UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departament de Sistemes Informàtics i Computació

Dense and sparse parallel linear algebra algorithms on graphics processing units

Author:
Alejandro Lamas Daviña

Director:
José E. Román Moltó

October 2018

To one, two, and three.

Acknowledgments

I would like to express my gratitude to my director José Román, for his permanent support during all these years of work. His wise advice and selfless guidance have been decisive for the culmination of this thesis. His door has always been open for me, and he has solved all my doubts with unlimited patience. For all that and for more, I thank him.

I would like to extend my gratitude to my colleagues of the SLEPc project. Here I thank again to José Román for his unique humor sense. To Carmen, for showing me the way and for all her good advices. To Enrique, who helped me to get rolling. And to the former members Andrés and Eloy, to whom I had the opportunity to meet and who enliven the group meals. I will keep good memories from these years.

I do not want to forget to mention to Xavier Cartoixà, Jeff Steward and Altuğ Aksoy, great researchers with whom I have had the opportunity to collaborate.

The afternoon snacks would not have been the same without the excellent discussions and comments of Fernando, David and of course Salva, who, without noticing it, also helped to improve this dissertation.

Last, I would like to thank to José Luis, IT staff of the department, for his high valuable work behind the scenes and his promptly response to any incidence.

To all of you who have supported me, thank you.

Abstract

One line of development followed in the field of supercomputing is the use of specific purpose processors to speed up certain types of computations. In this thesis we study the use of graphics processing units as computer accelerators and apply it to the field of linear algebra. In particular, we work with the SLEPc library to solve large-scale eigenvalue problems, and to apply matrix functions in scientific applications. SLEPc is a parallel library based on the MPI standard and is developed with the premise of being scalable, i.e. to allow solving larger problems by increasing the processing units.

We address the linear eigenvalue problem, $Ax = \lambda x$ in its standard form, using iterative techniques, in particular with Krylov's methods, with which we calculate a small portion of the eigenvalue spectrum. This type of algorithms is based on generating a subspace of reduced size (m) in which to project the large dimension problem (n), being $m \ll n$. Once the problem has been projected, it is solved by direct methods, which provide us with approximations of the eigenvalues of the initial problem we wanted to solve. The operations used in the expansion of the subspace vary depending on whether the desired eigenvalues are from the exterior or from the interior of the spectrum. In the case of searching for exterior eigenvalues, the expansion is done by matrix-vector multiplications. We do this on the GPU, either by using libraries or by creating functions that take advantage of the structure of the matrix. In the case of eigenvalues from the interior of the spectrum, the expansion requires solving linear systems of equations. In this thesis we implemented several algorithms to solve linear systems of equations for the specific case of matrices with a block-tridiagonal structure, that are run on GPU.

In the computation of matrix functions we have to distinguish between the direct application of a matrix function, $f(A)$, and the action of a matrix function on a vector, $f(A)b$. The first case involves a dense computation that limits the size of the problem. The second allows us to work with large sparse matrices, and to solve it we also make use of Krylov's methods. The expansion of subspace is done by matrix-vector multiplication, and we use GPUs in the same way as when solving eigenvalues. In this case the projected problem starts being of size m, but it is increased by m on each restart of the method. The solution of the projected problem is done by directly applying a matrix function. We have implemented several algorithms to compute the square root and the exponential matrix functions, in which the use of GPUs allows us to speed up the computation.

Resum

Una línia de desenvolupament seguida en el camp de la supercomputació és l'ús de processadors de propòsit específic per a accelerar determinats tipus de càlcul. En aquesta tesi estudiem l'ús de targetes gràfiques com a acceleradors de la computació i ho apliquem a l'àmbit de l'àlgebra lineal. En particular treballem amb la biblioteca SLEPc per a resoldre problemes de càlcul d'autovalors en matrius de gran dimensió, i per a aplicar funcions de matrius en els càlculs d'aplicacions científiques. SLEPc és una biblioteca paral·lela que es basa en l'estàndard MPI i està desenvolupada amb la premissa de ser escalable, açò és, de permetre resoldre problemes més grans en augmentar les unitats de processament.

El problema lineal d'autovalors, $Ax = \lambda x$ en la seua forma estàndard, ho abordem amb l'ús de tècniques iteratives, en concret amb mètodes de Krylov, amb els quals calculem una xicoteta porció de l'espectre d'autovalors. Aquest tipus d'algorismes es basa a generar un subespai de grandària reduïda (m) en el qual projectar el problema de gran dimensió (n), sent $m \ll n$. Una vegada s'ha projectat el problema, es resol aquest mitjançant mètodes directes, que ens proporcionen aproximacions als autovalors del problema inicial que volíem resoldre. Les operacions que s'utilitzen en l'expansió del subespai varien en funció de si els autovalors desitjats estan en l'exterior o a l'interior de l'espectre. En cas de cercar autovalors en l'exterior de l'espectre, l'expansió es fa mitjançant multiplicacions matriu-vector. Aquesta operació la realitzem en la GPU, bé mitjançant l'ús de biblioteques o mitjançant la creació de funcions que aprofiten l'estructura de la matriu. En cas d'autovalors a l'interior de l'espectre, l'expansió requereix resoldre sistemes d'equacions lineals. En aquesta tesi implementem diversos algorismes per a la resolució de sistemes d'equacions lineals per al cas específic de matrius amb estructura tridiagonal a blocs, que s'executen en GPU.

En el càlcul de les funcions de matrius hem de diferenciar entre l'aplicació directa d'una funció sobre una matriu, $f(A)$, i l'aplicació de l'acció d'una funció de matriu sobre un vector, $f(A)b$. El primer cas implica un càlcul dens que limita la grandària del problema. El segon permet treballar amb matrius disperses grans, i per a resoldre-ho també fem ús de mètodes de Krylov. L'expansió del subespai es fa mitjançant multiplicacions matriu-vector, i fem ús de GPUs de la mateixa forma que en resoldre autovalors. En aquest cas el problema projectat comença sent de grandària m, però s'incrementa en m en cada reinici del mètode. La resolució del problema projectat es fa aplicant una funció de matriu de forma directa. Nosaltres

hem implementat diversos algorismes per a calcular les funcions de matrius arrel quadrada i exponencial, en les quals l'ús de GPUs permet accelerar el càlcul.

Resumen

Una línea de desarrollo seguida en el campo de la supercomputación es el uso de procesadores de propósito específico para acelerar determinados tipos de cálculo. En esta tesis estudiamos el uso de tarjetas gráficas como aceleradores de la computación y lo aplicamos al ámbito del álgebra lineal. En particular trabajamos con la biblioteca SLEPc para resolver problemas de cálculo de autovalores en matrices de gran dimensión, y para aplicar funciones de matrices en los cálculos de aplicaciones científicas. SLEPc es una biblioteca paralela que se basa en el estándar MPI y está desarrollada con la premisa de ser escalable, esto es, de permitir resolver problemas más grandes al aumentar las unidades de procesado.

El problema lineal de autovalores, $Ax = \lambda x$ en su forma estándar, lo abordamos con el uso de técnicas iterativas, en concreto con métodos de Krylov, con los que calculamos una pequeña porción del espectro de autovalores. Este tipo de algoritmos se basa en generar un subespacio de tamaño reducido (m) en el que proyectar el problema de gran dimensión (n), siendo $m \ll n$. Una vez se ha proyectado el problema, se resuelve este mediante métodos directos, que nos proporcionan aproximaciones a los autovalores del problema inicial que queríamos resolver. Las operaciones que se utilizan en la expansión del subespacio varían en función de si los autovalores deseados están en el exterior o en el interior del espectro. En caso de buscar autovalores en el exterior del espectro, la expansión se hace mediante multiplicaciones matriz-vector. Esta operación la realizamos en la GPU, bien mediante el uso de bibliotecas o mediante la creación de funciones que aprovechan la estructura de la matriz. En caso de autovalores en el interior del espectro, la expansión requiere resolver sistemas de ecuaciones lineales. En esta tesis implementamos varios algoritmos para la resolución de sistemas de ecuaciones lineales para el caso específico de matrices con estructura tridiagonal a bloques, que se ejecutan en GPU.

En el cálculo de las funciones de matrices hemos de diferenciar entre la aplicación directa de una función sobre una matriz, $f(A)$, y la aplicación de la acción de una función de matriz sobre un vector, $f(A)b$. El primer caso implica un cálculo denso que limita el tamaño del problema. El segundo permite trabajar con matrices dispersas grandes, y para resolverlo también hacemos uso de métodos de Krylov. La expansión del subespacio se hace mediante multiplicaciones matriz-vector, y hacemos uso de GPUs de la misma forma que al resolver autovalores. En este caso el problema proyectado comienza siendo de tamaño m, pero se incrementa en m en cada reinicio del método. La resolución del problema proyectado se hace aplicando una función

de matriz de forma directa. Nosotros hemos implementado varios algoritmos para calcular las funciones de matrices raíz cuadrada y exponencial, en las que el uso de GPUs permite acelerar el cálculo.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

```
All work and no play makes
Jack a dull boy
All work and no play...
```

Scientific computing is an area of growing importance, not only because of its role in the progress of science, but also for helping companies in the design of new products or in the development of new processes to be competitive in a global economy. This thesis dissertation is undertaken within the framework of scientific computing, particularly in the context of linear systems of equations, eigenvalue problems and related fields. Typically this type of applications require an efficient and quick resolution of numerical problems of very large and increasing sizes, which means a challenge both in terms of computing and numerical analysis. Two main elements are required to solve these type of problems: a supercomputer, and specialized software that efficiently implements the required numerical algorithms.

1.1 Background and motivation

The architecture of supercomputers have experienced a huge transformation in the last years with the inclusion of new hardware accelerators. Many of the most powerful supercomputers in the world are equipped with some type of hardware accelerators. They can be GPUs, FPGAs, or other types of accelerators such as the Intel Xeon Phi. A close example for us is Minotauro, a supercomputer belonging to the Barcelona Supercomputer Center (BSC) that is furnished with GPUs. These technologies are gradually spreading to other more modest computing facilities like clusters of small research groups. Even a single server can be assembled with many accelerator cards and used as a small scale computing platform. But the high performance offered by these computing elements cannot be exploited to its limits without the availability of efficient software.

Today it is mandatory that scientific computing software makes use of specialized libraries to cope with the growing complexity of the applications. The use of software libraries allows the programmers to reduce the development cycle, and to benefit from high quality implementations. This last feature is very important when talking about numerical software, as numerical algorithms involve a high degree of complexity that makes them very difficult to implement in a robust and efficient way. Modern numerical libraries are designed to provide enough flexibility to allow its use in many contexts, but other properties like being robust, efficient, interoperable with other software, or portable to different computing platforms are also desired. A key characteristic of scientific software is its computational efficiency, as it is usually run on expensive supercomputers, and in that scenario it is imperative to fully exploit the hardware. The field of numerical software specialized to run on accelerators has evolved very much in the last years [73], but it still remains open to new theoretical research and developments of practical implementations.

Many scientific applications use the spectral analysis to study the behaviour of the underlying physical phenomena like structural vibration analysis, stability analysis, signal processing, etc. This type of analysis is associated with eigenvalue problems, whose matrices are typically large and sparse. It is usually enough solving a small part of the spectrum, either exterior eigenvalues or interior ones. Other emerging field of knowledge that can be applied to many scientific applications is the use of matrix functions. They can be used to solve problems like differential equations or applied in control theory. Directly applying $f(A)$ is usually avoided in practice when working with moderately large matrices due to the high computational and storage cost involved. When working with sparse matrices it is preferred to perform the computation of the action of the matrix function on a vector, $f(A)b$.

The computational requirements of the scientific problems grow constantly together with the evolution of the supercomputers. As a consequence of this, numerical problems arise affecting to the convergence of the methods. This makes it necessary to use more sophisticated strategies [12]. Innovative research proposals try to improve the convergence of the methods and reduce the cost per iteration. They also present new iterative methods or design alternatives that take advantage of the capabilities of the new hardware. Implementing parallel software in a robust and efficient way to solve large-scale eigenvalue problems in supercomputers becomes a challenging task. All these factors make desirable to provide the users with a software library for solving numerical problems, and with enough flexibility to fit in a wide range of applications. This is the objective of SLEPc.

