in an auxiliary vector.

Once the analysis has completed, the *solution phase* proceeds by assigning a *warp* to each column of the sparse matrix. The computations performed by each thread can be divided in three sub-stages: *lock-wait*, *critical section* and *lock-update*. At the first stage, the threads of a warp *busy-wait* until all their dependencies are resolved. This information is obtained through the use of a global vector of dependencies. Columns with no dependencies can advance to the *critical section* stage. This stage involves two main tasks. First, the solution element associated with that column is calculated, and second, the values of the column are multiplied by the solution element and subtracted from the solution vector using atomic operations, which was previously initialized with the right hand side. In the final stage, *lock-update*, atomic operations are used again to update the corresponding positions in the dependencies vector. The implementation is able to leverage the *shared memory* of the multiprocessors, by maintaining the data of the unknowns corresponding to warps of the same thread block in the fast memory.

A.3 Sync-free GPU triangular solver for CSR matrices

The algorithm that we shall present in the following lines falls under the *Synchronization-Free* category that we have discussed earlier. It is based on scheduling warps to solve unknowns as soon as their dependencies have been fulfilled. Although the main idea is similar to the one underlying the work by Liu et al. [71], our proposal is targeted for the CSR matrix storage format.

In our proposal, the procedure advances row-wise, assigning a warp for each unknown. The warp must busy wait until the rows on which it depends have been processed. In other words, the entries in the vector of unknowns that correspond to the column index of the nonzero elements of its row must have their final values. In order to keep track of this, we store an integer *ready* vector, which has an entry for each unknown, that is set to one if it has been solved, and is equal to zero otherwise.

The busy-waiting procedure that each warp must undergo before processing its assigned row consists on iteratively polling the corresponding entries of the *ready* vector until they all have been set to one by other warps. In each iteration, every thread of the warp fetches the value of the *ready* vector in the position equal to the column index of one of the non-zeros of its assigned row into a register, unless it has already obtained a value of *one* in a previous

iteration. The values are then reduced by a warp-voting primitive that returns one if the value in the corresponding register is nonzero for all the active threads in the warp, and returns zero otherwise. We keep two versions of the *ready* vector, one in global memory and the other distributed across the shared memory of the different thread blocks. Each time a warp updates its corresponding *ready* value, it must do it in the global memory and in the shared memory of its block. This allows that when a warp depends on a value that is to be updated by another warp of the same thread block, the thread polling for that value can do so in the much faster shared memory. This situation is common in matrices that concentrate most nonzero entries close to the diagonal. We also use a similar mechanism to update and fetch the value of the unknowns that are solved by a warp in the same thread block.

For this process to be successful, it is necessary that the warp scheduler of the SM activates the warps that will fulfill the dependencies of the current warp between one iteration and the other. We rely on the fact that accessing the *ready* vector will cause the warp to halt waiting for the value to be fetched from memory. Furthermore, although the *ready* vector is initially stored in global memory, there will be only a moderate number of entries being polled at a given moment, so they will hopefully be read from the cache most of the times.

The CUDA execution model specifies that there is only a number of resident blocks in each Streaming Multiprocessor (SM). The warps that belong to this blocks have all their resources allocated in the SM and can issue an instruction as soon as its operands are ready. Non-resident blocks, however, have to wait until resident blocks finish their execution, so no resident warp should wait for data that is to be produced by a non-resident one, since it would cause a deadlock. The triangular structure of the matrix ensures that each unknown depends on others of lower numbering. If each warp processes the row corresponding to its warp identifier, no deadlock should occur as long as the warp identifiers of the active (or resident) warps are always lower than those of the non-resident warps. Unfortunately, although the evidence suggests that this property holds for most GPU architectures (if not all), the order in which blocks are issued to the SMs is not specified by the manufacturer. For this reason, we use a global variable in the GPU memory that the first thread of each block must read and increment before commencing its computations. The value read from this variable is used instead of the block identifier to form the warp identifier. This has a cost of one atomicAdd per block, which is negligible in practice.

Once all the dependencies have been met, the algorithm can move on to the multiplying stage, in which each thread of the warp multiplies a nonzero value of the current row by the appropriate entry of the vector of unknowns, accumulating the result in a register.

If there are more than 32 nonzero elements in the row, the warp moves back to busy-waiting for the solution of the unknowns that correspond to the next 32 nonzero elements. The cycle is repeated until every nonzero in the row has been processed.

After the multiplying stage, the registers that accumulate the products performed by each thread of the warp are reduced using warp shuffle operations, and the result is then stored in the vector of unknowns, divided by the diagonal pivot, by the first thread of the warp.

Finally, the first thread of the warp is responsible of marking the unknown as solved in

the *ready* vector.

A simplified version of the source code of our GPU kernel (we omit the shared memory part and the selective fetching of the *ready* vector for clarity) is presented in Figure A.4. The input parameters are the three vectors representing the sparse matrix stored in CSR format, the right hand side, the dimension of the system, and a pointer to the memory reserved for the *ready* vector. As an output parameter, the function receives a pointer to the vector of unknowns. VALUE_TYPE stands for the floating point precision utilized.

Note that this algorithm requires no pre-processing stage other than setting to zero the *ready* vector. Moreover, working by rows makes possible to avoid the use of slow atomic operations.

A.4 A massively parallel level set analysis

As mentioned in Section A.1.1, the *depth* of a node, which is equivalent to the *level* that equation belongs to, can be computed by adding one to the maximum of the *depths* of the nodes it depends on. If the node has no dependencies its *depth* is simply one. Hence, the *depth* of a node can be computed entirely only after the *depths* of the nodes corresponding to its incoming edges are computed. As our parallel GPU algorithm assumes the graph is stored as a sparse adjacency matrix in CSR storage format, nodes map to rows of the matrix, and the incoming edges of each node map to the nonzero elements of each row.

As in the case of the Synchronization-Free solver routine, we launch a *warp* for each row of the sparse matrix. In order to compute the *depth* of the corresponding node, the warp must first busy wait until the rows on which it depends have been processed. In other words, the entries in the vector of *depths* that correspond to the column index of the nonzero elements of its row must have their final values. In order to keep track of this, we store an integer *ready* vector, which has an entry for each node, that is set to one if it has been processed, and is equal to zero otherwise. It should be noted that only one thread writes to each entry of these two vectors.

The outline of the algorithm is similar to that of the solver routine. The main differences lies in the *multiplying*, *reduction*, and *update* stages. Instead of multiplying the coefficient of the matrix by the value of the solution vector, accumulating the result in a local variable for each thread of the warp, the corresponding stage in the analysis routine has to fetch the value from the vector of *depths* in the corresponding position and store the maximum in each local variable. After this, the reduction stage computes the maximum by an intra-warp reduction. Finally, the update stage stores the reduced value plus one in the entry of the *depths* vector corresponding to the node processed by the warp.

A simplified version of the source code of our kernel is presented in Figure A.5. The input parameters are the two vectors representing the sparse adjacency matrix stored in CSR format, the dimension of the system, and a pointer to the memory reserved for the *ready* vector. As an output parameter, the function receives a pointer to the vector of *depths*.

In order to leverage some data locality, we maintain a copy of the entries of the *depths* and *ready* vectors that correspond with the nodes that will be processed by the thread block in the shared memory. When threads depend on nodes that are to be processed by warps

of the same thread block, we can fetch and update the corresponding values in the shared memory instead of using the much slower global memory. We do not include this feature in the outline of Figure A.5 for the sake of simplicity.

A.5 Combining the analysis and solution stages

It is clear that our analyzer follows the *self-scheduled* strategy to compute the level of each row instead of the solution of the linear system, and that both algorithms bare a remarkable resemblance. This situation motivates the combination of both analysis and solution phase of the SpTrSV in only one pass of our synchronization-free method. As the maximum depth of the nodes in the matrix, which is equivalent to the number levels, is a relevant factor on the final performance of the level-based strategy, it can be used to decide which of the two approaches is the most convenient for a given matrix.

Taking this into account,

we can extend the algorithm for the analysis so that it receives an initial vector of unknowns (set to zero), the right hand side vector and the values of the triangular matrix, computing the solution of the linear system in the same pass used to compute the depths of the nodes. This is useful in a scenario where it is necessary to solve a small number of linear systems with the same coefficient matrix, as it is often the case in ILU preconditioned iterative methods. Here, the solution of the first of the linear systems also produces the level information that can be used to solve the following triangular systems with the two-stage approach, if the performance estimation determines that this is the best strategy.

A.6 Experimental evaluation

This section summarizes the experiments conducted to evaluate the novel synchronization-free routines to perform the solution and level-set analysis of sparse triangular systems. The results for the solution routine are contrasted with the ones obtained by the cuSparse library, which we will refer to as *level-based*, since it is the most widely used implementation of this kernel, and with the proposal by Liu et al. [71], from here on called *CSC Sync. Free*, as it is, to our best knowledge, the only publicly available Synchronization-Free proposal for GPUs.

In turn, the results for the new analysis routine are contrasted with the ones obtained by the cuSparse library, since it is also the most widely used implementation of this operation. We are not aware of any other widespread and publicly available library that performs a level based solution of sparse triangular systems using a different approach.

All results were obtained using IEEE single- and double-precision floating point arithmetic.

A.6.1 Test cases

The benchmark utilized for the experiments is a set of moderate and large sparse matrices from the SuiteSparse Matrix Collection. In Table A.1 we display, for each matrix, its di-

Table A.1: Some features of the employed matrices. *Levels* is the number of levels in the DAG computed by the analysis phase.

Matrix	n	nnz	*Levels*
cant	62,451	2,034,917	2,397
chipcool0	20,082	150,616	534
crankseg_1	52,804	5,370,437	2,218
hollywood-2009	1,139,905	57,515,616	82,735
nlpkkt160	8,345,600	118,931,856	2
road_central	14,081,816	31,015,229	59
road_usa	23,947,347	52,801,659	77
ship_003	121,728	4,103,881	4,367
webbase-1M	1,000,005	2,348,442	512
wiki-Talk	2,394,385	3,072,221	515
cit-HepTh	27,770	4,244,363	183
dc2	116,835	666,173	14
epb3	84,617	3,313,794	879
g7jac140sc	41,490	10,102,488	291
lung2	109,460	273,646	479
rajat18	94,294	280,494	42
rajat25	87,190	1,458,168	137
soc-sign-epinions	131,828	271,947	444
TSOPF_RS_b162_c4	20,374	1,565,005	114
wordnet3	82,670	165,459	8

mension, number of non-zeros and the amount of levels produced by the analysis routine of cuSparse. As the exploitation of parallelism in the *level-based* approach consists in processing the rows that belong to a given level in parallel, dividing the number of rows by the number of levels can give a rough estimation of the parallelism available for that matrix.

The selection of test cases is the same than that of [72] in order to make sure that the results obtained for the *CSC Sync. Free* implementation are comparable. For the evaluation of the solver routine we used the first set of 10 matrices, while for the analysis routine we used the whole 20. For the evaluation of the combined solver we used only the first set, performing a similar study than the one conducted in [47].

A.6.2 Hardware platforms

The performance evaluation of our proposals was carried out using four different platforms. The first one is platform Beethoven, presented in Section 3.1, while the remaining are presented next.

Ravel

Equipped with an Intel(R) Core(TM) i7-6700 CPU processor (8 cores at 3.40 GHz) and 64 GB of DDR3 RAM. The platform also features a GTX1060 (Pascal) GPU with 1,152 CUDA Cores and 3 GB of GDDR5 RAM. The compiler for this platform is gcc 14.0.0 and for GPU codes we use the CUDA/CUSPARSE 8.0 libraries.

Sibelius

Equipped with an Intel(R) Core(TM) i7-6700 CPU processor (8 cores at 3.40 GHz) and 64 GB of DDR3 RAM. The platform also features a NVIDIA GTX 1080 Ti (Pascal) GPU with

3,584 CUDA Cores and 11 GB of GDDR5X RAM. The compiler for this platform is gcc 14.0.0 use the CUDA/CUSPARSE 8.0 for the GPU codes and libraries.

A.6.3 Evaluation of the solver routine

Tables A.2 and A.3 summarize the runtimes obtained for the execution in BEETHOVEN and RAVEL respectively. Both tables display the execution time of the *level-based* and *CSC Sync. Free*, which is separated in the analysis and solve phases, and the execution time for our CSR Synchronization-Free solver, in single and double precision. Additionally, we show the speedups obtained when comparing our solution to the two previous approaches.

The acceleration factors relative to the *level-based* implementation are heterogeneous, ranging from 3× in favour of the *level-based* approach to the 10× in favour of our solver that were obtained in BEETHOVEN using single precision. If the time devoted to the analysis phase is considered, the *level-based* implementation is outperformed by our solver for all the tested instances, with some speedups reaching values up to 25×. Although it is true that in some scenarios, as for example ILU-preconditioned iterative solvers, the analysis is performed only once for each matrix while the level information can be used to solve multiple linear systems, the benefits of the *level-based* strategy start to appear after a considerable amount of iterations. As an example, the results on RAVEL, using single precision, show that the solving phase of the *level-based* strategy is almost 2× faster than our synchronization-free variant for the *webbase-1M* problem. However, the *level-based* approach starts being convenient only after solving 13 linear systems with the same analysis information.

A more detailed analysis of the results shows that our solver is outperformed by the solving phase of CUSPARSE implementation for four of the matrices (*nlpkkt160, road_central*, and *webbase-1M*) and is on the same order of performance for the *wiki-Talk* matrix. This suggests that the *level-based* approach is superior in two situations. The *nlpkkt160* matrix is large, has an enormous amount of parallelism, and the cost of synchronization is negligible relative to the cost of the computations. In this case, the pre-processing allows to schedule the parallel solution of as many rows as possible, fully utilizing the computational units, while our busy-waiting strategy adds unnecessary overhead and may cause that rows that have no dependencies but are located in the bottom of the triangular matrix are delayed in their processing, as they belong to initially inactive blocks. The other three instances correspond to matrices that have fairly regular sparsity patterns in which the nonzero elements are clustered in small blocks. This can generate a great deal of dependencies between nearby rows, so the CUSPARSE solver can benefit from the global information obtained in the analysis phase in order to produce a better scheduling of the execution, while our strategy proceeds almost blindly, based on rather local information. Moreover, it should be remarked that in the cases where the CUSPARSE solver obtains the best performance, the cost of the analysis phase is higher in proportion. This can be observed by comparing the ratio between the speedup of our proposal against the solution phase of CUSPARSE and the speedup against both phases added together.

Compared to the *CSC Sync. Free* implementation, and without considering the double precision results in BEETHOVEN, where our proposal is clearly better, our solver presents

Table A.2: Runtime (in milliseconds) to solve triangular sparse linear systems with single and double precision in Beethoven.

Matrix	cuSparse			CSC Sync-free			Proposal	Speedup vs.			
	Analysis time	Resolve time	Total time	Analysis time	Resolve time	Total time		CUSPARSE		CSC S-F	
								Solve	+Anal.	Solve	+Anal.
single precision											
chipcool0	12,42	3,72	16,14	0,04	1,01	1,04	0,82	4,56	19,79	1,24	1,28
nlpkkt160	856,66	15,67	872,33	7,17	45,40	52,57	45,88	0,34	19,01	0,99	1,15
hollywood-2009	1.877,85	1.312,37	3.190,22	5,17	214,37	219,53	129,65	10,12	24,61	1,65	1,69
road_central	456,71	22,01	478,72	9,34	81,82	91,16	77,87	0,28	6,15	1,05	1,17
road_usa	789,55	34,66	824,20	5,05	165,47	170,52	157,06	0,22	5,25	1,05	1,09
ship_003	83,03	39,90	122,93	0,20	11,04	11,24	7,09	5,63	17,33	1,56	1,59
webbase-1M	40,04	4,70	44,74	0,23	7,10	7,33	7,10	0,66	6,30	1,00	1,03
wiki-Talk	75,65	15,02	90,67	0,36	10,54	10,91	10,04	1,50	9,03	1,05	1,09
crankseg_1	87,44	38,07	125,51	0,27	17,29	17,56	10,47	3,63	11,98	1,65	1,68
cant	44,12	23,49	67,60	0,12	5,12	5,25	4,13	5,69	16,37	1,24	1,27
double precision											
chipcool0	12,30	4,07	16,36	0,04	1,70	1,73	1,01	4,02	16,19	1,68	1,72
nlpkkt160	857,16	18,83	875,99	7,18	126,99	134,17	57,11	0,33	15,34	2,22	2,35
hollywood-2009	1.879,79	1.390,01	3.269,79	5,22	341,45	346,67	176,63	7,87	18,51	1,93	1,96
road_central	459,40	25,92	485,32	9,33	114,92	124,25	93,42	0,28	5,20	1,23	1,33
road_usa	790,26	40,62	830,88	5,09	239,26	244,35	192,00	0,21	4,33	1,25	1,27
ship_003	84,21	41,78	125,99	0,21	16,86	17,07	9,16	4,56	13,76	1,84	1,86
webbase-1M	39,90	4,99	44,89	0,23	11,33	11,56	8,72	0,57	5,15	1,30	1,32
wiki-Talk	76,10	15,73	91,83	0,36	14,54	14,91	12,91	1,22	7,11	1,13	1,15
crankseg_1	87,25	40,26	127,51	0,25	26,87	27,12	13,99	2,88	9,12	1,92	1,94
cant	43,62	24,52	68,14	0,12	9,29	9,41	5,09	4,82	13,39	1,83	1,85

Table A.3: Runtime (in milliseconds) to solve triangular sparse linear systems with single and double precision in RAVEL. The missing runtimes correspond to matrices that did not fit in the GPU memory of the GTX1060.

Matrix	cuSPARSE			CSC Sync-free			Proposal	Speedup vs.			
	Analysis time	Resolve time	Total time	Analysis time	Resolve time	Total time		CUSPARSE		CSC S-F	
								Solve	+Anal.	Solve	+Anal.
single precision											
chipcool0	7,42	1,61	9,02	0,06	0,38	0,44	0,37	4,31	24,19	1,02	1,17
nlpkkt160	-	-	-	-	-	-	-	-	-	-	-
hollywood-2009	1.186,62	515,20	1.701,82	6,88	123,95	130,83	78,00	6,61	21,82	1,59	1,68
road_central	393,85	18,75	412,59	10,14	64,12	74,26	63,11	0,30	6,54	1,02	1,18
road_usa	-	-	-	-	-	-	-	-	-	-	-
ship_003	60,46	17,20	77,66	0,79	6,38	7,17	4,47	3,85	17,39	1,43	1,61
webbase-1M	32,69	2,56	35,25	1,11	4,50	5,61	5,13	0,50	6,87	0,88	1,09
wiki-Talk	54,63	7,01	61,64	1,27	6,82	8,08	6,68	1,05	9,23	1,02	1,21
crankseg_1	61,05	15,91	76,95	0,74	9,78	10,51	6,12	2,60	12,57	1,60	1,72
cant	30,99	10,95	41,94	0,48	2,82	3,30	2,21	4,95	18,94	1,28	1,49
double precision											
chipcool0	7,45	1,88	9,34	0,30	0,56	0,86	0,61	3,07	15,23	0,91	1,40
nlpkkt160	-	-	-	-	-	-	-	-	-	-	-
hollywood-2009	-	-	-	-	-	-	-	-	-	-	-
road_central	394,08	29,37	423,45	10,22	86,05	96,27	101,41	0,29	4,18	0,85	0,95
road_usa	-	-	-	-	-	-	-	-	-	-	-
ship_003	59,53	18,96	78,49	0,76	7,59	8,34	7,48	2,53	10,49	1,01	1,11
webbase-1M	32,67	3,59	36,26	1,05	6,09	7,14	8,61	0,42	4,21	0,71	0,83
wiki-Talk	54,89	9,82	64,70	1,28	11,56	12,84	15,79	0,62	4,10	0,73	0,81
crankseg_1	60,69	18,43	79,11	0,71	11,60	12,31	10,57	1,74	7,49	1,10	1,17
cant	31,49	11,24	42,73	0,50	3,34	3,84	3,48	3,23	12,27	0,96	1,10

interesting performance gains in four instances (*hollywood-2009, ship-003, cantilever*, and *crankseg_1*), while performing similarly for the others. In this case, it is more difficult to derive the reasons for our superior performance, since both strategies are based on the same principles.

We have observed that, for the cases in which we obtain performance peaks, the amount of inactive cycles that a warp has to wait before processing its data is significantly lower for our solver. A key difference between the two approaches that could explain this results is that, in the *CSC Sync. Free* algorithm, the operations that need to be performed before a warp can exit the busy-waiting phase are necessarily serial (atomic additions to an entry in the *in_degree* vector), while in our case they could be performed in parallel by different warps. For this to be a real advantage, it is required that the unknowns for which a warp is waiting have no dependencies between them, which is completely dependent of the sparsity pattern of the matrix.

A.6.4 Evaluation of the analysis routine

The performance evaluation of the analysis routine was carried out in platforms RAVELand SIBELIUS. These two platforms are actually one server equipped with two NVIDIA graphic cards. Although both GPUs belong to the Pascal generation, they present different characteristics, as one of them is a low-end gaming GPU (the GTX 1060) and the other is one of the most powerful gaming graphic cards available (the GTX 1080 Ti).

Experimental Results

Table A.4 summarizes the runtimes obtained for the execution of *level-based* and our variant to perform the analysis phase of the SPTRSV in both the employed hardware platforms. The acceleration factor, computed as $\frac{level-based\ Time}{Proposal\ Time}$, is also provided.

The differences between the execution times obtained in the two graphic cards vary according to the test instance. In general, the benchmark executed faster in the SIBELIUSGPU, as it was expected, with differences that range from marginal to more than 2×. There are, however, a few counter intuitive results but, as in both algorithms the execution schedule can be greatly affected by the sparsity pattern of the matrix, a much deeper and complex analysis is required in order to explain them adequately.

The runtimes obtained in the experiments show a clear advantage of our approach. Our routine to compute the *depth* of the nodes in the DAG presents acceleration factors of up to 44× with respect to *level-based* analysis routine for the evaluated instances. The most significant acceleration is obtained in the SIBELIUS GPU for the *nlpkkt160* case, that is the largest instance in terms of memory usage.

It should be noted that the CUSPARSE analysis of the *nlpkkt160* and *road_usa* matrices failed in the RAVEL device, as the 3GB of DRAM in the accelerator were not sufficient to host the additional data structures generated by the routine. This reveals another advantage of our method, as we only need to allocate space for the auxiliary *is_solved* vector, while the CUSPARSE routine needs an amount of additional memory proportional to the number of

Table A.4: Runtime (in milliseconds) of the analysis phase of CUSPARSE and our analysis routine in platforms RAVEL and SIBELIUS.