SLEPc [60], is a software library for solving large-scale eigenvalue problems and related problems in parallel. It can be used to solve the standard and the generalized linear eigenvalue problem, and nonlinear problems as well, with symmetric and nonsymmetric matrices, in real or complex arithmetic. Other related problems like the singular value decomposition, or more recently the computation of functions of matrices are also available in it. SLEPc has been developed using modern software engineering techniques to ensure the efficiency and scalability of the software. Other characteristics are its flexibility to easily adapt to new types of problems, its robustness and portability. It is built on top of PETSc, a parallel library for the solution

of scientific applications modeled by partial differential equations.

Nowadays SLEPc has become one of the reference libraries in its area of knowledge in the scientific community, and its maturity ensures its suitability for many scientific applications. The use of the library by the scientific community has progressed from being manually installed by the interested researchers to be integrated in the supercomputers software portfolio and in general purpose operating systems like Debian, allowing a wider usage. Nevertheless, due to the constant improvements in the software, there are many researchers that still manually deploy in their platforms the development branch of the library to have access to new solvers and features added to SLEPc.

Many applications have obtained speedups of two or even three orders of magnitude when moving the computation to GPUs, with respect to previous versions that only used CPU. However, this remarkable results are not so frequent and only achievable when the computations involve dense matrix-matrix and similar operations. Algorithms involving computations with sparse matrices are expected to obtain less impressive results with speedups between 10-20x [7]. However, the use of GPUs is still very appealing in the sparse field, not only for the reduction in computation time, but also for the smaller power consumption offered by the accelerators and their better performance-cost share. Moreover, we can expect a faster evolution in the development of accelerators than in the development of general purpose CPUs.

At the beginning of this thesis, GPU support in SLEPc was more or less limited, and covered only a small percentage of the functionality. Some preliminary results in the context of specific applications had already been obtained [111], that allowed us to be optimistic about future developments. The use of GPU accelerators in SLEPc follows the approach plotted by its companion library PETSc. The design of GPU support in PETSc started as described in [96], based on performing some operations on the GPU and minimizing the data transfers between GPU and CPU.

We began the thesis work being aware of the fact that SLEPc must adopt the emerging technologies to keep being a reference in the field of high-performance eigenvalue computations. The possible lines of action are related with the three main operations in the sparse solvers used by SLEPc:

Expansion of the subspace. The most simple case entails the product of a sparse matrix times a vector, that can be done in a relatively efficient way on GPU [16]. In other cases, this step implies solving a sparse linear system of equations.

Orthogonalization of the basis vectors. This is the part of the computation that can represent a high percentage of the total computing time, and is more suitable for its execution on GPU, as it involves working with data stored contiguously, taking advantage of the spatial locality.

Solution of the projected problem. This step implies dense computations. They usually are of small size and can be performed fast on CPU, but sometimes they reach sizes of several thousands, thing that requires their parallelization, being an ideal case for using GPUs.

1.2 Objectives

In accordance with the discussed context, in this thesis we tackle several paths to try to improve the performance of SLEPc computations by means of graphics processing units. The main goal of the dissertation is then the acceleration of SLEPc solvers via parallel implementations of linear algebra algorithms on GPUs.

One of the targets of the thesis, concerning the expansion of the subspace is to make progress in the efficient solution of linear systems of equations on GPU, with both direct methods [67,78,145] and preconditioned iterative methods [89,112,113]. The main obstacle to reach good efficiency when using direct methods on GPU is the triangular solve [67]. This issue also appears in the case of incomplete factorization preconditioners [113].

In order to achieve good performance in the orthogonalization, it is essential to redefine the methods to work in a block oriented fashion, that is, to operate with collections of vectors instead of treating them individually. It is not a straightforward modification and requires a redesign of a significant part of SLEPc.

Last, solving the projected problem on GPU allows for interesting contributions, as solving dense eigenvalue problems on GPU is still an emergent topic [52, 56, 138, 144] and there is plenty of room to propose innovative solutions.

On any algorithm or development proposed it is important to consider that it is necessary to minimize the data transfers between GPU and CPU, as it is a well-known bottleneck. It is also important to provide multi-GPU support, that is, being able to use all the available GPUs on a system, wherever they are, in multiple nodes or in a single node [144].

1.3 Structure of the document

The order in which the work is presented along the chapters of this thesis follows the natural timeline of its creation. Throughout these years, both the supercomputers used in the development and testing performed, and the software versions used have experienced major changes. Some of the supercomputers used have increased their computing capacity, and others have been decommissioned to be replaced for newer ones. In the same way, several characteristics of the software used have been superseded by better ones in the latest versions. Due to this, it would not be fair to establish a comparison with results from different experiments.

In Chapter 2 a small historical review of the evolution of the parallel computing is made. In it we talk about the progress in the manufacturing process and in the hardware architectures. We show the main parallel architectures and their programming models, and present GPUs as hardware accelerators.

In Chapter 3 we introduce the eigenvalue problem with which we will work and we make a short overview of the different methods for solving it. We present SLEPc and PETSc libraries, showing design and implementation details that are important for the use of GPUs. We also comment about the most robust algorithms implemented in SLEPc for solving eigenproblems, and how using GPUs on them can reduce the

execution time. In this chapter we show the first results of the use of GPUs as accelerators with a scientific application.

In Chapter 4 we focus on solving eigenvalues of matrices with a block-tridiagonal structure. We optimize the computation of eigenvalues in the exterior of the spectrum with the development of a kernel that exploits the storage structure used. For computing interior eigenvalues we develop several implementations of algorithms that solve linear systems of equations with that structure, and show the gain provided by the GPUs.

In Chapter 5 we work with methods for computing functions of matrices that, as we have already said, is an emerging field with a great potential for improvement. We study and implement dense methods for computing matrix functions, and sparse methods where the action of a matrix function on a vector is computed. We also provide a comparison of the executions on CPU and on GPU.

Finally, in Chapter 6 we summarize the main contributions of this thesis, report about the publications generated and the transfer activities performed. Lastly, we show some possible research lines to follow.

Chapter 2

High-performance computing

Festina lente

Scientific research and engineering provide challenging problems from the point of view of computing. Climate, economic or molecular modeling, weather forecasting or physical simulations, such as vehicle aerodynamics or development of earthquake-resistant structures imply doing complex computations with very large datasets, and in some cases, including limited response times. Such demands make those problems not addressable with general-purpose computers, and are the root of the development of special computers able to cope with them.

Supercomputers are those special highly performing machines, frequently designed with new architectures and developed with the latest technology available, that is usually developed ad-hoc to improve their capabilities. High-performance computing (HPC) involves not just a big and fast computer, but all the necessary elements that make it able to solve very demanding problems in an efficient way. It brings together several elements such as: computer architecture, electronics, algorithms and programming. The software used on supercomputers plays a key role in HPC in order to effectively exploit the computing power.

In this chapter we present the concept of high-performance computing, and show the evolution that it has gone through until reaching its current status. Several computer architectures coming from different classifications and the two main parallel programming paradigms are described. We place emphasis on existing accelerators, providing a detailed view of GPUs, the ones used in this work. Finally, we introduce and explain some performance factors and indicators like floating-point operations per second (FLOPs/s), time, speedup, efficiency and scalability, that are used on the following chapters.

2.1 Hardware evolution

On the early days of computing, scientific computing was the driving force on the development of new computing systems, and reciprocally, it benefited from those new systems. Both of them have changed dramatically during their few years of life. The constant improvement of the computers has allowed them to increase their performance in several orders of magnitude, moving from solving simplified scientific problems to be able to solve more realistic ones. The roadmap to improve the computing performance has taken multiple parallel lines. We can name the improvements in the manufacturing process, in the architecture of the computing systems, in the development of software and specialized libraries, or the origin of parallel computing as some of them.

The advances on the manufacturing process, such as the transition from vacuum tubes to silicon junction transistors allowed the development of complementary metal-oxide-semiconductors (CMOS) based integrated circuits (microchips), that lead to large-scale and very-large-scale integrated microprocessors. The miniaturization of the processors made possible to reduce the voltage used and reach working frequencies in the range of gigahertz. The evolution of the technology has well-known physical limits, though. The ultimate limit of the downsizing of silicon transistors is about 0.3 nm, corresponding with the distance of atoms in silicon crystals. Current microprocessors have transistors manufactured with a process in the range between 14 and 10 nm[1], and the expected evolution is to reach soon the downscaling limit around 5 nm [72]. The propagation speed of electromagnetic waves is also a well-known limit for signal transmissions.

Parallel computing refers to the ability of the hardware to carry out several instructions simultaneously. The simultaneity can be the execution of several independent processes or several operations within a process, that can come from the multiplexing of the process into several execution threads or just the parallelization of different instructions. Different levels of parallelism, such as parallelism at bit, instruction, thread, task or data level were generated with the improvements in the architectures. These levels try to increase the rate at which the elements on each level (bits, instructions, etc.) are processed.

At bit level, the increase of the word size allowed the processors to operate with more information per instruction, and increased the amount of addressable memory. Although larger word sizes have been used, standard modern processors use word sizes of 32 or 64 bits.

The execution (or functional) unit is a component of the processors that performs the operations instructed by the program. On a simple processor architecture design, such as a scalar processor, one instruction is executed on each clock cycle. The development of pipelined and superscalar processors allowed to increase this ratio. An instruction pipeline consists in the partition of the instructions into a series of sequential steps that are performed by multiple execution parts coming from the segmentation of the execution unit, in a similar way as the steps carried out on a

[1] Intel launched its first 14 nm processor in 2014, and IBM in 2017, NVIDIA launched its first 12 nm processor in 2017 and AMD in 2018, and Samsung launched its first 10 nm processor in 2017.

production line. Each one of these execution parts is responsible of a particular task, like fetching, decoding or executing the instruction. Once fully fed, the pipeline gives a throughput of as many instructions per clock cycle as partitions originated from the segmentation of the execution unit. Superscalar processors enable to complete several instructions on a clock cycle by increasing the number of available execution units. Nowadays, most of the computers combine these two techniques of instruction-level parallelism on pipelined superscalar processors.

The programming of a single-processor computer is serial, but different processes can be computed concurrently by time sharing the CPU. More performance on such a machine can be obtained by increasing the processor's working frequency. But the dynamic power consumed by a switching circuit

$$P = CV^2 f, \tag{2.1}$$

increases linearly with the frequency, where C is capacitance (approximately proportional to the chip area), V is voltage, and f is frequency. Also, as a consequence of reaching the nanometre scaling, the leaked current of the transistors become an important issue, reaching to signify 50 % of the total power consumption [100]. Dissipating all that heat from the chip, to keep it under a safe operating temperature range, is a major constraint in the development of integrated circuits. Once reached the gigahertz range, the leaked currents and the heat, made the CPU manufacturers turn their developments into a different approach.

The increase of the processors working frequency soon revealed an important constraint of the computers performance. While the processing capabilities were growing, the time to fetch data from memory did not improve equally, and became much greater than the time to process the same data. This issue, known as the memory wall, limits the performance, as the processor remains idle while waiting for the data. The problem can be addressed with the use of intermediate memories between the CPU and the main memory, and faster than this one. Cache memories store data of recent operations and instructions to accelerate subsequent accesses to them. Obviously, the data has to be reused to benefit from the faster access. Normally, several levels of cache memories with different speeds are used. The level 1 is the closest one to the processor, and is the fastest and smallest one in storage capacity. Each superior levels grow in size and become slower as they move away from the CPU.

Another old technique to reduce the memory wall is the use of hardware multithreading. Superscalar processors can implement simultaneous multithreading (SMT), in which a pipeline stage fetches instructions from several processes (or threads) in the same cycle, and alternates their processing. This thread-level parallelism hides the memory latency and improves the processor utilization by reducing the idle waits. The execution of a specific instruction can load into the cache some data also needed from other concurrent threads, speeding up their execution, but at the same time may produce or increase cache trashing.

Task-level parallelism can also be achieved by adding redundant computing elements, in a similar way as it is done in superscalar processors. The addition of several

processors to the same machine was implemented on early computers making possible computing different processes in parallel. But a larger step was the assembly of several computational cores into a single processor, entering in the multi core era [133].

In 2001, IBM anticipated to the frequency ceiling faced three years later by launching the first (non-embedded) general-purpose multicore processor, the POWER4. The cores of such a multicore processor are full independent processing devices, allowing to process several instructions in parallel on the available cores.

The path to higher performance by means of the explicit parallelism of multicore processors, implies that on the same integration scale (and same voltage used), the frequency and power consumption per core must be reduced, to maintain the processor under the heat dissipation limits. This frequency reduction implies manufacturing multicore processors with less performance per core than previous single core processors. By reducing their frequencies, multicore processors also alleviate a bit the memory wall problem, reducing the latency gap. On the other side, the memory bandwidth requirements grow with the cores competing for the shared resources, in a situation that aggravates with the use of hardware multithreading to boost core utilization.

The coexistence of several cores in the same die implies the use of inter-core communication mechanisms, that even with a small number of cores becomes a serious issue, due to the huge impact on the performance that it can signify. The simple solution of using a common bus facilitates the cache coherency, but increases the latency, reduces the bandwidth, and limits the scalability. The need for an efficient inter-core communication mechanism made the design paradigm of network on chip (NoC) [103] emerge. It is a communication solution based on modular packet-switched mechanisms, and implies the use of network topologies, switching techniques, and routing algorithms as in standard computer networks, but on a chip level. Mesh topology is often used on NoC because it provides a simple routing and low network overhead, making it very scalable.

The multiple cores of a processor do not necessarily share the same design and/or functionality. Heterogeneous architectures like Cell Broadband Engine [54] or big.-LITTLE appeared, that include different types of cores in the same processor. The former integrates a single general-purpose computational core with several coprocessors to accelerate multimedia computation, the latter combines cores with reduced performance and energy requirements with others having the opposite characteristics. Other possible features of this processors include the use of different working frequencies for groups of cores, or the implementation of different instruction sets.

Most of the processors manufactured nowadays belong to the multicore family, and they are used in pairs, or more, to form the computational core of a single HPC machine.

A different level of parallelism is the data-level parallelism, that consists in distributing the data among several processors, so they can run in parallel the same operations over the different subsets of the data in a synchronous way. This technique of parallelism is widely exploited by MPI and CUDA programming as we will see later.

Supercomputers incorporate and combine most of these parallelization techniques and exploit them to the limit. Current supercomputers [131] are formed by thousands of nodes with several processors and accelerator devices with multiple computational cores. The interconnection network of the supercomputers is as important as the individual computing power of the nodes. The type of network used implies a specific bandwidth and latency that together with the topology and the routing employed define the scalability limits of the supercomputer.

Current trends in computers development try to replace silicon transistors with transistors based on different materials, or enhance the production of computers based on quantum computing.

2.2 Parallel architectures

As we have seen, the increase of the raw computing performance comes form the parallelization techniques used. Those arisen parallel computers can be classified within different criteria such as the number and type of the processors used, the mechanisms used by the processors to communicate with each other, or the existence of a central control mechanism. Here we present different parallel architectures based on the last two classifications.

2.2.1 Central control mechanism

One of the most recognized and used computer architectures classifications is Flynn's taxonomy [44], that establishes four categories of computers based on the existence of a central control mechanism that orchestrates the (possibly multiple) instructions and data streams. Parallel computers are characterized by the classes with multiple data streams, and current computers combine both classes.

Single instruction stream on multiple data stream (SIMD)

On this type of computers, a single control unit instructs multiple execution units to perform the same instruction in parallel over different data streams, providing data-level parallelism. It encompasses vector processors as the ones used in Cray machines, and the vectorial instruction sets of current processors such as SSE (Streaming SIMD Extensions), 3DNow! or Altivec.

Multiple instruction stream on multiple data stream (MIMD)

This class of computers have one control unit per execution unit, and it allows them to run different instructions on different data streams, in parallel. The pairs instructions-data streams are totally independent between them, but can be coordinated. Multicore computers, that provide task-level parallelism, are an example of this class.

2.2.2 Communication mechanism

According to the communication mechanism used between the processors, the computers can be classified in shared-memory and message-passing architectures.

Shared memory

This architecture is characterized by having a common memory address space that is accessed by all the processors through an interconnection network. Although it is the most common implementation, the shared-memory concept does not imply sharing a common physical memory, it can be a logically shared-memory composed by multiple physical memories distributed among the processors. On this architecture, different processors communicate with each other by writing and reading information on the shared memory. This shared access makes it necessary to use control mechanisms to avoid race conditions, in which several processors operate on the same memory address at the same time, and at least one of them write to it. Also, due to the use of local cache memories by the processors to improve the performance, this architecture usually implements mechanisms to ensure the coherence between them. As several processes can have the same variable stored in their cache memory, if a processor modifies the value of the variable on its local cache, the change must be propagated to the other cache memories, or invalidated on them. Although all the processors have access to all the memory, the access time for a specific address is not necessarily the same for all the processors, as it can depend on the physical distribution of the processors and the memory banks. Shared memory computers can be classified based on the existence (or absence) of homogeneity on the access time.

Uniform memory access (UMA) On this class of computers, the latency time and bandwidth in the access to any address of the memory by any of the processors is the same.

Non uniform memory access (NUMA) On this other class, the time to access to different addresses by one processor varies depending on physical factors. In the same way, different processors can obtain different latency time and bandwidth when accessing the same memory address.

Message passing

On this architecture, each processor has exclusive access to a local address space, and the communication with other processors is done by explicitly sending messages through an interconnection network. The messages can be used to transmit data and to synchronize different processors. In the same way as in shared-memory, here the non-shared-memory concept is also logical, although usually implies a physical distribution of the memory. The communication between the processors is a key factor on this architecture, and its management is as relevant as the control mechanism

necessary in shared-memory architectures. Given its distributed nature, message-passing computers can be very scalable, depending on the type of interconnection network used, and allows implementations with very large number of processors. Computers of this class usually have also a local shared-memory architecture.

2.3 Programming models and parallel software

The hardware improvements are fully bound to parallel programming and an equal software evolution to maximize the performance capabilities offered. The final goal of parallel programming is the execution of the software in less time than a serial version. The reduction in time is achieved by fully employing all the computational resources available. If a serial program is executed on a parallel computer, the transparent parallelism at instruction-level or thread-level are the maximum levels of parallelism that it can benefit from, as higher levels like task-level or data-level parallelism imply modifications in the program. An efficient parallelization of a program is a difficult task and usually requires a good knowledge of the architecture where it is going to be executed. Parallel processes replicate the executed program and are separately run on different computing elements.

2.3.1 Threads and OpenMP

Almost all operating systems allow programming with a shared-memory model by providing interprocess communication (IPC) mechanisms. They enable that different processes communicate with each other through the memory. Other mechanism consists in creating multiple execution threads within a single process that can run concurrently, or in parallel if the hardware allows it. There exist multiple application programming interfaces (API) that allow the users to employ this multithreading model.

The portable operating system interface (POSIX) standard [70] defines the specification of a threading API, pthreads (POSIX threads), that has been implemented in many operating systems, Unix and not Unix. Pthreads defines a low-level set of C language types and functions that allow the user to manage a parallel execution. The programmer has to decompose the work to be done and must explicitly manage the creation of the threads, and the access to the memory, by means of condition variables, mutexes, synchronization barriers and locks. This API is broadly used by operating systems, but is less frequent in generic applications as it requires a considerable programming effort.

Another specification for multithreading programming is OpenMP [28], which defines a set of compiler directives, an API, and environment variables to manage a high-level parallelization. OpenMP includes a runtime environment that provides an automatic management of the threads, releasing the programmer from that task. It is a multi-platform solution available for C, C++ and Fortran languages and its main use is by compiler directives (`#pragma`), that allow the programmer to incrementally specify parallel regions within the code. This enables the programmer to focus

on parallelizing first the most time-consuming parts of the algorithm. Another interesting feature is that it can use accelerators like digital signal processors (DSP), or graphics processing units (GPU), by defining regions of code in which the data and/or the computation is moved to be processed on them. All these features have made OpenMP the shared-memory de facto standard.

2.3.2 Message Passing Interface

With the use of multi-processor machines, multiple message-passing systems proliferated among the supercomputer manufacturers and the academia. Those first implementations were incompatible between them, and even hardware specific. The existence of such a great variety of systems made the development of parallel software very costly. The developers had to use the solutions offered by the vendors, and the migration to a different machine involved a complete rework of the software. This situation made it necessary to develop a unified standard system for parallel computing on distributed memory systems.

Message Passing Interface (MPI) [98] is that standard library interface specification focused on the message-passing parallel programming model. It born as the result of a standardization effort that selected the most prominent characteristics of the different message-passing systems existing at the moment, as none of them was the ideal solution, but all had their own strengths.

The specification includes C and Fortran bindings, and its portability, efficiency and flexibility are some of its strong points. It allows MIMD programs, although it is common to use it with a single program multiple data (SPMD) approach. MPI provides derived datatypes that allow forming new datatypes from previously defined ones; process groups, that establish an order among processes by using identifiers; communication contexts, that map a group of processes with a communication namespace; point-to-point communications, in which only one sender and one receiver take part; collective operations, in which all the processes of a communication context participate; process creation and management operations, one sided communications and parallel file input/output, among others. MPI is the de facto standard for distributed memory environments, for its portability and scalability.

Datatypes

MPI defines several basic datatypes with correspondence of C and Fortran datatypes, like `MPI_CHAR` (`char`), `MPI_INT` (`signed int`) and `MPI_FLOAT` (`float`) for C, and `MPI_CHARACTER`, (`CHARACTER(1)`) `MPI_INTEGER` (`INTEGER`) and `MPI_REAL` (`REAL`) for Fortran. Other basic datatypes like `MPI_BYTE` and `MPI_PACKED`, that have no correspondence with the language used, are also included. `MPI_BYTE` is included as an uninterpreted type that has the same meaning on all machines (eight bits), as opposed to a character, that may have different representations on different machines.

MPI expects the data transmitted on a message to be formed by consecutive elements of a single type. This characteristic would result in a limitation when having to transmit different types of data, or non-consecutive elements of the same

type. The use of the type `MPI_PACKED`, to explicitly pack multiple sequences of consecutive elements of different types and lengths into a single buffer, to be sent in a single transmission, was a first attempt of avoiding the issue. This approach was followed to maintain compatibility with previous libraries.

A drawback of the `MPI_PACKED` strategy, is that it requires additional memory-to-memory copies on the sender (to the packed buffer) and on the receiver side (from the packed buffer). Derived datatypes are other datatypes defined in MPI, that allow the programmer to avoid the explicit packing and unpacking of `MPI_PACKED`, and elude the memory copies.

Derived datatypes allow the programmer to transmit, in a single transfer, objects composed by variables of different types and sizes, and not necessarily stored contiguously in memory. They are formed from basic datatypes, and once defined, the new types can also be recursively used to form new derived types.

Groups and communicators

All the processes in MPI are uniquely identified by their rank, an integer index that starts from zero and increases without jumps. This rank is unique within a group, that represents an ordered set of processes. There can be as many groups as desired in a single program execution, as a group can be composed of any number of processes, and can include the same set used in other group. Groups are mapped to communicator contexts and identify the processes involved in their communications.

Communicator contexts are the tool used by MPI that allow to partition the communication realm into subdomains. As with groups, there is no limit to its number. Within a specific context, the processes can communicate with each other without collision with communications of other contexts, even if the sets of processes overlap. After the program initialization, a process can establish communication with two predefined communicators, `MPI_COMM_WORLD`, that includes all the existing processes, and `MPI_COMM_SELF`, including only itself.

Each message sent must explicitly name the context to where it belongs, and set the source and destination processes by using their ranks within the corresponding group. The specification includes operations to consult the rank of a process in a particular context, and the size of the group associated with that context.

Point to point communications

The simplest form of communication involves only two processes, one of them acting as a sender and the other as receiver of the message. For the communication to work, both processes must belong to the same communicator, and they must explicitly indicate it on the call. The programmer has to consult and use the ranks of the processes in the communicator to establish the direction of the message. Although it is not necessary[2], it is common to use a condition clause to make the process with the desired rank to act as source of the message, invoke the appropriate send call. In the same way, the process with the rank expected to be the destination,

[2]There exist single calls that fusion send and receive operations like `MPI_Sendrecv`.

has to invoke the receive call. The calls have also to indicate a tag that allows to differentiate between multiple messages with the same origin and destination pair on the same communicator. The coordination in the calls is essential, if one of the processes misses the call, the message will not be transmitted.

Those four elements, (source, destination, tag and communicator), are the necessary data that allows the system to transmit the message. But besides them, the calls must also agree on the amount of data to be transmitted. The processes have to indicate the datatype, by using any valid MPI datatype, including derived types, and the amount of elements of that datatype to be transmitted. As far as the amount of data specified on the destination process is not greater than on the origin, they can use different counts. If the destination buffer is bigger than the data transmitted, the remaining memory positions remain unmodified, but if it is smaller, an overflow occurs. By using derived datatypes, it is possible to store the data with a different shape (or object abstraction) in the destination process than in the origin, as the underneath basic datatypes must match on both calls, but the layout in memory can vary.