Matrix	cuSPARSE time	Proposal time	Speedup
RAVEL			
cant	32.00	2.18	14.68
chipcool0	8.27	0.46	17.98
crankseg_1	61.43	5.94	10.34
hollywood-2009	1,179.61	76.48	15.42
nlpkkt160	-	25.61	-
road_central	465.00	57.59	8.07
road_usa	-	128.20	-
ship_003	61.00	4.30	14.19
webbase-1M	37.49	4.67	8.03
wiki-Talk	63.87	5.85	10.92
cit-HepTh	5.70	0.29	19.66
dc2	13.75	0.83	16.57
epb3	12.61	11.65	1.08
g7jac140sc	7.04	0.43	16.37
lung2	9.58	1.34	7.15
rajat18	7.64	0.71	10.76
rajat25	8.66	0.94	9.21
soc-sign-epinions	13.39	0.72	18.60
TSOPF_RS_b162_c4	7.29	0.69	10.57
wordnet3	5.61	0.14	5.61
SIBELIUS			
cant	29.73	3.14	9.47
chipcool0	8.31	0.53	15.68
crankseg_1	54.37	4.33	12.56
hollywood-2009	1,113.87	114.75	9.71
nlpkkt160	424.75	9.66	43.97
road_central	271.23	20.77	13.06
road_usa	464.23	46.19	10.05
ship_003	54.27	2.71	20.03
webbase-1M	30.27	2.15	14.08
wiki-Talk	48.80	2.87	17.00
cit-HepTh	5.67	0.31	18.29
dc2	13.59	0.38	35.76
epb3	13.02	5.65	2.30
g7jac140sc	7.46	0.30	24.87
lung2	10.39	0.91	11.42
rajat18	8.47	0.42	20.17
rajat25	9.30	0.58	16.03
soc-sign-epinions	13.46	0.58	23.21
TSOPF_RS_b162_c4	6.07	0.55	11.04
wordnet3	5.59	0.33	16.94

Table A.5: Runtime (in ms) of our combined analysis and solution routine in platform RAVEL. The column overhead represents the additional cost of solving one triangular system during the analysis. The values in the last column are calculated as the added time of performing our analysis phase followed by CUSPARSE's solution phase. divided by the execution time of the combined analysis and solution routine.

Matrix	CUSPARSE solution	Proposal analysis	CUSPARSE sol. + prop. analysis	Combined routine	Overhead %	Accel. factor
single precision						
cant	11.01	2.18	13.19	2.75	20.73	4.80
chipcool0	1.79	0.46	2.25	0.52	11.54	4.33
crankseg_1	15.87	5.94	21.81	7.70	22.86	2.83
hollywood-2009	516.87	76.48	593.35	97.17	21.29	6.11
nlpkkt160	-	25.61	-	34.02	24.72	-
road_central	19.46	57.59	77.05	71.26	19.18	1.08
road_usa	-	128.20	-	146.70	12.38	
ship_003	17.02	4.30	21.32	5.49	21.68	3.88
webbase-1M	2.56	4.67	7.23	5.82	19.76	1.24
wiki-Talk	7.07	5.85	12.92	7.70	24.03	1.68
double precision						
cant	11.04	2.13	13.17	3.23	34.06	4.08
chipcool0	1.87	0.36	2.23	0.51	29.41	4.37
crankseg_1	18.38	6.00	24.38	8.94	32.89	2.73
hollywood-2009	584.40	76.63	661.03	112.00	31.58	5.90
nlpkkt160	-	25.61	-	61.89	58.46	-
road_central	29.60	58.66	88.26	110.77	47.04	0.80
road_usa	-	128.20	-	205.99	37.76	-
ship_003	18.92	4.29	23.21	6.53	34.30	3.55
webbase-1M	3.61	4.71	8.32	8.91	47.14	0.93
wiki-Talk	9.83	5.86	15.69	17.35	66.22	0.90

non-zeros of the sparse matrix[3].

A.6.5 Evaluation of the combined routine

As our routine is capable of solving a triangular linear system on the same pass that computes the level analysis, we compare the combined analysis and solution against performing our analysis (as it is clearly the best option for our benchmark) and then performing the solution of the linear system using the CUSPARSE solver. Here, we are assuming that we should be able to develop a sparse triangular solver that uses our analysis information with a performance similar to that of CUSPARSE solver. We also record the overhead implied by the solution of the system relative to the cost of the analysis, which we calculate as

$$\frac{T_{\text{combined}} - T_{\text{analysis}}}{T_{\text{combined}}} \times 100$$

Tables A.5 and A.6 show the results obtained for our first set of matrices in platforms RAVEL and the SIBELIUS, respectively, considering the solution of the system using single and double precision arithmetic. The results obtained in the two platforms coincide in that the simultaneous solution of the triangular system has a relatively small impact on the performance of the analysis when using single precision, while it can imply a much larger overhead when using double. This is expected and it is related to the large performance gap between single and double precision in this type of graphic cards.

[3] See for example, CUSPARSE Library user's guide DU-06709-001_v8.0, January 2017.

Table A.6: Runtime (in milliseconds) of our combined analysis and solution routine in platform SIBELIUS. The column overhead represents the additional cost of solving one triangular system during the analysis. The values in the last column are calculated as the added time of performing our analysis phase followed by cuSPARSE's solution phase, divided by the execution time of the combined analysis and solution routine.

Matrix	cuSPARSE solution	Proposal analysis	cuSPARSE sol. + prop. analysis	Combined routine	Overhead %	Accel. factor
single precision						
cant	10,98	3,14	14,12	3,59	12,53	3,93
chipcool0	1,60	0,53	2,13	0,74	28,38	2,88
crankseg_1	16,40	4,33	20,73	5,54	21,84	3,74
hollywood-2009	524,30	114,75	639,05	131,86	12,98	4,85
nlpkkt160	5,56	9,66	15,22	13,17	26,65	1,16
road_central	9,26	20,77	30,03	26,03	20,21	1,15
road_usa	14,30	46,19	60,49	56,53	18,29	1,07
ship_003	17,45	2,71	20,16	3,48	22,13	5,79
webbase-1M	2,02	2,15	4,17	2,59	16,99	1,61
wiki-Talk	6,10	2,87	8,97	3,66	21,58	2,45
double precision						
cant	11,61	3,35	14,96	4,01	16,46	3,73
chipcool0	1,86	0,54	2,40	0,69	21,74	3,48
crankseg_1	18,91	4,34	23,25	6,31	31,22	3,68
hollywood-2009	597,72	115,77	713,49	141,15	17,98	5,05
nlpkkt160	8,04	9,70	17,74	21,21	54,27	0,84
road_central	12,53	20,82	33,35	37,97	45,17	0,88
road_usa	19,24	46,38	65,62	72,94	36,41	0,90
ship_003	19,67	2,76	22,43	4,07	32,19	5,51
webbase-1M	2,47	2,18	4,65	3,56	38,76	1,31
wiki-Talk	7,23	2,88	10,11	6,94	58,50	1,46

Regarding the convenience of the combined solution, the results suggest that this strategy could yield important performance benefits in cases where only a few triangular linear systems need to be solved. Moreover, for some of the tested instances, the runtime of our combined routine is even smaller than that of cuSPARSE solve phase. This means that it is better to solve the future triangular linear systems using our strategy (without performing the analysis) instead of using cuSPARSE.

```
__global__
void sptrsv_kernel(
        const int* __restrict__ row_ptr,
        const int* __restrict__ col_idx,
        const VALUE_TYPE* __restrict__ val,
        const VALUE_TYPE* __restrict__ b,
        VALUE_TYPE* x,
        int * is_solved, int n) {

    __shared__ int blIdx;

    //row_ctr is a global counter initialized in 0

    if(threadIdx.x==0) blIdx = atomicAdd(&row_ctr, 1);

    //Global thread warp identifier
    int wrp = threadIdx.x + blIdx * blockDim.x;
    wrp /= WARP_SIZE;

    if(wrp >= n) return;

    //Identifies the thread relative to the warp
    int lne = threadIdx.x & 0x1f;

    int row = row_ptr[wrp];
    int nxt_row = row_ptr[wrp+1];

    VALUE_TYPE left_sum = 0;
    VALUE_TYPE piv = 1 / val[nxt_row-1];

    int ready;
    int lock = 0;

    if(lne==0)
        left_sum = b[wrp];

    int off = row+lne;

    // Wait and multiply
    while(off < nxt_row - 1)
    {
        ready = is_solved[col_idx[off]];
        lock = __all(ready);

        if(lock)
        {
            left_sum -= val[off] * x[col_idx[off]];
            off+=WARP_SIZE;
        }
    }

    // Reduction
    for (int i=16; i>=1; i/=2)
        left_sum += __shfl_xor(left_sum, i, 32);

    if(lne==0){
        //Write the result
        x[wrp] = left_sum * piv;

        __threadfence();

        //Mark the equation as solved
        is_solved[wrp] = 1;
    }
}
```

Figure A.4: Simplified CUDA source code of our solution kernel.

```
__global__
void sptrsv_kernel(
        const int* row_ptr,
        const int* col_idx,
        int * depths,
        int * is_solved, int n) {

    __shared__ int blIdx;

    //row_ctr is a global counter initialized in 0

    if(threadIdx.x==0) blIdx = atomicAdd(&row_ctr, 1);

    //Global thread warp identifier
    int wrp = threadIdx.x + blIdx * blockDim.x;
    wrp /= WARP_SIZE;

    if(wrp >= n) return;

    //Identifies the thread relative to the warp
    int lne = threadIdx.x & 0x1f;

    int row = row_ptr[wrp];
    int nxt_row = row_ptr[wrp+1];

    int depht = 0;
    int ready = 0;
    int off = row+lne;
    int colidx = col_idx[off];

    while(off < nxt_row - 1)
    {

        // Wait and update local depth
        if(!ready)
        {
            ready = is_solved[colidx];
            if (ready){
                depth = max(depth, depths[colidx]);
            }
        }

        if( __all(ready) ){
            off+=WARP_SIZE;
            colidx = col_idx[off];
            ready=0;
        }

    }

    // Reduction
    for (int i=16; i>=1; i/=2)
        depth = max(depth, __shfl_down(depth, i));

    if(lne==0){
        //Write the result
        depths[wrp] = 1+depth;

        __threadfence();

        //Mark the equation as solved
        is_solved[wrp] = 1;
    }
}
```

Figure A.5: Simplified CUDA source code of our analysis kernel.

APPENDIX B

Balancing energy and efficiency in ILUPACK

For over two decades, the *LINPACK benchmark* [43] has been employed to compile performance and throughput-per-power unit rankings of most of the world's fastest supercomputers twice per year [1]. Unfortunately, this test boils down to the LU factorization [56], a compute-bound operation that may not be representative of the performance and power dissipation experienced by many of the complex applications running in current high performance computing (HPC) sites.

The alternative *High Performance Conjugate Gradients (HPCG) benchmark* [2, 41] has been recently introduced with the specific purpose of exercising computational units and producing data access patterns that mimic those present in an ample set of important HPC applications. This attempt to change the reference benchmark is crucial because such metrics may guide computer architecture designers, e.g. from AMD, ARM, IBM, Intel and NVIDIA, to invest in future hardware features and components with a real impact on the performance and energy efficiency of these applications.

The HPCG benchmark consists of basic numerical kernels such as the sparse matrix-vector multiplication (SpMV) and sparse triangular solve; basic vector operations as e.g. vector updates and dot products; and a simple smoother combined with a multigrid preconditioner. The reference implementation is written in C++, with parallelism extracted via MPI and OpenMP [2]. However, in an era where general-purpose processors (CPUs) as well as the Intel Xeon Phi accelerator contain dozens of cores, the concurrency level that is targeted by this legacy implementation may be too fine-grain for these architectures. Furthermore, the reference implementation is certainly not portable to heterogeneous platforms equipped with graphics processing units (GPUs) comprising thousands of simple arithmetic processing units (e.g., NVIDIA's CUDA cores).