Point to point communications can be done through blocking or non-blocking calls. Blocking sends are those that do not return until the buffer used in the call is not necessary any more, and can be modified. Blocking receives do not return until the message has been fully stored in the destination buffer. These blocking calls limit the performance of the computation, as it is stopped while the communication is being done. On the contrary, non-blocking calls allow to overlap the communication and/or the computation, with the cost of needing to verify the completion of the non-blocking send and receive before using the buffers.

Collective operations

Collective operations involve and must be invoked from all the processes in a communicator. Most of the available collective operations are transmission communications, but there also exist barriers to synchronize the processes, and reduction operations. Any program can be made out of point-to-point communications exclusively, but there are cases in which using them makes the code excessively complex or tedious to program. Collective operations provide a higher level of abstraction, simplify the code, and allow the libraries to provide several implementations of the operations, optimized for different network topologies.

Different data flows are possible when using collective operations. If there is a single origin or destination process of the data within the operation, it is called the root process. It is the case of *One to All* operations, like `MPI_Bcast` or `MPI_Scatter`, which transmit some information from a single root process to the rest of the processes in the group, and *All to One* operations, like `MPI_Gather` or `MPI_Reduce`, which transmit information from all the processes in the group to the root process. *All to All* operations, like `MPI_Allgather` or `MPI_Allreduce`, transmit information from all the processes in the group to all the processes.

As in the case of point-to-point communications, there exists a blocking and a non-blocking version of the collective operations.

2.3.3 Software

The development of applications experienced a evolution of magnitude similar to the hardware transformation. Several milestones of the software development were the adoption of open standards, the use of software libraries, the object oriented programming, the parallel computing, and the free and collaborative software.

Standards, as a formal document that describes and establishes requirements and procedures to unify criteria, not as the dominant position on the field, have been a huge step in the evolution of computing systems. Standards allow the interoperability of the systems, by setting formats and protocols to be followed for exchanging information. The descriptive term, open, of open standards, is significant, as the specification has to be, at least, public to encourage its adoption, and must not limit its implementations or use, now or in the future. Another desired connotation of open is to also accept to be modified, to fix or extend its specification, with a consensus between the affected parties whoever they be. In computing, from the point of view of the software evolution, the desired scenario is to use free software on the local machine, and open standards to transmit information and communicate with other software on remote computers.

A library is a compendium of software, written in a specific programming language, that specializes in solving a problem or family of problems, and facilitates an interface to higher level programs to use its routines.

Years ago, if a development team worked on several applications with common functionalities, they could share that common portion of the code between the programs, just by copying the appropriate lines. If changes were needed to be made on that part of the code, they had to be replicated on the multiple applications using it, on every single change, to maintain the coherence. The most they could do to alleviate the issue is to isolate the common code in a file, that could be shared by the applications. If a different team wanted to solve the same problem, and unless the original team had published the code of its implementation, it had to face the complex, time-consuming, and error prone task of implementing that functionality by writing a similar code from scratch. Different implementations usually have different interfaces for the same functionality. That situation involved a considerably amount of redundant work in the software development area. Additionally, the programming errors in the algorithm and in the implementation, depended completely on the expertise of each development team. If the functionality needed was not in the field of knowledge of the developers, the quality of the final product could be affected.

The employment of libraries provides the opposite situation. The code is developed once and reused in countless projects. Libraries offer an abstraction layer to the applications, that simplifies the code when a specific functionality that they provide is needed. A multi-line ad-hoc implementation can be replaced with a single call to the library, improving the readability and maintainability of the application. The development of a library is usually done by experts in the field that can optimize the algorithms, and any application software using it automatically benefits from its improvements in performance or accuracy. Even novel applications can provide

good functionality by relying on the well-known behaviour of mature libraries.

A single standard can have multiple implementations in the form of *competing libraries* that share the same interface. The programmers benefit from that circumstance, by being able to migrate between different implementations without the need of modifying the application code.

The common procedure is to develop the software in a modular fashion, by using libraries specialized in a target field. A key element in scientific computation is the numerical software, that relies widely on libraries providing computational kernels used to solve higher level operations.

As in this thesis we focus on solving linear algebra problems, we turn our attention to software libraries that implement related algorithms. According to the density of the data (vectors or matrices) involved, linear algebra problems can be classified in dense and sparse. For dense problems, we must highlight two libraries that have settled the foundation for the subsequent linear algebra libraries developed, as they are not just plain libraries, but also act as specifications, and are the de facto standard of a common interface to perform linear algebra operations. These libraries are BLAS and LAPACK.

Basic linear algebra subprograms (BLAS) [87] are a set of routines developed originally in Fortran. Their development started by identifying low-level recurrent mathematical operations, called kernels, that appear on the most common algorithms of numerical software, and consume most of the execution time. Those operations were provided with an interface and used as basic building blocks to perform more complex operations.

For the same routine, BLAS defines the interface for real and complex arithmetic, both in single and double precision. Routines are organized in three levels, that gradually have extended the library, associated by the type of data they work with.

Level 1: composed of commonly used vector operations on stridden arrays. Involve $\mathcal{O}(n)$ floating-point operations and $\mathcal{O}(n)$ data elements accessed, being n the length of the vectors. The performance obtained with operations of this level is limited by the poor ratio of floating-point operations to data movement. A typical operation of this level is the addition of two vectors, $y \leftarrow \alpha x + y$.

Level 2: composed of matrix-vector operations. Involve $\mathcal{O}\left(n^2\right)$ floating-point operations and $\mathcal{O}\left(n^2\right)$ data elements accessed, being n the dimension of the matrix. This set of operations try to exploit the memory hierarchy of the computers by reusing data stored in cache memory, and this way avoiding duplicated data movements from the main memory. But in the same manner as the level 1 operations, the performance of this level is memory bound, as the floating-point operations to data movement ratio is the same. A typical operation of this level is the matrix-vector product, $y \leftarrow \alpha Ax + \beta y$.

Level 3: composed of block oriented matrix-matrix operations. This level encourages block oriented algorithms to fully reuse the data of the blocks stored in cache memory. This is the level with better performance, as the routines involve $\mathcal{O}\left(n^3\right)$ floating-point operations and $\mathcal{O}\left(n^2\right)$ data elements accessed,

being n the dimension of the matrices. This makes the performance to be defined by the CPU instead of by the memory. A typical operation of this level is the matrix-matrix product, $C \leftarrow \alpha AB + \beta C$.

Linear algebra package (LAPACK) [6] is an efficient and portable library for solving the most common linear algebra problems encountered in numeric computations. It originated from two preexisting and widely used linear algebra libraries, LINPACK [37], oriented to the resolution of linear systems of equations and least square problems, and EISPACK [123], focused on solving eigenvalue problems. The algorithms used on these two libraries were not developed taking into account that the cost of accessing the memory can limit the performance of the software, so they are not data-reuse oriented. LAPACK is a collaborative project that employs more modern design methodologies, simplified the algorithms to be more modular, and obtains a better performance with code structured to reuse data in faster memories, and reduce the movement of data from/to main memory. It includes and expands the functionality of LINPACK and EISPACK by solving systems of linear equations, least square solutions to linear systems of equations, eigenproblems and singular value problems. It also performs the related matrix factorizations like LU, Cholesky, QR, singular value decompositions, and Schur decompositions.

LAPACK routines are divided in three classes:

Computational routines: perform specific computational tasks like matrix factorizations.

Driver routines: based on the computational routines, solve a full problem like a linear system, eigenvalue problem, etc.

Auxiliary routines: perform auxiliary tasks for the computational routines. Operations similar to BLAS but not present on it, or extensions of BLAS routines.

LAPACK routines are built using BLAS, so they benefit from its portability, and the performance of LAPACK is linked to the implementation of BLAS employed. Level 3 BLAS is used when possible. The usage of LAPACK is similar to the usage of BLAS, and in the same way as BLAS does, LAPACK also defines interfaces for real and complex arithmetic, both in single and double precision.

The reference implementations of BLAS and LAPACK developed in Fortran are serial. Other numerical libraries developed to run in parallel are ScaLAPACK (Scalable Linear Algebra PACKage) [19] that provides high-performance linear algebra routines for parallel distributed memory architectures; PLASMA (Parallel Linear Algebra for Scalable Multicore Architectures) [23] focused on multicore architectures; MKL (Math Kernel Library) [71] a proprietary library optimized for Intel processors; or OpenBLAS [142] an optimized free software library that implements BLAS with a performance comparable to the Intel MKL.

BLAS considers only dense computations on which all the elements of the corresponding operators are involved. In such case, both vectors and matrices are stored in memory with a dense format. However, as many times scientific problems entail working with very large and sparse matrices, using a dense storage format incurs in

a excessive storage and computational overhead. The structure of the matrix becomes relevant for sparse matrices, as the performance obtained is directly related with the performance of the matrix-vector product, that in turn is determined by the matrix structure and storage scheme used. Structured matrices like diagonal, band, triangular or even block-tridiagonal allow the use of specific representations that save both memory and computational effort. Commonly, the particular type of problem to be solved indirectly yields the storage scheme to be used.

Eventually, the use of sparse storage formats concerned with unstructured sparse matrices was addressed by the BLAS Technical Forum. It published an interface to operate with dense-sparse operands[3]. Sparse storage schemes represent only the nonzero[4] elements of the matrix and store them contiguously in memory, so the memory required is significantly reduced with respect to the dimension of the matrix, the access to the elements is fast, and zero elements are not involved in the computation. Each storage scheme intends to optimize the most frequent operations like the matrix-vector product, the factorization of the matrix or the extraction of the diagonal elements. We name here some of the sparse matrix storage schemes.

Compressed Sparse Row format (CSR). It is the most commonly used storage scheme. It stores only the nonzero values without caring about the sparse pattern of the matrix. It uses three arrays, one to store the values in row-major order, a second array stores the column indexes, and a third one stores pointers to the begin of the rows. Row indexes are not stored explicitly.

Compressed Sparse Column format (CSC). It is also a very common scheme, analogous to CSR, but storing the values in column-major order.

Coordinate format (COO). It stores each element of the matrix using its Cartesian coordinates. It provides a very efficient sequential access, and an inefficient access to a specific element (search).

Block Sparse Row format (BSR). It is a variant of the CSR format for matrices with square dense blocks in a regular pattern. Instead of storing elements like CSR, it stores the dense blocks.

Modified Compressed Sparse Row format (MSR). It is a variant of the CSR format that stores the elements of the diagonal, that can be referenced with a single coordinate, in a separate array.

Diagonal format (DIA). It is a format specific for storing banded matrices. It stores the elements of each diagonal, in a bidimensional array of size $d \times n$, where d is the number of diagonals containing at least one nonzero element, and n is the number of rows of the matrix. Its arrangement leaves unused memory on all but one of the diagonals.

[3]Sparse-sparse operations are not considered in the sparse interface.
[4]Some zero elements may be stored, depending on the storage scheme used.

Ellpack-Itpack generalized diagonal format (ELL). It stores the matrix in two bidimensional arrays, one to store the values and other to store the column indexes. The size of these arrays is $m \times k$, where m is the number of rows of the matrix and k is the maximum number of nonzero elements aggregated in a row. This number k determines the storage overhead. This format is most efficient with a homogeneous number of nonzero elements per row.

For parallel software on distributed memory architectures, there are multiple communication libraries implementing MPI, two of them are OpenMPI [49] and MPICH. Both of them are open-source libraries, high-performance computing oriented, widely used on supercomputers. OpenMPI is an initiative that blends resources from several previous MPI implementations like FT-MPI, LA-MPI, LAM/-MPI and PACX-MPI aiming to reduce the fragmentation of the implementations and at the same time, provide a high-quality library by selecting the best of each contributor. The MPICH implementation was originally developed to provide feedback to the MPI Forum during the MPI standardization. Today it is the reference implementation for the latest MPI standards and it is common that manufacturers fork the project to develop their own MPI implementation, as it is the first library that implements new features.

2.4 Hardware accelerators

Hardware accelerators are specific pieces of hardware designed to perform certain tasks very efficiently and allow to offload that work from the CPU. The first elements used to accelerate the performance of the computers were processors separated from the CPU, referred to as coprocessors.