This appendix describes our study of the performance and energy efficiency of state-of-the-art multicore CPUs and many-core accelerators, using the task and data-parallel versions of ILUPACK described in this thesis. Compared with the HPCG benchmark, these multi-

$A \to M$	O0. Preconditioner computation
Initialize $x_0, r_0, z_0, d_0, \beta_0, \tau_0; k := 0$	
while $(\tau_k > \tau_{\max})$	Loop for iterative PCG solver
$w_k := Ap_k$	O1. SpMV
$\alpha_k := \rho_k / p_k^T w_k$	O2. DOT product
$x_{k+1} := x_k + \alpha_k p_k$	O3. AXPY
$r_{k+1} := r_k - \alpha_k w_k$	O4. AXPY
$z_{k+1} := M^{-1} r_{k+1}$	O5. Apply preconditioner
$\rho_{k+1} := r_{k+1}^T z_{k+1}$	O6. DOT product
$p_{k+1} := z_{k+1} + (\rho_{k+1}/\rho_k) p_k$	O7. AXPY-like
$\tau_{k+1} :=\| r_{k+1} \|_2$	O8. vector 2-norm
$k := k + 1$	
endwhile	

Figure B.1: Algorithmic formulation of the preconditioned CG method. Here, $\tau_{\max}$ is an upper bound on the relative residual for the computed approximation to the solution.

threaded implementations of ILUPACK are composed of the same sort of numerical kernels and, therefore, exhibit analogous data access patterns and arithmetic-to-memory operations ratios. On the other hand, our task-parallel version of ILUPACK is likely better suited to exploit the hardware parallelism of both general-purpose processors and the Intel Xeon Phi, while our data-parallel implementation targets the large volume of CUDA cores in NVIDIA architectures.

Additionally, we analyze a data-parallel variant of ILUPACK adapted to low-power hardware platforms such as the NVIDIA Jestson TX1.

B.1 Characterizing the efficiency of hardware platforms using ILUPACK

In analogy with the HPCG benchmark, here we only consider linear systems with symmetric positive definite (SPD) coefficient matrix A, on which the preconditioned CG (PCG) solver underlying ILUPACK is applied.

Figure B.1 describes the PCG method algorithmically. The first step of the solver (O0) corresponds to the computation of the preconditioner M, while the subsequent iteration involves a SpMV (O1), the application of the preconditioner (O5), and several vector operations (DOT products, AXPY-like updates, vector norm; in O2–O4 and O6–O8). We emphasize that this same PCG iteration is the basis of the HPCG benchmark.

Exploiting task-parallelism in ILUPACK's PCG

Our task-parallel version of ILUPACK employs the task-based programming model embedded in the OmpSs[1] framework to decompose the solver into tasks (routines annotated by the user via OpenMP-like directives) as well as to detect data dependencies between tasks at execution time (with the help of directive clauses that specify the directionality and size of the task operands). With this information, OmpSs implicitly generates a task graph during the execution, which is utilized by the CPU threads in order to exploit the task parallelism implicit to the operation via a dynamic out-of-order but dependency-aware schedule.

[1] https://pm.bsc.es/ompss

Let us consider the PCG iteration. The variables that appear in these operations define a partial order which enforces an almost strict serial execution. Specifically, at the $(k+1)$-th iteration,

$$\ldots \to \mathrm{O7} \to \overbrace{\mathrm{O1} \to \mathrm{O2} \to \mathrm{O4} \to \mathrm{O5} \to \mathrm{O6} \to \mathrm{O7}}^{(k+1)\text{-th iteration}} \to \mathrm{O1} \to \ldots$$

must be computed in that order, but O3 and O8 can be computed any time once O2 and O4 are respectively available. Further concurrency is exposed by dividing the application of the preconditioner into subtasks of finer granularity, in a form analogous to that described in Section 2.4.2.

The data-parallel version used for the energy evaluation is the baseline GPU-aware variant of the CG method described in Section 3.2.

B.1.1 Hardware and software configurations

Here we present the hardware and software setup used for the energy evaluation. This is necessary to understand the ideas behind the optimization of the OmpSs implementation in the next section.

The evaluation involves INTEL processors from two different generations, as well as platforms equipped with GPU and XeonPhi accelerators.

All the experiments employed IEEE 754 real double-precision (DP) arithmetic on the following four platforms:

- SANDY: A server equipped with two hexacore Intel Xeon E5-2620 ("Sandy Bridge") processors (total of 12 cores) running at 2.0 GHz with 32 Gbytes of DDR3 RAM. The compiler is `gcc` 4.4.7.
- HASWELL: A system with two hexacore Intel Xeon E5-2603v3 ("Haswell") processors (total of 12 cores) at 1.6 GHz with 32 Gbytes of DDR4 RAM. The compiler is `gcc` 4.4.7.
- XEON PHI: A board with an Intel Xeon Phi 5110P co-processor. (The tests on this board were ran in native mode and, therefore, the specifications of the server are irrelevant.) The accelerator comprises 60 x86 cores running at 1,053 MHz and 8 Gbytes of GDDR5 RAM. The Intel compiler is `icc` 13.1.3.
- KEPLER: An NVIDIA K40 board ("Kepler" GK110B GPU with 2,880 cores) with 12 Gbytes of GDDR5 RAM, connected via a PCI-e Gen3 slot to a server equipped with an Intel i7-4770 processor (4 cores at 3.40 GHz) and 16 Gbytes of DDR3 RAM. The compiler for this platform is `gcc` 4.9.2, and the codes are linked to CUDA/cuSPARSE 6.5. This platform is the same as BEETHOVEN, but is renamed in this section for clarity purposes.

Other software included ILUPACK (2.4), the Mercurium C/C++ compiler/Nanox (releases 1.99.7/0.9a for SANDY, HASWELL and XEON PHI) with support for OmpSs, and METIS (5.0.2) for the graph reorderings.

Power/energy was measured via RAPL in SANDY and HASWELL, reporting the aggregated dissipation from the packages (sockets) and the DRAM chips. For XEON PHI we leveraged

routine `mic_get_inst_power_readings` from the `libmicmgmt` library to obtain the power of the accelerator. In KEPLER, we use RAPL to measure the consumption from the server's package and DRAM, and NVML library to obtain the dissipation from the GPU.

Table B.1: Laplace matrices employed in the evaluation.

Matrix	Size (n)	#Nonzeros (n_z)	Row density (n_z/n)
A171	5,000,211	19,913,121	3.98
A252	16,003,008	63,821,520	3.98
A318	32,157,432	128,326,356	3.98

For the analysis, we employed the SPD linear system arising from the finite difference discretization of a 3D Laplace problem in Section 3.1, with three instances of different size; see Table B.1. In the experiments, all entries of the right-hand side vector b were initialized to 1, and the PCG was started with the initial guess $x_0 \equiv 0$. For the tests, the parameters that control the fill-in and convergence of the iterative process in ILUPACK were set as `droptool` = 1.0E-2, `condest` = 5, `elbow` = 10, and `restol` = 1.0E-6.

We use GFLOPS and GFLOPS/W to assess, respectively, the performance and energy consumption of the parallel codes/platforms. ILUPACK is in part a memory-bound computation. Therefore, an alternative performance metric could have been based on the attained memory transfer rate (Gbytes/s). Nevertheless, given that the data matrices are all off-chip, and ILUPACK performs a number of flops that is proportional to the volume of memory accesses, we prefer to stand with the GFLOPS metric. This measure has the advantage of being more traditional among the HPC community.

B.1.2 Optimizing energy and performance

The task-parallel version of ILUPACK based on OmpSs applies two architecture-aware optimization strategies:

- For multisocket servers, (e.g. SANDY and HASWELL,) we accommodate a NUMA-aware execution via the NANOS[2] environment variable `NX_ARGS` with the argument `--schedule=socket` combined with a careful modification of the ILUPACK code. Concretely, our code records in which socket each task was executed during the initial calculation of the preconditioner. This information is subsequently leveraged, during all iterations of the PCG solve, to enforce that tasks which operate on the same data that was generated/accessed during the preconditioner calculation are mapped to the same socket where they were originally executed.
- A critical aspect in the Intel Xeon Phi is how to bind the OmpSs threads to the hardware threads/cores in order to distribute the workload. In our executions, this mapping is controlled using the NANOS runtime environment variable `NX_ARGS`, passing the appropriate values via arguments `--binding_stride`, `--binding_start` and

[2] http://pm.bsc.es/nanox

`--smp_workers`. In our experiments we evaluate several configurations of these parameters to balance the workload distribution and achieve an optimal saturation of the hardware cores.

Saving energy in the OmpSs version

The OmpSs runtime allows the user to trade off performance for power (and, hopefully, energy) consumption by controlling the behaviour of idle OmpSs threads, setting it to a range of modes that vary between pure blocking (idle-wait) and polling (busy-wait). To execute our application in blocking mode, we set the arguments `--enable-block` and `--spins=1` in the `NX_ARGS` NANOS environment variable. The first parameter enables the blocking mode while the second one indicates the number of spins before an idle thread is blocked. For the polling mode, we simply do not include the option `--enable-block`; we set `--enable-yield`, which forces threads to yield on an idle loop and a conditional wait loop; and we set `--yields=1000000` to specify the number of yields before blocking an idle thread.

Saving energy in the data-parallel version

On heterogeneous platforms, consisting of a CPU and a GPU, our data-parallel version offloads a significant part of the computations to the graphics accelerator rendering the CPU idle for a significant fraction of the execution. In this scenario, a potential source of energy savings is to operate in the CUDA blocking synchronization mode, which allows that the operating system puts the CPU to sleep (i.e., to promote it to a deep C-state) when idle.

B.1.3 Experimental Evaluation

In order to characterize the energy-efficiency of the studied computing platforms, it is first necessary to count with parallel implementations that are correctly tuned to leverage the features of each device. There exists a considerable variety of factors that affect the efficiency of a parallel application on a target hardware. Among these, we next analyze the following configuration parameters:

- *Degree of software concurrency*, i.e., the number of threads that execute the application.
- *Operation "behaviour" of idle threads (CPU power states or C-states).* A thread without work can remain in an active state, polling for a new job to execute. Alternatively, it can be suspended (blocked) and awakened when a new job is assigned to it. The polling mode favors performance at the expense of higher power consumption in some platforms. The blocking mode, on the other hand, can produce lower power consumption, by allowing the operating system to promote the suspended core into a power-saving C-state, but may negatively impact the execution time because of the time it takes to reset the core into the active C0 state. The effect of these two modes on energy efficiency is uncertain, as energy is the product of time and power.
- *Operation frequency of active threads (CPU performance states or P-states).* Active threads can operate on a range of frequency/voltage pairs (P-states) that, for the Intel

platforms evaluated in this work, can only be set on a per socket basis (i.e., for all cores of the same socket). These modes are controlled by the operating system, though the user can provide some general guidelines via the *Linux governor modes*. In general, the P-states provide a means to trade off performance for power dissipation for active cores/sockets.

- *Binding of threads to hardware cores.* The degree of software concurrency translates into the exploitation of a certain level of hardware parallelism depending on the mapping of the software threads to the hardware (physical) cores. For the execution of numerical codes on general-purpose x86 CPUs, the standard approach maps one thread per core. For specialized hardware such as the Intel Xeon Phi (as well as the IBM Power A2), better results may be obtained by using 2 or 4 software threads per core.

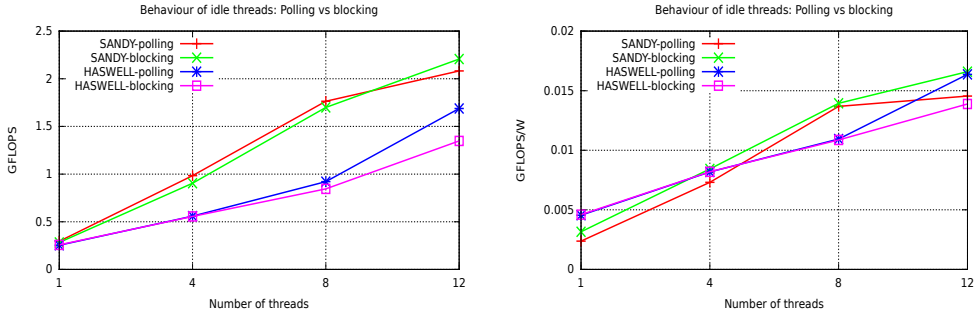


Figure B.2: GFLOPS (left) and GFLOPS/W (right) obtained with the task-parallel version of ILUPACK on SANDY and HASWELL, using the blocking and polling modes for benchmark A318.

The initial experiments in the remainder of this subsection aim to tune the previous configuration parameters for the execution of the task-parallel version of ILUPACK on SANDY, HASWELL and XEON PHI. For this purpose, we select the largest dataset that fits into the memory of each platform (A318 for both SANDY and HASWELL, and A171 on XEON PHI), and evaluate the GFLOPS and GFLOPS/W metrics as the number of threads grows. A direct comparison between the XEON PHI and the two general-purpose x86 platforms cannot be done at this point. For the task-parallel version of ILUPACK, the degree of software concurrency determines the number of tasks that should be present in the bottom level of the dependency tree (see Section B.1), and the actual number of flops that is required for the solution of each problem case.

Figure B.2 reports the performance and energy efficiency attained with the task-parallel version of ILUPACK, on SANDY and HASWELL, when OmpSs is instructed to operate in either the polling and blocking modes (see subsection B.1.2). This first experiment reveals that the impact of these modes on both metrics is minor when up to 8 threads are employed. However, for 12 threads, we can observe quite a different behaviour depending on the target platform. Concretely, for SANDY, it is more convenient to rely on the blocking mode, especially from the point of view of GFLOPS/W while, for HASWELL, the polling mode yields superior performance and energy efficiency over its blocking counterpart. According to these results, in the following experiments we select the blocking and polling modes for SANDY and HASWELL, respectively.

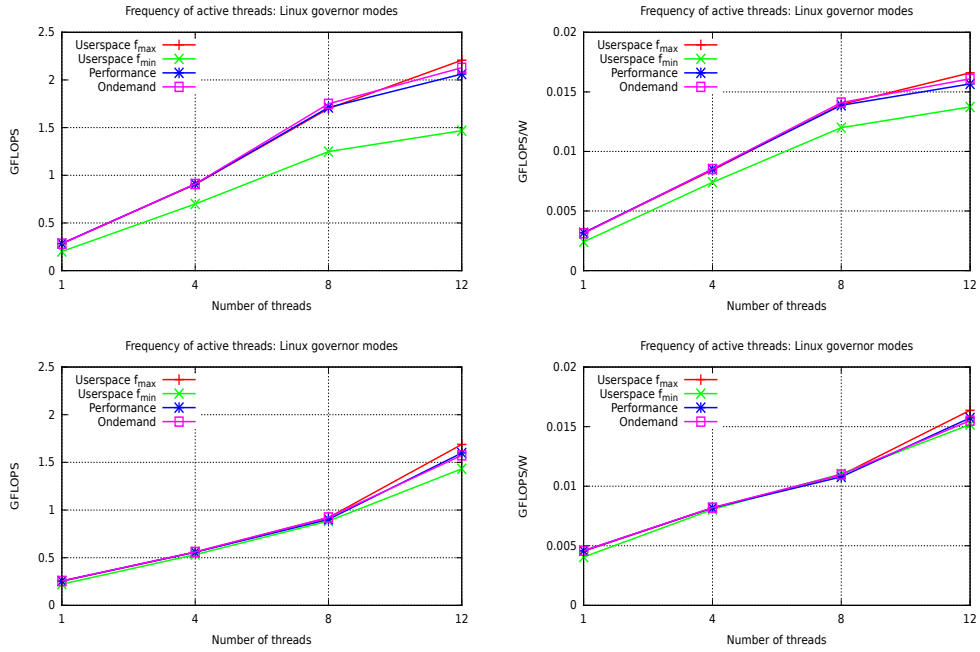


Figure B.3: GFLOPS (left) and GFLOPS/W (right) obtained with the task-parallel version of ILUPACK on SANDY (top) and HASWELL (bottom), using different Linux governors for benchmark A318.

Figure B.3 evaluates the impact of three Linux governors available in SANDY and HASWELL: `performance`, `ondemand` and `userspace`, with the latter set so that the sockets operate in either the maximum or minimum frequencies (f_{max} and f_{min}, respectively) of the corresponding platforms (f_{max}=2.0 GHz and f_{min}=1.2 GHz for SANDY; and f_{max}=1.6 GHz and f_{min}=1.2 GHz for HASWELL). The four plots in the figure reveal the small impact of this configuration parameter on the performance and energy efficiency of the task-parallel version of ILUPACK on both servers, which is only visible when 12 threads/cores are employed in the execution. Given these results, we select the `userspace` governor, with the P0 state (i.e., maximum frequency), in the remaining experiments with these two platforms.

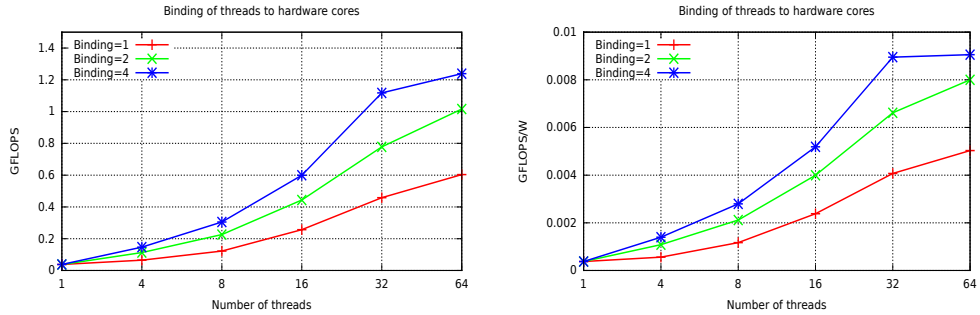


Figure B.4: GFLOPS (left) and GFLOPS/W (right) obtained with the task-parallel version of ILUPACK on XEON PHI, using different binding options for benchmark A171.

The last experiment with the configuration parameters, in Figure B.4, exposes the effect of populating each hardware core of XEON PHI with 1, 2, 4 (software) threads

(`Binding=4,2,1`, respectively). The best choice is clearly the first option which, given a fixed number of threads, maximizes the number of hardware cores employed in the experiment. This will be the configuration adopted for the following experiments with this platform.

Characterization of the platforms

Table B.2: Characterization of the four platforms obtained with the task-parallel and data-parallel versions of ILUPACK.

Platform	Matrix	Time (s)	GFLOPS	Energy (J)	GFLOPS/W
SANDY	A171	21.12	2.95	2,827.89	0.0221
	A252	101.42	2.74	13,843.17	0.0201
	A318	322.06	2.21	42,827.13	0.0166
HASWELL	A171	31.89	1.95	3,277.67	0.0193
	A252	154.04	1.80	15,933.05	0.0174
	A318	421.13	1.69	43,419.49	0.0164
XEON PHI	A171	58.69	1.24	8,032.32	0.0090
KEPLER	A171	23.09	2.49	2,909.34	0.0198
	A252	83.82	3.16	11,449.81	0.0231

Table B.2 evaluates the task-parallel version of ILUPACK running on SANDY, HASWELL or XEON PHI, compared with the data-parallel version of the solver executed on KEPLER, using four efficiency metrics: execution time, GFLOPS, (total) energy-to-solution, and GFLOPS/W. For the Intel-based platforms, we use 12 threads in both SANDY and HASWELL, and 64 for XEON PHI.

A direct comparison of the platforms, using the same problem case is difficult: First, due to the small memory of the XEON PHI, the largest problem that could be solved in this platform (A171) seems too small to exploit the large amount of hardware parallelism of this accelerator. In addition, increasing the problem dimension shows different trends depending on the platform, with a decline in the GFLOPS and GFLOPS/W rates for SANDY and HASWELL, but a raise for KEPLER. Finally, even if the problem case is the same, the solvers do not necessarily perform the same amount of operations to converge as the exact number of flops depends, e.g., of the level of task-parallelism tackled by each solver/platform (12 tasks in the bottom level of the DAG for SANDY and HASWELL, 64 for XEON PHI, and a single task for KEPLER) as well as variations due to rounding errors, which affect the convergence rate.

As an alternative, let us perform a comparison based on the largest problem case that can be tackled on each platform: A318 for SANDY and HASWELL, A171 for XEON PHI and A252 for KEPLER. Consider first the two platforms equipped with general-purpose CPUs. As the two system comprise 12 cores, in principle we could expect better performance from HASWELL because the floating-point units (FPUs) available in this recent architecture can produce up to 16 DP flops/cycle compared with the 8 DP flops/cycle of SANDY. However, the irregular data access patterns present in ILUPACK turns the exploitation of the wide vector units (SIMD) into a difficult task which, combined with the higher maximum frequency of

SANDY, explains why this platform outperforms HASWELL. Interestingly, the gap between the GFLOPS rates of these two platforms, a factor of about 2.21/1.66=1.30, is captured to high accuracy by the difference between their maximum frequencies, 2.0/1.6=1.25. This variation is not reflected in the energy and GFLOPS/W metrics, where HASWELL is only slightly behind SANDY. These particular trends make us believe that HASWELL could match SANDY's performance and outperform its energy efficiency if both platform were operated with the same maximum frequency. Let us include KEPLER and benchmark A252 in the comparison now. As the problems being solved are different, we will perform the comparison in terms of GFLOPS and GFLOPS/W, and obviate time and energy. In spite of the large number of FPUs in KEPLER, we see that the difference in favor of this data-parallel architecture is moderate, with a factor of 1.43 and 1.86 over SANDY and HASWELL, respectively, in the GFLOPS rate; and roughly 1.40 over any of the two systems in the GFLOPS/W metric. Finally, we note that XEON PHI lags behind any of the other three platforms in both GFLOPS and GFLOPS/W.

B.2 Adaptation of ILUPACK to low power devices

In this section we describe the particular strategies that were adopted in order to execute ILUPACK on low power devices as the Jetson TX1.

The development of this low power variant is based on the SPD data parallel implementation of ILUPACK, as we consider this can be the first step towards a distributed version of ILUPACK capable of executing on clusters of low power devices.

As in the previously presented parallel variants, it was necessary to transform ILUPACK data structures in order to use CUSPARSE. This transformation is done only once, during the construction of each level of the preconditioner, and occurs entirely in the CPU. In devices equipped with physical Unified Memory[3], like the Jetson TX1, no transference is needed once this transformation has been done.

Regarding the computation of the SPMV in the GPU, it is important to remark that in this case each level of the preconditioner involves a matrix-vector multiplication with F and F^T. As in the baseline SPD version described in Section 3.2, we store both F and F^T in the GPU memory, accepting some storage overhead in order to avoid using the transposed routine offered by CUSPARSE. Future implementations could consider the development of custom kernels and the use of adequate sparse storage formats that allow a more balanced performance ratio between the regular and transposed SPMV.

For the rest of the work we will consider two different implementations:

- JTX1_GPU: computes the entire preconditioner application on the GPU while the rest of the computations are carried out in the ARM processor.
- JTX1_ARM: makes all the computations in the ARM processor. This variant does not leverage any data parallelism.

[3]See: JETSON TX1 DATASHEET DS-07224-010_v1.1

Jetson TX1

The NVIDIA Jetson TX1 it is a low-power GPU enabled system that includes a 256-core Maxwell GPU and a 64-bit quad-core ARM A57 processor, which was configured in maximum performance. The platform is also equipped with 4GB of unified LPDDR4 RAM that has a theoretical bandwidth of 25.6 GB/s (see [80]).

Experimental evaluation on low-power platforms

Motivated by the rapid development of low-power computing platforms, we adapted ILUPACK in order to execute it in the NVIDIA Jetson TX1.

The code was slightly modified to be able to take advantage of the unified physical memory and cross-compiled using the compiler GCC 4.8.5 for `aarch64` with the `-O3` flag enabled, and the corresponding variant of CUDA Toolkit 8.0 for the Jetson, employing the appropriate libraries.