Time-consuming operations like input/output penalize the performance due to the time the CPU is waiting idle for the data to be read or written. If those operations are delegated to a specialized device managed by the CPU, who instructs this coprocessor to carry out the desired operation, the CPU can carry on performing other computations. Floating-point operations are another example that on ancient computers could be done much faster in a mathematical coprocessor than in the CPU. Other accelerators specialize in network transmissions like the TCP offload engines (TOE), or in digital signal processing. A variety of technologies have been used in a specialized way to accelerate different parts of the computation. Along with the evolution of the hardware, it is not uncommon that the coprocessor functionalities, like the one provided by the floating-point units or the graphics processors, end up integrated in the CPU, completing the *wheel of reincarnation* [99].

Coprocessors can be simple processors operated by means of specific instructions belonging to the CPU instruction set, or fully independent processors. Here we present some of the multiple technologies that have been used to accelerate the computation.

2.4.1 Integrated circuits

Application-specific integrated circuits (ASIC) are customized circuits to perform particular tasks very fast, instead of being a general-purpose solution. An ASIC can implement the full logic of a microprocessor, including ROM and RAM memories or other components like network modules. Such integration is referred to as system on a chip (SoC).

Programmable logic devices like field-programmable gate arrays (FPGA) are integrated circuits composed by a grid of logic gates, designed to be programmed by the user, instead of by the manufacturer. These devices provide the flexibility of being reprogrammable even at run time. One of their traditional applications, given their multipurpose character, is the rapid ASIC development, acting as low-cost prototypes.

FPGAs are less dense, more energy demanding and do not reach the performance of ASICs, but its development has reached a level of maturity with capabilities competitive with them. They have been widely used as digital signal processors (DSP) but can also implement more complex logics, like SoC. FPGA cards are included in servers by manufacturers, and are intensively used in HPC [108] as fine grained accelerators to speedup some (parts of) workflows for their flexibility and low-cost, and for the good performance per watt ratio that they provide, giving them an advantage respect to CPUs or other devices like graphics processing units. Another characteristic that emphasizes their versatility is that developers can program them with hardware description languages like VHDL or by means of traditional software languages like C, C++, Python or OpenCL.

2.4.2 Manycore processors

The natural trend in multicore chip design is to increase the number of integrated cores in a single die, but an additional level of scalability in the integration process is obtained by assembling several cores in tiles and several tiles on the same chip. Two different strategies are followed for homogeneous multicore processors that allow us to make a distinction based on the type of cores that they use.

A first family of manycore processors integrate a large number of simple low-performance cores running at not very high frequencies, with simple architectures, and with their own memories. This kind of processors are used as accelerators, in the form of an external card connected to standard expansion buses like PCIe, or directly connected to a high-performance interconnection fabric that alleviates the bottleneck of using the PCIe bus when copying data to and from the memory of the devices. A typical example of this family of manycore processors are GPUs, that originated this category.

A second family are those using complex high-performance cores. This fact allows to maintain a good speed on sequential programs and at the same time provides much performance when parallelizing the code. Standalone processors from IBM, Intel, AMD or ARM-based processors employed as CPUs, are expected to eventually fit in this category. A good example of scalable manycore is the tile-based architecture

family of processors from Tilera[5] [18] (successor of the Raw microprocessor [130]), that includes one ARM-based processor per tile, and the tiles, arranged in a grid, are interconnected with an on-chip mesh network.

The Xeon Phi processor from Intel is a similar product that, due to the changes experienced, somewhat lies in between both families. It is referred to as many integrated core (MIC) processor, so as such it has a considerable large number of cores, and on its first generations it was exclusively a coprocessor attached to the PCIe bus. The distinguishing feature of Xeon Phi with respect to the first family of manycore systems is that their cores are full performance multithreaded cores that share the instruction set architecture (ISA) of common CPUs (x86). The fact of sharing the ISA makes MICs more general-purpose than GPUs, allowing them to be programmable as a normal general-purpose processor, without the need to port the software.

The current generation of Xeon Phi combines two cores on a single tile and replicates the tiles on the die. Each tile has two vector processing units (VPU) per core and 1 MB of level 2 cache shared between the two cores. The tiles, that on previous generations were interconnected through a cache coherent one-dimensional ring bus, now use a cache coherent, two-dimensional mesh to communicate with other parts of the chip. As other manycore products, current generations can be attached to a standard bus or directly to a specific interconnection fabric, but Xeon Phi can also be used both as a coprocessor and as standalone processor, capable of booting the operating system, so its capabilities are closer to the second family than to the first one.

A different family of manycore processors are the heterogeneous systems, that integrate cores with different ISAs and capabilities. An important example of this family is the Chinese SW26010 [147], a heterogeneous chip used in the supercomputer Sunway TaihuLight [47], number one of the Top500 [131] from June 2016 to November 2017.

2.4.3 Graphics processing units

The use of graphics devices as accelerators has followed a different path than the manycore processors evolved from the CPUs. Even before IBM developed its first multicore processor, graphics devices were already in use as massively parallel devices due to the parallel nature of graphics algorithms.

The process of generating an image to be presented on a display consists of a series of sequential stages called the *graphics pipeline* [69]. The classical four stages of the pipeline are the vertex processing, the triangle rasterization, the pixel texturing and lighting, and the final merge of the generated shading with color and depth. As a pipeline, the output from one stage is treated as the input for the next, and although the processing is serial, some of these steps, like the vertex and pixel stages, perform floating-point operations without any type of data dependency, so with the appropriate hardware, the stage can operate on all the data in parallel.

[5]Now part of Mellanox.

With the first graphics devices, the applications interested in displaying an image had to render it using the CPU to go through the pipeline, and once formed, sent it to the graphics card. That behaviour changed with the introduction of programmable graphics devices, that enabled the use of callback functions, known as *shaders*, that are executed on the device for each vertex, or pixel, on their respective stages, unloading the CPU from those operations. These devices usually have to have different processors specialized on each stage, causing the idle state of the processors when the computation was taking place on another stage.

Early efforts on the field of general-purpose computation on graphics processing units (GPGPU) had started in 1980 with the Ikonas Raster Display System, a fully programmable graphics processor [41, 42], that through the use of an ad-hoc high-level command language, the Ikonas display language, allowed the users to send programs to the devices to instruct the cards on how to process, in parallel, the polygons and images to be displayed [134]. But it was not until 2001 when OpenGL [76] and DirectX included support for programmable shaders, that a major transformation was experienced in the computer graphics world. Such feature allows programmers to port data parallel algorithms to the GPU in the form of shaders. In order to execute generic programs on the GPU, the data is moved to the memory of the GPU as a 2D texture, and a rectangle fitted to that texture is rasterized to produce a result that is stored in the framebuffer. The same year, NVIDIA released the first GPU with programmable pixel and vertex shaders, the GeForce 3, with which the first matrix-matrix multiplication on a GPU took place [85].

The next and even more important milestone in GPU computing, was the release of the G80 architecture as part of the NVIDIA GeForce 8 series in 2006. The G80 came with the introduction of the Compute Unified Device Architecture (CUDA), the first GPU architecture specially designed as a general-purpose computing platform.

The adoption of CUDA made the GeForce 8800 a revolutionary device, being the first one that replaced separate vertex and pixel specific-purpose processors with a single unified processor, in which each one of the pipeline stages is executed. The fact of providing a unified processor avoids the idle cycles of the old dedicated processors when processing graphics, and enables the possibility of executing generic computing programs. CUDA-enabled cards replaced the pipeline model by adopting a single instruction multiple thread (SIMT) execution model, and changed the programmers job from managing vertex and pixels through the use of callbacks and accessing the texture memory, to handle threads and accessing a shared, generic, global, byte-addressable memory, also used for inter-thread communications.

The fact that a common device included in almost any computer, like a graphics card could be used as a general-purpose processor was accompanied with the support for C and C++ programming languages, feature that removed barriers for its adoption and widely increased the number of potential users, turning CUDA into the most popular GPU-computing platform.

CUDA overview

We focus now on the GPGPU perspective of CUDA, rather than on its graphics origin or functionality, but without forgetting them, as otherwise one could have the feeling that something constraints its general-purpose orientation.

CUDA implies heterogeneous programming, involving CPU and GPU processors. The convention is to use *host* and *device* names to denote the CPU and GPU domains respectively, and *kernel* to denote a function compiled to run on the device. As accelerator, a process can not run exclusively on the device, but needs to be launched as a normal process from the host. Once the execution has started, when the process reaches a specific point in the code, it instructs the device to perform the computation of a particular kernel. Before and after calling the kernel, the host program can optionally copy data to and from the device memory to place operands and/or retrieve a result. From the point of view of the host code, the kernel call is asynchronous, enabling to perform different computations in parallel on CPU and GPU.

When talking about CUDA it is necessary to establish a distinction between the hardware architecture and the associated software platform. Different generations of the hardware not only have different design but also support different features that are specified by its compute capability version. The software platform allows code to run on CUDA devices, independently of the hardware generation. It provides support for hardware-agnostic features and other ones that are compute-capability specific.

The physical design of the architecture follows a hierarchical structure, having the *Streaming Processors* (SP) as its computational core, on which the threads are executed. The card's layout if formed by several *Graphics Processing Clusters* (GPC), each of them having multiple *Streaming Multiprocessors* (SM), that are the ones that contain the SPs.

Unlike on previous graphics architectures that employed vector processors, the SP is an *in-order* scalar processor that contains an integer arithmetic logic unit (ALU) and a floating-point unit (FPU) to perform general-purpose computations. SMs on their side, are composed of a collection of SPs, *Special Function Units* (SFU), load/store units, a registers memory, and a shared memory. GPCs contain several multiprocessors and a raster engine. Registers and shared memory are *on-chip* SRAM memories, and are analogous in latency and bandwidth to the cache memory of a CPU. A larger *off-chip* DRAM memory acts as the main memory of the device. It is reachable from all the SPs and encompasses both local and global memory. Figure 2.1 represents the CUDA memory hierarchy on which the different types of memory appear, and Table 2.1 shows the main characteristics of each of those memories.

From a logical point of view, a kernel is executed (potentially) in parallel on the device by multiple threads. The set of threads executing the same kernel forms a *grid*, and the threads of a grid that are mapped to run on the same SM are organized in blocks within the grid. The threads that belong to the same block have access to the shared memory of the SM and can be synchronized explicitly. Thread blocks

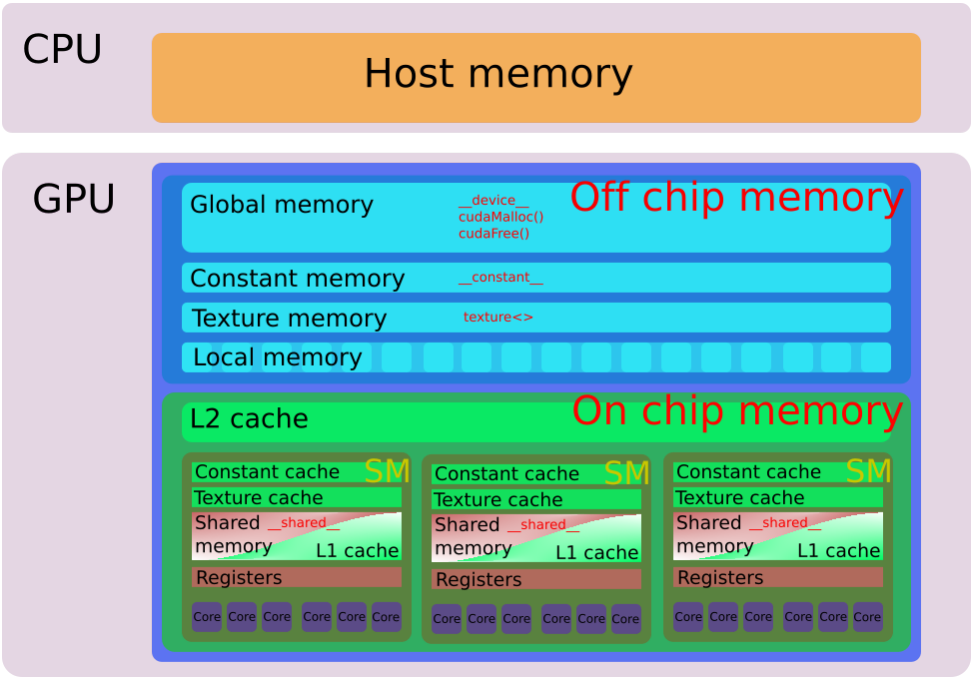

Figure 2.1. CUDA device memory hierarchy.

Table 2.1. CUDA device memory.

Memory	Location	Cached	Access	Scope	Lifetime
Register	On chip	No	R/W	Single thread	Thread
Local	Off chip	Yes	R/W	Single thread	Thread
Shared	On chip	No	R/W	All threads in block	Block
Global	Off chip	Yes	R/W	All threads + host	Host allocation
Constant	Off chip	Yes	R	All threads + host	Host allocation
Texture	Off chip	Yes	R	All threads + host	Host allocation

can be organized in one, two or three dimensions into the grid and are required to execute independently, so they can be executed in-order within a dimension, in parallel, or in any other arbitrary order. Threads within a block can also form 1D, 2D, or 3D structures.

During the execution, an SM partitions its assigned thread blocks in groups of consecutive threads denominated *warps*[6], that are the minimal scheduling unit managed by the multiprocessor. Each one of the threads belonging to a warp has its own program counter[7] and registers to keep track of its state, allowing them to follow different execution branches. Within a warp, all the threads begin the execution at the same instruction, and a warp executes a single common instruction at a time, so if branch divergences occur, only the threads on that instruction's branch will be active on the current cycle. The scheduling of the warps has important implications in the performance. On the one hand, it increases the performance by hiding the memory latency, as it selects from a pool the warps that are ready for execution. On the other hand, only a number of threads per block multiple of the warp size is able to optimally populate the SMs, and the more divergent branches appear during the execution, the more the performance is reduced.

The architecture has experienced a lot of improvements through the years. One expected (and accomplished) enhancement is the increase of the assembled multiprocessors, and SPs per multiprocessor, on each new generation, thanks to the reduction of the manufacturing process and the increase of the die size, reaching thousands of cores in a single device.

Some other improvements on *Fermi* architecture (2010) were the support for double precision floating-point operations, the inclusion of a hierarchy of cache memories on the SMs, and the increase of *warp* schedulers per SM. *Kepler* architecture (2012) added a *Grid Management Unit* (GMU) that performs the scheduling among the different existing grids, queueing those who are not ready to be executed and prioritizing those who are, trying to improve the performance. Kepler also incorporates *GPUDirect*, a proprietary technology that allows data transfers between different GPUs without the intervention of the CPU or the memory system, by allowing network adapters to directly access the memory of the GPUs. *Pascal* (2016) introduced the *NVLink* high-bandwidth proprietary communication channel that replaces the use of a PCIe bus to perform GPU-GPU and GPU-CPU transfers, and the unified memory, a single virtual address space shared between the CPUs and the GPUs. Independent thread scheduling and special-purpose computing units that perform fused multiply-add operations working with 4×4 matrices were added in *Volta* (2017), with the name of *tensor cores*.

The general-purpose orientation of the architecture, the growth of the complexity, the creation of new data channels to connect GPUs with CPUs, the unification of the GPU and CPU memory, and the addition of specialized computing elements like the tensor cores inevitably reminds the old wheel of reincarnation, on which the design of the graphics processors results in a never-ending increase of complexity and the addition of auxiliary processors, that in turn become more complex.

[6] Warp size depends on the compute capability, being 32 threads on all existing versions.

[7] Originally a single program counter was shared between all the threads of a warp.

CUDA programming

The programming of CUDA devices is done mainly in C and C++ languages[8], as the architecture provides a small extension for these languages that allows to define code to be run on the device. As CUDA programs can not be launched to run exclusively on the device, using the CUDA extension implies having two types of code on the program: code that runs on the CPU (host code) and code that runs on the GPU (device code). The language extension [30] provides a series of function and variable qualifiers that determine the platform target of the corresponding function (CPU or GPU), and set the type of memory on which the variable has to be allocated on the GPU memory, respectively. There exist three function qualifiers:

`__host__`: declares a normal C function, invoked from the host and executed on the host.

`__global__`: declares a function invoked from the host and executed on the device. These functions, also called kernels, are the ones whose invocation switches the execution to take place on the GPU.

`__device__`: declares a function invoked from the device and executed on the device.

Note that the host-device relation is not bidirectional, and it is not possible to launch a host function from device code, or a kernel from a `__device__` function. The rationale of having a `__host__` qualifier, equivalent to not using any qualifier at all, is that a function can have multiple qualifiers, being possible to have `__host__` and `__device__` qualifiers, and in that case, the code is compiled for running on both the host and the device.

Another five qualifiers are used for variables:

`__device__`: declares a variable that resides on the device. It is allocated on global memory if no other qualifier is used.

`__constant__`: declares a variable that resides on constant memory. Optionally used with `__device__`.

`__shared__`: declares a variable that resides on shared memory. Optionally used with `__device__`.

`__managed__`: declares a variable that can be directly referenced from both, host and device code. Optionally used with `__device__`.

`__restrict__`: used to specify non-aliased pointers.

Any automatic variable (variables without qualifiers) declared in device code should be placed in register memory by default, but as the amount of memory is limited, the compiler could have to place some variables on local memory. That

[8]There are Fortran compilers from PGI and IBM, and several bindings for other programming languages such as Python.

Table 2.2. CUDA variable declaration and associated memory.

Declaration of variables	Memory where they reside
Automatic variables	Register
Automatic variables (large)	Local
`__device__ __shared__`	Shared
`__device__`	Global
`__device__ __constant__`	Constant

can happen when the function has many variables or with some types of arrays. Table 2.2 shows the type of memory where the variables are allocated with the different qualifiers. There exist some restrictions to the variable qualifiers. The `__device__`, `__shared__` and `__constant__` specifiers are not allowed on classes, structures, unions, formal function parameters, and local variables on host code.

CUDA defines some built-in data types and variables present at compile time and run time, respectively. The data types defined are integer and floating-point structures with up to four components, that allow to specify four-dimensional elements. One relevant data type defined is `dim3`, a three-dimensional integer vector type. If a variable is defined with this type, the unspecified components are initialized to one, reducing the actual number of dimensions. Four built-in variables of types `dim3` and `uint3` are present on all `__device__` and `__global__` functions:

`dim3 gridDim`: provides the dimensions of the grid (in blocks).

`dim3 blockDim`: provides the dimensions of the blocks (in threads).

`uint3 blockIdx`: provides the block index within the grid.

`uint3 threadIdx`: provides the thread index within the block.

These variables are important for the correctness of the code as they allow it to determine things like the total number of threads and blocks or unique thread IDs. Kernel functions have to be written from the point of view of a single thread, identifying it by its ID in a similar way as the rank identifies a MPI process. Another built-in variable of type `int` is `warpSize`, that specifies the warp size in threads.

As already mentioned, kernel launches are asynchronous with respect to the host, allowing concurrent execution of host and device code. Other tasks that can operate concurrently are memory transfers, from the host to the device and vice versa, within the memory of a specific device or among devices. All these operations may operate concurrently given that they use different *streams*. Streams are series of commands that can operate concurrently or out-of-order between them. The commands within a given stream are executed in-order, though.

If invoking a function in C may require passing some arguments to it, launching a kernel implies the use of additional arguments in the call. These additional kernel-specific arguments indicate how the functions have to be executed on the device.

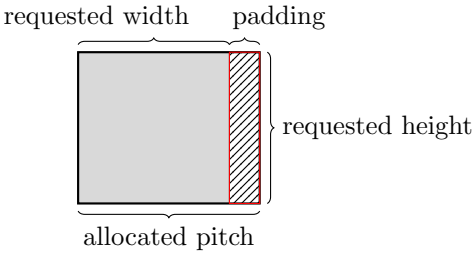

Figure 2.2. Schema of 2D memory allocated in the GPU with `cudaMallocPitch`.

The full kernel invocation

```
kernel <<<Dimgrid, Dimblock, Shr, Str>>>([kernel arguments]);
```

has four parameters given via the `<<< >>>` operator. `Dimgrid` specifies the number of blocks that constitute the dimensions of the grid, `Dimblock` specifies the number of threads per block, `Shr` specifies the number of bytes of shared memory dynamically allocated per thread block, and `Str` specifies the stream linked to the call. Multiple kernels can be executed simultaneously by sending them to different streams[9]. The last two parameters are optional and default to zero if not employed, allowing a kernel to be launched as

```
kernel <<<Dimgrid, Dimblock>>>([kernel arguments]);
```

All the kernel launches which do not specify a stream parameter or explicitly set it to zero are associated with the default stream, and therefore executed in-order within that stream.

CUDA runtime

Besides the user-provided kernels, there exists a broad functionality offered by the CUDA runtime API[10]. It offers functions to allocate/deallocate the host and device memory, to set or copy memory, to define the ratio of shared memory and L1 cache, to manage the devices, or to manage the streams. It also provides an automatic context initialization and management.

We want to highlight some details about the memory management offered by the runtime. The runtime allows to allocate memory on both the host and the device. On the device it provides specific functions to allocate memory on a structure basis. Two-dimensional arrays can be allocated with the function `cudaMallocPitch`, that may pad the rows to ensure the hardware alignment for a coalesced access to the memory. See Figure 2.2. Three-dimensional arrays can be allocated aligned in a similar way with `cudaMalloc3D`. Once the memory is allocated aligned, specific

[9] It is not possible to have more than 32 concurrent kernels executing at the same time.
[10] A low level driver API offering a more fine-grained control, similar to OpenCL, is also available.

functions can be used to copy or set it with values in a efficient way. For the host, the runtime provides the function `cudaHostAlloc` to allocate page-locked (pinned) memory. The CPU implements a virtual memory system that allows programs to use more memory than the available in the system by swapping out unused pages and swapping them in again when needed. Page-locked memory ensures that the operating system does not page the memory out to swap memory (to disk or any other secondary memory), and that it can be accessed directly by the device by means of direct memory access (DMA) without using intermediate buffers[11]. This memory can be read or written with a higher bandwidth than pageable memory allocated with `malloc`. Memory allocated with `malloc` can also be page-locked with the function `cudaHostRegister`.

CUDA software

The software toolkit also includes a mathematical library, and multiple numerical libraries like cuBLAS [101] and cuSPARSE that implement BLAS on top of the CUDA runtime, for dense and sparse operators respectively.

cuBLAS implements all the three levels of BLAS, and provides the complete set of BLAS standard routines. Moreover, it includes an additional set of routines that extend BLAS functionality with mixed precision and batched operations, including some LAPACK routines. Batched operations allow to compute concurrently the same operation on multiple small data in a single kernel. For instance, the batched _gemm routine performs the matrix-matrix multiplication of a batch of matrices. Batched operations allow to fully utilize the CUDA streaming processors and obtain good performance with small input data. It allows the concurrent execution of multiple kernels through the use of streams and has partial support for multi-GPU operations, that in the case of large problems can perform hybrid CPU-GPU computations. The functionality provided by cuBLAS can be used from host code and from user defined kernels.

cuSPARSE provides an implementation of the sparse interface for basic linear algebra subroutines. It supports three levels of operation.

Level 1: routines for vector-vector operations involving a sparse vector and a dense vector.

Level 2: routines for matrix-vector operations involving a sparse matrix and a dense vector.

Level 3: routines for matrix-matrix operations involving a sparse matrix and a dense matrix[12].

Besides dense matrix storage, it supports a variety of sparse formats like COO, CSR, CSC, and Blocked CSR, and provides conversion routines that allow the conversion between these formats. These sparse formats, that are suitable for cache-aware CPU

[11] Pageable memory is copied to a page-locked intermediate buffer previous to its copy when using DMA.

[12] Additionally, it has support for sparse matrix by sparse matrix addition and multiplication.

platforms, usually provide poor performance on GPUs. For this, new formats are developed to allow an efficient implementation of a sparse matrix-vector product on GPUs [17, 68]. A sparse format included in cuSPARSE, specifically designed for the use on GPU is HYB.

Hybrid ELL and COO format (HYB). It is a scheme that combines ELL and COO formats. As the efficiency of ELL decreases when the number of nonzero elements per row varies significantly, the ELL is combined with the COO format that is not affected by the number of nonzeros per row.

On top of cuBLAS and cuSPARSE is built cuSOLVER, a library that provides support for LAPACK-like operations on the GPU. The two main components of cuSOLVER are cuSolverDN for dense LAPACK routines, and cuSolverSP for sparse LAPACK routines.

A different library included in the CUDA toolkit is Thrust. It is a high-level (object-oriented) C++ template library that provides an interface based on the C++ Standard Template Library (STL). Thrust allows users to benefit from a parallel execution on the GPU without the need to write any kernel, or explicitly manage device memory, or use other CUDA libraries. The underlying idea is to simplify the programming by allowing users to exclusively use C++ code, and transparently allocate certain objects on the GPU to perform the operations. Thrust code can also be executed on CPU.