The primary purpose of this evaluation is to determine if this kind of devices are able to solve sparse linear systems of moderate dimensions efficiently using ILUPACK. Since the support of double-precision arithmetic of the Jetson GPU is more than limited, all the experiments reported in this section were computed using IEEE single-precision floating point numbers.

The benchmark utilized for the test is the SPD case of scalable size presented in Section 3.1.2.

Table B.3: Runtime (in seconds) of the data-parallel variant of ILUPACK in Jetson TX1. *Prec. time* corresponds to the time spent applying the preconditioner during the entire solver.

variant	Case	*iters*	*Prec. time*	*Total time*	*error*	*Prec. speedup*	*Total speedup*
JTX1_ARM		156	60.33	84,57	2.31E-07	-	-
JTX1_GPU	A126	156	44.36	70.30	2.45E-07	1.36	1.20
JTX1_ARM		206	161.90	228.26	3.07E-07	-	-
JTX1_GPU	A159	206	123.93	187.93	3.15E-07	1.31	1.21
JTX1_ARM		222	218.53	306,78	3.02E-07	-	
JTX1_GPU	A171	222	146.09	229.45	3.10E-07	1.50	1.34

Table B.3 shows the comparison between the performance of the sequential and data-parallel versions of ILUPACK executed on the Jetson module. It can be observed that the JTX1_GPU version implies lower runtimes than the JTX1_ARM variant, but these improvements are decreasing with the dimension of the addressed problem. This result is consistent with other experiments, and relates to the fact that GPUs requires large volumes of data to effectively exploit their computational power.

If we take the performance of JTX1_GPU in other kind of hardware platform into consideration, the benefits offered by the Jetson device are easy to see. To illustrate this aspect we compared our previous results for the GPU-based ILUPACK from [5], run on an NVIDIA K20 GPU, to compare the runtimes. Table B.4 summarizes these results, focusing only in the preconditioner application runtime, and contrasting it with the one obtained in the Jetson TX1 platform.

Table B.4: Runtime (in seconds) of GPU-based ILUPACK in a K20 (from [5], using double-precision arithmetic) and the JTX1_GPUvariant of ILUPACK in Jetson TX1 (in single-precision).

Case	K20			Jetson		
	iters	*Prec. time*	*Time by iter*	*iters*	*Prec. time*	*Time by iter*
A126	44	11.38	.26	156	44.36	.28
A159	52	19.75	.38	206	123.93	.60
A171	-	-	-	222	146.09	.66
A200	76	28.28	.37	-	-	-

It should be recalled that ILUPACK, as a typical iterative linear system solver, is a memory-bounded algorithm. Hence, when comparing the performance allowed by the Jetson with other GPU-based general hardware platforms, it is necessary to analyze the differences between their memory bandwidth. As an example, the NVIDIA K20 GPU offers a peak memory bandwidth of 208 GB/s ([78]), while the NVIDIA Jetson only allows to reach 25.6 GB/s, i.e. a difference above 8×.

Before analyzing the results, it is important to remark that the computations in both works are not exactly the same. Because of the reduced memory capacity of the Jetson, the preconditioners for this device where generated using a higher drop tolerance, which allows less fill-in in the triangular factors. For this reason the number of iterations performed by the solvers in the two platforms are different. However, the runtime per iteration is an acceptable estimator for the performance of each version.

The results summarized in Table B.4 show that the time per iteration for the smallest case is similar in the two platforms. Considering that the experiments in the K20 GPU were performed using double precision, it is reasonable to expect this runtimes to be reduced in half[4] if single precision is used. This means that the difference in performance is of about 2× in favour of the K20. Nevertheless, this gap is considerably smaller than the difference in the bandwidth of both devices, which is of approximately 8×. However, it can also be observed that the benefits offered by the Jetson hardware start to diminish when the dimension of the test cases grow (note that in the case A159 is near to 3× if we estimate the single precision performance of the K20 as before).

This result shows that when the dimension of the addressed test case is enough to leverage the computational power of high end GPUs this kind of lightweight devices are not competitive. On the other hand, in contexts where the problem characteristics do not allow exploiting commodity GPUs efficiently, this kind of devices (e.g. the Jetson TX1) are a really good option. Additionally, the important difference in power consumption between the two devices (the K20 has a peak power consumption of 225W[5], while the Jetson only 15W[6]) should also be taken into account.

With the obtained results, our next step is to develop a distributed variant of ILUPACK specially designed to run on a cluster of low power devices, as the NVIDIA Jetson TX1, and evaluate the energy consumption aspects. It should be noted that this kind of clusters are

[4] Assuming a memory-bound procedure.
[5] TESLA K20 GPU ACCELERATOR - Board Specifications - BD-06455-001_v05
[6] JETSON TX1 DATASHEET DS-07224-010_v1.1

not yet widespread, but some examples are the one built in the context of the Mont-Blanc project, leaded by the Barcelona Super Computing (BSC) Spain [37], and the one constructed by the ICARUS project of the Institute for Applied Mathematics, TU Dortmund, Germany [55, 54].

APPENDIX C

Description of the GPU architectures used in this work

This section includes the description of the most remarkable characteristics of the three NVIDIA GPU architectures employed in this thesis, namely the Fermi, Kepler and Maxwell architectures.

C.0.1 Fermi architecture

Fermi architecture, illustrated in Figure C.1, presents up to 512 CUDA cores organized in 16 Streaming Multiprocessors (SM) of 32 cores each. The number of CUDA cores and SMs varies according to the GPU model. The architecture supports up to 6 GB of GDDR5 RAM memory with an interface bandwidth of 384 bits.

Each CUDA core contains an Arithmetic-Logic Unit (ALU) and a Floating Point Unit (FPU). Unlike its preceding architecture (Tesla), Fermi GPUs completely support the IEEE 754-2008 floating-point standard, providing the *fused multiply-add* (FMA) operation, which performs a multiplication and an addition in the same rounding step, avoiding the loss of precision in the addition [82].

A diagram of the Fermi multiprocessor can be observed in the right part of Figure C.1.

C.0.2 Kepler architecture

The GK110 microarchitecture (code-name Kepler) was developed by NVIDIA [78] in 2012, introducing several changes to the earlier Fermi architecture, most of them with focus on improving the device energy efficiency, usually measured as the performance-per-watt ratio. A diagram of this architecture is presented in Figure C.2. The use of a unified GPU clock (abandoning the shader clock) simplified the static scheduling of instructions. This factor and the more powerfriendly nature of the cores included in Kepler, generated a notable reduction in power consumption. However, to archive the same level of performance than the previous microarchitecture, it was necessary to increase the number of cores. According

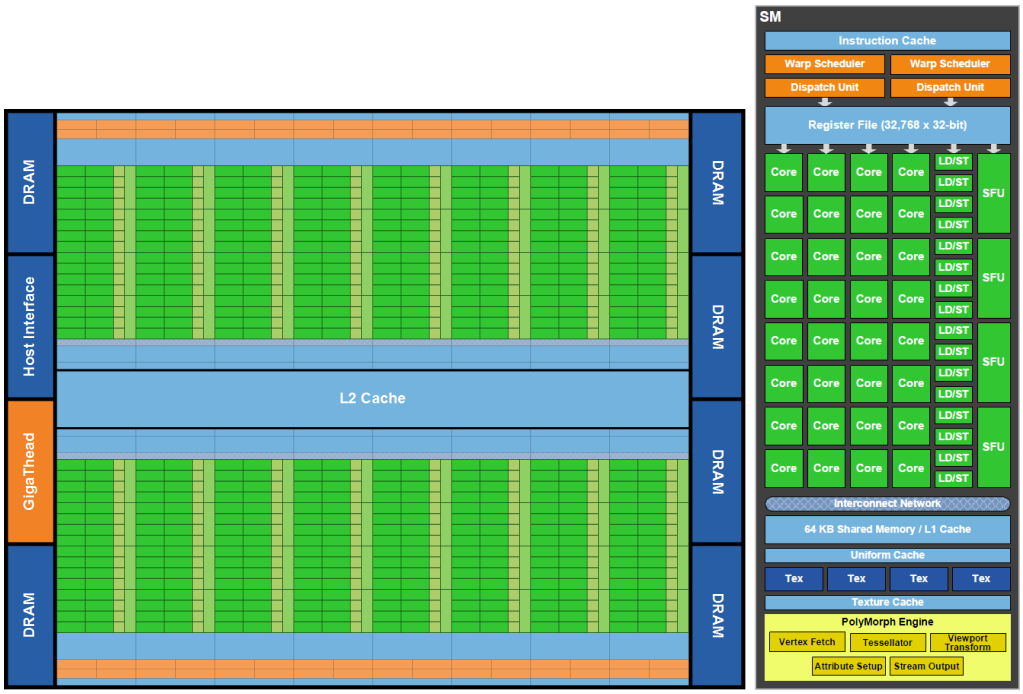


Figure C.1: Block diagram of Fermi GF100 architecture (left) and its SM (right). Extracted from *NVIDIA Fermi GF100 Architecture Whitepaper*. Available online.

to NVIDIA reports, two Kepler cores perform similar than one Fermi core but consuming less than 50% of the energy.

From a hardware architecture point of view, Kepler employs a new generation of Streaming Multiprocessor called SMX (see Figure C.3). Each SMX contains 192 single-precision CUDA cores, 64 double-precision units, 32 special function units (SFU), and 32 load/store units (LD/ST). To increase the double precision performance offered by the previous generation, the SMX of Kepler has 8× more SFUs than the SM in Fermi. As before, the SMX schedules threads in groups of 32, implicitly synchronized, parallel threads called warps. Each Kepler SMX contains 4 Warp Schedulers, each with dual Instruction Dispatch Units, allowing the concurrent execution of up to four warps. Additionally, it allows simple and double precision instructions to be paired with other instructions.

The new design of the SMX also introduced a new high performance operation, and significantly improved another one. On the one hand, the warp shuffle operations (SHFL) appeared, which allow the threads within a warp to exchange data resident in registers without using slower memory spaces like the the shared or global memory. This kind of operations has four variants (indexed anytoany, shift right/left to nth neighbour and butterfly (XOR) exchange) and are extremely useful in the implementation of many high performance kernels. On the other hand, the atomic instructions got a 10× performance gain through the inclusion of more atomic processors and a shorter processing pipeline. Kepler supports five atomic operations: `atomicMin`, `atomicMax`, `atomicAnd`, `atomicOr` and `atomicXor`.

Regarding memories, each SMX has 64 KB of on-chip memory that can be configured as 48 KB of Shared memory with 16 KB of L1 cache, 16 KB of shared memory with 48 KB of

Figure C.2: Block diagram of Kepler GK110 architecture. Extracted from *NVIDIA Kepler GK110 Architecture Whitepaper*. Available online.

L1 cache, or 32 KB of Shared memory with 32 KB of L1 cache, which was not available in Fermi. In addition to the L1 cache, Kepler introduces a 48KB cache for data that is read-only. The cache is directly accessible to the SMX for general load operations, the Read-Only Data Cache's higher tag bandwidth supports full speed unaligned memory access patterns. The Kepler features a 1536KB L2 cache memory, doubling the amount of L2 as well as the L2 bandwidth per clock available in Fermi.

Unlike in previous architectures, kernels executing on Kepler have the capacity to launch other kernels, which is called Dynamic Parallelism, and can also create the necessary streams, events and manage the dependencies required to process additional work without the need of the CPU interaction. This allows to avoid the need to return to the CPU to lunch new kernels, and the cost associated with it. The Fermi architecture supports 16-way concurrent kernel launches, but finally all arrive to the same hardware work queue. Kepler improves this with the new Hyper-Q feature, which contains 32 simultaneous work queues. Each CUDA stream is managed within its own work queue, so an operation in one stream will no longer block others streams.