On top of Thrust is developed CUSP [33], other parallel linear algebra library, released under an Apache license, that provides a high-level interface for manipulating sparse matrices on GPU. CUSP has support for dense and a variety of sparse matrix formats like COO, CSR, DIA, ELL, and HYB. And it also allows the conversion between these formats. It furnishes iterative methods for solving linear systems and for computing eigenpairs, and also several preconditioners for the iterative solvers. It provides sparse matrix-vector and matrix-matrix multiplication, and other sparse matrix operations like add, subtract, or transpose. CUSP also includes a subset of BLAS and LAPACK routines for performing operations with dense matrices.

Lastly, we want to mention MAGMA (Matrix Algebra on GPU and Multicore Architectures) [132], a high-performance library focused on providing dense linear algebra functionality similar to LAPACK, on heterogeneous multi-GPU and multicore architectures. It tries to obtain the best performance by combining executions on CPU and on GPU. It provides two interfaces, one that takes the operators and stores the result using CPU memory, and other analogous that uses GPU memory. It makes use of cuBLAS for BLAS operations, but also includes its own implementation of BLAS routines. Contains a sparse module that implements sparse BLAS routines as well as functions to handle the iterative solution of a sparse linear system of equations. MAGMA supports three sparse formats: CSR, ELL and SELL-P [8], a variant of SELL-C [77].

SELL-C. Format that tries to reduce the storage overhead of the ELL format by splitting the original matrix into blocks of rows, and storing each partition using ELL.

Padded sliced ELLPACK format (SELL-P). Modifies SELL-C by adding rows with zeros such that the row length of each block becomes a multiple of the number of (CUDA) threads assigned to each row. It allows instruction parallelism and coalesced memory access.

MAGMA is published with a BSD license.

2.5 Performance indicators

When developing new algorithms to solve a well-known problem, it is important to have a method to compare them. Assuming that several of them solve the problem accurately, the desired behaviour is to obtain the solution fast. Usually, the faster the better. In the same way that the runners can compare their performance with the time set by the winners of the world championships in athletics, the software needs a benchmark to compare with. The easiest way to obtain a benchmark is by using the time of the fastest existing algorithm. If no previous version exists, the new algorithm will be the benchmark for future algorithms.

The reason behind the hardware innovation process is that the performance of a program is always going to be restricted by the hardware it runs on. The hardware sets the upper limit. If a specific software is able to obtain the maximum production of the hardware, it has accomplished the objectives. The *performance* of HPC implies obtaining, if not the maximum, at least a significant fulfillment. The issue with the parallel hardware is that reaching its limits is not possible with a single serial program, and even parallel software can have difficulties to succeed.

Although the natural indicator to measure the performance of the software is the time, other factors like the number of floating-point operations per second (FLOPs/s) done by a program, the relative speedup obtained between two implementations, and the concepts of efficiency and scalability, are presented along this section. We will not consider other performance metrics like performance per watt, known as green computing, or performance per monetary unit.

2.5.1 Execution time

The main goal of parallel computing is the reduction of the computational time. A parallel program can make a more intensive use of the hardware by fully using, at the same time, all the computational resources available. Is the time then, the principal performance indicator of a parallelization, and this performance is going to be expressed as a relation to the time of the execution of the equivalent serial algorithm.

The concept of execution time is easy to understand, but the performance obtained with a parallel version of an algorithm is not straightforward to estimate just from the time of a serial execution. For a serial program, the execution time is the elapsed time since the program starts until its termination, and for a parallel program, the execution time begins when the first of the involved processors starts the computation, and ends when the last remaining working processor finalizes. The

intuition tells us that the more processors a parallel program uses, the less time it should need to complete. A naive expected result is that the time of a parallel execution, that uses two processors, is going to be reduced to half the time of the serial execution. But parallel programming has additional factors that overload the computation and increase the processing time, reducing the gain obtained.

One characteristic of a parallel program is the need of extra steps not present on the serial implementation. The memory access management on shared-memory systems, or the communication non-overlapped with computation on distributed systems, imply wasting time in management activities.

If the processes have to perform different computations, it is possible that some of them have to stand idle for others, slowing down the whole program. For this reason, it is very important to load-balance the computation between the processes during all the computing process, and this usually means to equally distribute the data among them. Also, the data distribution used, not speaking of the amount of data but to what data are mapped to what process, is linked with the communication that the different processes have to perform during the execution.

On distributed memory systems, the communication time is directly affected by the latency and transfer rate of the intercommunication network. So the use of low-latency and high-transfer rate networks is mandatory on these systems. Even with low latencies, as it is a fixed cost that does not depend on the message size, it is useful to unify several small messages into a larger one to reduce the communication time.

Our final comment goes to the fact that it is not always possible to parallelize all the steps of an algorithm. This is known as the Amdahl's law [4], and has a big impact on the final performance. Being

$$T_1 = T_{(s)} + T_{(p)}, \tag{2.2}$$

the total serial execution time of a program, where $T_{(s)}$ represents the time of the intrinsically serial portion of the program, and $T_{(p)}$ the time of the parallelizable portion. Amdahl's law says that the minimal parallel time that can be obtained with p processors is limited to

$$T_p = T_{(s)} + \frac{T_{(p)}}{p}. \tag{2.3}$$

2.5.2 Speedup

Viewing two versions of the same program, one of them serial and other one parallel, as black boxes that given some inputs both return the same outputs, the different time that both versions take to process the information is going to define their relative performance, called speedup. Formally, the speedup is

$$S_p = \frac{T_1}{T_p}, \tag{2.4}$$

where T_1 represents the execution time of the best serial implementation, or the execution of the parallel algorithm with a single processor; and T_p represents the

execution time of the parallel implementation, were p is the number of processors used. Theoretically, the speedup is $S_p \leq p$ having p as limit, but there exists the possibility for some parallel algorithms to obtain super-linear speedups (greater that p). Super-linear speedups occur due to a more optimal or intense use of the memory of the system, normally the cache memory.

The common case, though, is to reach speedup values smaller than p because of the factors mentioned in § 2.5.1, and as Amdahl's law states, the maximum speedup is limited to

$$S_p = 1 + \frac{T_{(p)}}{T_{(s)}}. \tag{2.5}$$

2.5.3 Efficiency

The speedup allows us to compute the efficiency of a parallel algorithm, that gives the degree of utilization of the parallel resources with that algorithm, in proportion to the use of the resources achieved with the serial algorithm. This metric relates the obtained speedup with the number of processors employed

$$E_p = \frac{S_p}{p}. \tag{2.6}$$

In the same way that the speedup is usually smaller than p, in the practice the efficiency of the parallel algorithms usually is smaller than 1.

2.5.4 Scalability

Two important things to consider when measuring the performance of the parallel software is the influence of the hardware used, and the size of the problem to be solved. The normal procedure to evaluate the performance is to vary the parameters of the execution, to know how the parallel version works on different conditions.

The scalability of an algorithm tells us if it will take advantage of the increment of the hardware resources, by maintaining its efficiency. Two types of scalability can be measured, depending on if the increase of hardware resources goes with an equivalent increment of the problem size, or not.

Strong scaling studies how the number of processors used affects to the execution time, while keeping a fixed total problem size.

Weak scaling studies how the execution time vary when modifying the number of processors, while modifying accordingly the total problem size, maintaining a fixed problem size per processor.

2.5.5 FLOPs/s

The use of the time as performance indicator that we have seen until now is empirical, only possible by executing a working implementation on a specific hardware. A different way of measuring the numerical performance of a computer, or the cost

of an algorithm is by counting the number of floating-point operations. Floating-point operations are employed as performance metric in computing as they are the core operations in scientific software, contrary to integer operations that can be considered negligible.

In case of hardware performance, the target is to provide as many FLOPs/s as possible. It results in an architecture independent metric that allows to compare different pieces of hardware. Traditionally, for nodes with one single-core superscalar processor, that works with a fixed frequency, the theoretical floating-point peak performance of a computer could be determined by multiplying the number of floating-point operations per cycle, by the frequency. The addition of several sockets and cores per processor expands the formula

$$\frac{FLOPs}{node} = \left(\frac{FLOPs}{cycle}\right)\left(\frac{cycles}{second}\right)\left(\frac{cores}{socket}\right)\left(\frac{sockets}{node}\right), \tag{2.7}$$

but it turns out to be still too simplistic, as it assumes a fully unified performance on each stage.

However, the current sophistication of the processors and the hardware architecture make this calculus not so straightforward [36]. The number of FLOPs per cycle coming from the micro architecture elements

$$m_{FLOPs} = \left(\frac{FLOPs}{operation}\right)\left(\frac{operation}{instructions}\right)\left(\frac{instructions}{cycle}\right), \tag{2.8}$$

and the number of cycles per second (per node) coming from the machine architecture elements

$$M_{cycles} = \left(\frac{cycles}{second}\right)\left(\frac{cores}{socket}\right)\left(\frac{sockets}{node}\right). \tag{2.9}$$

can be combined to give an estimation of FLOPs per node

$$\frac{FLOPs}{node} = \frac{m_{FLOPs}}{M_{cycles}}. \tag{2.10}$$

But hardware evolution turns this calculus into a complicated enterprise, in which vector operations, dynamic frequencies (including different frequencies per core), fused operations, multithreading behaviour, or the support for several instruction sets are characteristics that influence the performance.

FLOPs can also be used as a theoretical metric for the cost of an algorithm, independent of the hardware, that can be obtained a priori, just by examining the operations in the algorithm. It allows us to compare different algorithms that solve the same problem without the need of implementing them. Traditionally, algorithms with less FLOPs would finish in less time, but that is not a golden rule with parallel hardware, where high-FLOPs-demanding algorithms can exploit the parallel resources and finish sooner than less-demanding ones.

Chapter 3

Eigenvalue problems

Il meglio è nemico del bene

Computing eigenvalues and eigenvectors is required by a wide range of scientific and engineering applications.

Among others, eigenvalues are used in engineering to determine the natural frequencies of vibration (vibration analysis) of large structures like buildings and bridges during their design. In control theory, the eigenvalues are used to determine the stability and response of dynamical systems like the analysis of the dynamic stability of electrical circuits and fluid flow. In physics, they allow the analysis of systems modeled by the Schrödinger equation, like the parametrization of quantum cascade lasers and superlattices. They also allow to study the behavior of mechanical engineering and theoretical physics modeled with Poisson Equation like the behaviour of plasma. In chemistry they allow the simulation of molecular clusters.

Numerical linear algebra is a very active field of research and many mathematical forms of solving the eigenvalue problem have arisen from a variety of authors. A good description of the eigenvalue problem is available [12, 115, 129], including a good article with a historical perspective [53].

The standard eigenvalue problem is formulated as

$$Ax = \lambda x, \quad x \neq 0, \tag{3.1}$$

where $A \in \mathbb{C}^{n\times n}$ is the matrix that defines the problem, $\lambda \in \mathbb{C}$ is an eigenvalue and $x \in \mathbb{C}^n$ is a right eigenvector of A. The pair $(\lambda, x) \in \mathbb{C} \times \mathbb{C}^n$ is called an eigenpair of A.

Although the eigenvalues are complex in general, the eigenvalues of a Hermitian matrix $H \in \mathbb{C}^{n\times n}$ are real, and if the matrix is real symmetric $H \in \mathbb{R}^{n\times n}$, then the eigenvectors are also real.

Other types of eigenvalue problems that frequently arise in applications lead to

the generalized form, in which there are two intervening matrices,

$$Ax = \lambda Bx. \tag{3.2}$$

The pair of matrices of equation (3.2) define the *matrix pencil*

$$A - \lambda B, \tag{3.3}$$

and the eigenvalues of the matrix pencil are those λ for which

$$\det(A - \lambda B) = 0. \tag{3.4}$$

Under some assumptions, the generalized problem can be reduced to a standard eigenvalue problem and solved as such. For example, being matrix B non-singular, the problem can be transformed in

$$B^{-1}Ax = \lambda x. \tag{3.5}$$

3.1 Methods to compute eigenvalues

Generally speaking, there are two broad classes of methods for solving the eigenvalue problem, that we could call direct and iterative methods. When the matrices are dense and all the eigenpairs are required, the direct methods are usually the preferred choice. They reduce the matrix to a condensed form by applying orthogonal transformations, from which the eigenvalues can be easily obtained. On the contrary, if only a few eigenpairs are required and the matrices are large and sparse, projecting the eigenproblem on a low-dimensional subspace is normally a better option.