C.0.3 Maxwell architecture

Maxwell is the microarchitecture developed by NVIDIA [79] as the successor of Kepler. Maxwell introduces an improved Streaming Multiprocessor (SM), also designed to increase the power efficiency ratio, delivering 2× the performance per watt of Kepler devices. The

Figure C.3: Block diagram of the SMX presented by Kepler. Extracted from *NVIDIA Kepler GK110 Architecture Whitepaper*. Available online.

Maxwell architecture is composed of an array of Graphics Processing Clusters (GPCs) with, 16 Maxwell Streaming Multiprocessors (SMM), and four memory controllers each.

The new SMM design contains four warp schedulers, and each warp scheduler is capable of dispatching two instructions per warp every clock. The Maxwell SMM is partitioned into four distinct 32-CUDA core processing blocks (128 CUDA cores total per SM), each with its own dedicated resources for scheduling and instruction buffering. Each Maxwell CUDA core is able to deliver roughly 1.4× more performance per core compared to a Kepler CUDA core.

Maxwell offers a 256-bit memory interface with 7 Gbps GDDR5 memory, and also features a unified 2048 KB L2 cache that is shared across the GPU.

C.1 Summary of computing platforms

The tables in this section describe the main features of the computational platforms used in this document. The information is split in three tables. Table C.1 enumarates the main characteristics of the CPU of each platform, Table C.2 details the most relevant properties of the accelerators, and Table C.3 enumerates the versions of the software employed.

Table C.1: CPU and main memory features of each of the utilized platforms.

	Model	Cores	Freq. (GHz)	Cache (MB)	RAM (GB)	Max. bw. (GB/s)
Bach	Intel Core i7-2600	4	3.40	8	8	21.0
Mozart	Intel Core i3-3220	2	3.30	3	16	25.6
Beethoven	Intel Core i7-4770	4	3.40	8	16	25.6
Brahms	Intel Xeon E5-2620 v2	12	2.10	15	128	51.2
Falla	Intel Xeon E5-2680 v3	12	2.50	30	64	68.0
Jetson TX1	ARM A57	4	1.73	2	4 (unified)	25.6
Ravel	Intel Core i7-6700	4	3.40	8	64	34.1
Sibelius	Intel Core i7-6700	4	3.40	8	64	34.1
sandy	Intel Xeon E5-2620	12	2.00	15	32	51.2
haswell	Intel Xeon E5-2603 v3	12	1.60	15	32	51.0

Table C.2: Main features of the accelerators in each of the utilized platforms.

	Model	Arch.	Cores	Freq. (GHz)	RAM (GB)	Max. bw. (GB/s)
Bach	Tesla C2070	Fermi	448	1.50	5	144
Mozart	Tesla K20	Kepler	2496	0.70	6	250
Beethoven	Tesla K40c	Kepler	2880	0.75	11	288
Brahms	2 × Tesla K40m	Kepler	2880	0.75	11	288
Falla	4 × 2 × Tesla K40m	Kepler	2880	0.75	11	288
Jetson TX1	Tegra X1	Maxwell	256	0.99	4 (unified)	25.6
Ravel	GTX 1060	Pascal	1152	1.50	3	192
Sibelius	GTX 1080 Ti	Pascal	3584	1.58	11	484
xeon phi	Intel Xeon Phi 5110P	Intel-MIC	60	1.05	8	320

Table C.3: Versions of the C and Fortran compilers and the CUDA Toolkit for each of the utilized platforms.

	C and Fortran Compiler	CUDA Toolkit	Other software
Bach	gcc 4.4.6	4.1	
Mozart	gcc 4.4.7	5.0	
Beethoven	gcc 4.9.2	6.5	
Brahms	Intel Parallel Studio 2016	6.5	
Falla	gcc 4.8.5	7.5	OpenMPI 10.3.1
Jetson TX1	gcc 4.8.5	8.0	
Ravel	gcc 14.0.0	8.0	
Sibelius	gcc 14.0.0	8.0	
xeon phi	icc 13.1.3	-	Mercurium C/C++ compiler/Nanox 1.99.7/0.9a
sandy	gcc 4.4.7	-	Mercurium C/C++ compiler/Nanox 1.99.7/0.9a
haswell	gcc 4.4.7	-	Mercurium C/C++ compiler/Nanox 1.99.7/0.9a

Bibliography

[1] (2013). The top500 list. Available at http://www.top500.org.

[2] (2015). HPCG - high performance Conjugate Gradients. https://software.sandia.gov/hpcg.

[3] Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. (2011). Exploiting thread-level parallelism in the iterative solution of sparse linear systems. *Parallel Computing*, **37**(3), 183–202.

[4] Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. (2012). Parallelization of multilevel ILU preconditioners on distributed-memory multiprocessors. In *Applied Parallel and Scientific Computing, LNCS*, volume 7133, pages 162–172. Springer, Berlin, Heidelberg.

[5] Aliaga, J. I., Bollhöfer, M., Dufrechou, E., Ezzatti, P., and Quintana-Ortí, E. S. (2014). Leveraging data-parallelism in ILUPACK using graphics processors. In *IEEE 13th International Symposium on Parallel and Distributed Computing, ISPDC 2014, Marseille, France, June 24-27, 2014*, pages 119–126.

[6] Aliaga, J. I., Bollhöfer, M., Dufrechou, E., Ezzatti, P., and Quintana-Ortí, E. S. (2016a). A data-parallel ILUPACK for sparse general and symmetric indefinite linear systems. In *Euro-Par 2016: Parallel Processing Workshops - Euro-Par 2016 International Workshops, Grenoble, France, August 24-26, 2016, Revised Selected Papers*, pages 121–133.

[7] Aliaga, J. I., Dufrechou, E., Ezzatti, P., and Quintana-Ortí, E. S. (2016b). Design of a task-parallel version of ILUPACK for graphics processors. In *High Performance Computing - Third Latin American Conference, CARLA 2016, Mexico City, Mexico, August 29 - September 2, 2016, Revised Selected Papers*, pages 91–103.

[8] Aliaga, J. I., Badia, R. M., Barreda, M., Bollhöfer, M., Dufrechou, E., Ezzatti, P., and Quintana-Ortí, E. S. (2016c). Exploiting task and data parallelism in ilupack's preconditioned CG solver on NUMA architectures and many-core accelerators. *Parallel Computing*, **54**, 97–107.

[9] Aliaga, J. I., Dufrechou, E., Ezzatti, P., and Quintana-Ortí, E. S. (2017a). Evaluating the NVIDIA tegra processor as a low-power alternative for sparse GPU computations. In *High Performance Computing - 4th Latin American Conference, CARLA 2017, Buenos Aires, Argentina, and Colonia del Sacramento, Uruguay, September 20-22, 2017, Revised Selected Papers*, pages 111–122.

[10] Aliaga, J. I., Dufrechou, E., Ezzatti, P., and Quintana-Ortí, E. S. (2017b). Overcoming memory-capacity constraints in the use of ILUPACK on graphics processors. In *29th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2017, Campinas, Brazil, October 17-20, 2017*, pages 41–48.

[11] Aliaga, J. I., Bollhöfer, M., Dufrechou, E., Ezzatti, P., and Quintana-Ortí, E. S. (2018). Extending ILUPACK with a task-parallel version of BiCG for dual-GPU servers. In *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM@PPoPP 2018, February 25, 2018, Vienna, Austria*, pages 71–78.

[12] Alvarado, F. and Schreiber, R. (1991). Fast parallel solution of sparse triangular systems. In *13th IMACS World Congress on Computation and Applied Mathematics, Dublin*.

[13] Alvarado, F., Yu, D., and Betancourt, R. (1989). Ordering schemes for partitioned sparse inverses. In *SIAM Symposium on Sparse Matrices, Salishan Lodge, Gleneden Beach, Oregon*.

[14] Alvarado, F. L. and Schreiber, R. (1993). Optimal parallel solution of sparse triangular systems. *SIAM Journal on Scientific Computing*, **14**(2), 446–460.

[15] Ament, M., Knittel, G., Weiskopf, D., and Straßer, W. (2010). A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-gpu platform. In *PDP '10: Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Networkbased Processing*, pages 583–592.

[16] Amestoy, P. R., Davis, T. A., and Duff, I. S. (1996). An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.*, **17**(4), 886–905.

[17] Anderson, E. and Saad, Y. (1989a). Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing*, **1**(01), 73–95.

[18] Anderson, E. and Saad, Y. (1989b). Solving sparse triangularclinear systems on parallel computers. *International Journal of High Speed Computing*, **01**(01), 73–95.

[19] Anzt, H., Chow, E., and Dongarra, J. (2015). *Iterative Sparse Triangular Solves for Preconditioning*, pages 650–661. Springer Berlin Heidelberg, Berlin, Heidelberg.

[20] Arnoldi, W. E. (1951). The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of applied mathematics*, **9**(1), 17–29.

[21] Bell, N. and Garland, M. (2009). 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.

[22] Bell, N. and Garland, M. (2012). Cusp: Generic parallel algorithms for sparse matrix and graph computations. Version 0.3.0.

[23] Benzi, M. and Tuma, M. (1998). A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM Journal on Scientific Computing*, **19**(3), 968–994.

[24] Bientinesi, P., Gunnels, J. A., Myers, M. E., Quintana-Ortí, E. S., and Geijn, R. A. v. d. (2005). The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Softw.*, **31**(1), 1–26.

[25] Blanchard, P. and Brüning, E. (2015). *Mathematical Methods in Physics: Distributions, Hilbert Space Operators, Variational Methods, and Applications in Quantum Physics*, volume 69. Birkhäuser.

[26] Bollhoefer, M. and Saad, Y. (2002). On the relations between ILUs and factored approximate inverses. *SIAM J. Matrix Anal. Appl.*, **24**(1), 219–237.

[27] Bollhöfer, M. (2001). A robust ILU with pivoting based on monitoring the growth of the inverse factors. *Linear Algebra and its Applications*, **338**(1), 201 – 218.

[28] Bollhöfer, M. (2003). A robust and efficient ILU that incorporates the growth of the inverse triangular factors. *SIAM Journal on Scientific Computing*, **25**(1), 86–103.

[29] Bollhöfer, M. and Saad, Y. (2002a). A factored approximate inverse preconditioner with pivoting. *SIAM Journal on Matrix Analysis and Applications*, **23**(3), 692–705.

[30] Bollhöfer, M. and Saad, Y. (2002b). On the relations between ILUs and factored approximate inverses. *SIAM Journal on Matrix Analysis and Applications*, **24**(1), 219–237.

[31] Bollhöfer, M. and Saad, Y. (2006). Multilevel preconditioners constructed from inverse-based ILUs. *SIAM Journal on Scientific Computing*, **27**(5), 1627–1650.

[32] Bolz, J., Farmer, I., Grinspun, E., and Schröoder, P. (2003). Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. *ACM Trans. Graph.*, **22**(3), 917–924.

[33] Buatois, L., Caumon, G., and Levy, B. (2007). Concurrent number cruncher: An efficient sparse linear solver on the GPU. In *High Performance Computation Conference (HPCC), Springer Lecture Notes in Computer Sciences*.

[34] Chiang, A. C. (1984). *Fundamental methods of mathematical economics*. Aukland (New Zealand) McGraw-Hill.

[35] Chow, E. and Patel, A. (2015). Fine-grained parallel incomplete lu factorization. *SIAM Journal on Scientific Computing*, **37**(2), C169–C193.

[36] Cline, A. K., Moler, C. B., Stewart, G. W., and Wilkinson, J. H. (1979). An estimate for the condition number of a matrix. *SIAM Journal on Numerical Analysis*, **16**(2), 368–375.

[37] Contributors, A. (2016). start — mont-blanc prototype. [Online; accessed 10-July-2017].

[38] Cuthill, E. (1972). Several strategies for reducing the bandwidth of matrices. In D. J. Rose and R. A. Willoughby, editors, *Sparse Matrices and their Applications: Proceedings of a Symposium on Sparse Matrices and Their Applications, held September 9–10, 1971, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, the National Science Foundation, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department.*, pages 157–166, Boston, MA. Springer US.

[39] Davis, T. (2006). *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics.

[40] Davis, T. A. and Hu, Y. (2011). The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, **38**(1), 1:1–1:25.

[41] Dongarra, J. and Heroux, M. A. (2013). Toward a new metric for ranking high performance computing systems. Sandia Report SAND2013-4744, Sandia National Lab.

[42] Dongarra, J., Duff, I., Sorensen, D., and van der Vorst, H. (1998). *Numerical Linear Algebra for High-Performance Computers*. Society for Industrial and Applied Mathematics.

[43] Dongarra, J. J., Luszczek, P., and Petitet, A. (2003). The LINPACK benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, **15**(9), 803–820.

[44] Duff, I. S. and Koster, J. (2001). On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, **22**(4), 973–996.

[45] Dufrechou, E. and Ezzatti, P. (2018). Solving sparse triangular linear systems in modern gpus: A synchronization-free algorithm. In *26th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2018, Cambridge, United Kingdom, March 21-23, 2018*, pages 196–203.

[46] (Eds.), G. M. (1999). *Computer Solution of Large Linear Systems*. Studies in Mathematics and Its Applications 28. Elsevier.

[47] Erguiz, D., Dufrechou, E., and Ezzatti, P. (2017). Assessing sparse triangular linear system solvers on gpus. In *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pages 37–42.

[48] Fletcher, R. (1976). Conjugate gradient methods for indefinite systems. In *Numerical analysis*, pages 73–89. Springer.

[49] Freund, R. and Zha, H. (1994). Simplifications of the nonsymmetric lanczos process and a new algorithm for hermitian indefinite linear systems. *AT&T Numerical Analysis Manuscript, Bell Laboratories, Murray Hill, NJ*.

[50] Freund, R. W. and Szeto, T. L. D. (1992). *A quasi-minimal residual squared algorithm for non-Hermitian linear systems*. Department of Mathematics, University of California, Los Angeles.

[51] George, A. (1973). Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, **10**(2), 345–363.

[52] George, A., Heath, M. T., Liu, J., and Ng, E. (1986). Solution of sparse positive definite systems on a shared-memory multiprocessor. *International Journal of Parallel Programming*, **15**(4), 309–325.

[53] George, A., Liu, J., and Ng, E. (1994). Computer solution of sparse linear systems. *Academic, Orlando*.

[54] Geveler, M. and Turek, S. (2017). How applied sciences can accelerate the energy revolution–a pleading for energy awareness in scientific computing. In *Newsletter of the European Community on Computational Methods in Apllied Sciences*. -. accepted.

[55] Geveler, M., Ribbrock, D., Donner, D., Ruelmann, H., Höppke, C., Schneider, D., Tomaschewski, D., and Turek, S. (2016). The icarus white paper: A scalable, energy–efficient, solar–powered HPC center based on low power GPUs. In F. Desprez, P. Dutot, C. Kaklamanis, L. Marchal, K. Molitorisz, L. Ricci, V. Scarano, M. Vega-Rodriguez, A. Varbanescu, S. Hunold, S. Scott, S. Lankes, and J. Weidendorfer, editors, *Workshop on Unconventional HPC*, volume Euro-Par 2016: Parallel Processing Workshops: Euro-Par 2016 International Workshops, Grenoble, France, August 24-26, 2016, Revised Selected Papers of *LNCS, Euro-Par'16*, pages 737–749. Springer. isbn 978-3-319-58943-5; http://dx.doi.org/10.1007/978-3-319-58943-5_59.

[56] Golub, G. H. and Loan, C. F. V. (1996). *Matrix Computations*. The Johns Hopkins Univ. Press, Baltimore, 3rd edition.

[57] Golub, G. H. and Loan, C. F. V. (2012). *Matrix Computations*. The Johns Hopkins Univ. Press, Baltimore, 4th edition.

[58] Goodnight, N., Woolley, C., Lewin, G., Luebke, D., and Humphreys, G. (2003). A multigrid solver for boundary value problems using programmable graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 102–111, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

[59] Greenbaum, A. (1987). *Iterative methods for solving linear systems*. Frontiers in applied mathematics 17. Society for Industrial and Applied Mathematics, 1 edition.

[60] Gupta, R., van Gijzen, M. B., and Vuik, C. (2013). 3d bubbly flow simulation on the gpu - iterative solution of a linear system using sub-domain and level-set deflation. In *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, volume 0, pages 359–366, Los Alamitos, CA, USA. IEEE Computer Society.

[61] He, K., Tan, S. X., Zhao, H., Liu, X.-X., Wang, H., and Shi, G. (2016). Parallel GMRES solver for fast analysis of large linear dynamic systems on GPU platforms. *Integration, the VLSI Journal*, **52**, 10 – 22.

[62] Hennessy, J. L. and Patterson, D. A. (2011). *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition.

[63] Hestenes, M. R. and Stiefel, E. (1952). Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, **49**(1).

[64] Hénon, P. and Saad, Y. (2006). A parallel multistage ilu factorization based on a hierarchical graph decomposition. *SIAM Journal on Scientific Computing*, **28**(6), 2266–2293.

[65] Jones, M. and Patrick, M. (1993). Bunch–kaufman factorization for real symmetric indefinite banded matrices. *SIAM Journal on Matrix Analysis and Applications*, **14**(2), 553–559.

[66] Kahn, A. B. (1962). Topological sorting of large networks. *Commun. ACM*, **5**(11), 558–562.

[67] Kirk, D. and Hwu, W. (2012). *Programming Massively Parallel Processors, Second Edition: A Hands-on Approach*. Morgan Kaufmann.

[68] Krüger, J., Schiwietz, T., Kipfer, P., and Westermann, R. (2004). Numerical simulations on PC graphics hardware. In *ParSim 2004 (Special Session of EuroPVM/MPI 2004)*, Budapest, Hungary.

[69] Lanczos, C. (1952). Solution of systems of linear equations by minimized iterations. *J. Res. Nat. Bur. Standards*, **49**(1), 33–53.

[70] Li, R. and Saad, Y. (2013). Gpu-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing*, **63**(2), 443–466.

[71] Liu, W., Li, A., Hogg, J., Duff, I. S., and Vinter, B. (2016). A synchronization-free algorithm for parallel sparse triangular solves. In *European Conference on Parallel Processing*, pages 617–630. Springer.

[72] Liu, W., Li, A., Hogg, J. D., Duff, I. S., and Vinter, B. (2017). Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides. *Concurrency and Computation: Practice and Experience*, **29**(21).

[73] Mayer, J. (2006). Alternative weighted dropping strategies for ilutp. *SIAM Journal on Scientific Computing*, **27**(4), 1424–1437.

[74] Naumov, M. (2011a). Incomplete-LU and Cholesky preconditioned. Iterative methods using CUSPARSE and CUBLAS. Nvidia white paper, NVIDIA corporation.

[75] Naumov, M. (2011b). Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. Nvidia white paper, NVIDIA corporation.

[76] Naumov, M. (2011c). Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011*, **1**.

[77] NVIDIA (1999). Nvidia launches the world's first graphics processing unit: Geforce 256. [Online; accessed 10-June-2017].

[78] NVIDIA (2013). TESLA K20 GPU Accelerator. https://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v05.pdf,[Online; accessed 10-July-2017].

[79] NVIDIA (2014). Nvidia geforce gtx 980 featuring maxwell, the most advanced gpu ever made. [Online; accessed 10-June-2017].

[80] NVIDIA (2015). NVIDIA Tegra X1 NVIDIAs New Mobile Superchip. http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf, [Online; accessed 10-July-2017].

[81] Nvidia, C. (2008). Cublas library. *NVIDIA Corporation, Santa Clara, California*, **15**(27), 31.

[82] Nvidia, C. (2017). Cuda C programming guide v8.0. *Nvidia Corporation*.

[83] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008). Gpu computing. *Proceedings of the IEEE*, **96**(5), 879–899.

[84] Pothen, A. and Alvarado, F. L. (1992). A fast reordering algorithm for parallel sparse triangular solution. *SIAM journal on scientific and statistical computing*, **13**(2), 645–653.

[85] Rothberg, E. and Gupta, A. (1992). Parallel ICCG on a hierarchical memory multiprocessor - addressing the triangular solve bottleneck. *Parallel Computing*, **18**(7), 719 – 741.

[86] Rumpf, M. and Strzodka, R. (2001). Using graphics cards for quantized fem computations. In *in IASTED Visualization, Imaging and Image Processing Conference*, pages 193–202.

[87] Saad, Y. (1994). ILUT: a dual threshold incomplete LU factorization. *Numerical Linear Algebra with Applications*, **1**(4), 387–402.

[88] Saad, Y. (2003a). *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA, USA, 2nd edition.

[89] Saad, Y. (2003b). *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, 2 edition.

[90] Saad, Y. (2005). Multilevel ILU with reorderings for diagonal dominance. *SIAM Journal on Scientific Computing*, **27**(3), 1032–1057.

[91] Saad, Y. and Schultz, M. H. (1986). Parallel implementations of preconditioned conjugate gradient methods. *Mathematical and Computational Methods in Seismic Exploration and Reservoir Modeling*, pages 108–127.

[92] Saltz, J. H. (1990). Aggregation methods for solving sparse triangular systems on multiprocessors. *SIAM J. Sci. Stat. Comput.*, **11**(1), 123–144.

[93] Saltz, J. H., Alyc, S., Syacercnizatxoi, B. C., Aeronauticsand, N., and Saltz, J. E. (1987). Automated problem scheduling and reduction of synchronization delay effects. Technical report.

[94] Schaller, R. R. (1997). Moore's law: past, present and future. *IEEE Spectrum*, **34**(6), 52–59.

[95] Schenk, O., Röllin, S., and Gupta, A. (2004). The effects of unsymmetric matrix permutations and scalings in semiconductor device and circuit simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **23**, 400–411.

[96] Schenk, O., Wächter, A., and Weiser, M. (2009). Inertia-revealing preconditioning for large-scale nonconvex constrained optimization. *SIAM J. Sci. Comp.*, **31**(2), 939–960.

[97] Sonneveld, P. (1989). CGS, A fast Lanczos-type solver for nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, **10**(1), 36–52.

[98] Stewart, G. W. (1998). *Matrix Algorithms*. Society for Industrial and Applied Mathematics, 1 edition.

[99] Sudan, H., Klie, H., Li, R., and Saad, Y. (2010). High performance manycore solvers for reservoir simulation. In *12th European conference on the mathematics of oil recovery*.

[100] Team, C. D. (2018). *Cusparse Library v9.0*. NVIDIA Corporation.

[101] Thomas, L. H. (1949). Elliptic Problems in Linear Differential Equations over a Network. Technical report, Columbia University.

[102] van der Vorst, H. A. (1982). A vectorizable variant of some iccg methods. *SIAM Journal on Scientific and Statistical Computing*, **3**(3), 350–356.

[103] Van der Vorst, H. A. (1990). The convergence behaviour of preconditioned CG and CG-S in the presence of rounding errors. In O. Axelsson and L. Y. Kolotilina, editors, *Preconditioned Conjugate Gradient Methods*, pages 126–136, Berlin, Heidelberg. Springer Berlin Heidelberg.

[104] van der Vorst, H. A. (1992). Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, **13**(2), 631–644.

[105] Van Duin, A. C. (1999). Scalable parallel preconditioning with the sparse approximate inverse of triangular matrices. *SIAM journal on matrix analysis and applications*, **20**(4), 987–1006.

[106] Wing, O. and Huang, J. W. (1980). A computation model of parallel solution of linear equations. *IEEE Trans. Computers*, **29**(7), 632–638.