3.1.1 Direct methods

Direct methods are based on applying orthogonal transformations to reduce the matrix involved in (3.1) to a condensed canonical form from which eigenpairs can be recovered almost trivially, such as the (generalized) Schur form. For example, given a non-singular matrix X, the transformation XAX^{-1} has the same eigenvalues as A and its eigenvectors are of the form Xx, where x is an eigenvector of A.

The Schur form is the most practical and used form, as it can always be obtained in a stable way by means of unitary transformations. For the generic non-symmetric case, the Schur decomposition of A can be expressed as

$$A = QTQ^*, \tag{3.6}$$

where T is an upper triangular matrix, whose diagonal elements are the eigenvalues of A, and Q is a unitary[1] matrix whose columns q_i are called Schur vectors. If s_i is an eigenvector of T, then Qs_i is an eigenvector of A.

[1] A unitary matrix has as inverse its conjugate transpose.

The symmetric case is a particularly simple case of the Schur form that, if A is real reads

$$A = UDU^*, \tag{3.7}$$

where D is a diagonal matrix with the eigenvalues of A on its diagonal, and the columns of U are eigenvectors of A.

In order to reduce the cost of applying similarity transformations, direct methods begin by reducing matrix A to either tridiagonal or upper Hessenberg form, in the symmetric or non-symmetric case, respectively. Once the problem has been reduced to a condensed form, various algorithms can be applied to compute the eigenvalues, such as the QR iteration, or specific methods for symmetric tridiagonals, such as divide-and-conquer [31]. The QR iteration is a numerically stable algorithm that converges to the Schur form of the initial matrix.

When the matrices are dense and not too large, direct reduction methods are appropriate, but usually they are not so for the case of sparse matrices because they destroy sparsity, and furthermore they compute all eigenvalues, while often it is enough to compute just a few, especially in large-scale problems with sparse matrices.

3.1.2 Iterative methods

Iterative methods are based on obtaining approximations of the eigenvalues by generating a sequence of operations that given an initial guess converge to the desired values. On each iteration, the approximation obtained is derived from the approximation of the previous iteration. These techniques preserve sparsity and hence they are often the only viable strategy for large, sparse matrices. They are usually better suited for computing only a few eigenpairs. They have a low memory footprint, are amenable to be parallelized, and are scalable.

The simplest iterative method to solve the eigenvalue problem is the power method. It consists in repeatedly performing an iteration that multiplies the matrix A by a vector

$$x_{k+1} = Ax_k/\|Ax_k\|. \tag{3.8}$$

The iteration magnifies the component of that vector in the direction of the eigenvector with largest eigenvalue (in absolute value), relative to the other components. The convergence of this method depends on the ratio of the second largest eigenvalue with respect to the largest one. If both eigenvalues are very close (in magnitude) the convergence will be very slow. It is very easy to implement as it only requires a matrix-vector multiplication, but it can only approximate the dominant eigenvector. Having the eigenvector x, the Rayleigh quotient

$$\lambda = \frac{x^*Ax}{x^*x}, \tag{3.9}$$

allows to obtain the eigenvalue λ associated to x.

One class of iterative methods are those that generate a sequence of subspaces $\mathcal{V}^{(i)}$ and extract approximate solutions from them. Iterative subspace methods are

an extension to the power method that work with multiple vectors instead of with only one. For instance, a good example of them is the original simple subspace iteration method, that for a specific power k it computes the matrix

$$X_k = A^k X_0, \tag{3.10}$$

where X_0 is an initial block of m vectors $[x_1, ..., x_m]$. All of the vectors of the generated subspace converge to the eigenvector associated with the largest eigenvalue. A drawback of this method is that the greater the power used, the poorer the linear independence of the system X_k becomes.

To improve the linear independence of the vectors, the simple subspace iteration can be extended with a factorization, as shown in Algorithm 3.1. The QR decomposition performed in step 2 is a step comparable with the normalization of the power method, and ensures linear independence between the columns of X_k.

Algorithm 3.1. Simple subspace iteration

1 $W = AX_k$
2 Compute QR decomposition of W
3 $X_{k+1} = Q$

A common technique used in eigenproblem algorithms is the use of a projection subspace. The method consists in approximating the eigenvectors of A by means of such subspace $\mathcal{K}$ of a reduced dimension $m \leq n$. The Rayleigh–Ritz process approximates the eigenvectors in that way. It seeks to ensure that the residual of the approximations satisfy the Ritz–Galerkin condition, that in the case of the standard problem it reads

$$A\tilde{x} - \tilde{\lambda}\tilde{x} \perp \mathcal{K}. \tag{3.11}$$

Being $\tilde{x} = Vy$ a linear combination of the m vectors of the subspace $\mathcal{K}$, of which $V = [v_1, v_2, ..., v_m]$ is a basis, this results in

$$V^*AVy - \tilde{\lambda}V^*Vy = 0, \tag{3.12}$$

that simplifies to

$$V^*AVy - \tilde{\lambda}y = 0, \tag{3.13}$$

and then, $\tilde{\lambda}$ and y must satisfy

$$B_m y = \tilde{\lambda}y. \tag{3.14}$$

The Rayleigh–Ritz process, detailed in Algorithm 3.2, is used to accelerate the subspace iteration [13]. Once the subspace is built, the steps 3 to 5 perform the Rayleigh–Ritz method.

The values and vectors obtained with this method in steps 3 and 5 are known as Ritz values and Ritz vectors. The approximate eigenvectors of the original eigenvalue problem obtained in the step 5 are sensitive to rounding errors, but the Schur vectors can be obtained instead in a stable way.

Algorithm 3.2. Rayleigh–Ritz method

1 Compute an orthonormal basis $\{v_1, v_2, ..., v_m\}$ of the subspace $\mathcal{K}$, and let $V = [v_1, v_2, ..., v_m]$.
2 Compute the projected matrix $B_m = V^* A V$.
3 Compute the eigenvalues $\tilde{\lambda}$ of B_m, and select the $k < m$ desired eigenvalues $\tilde{\lambda}_i, i = 1, ..., k$.
4 Compute the k eigenvectors y_i of B_m associated with $\tilde{\lambda}_i$.
5 Compute the corresponding approximate eigenvectors of A, $\tilde{x}_i = V y_i$.

3.2 Krylov methods for eigenvalue problems

One important class of the projection methods is the one based on Krylov subspaces, that allows to compute approximations of the eigenpairs of A from a Krylov subspace

$$\mathcal{K}_m(A, v_1) \equiv \text{span}\{v_1, Av_1, A^2 v_1, \ldots, A^{m-1} v_1\}. \tag{3.15}$$

One example of these methods is the Arnoldi algorithm [9], that builds an orthogonal basis of the Krylov subspace $\mathcal{K}_m$ that constitutes steps 1 and 2 of Algorithm 3.2.

Algorithm 3.3. Arnoldi

Data: Matrix $A \in \mathbb{C}^{n \times n}$ defining (3.1), initial vector $v_1 \in \mathbb{C}^n$
Result: Matrices $H_{m+1} \in \mathbb{C}^{(m+1) \times m}$, $V_{m+1} \in \mathbb{C}^{n \times (m+1)}$ and $h_{m+1,m} \in \mathbb{R}$

1 $v_1 = v_1 / \|v_1\|$
2 $V_1 \leftarrow [v_1]$
3 $H_1 \leftarrow [\,]$
4 **for** $j = 1, 2, \ldots, m$ **do**
5 $\quad w = A v_j$
6 $\quad h_j = V_j^* w$
7 $\quad \tilde{w} = w - V_j h_j$
8 $\quad h_{j+1,j} = \|\tilde{w}\|$
9 $\quad$ **if** $h_{j+1,j} = 0$ **then** break
10 $\quad v_{j+1} = \tilde{w} / h_{j+1,j}$
11 $\quad V_{j+1} \leftarrow [\, V_j \;\; v_{j+1} \,]$
12 $\quad H_{j+1} \leftarrow \begin{bmatrix} H_j & h_j \\ 0 & h_{j+1,j} \end{bmatrix}$
13 **end**

Starting with an initial vector v_1 of norm 1, the method obtains an orthonormal basis $\{v_1, \ldots, v_{m+1}\}$ of $\mathcal{K}_m(A, v_1)$ in an iterative way, and an upper Hessemberg matrix H_m, verifying the Arnoldi relation

$$A V_m = V_m H_m + h_{m+1,m} v_{m+1} e_m^*, \tag{3.16}$$

in which $V_m = \begin{bmatrix} v_1 & \cdots & v_m \end{bmatrix}$. Algorithm 3.3 details the steps followed by Arnoldi's method to generate (3.16). The main steps of the algorithm are the expansion of the Krylov subspace (step 5), the orthogonalization and normalization of the new vector (steps 6, 7, and 10), and the update of the upper Hessenberg matrix H_m (step 12) with the coefficients obtained in the orthogonalization and normalization process. The eigenpairs (θ, s) of H_m generate Ritz vectors $\tilde{x} = V_m s$, and Ritz values, $\tilde{\lambda} = \theta$ that provide approximations of the original eigenvalue problem.

In the case of A being an Hermitian matrix, the matrix H_m becomes a symmetric real tridiagonal matrix

$$T_m = \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \vdots & \vdots & \vdots & \\ & & \beta_{m-2} & \alpha_{m-1} & \beta_{m-1} \\ & & & \beta_{m-1} & \alpha_m \end{bmatrix}. \tag{3.17}$$

Starting from an initial vector v_1 and $\beta_0 = 0$, the recurrence

$$\beta_j v_{j+1} = A v_j - \alpha_j v_j - \beta_{j-1} v_{j-1}, \tag{3.18}$$

with $\alpha_j = v_j^* A v_j$ and $\beta_j = v_{j+1}^* A v_j$ generates a matrix V_m whose columns are the orthogonal $v_1, v_2, ..., v_m$ Lanczos vectors and that satisfies the Lanczos relation

$$A V_m - V_m T_m = \beta_m v_{m+1} e_m^*, \tag{3.19}$$

an expression analogous to the Arnoldi relation (3.16). The Lanczos iteration represented in Algorithm 3.4 can be considered a particular case of the Arnoldi method. The computation involving v_{j+1} can be seen as the orthogonalization of Av_j with

Algorithm 3.4. Lanczos iteration

```
1 for j = 1, 2, ..., m do
2 |   w = A v_j
3 |   w̃ = w − β_{j−1} v_{j−1}
4 |   α_j = v_j^* w̃
5 |   ŵ = w̃ − α_j v_j
6 |   β_j = ‖ŵ‖_2
7 |   if β_j = 0 then break
8 |   v_{j+1} = ŵ/β_j
9 end
```

respect to v_{j-1} and v_j. In exact arithmetic, v_{j+1} is orthogonal to $v_1, v_2, ..., v_{j-2}$, so it is not necessary to use those vectors in the orthogonalization process, reducing the number of operations with respect to the Arnoldi algorithm. However, in practical implementations where finite precision arithmetic is used, the vectors suffer a loss of orthogonality when a Ritz value is close to converge [102], and spurious copies

of the eigenvalues appear. An obvious solution is the use of Arnoldi to keep the orthogonality, but as the projected matrix is tridiagonal symmetric it can be taken into account during the computation. Other techniques try to exploit the fact that the loss of orthogonality occurs when a Ritz value is converging to re-orthogonalize only on such case.

A well-known particularity of the Krylov subspace methods is the constant growth of the subspace on each iteration of the algorithm. In many cases, the algorithm convergence is slow and Ritz approximations may require many iterations to converge. This feature becomes a serious two-fold issue when solving large-scale problems. In the one hand the amount of memory necessary to store the subspace may be too large, and in the other hand the cost of the orthogonalization increases on each iteration, as the dimension of the basis of the Krylov subspace increases. In practice, to have some control over the memory requirements of the method and over the cost of the orthogonalization, the expansion of the subspace has to be limited in some way.

Restart techniques can be used to avoid managing very large subspaces. They can be explicit or implicit and consist in stopping the algorithm when a desired size has been reached, and build a new basis reusing part of the information from the previous basis. The explicit restart explicitly builds an initial vector as start point for the new basis. It can use a linear combination of some of the Ritz vectors of the previous basis, being the simplest form of restart. However, in practice, this approach does not work very well.

The implicit restart [124] improves significantly the convergence with respect to the explicit restart by means of keeping several of the Schur vectors of the previous basis on the new one.

3.2.1 Krylov-Schur

The Krylov-Schur method [128] implements an implicit restart, and extends and contracts a Krylov decomposition in an iterative way. The restart of the method is based on using the Krylov relation

$$AV_m = V_m C + v_{m+1} b^*, \tag{3.20}$$

that is a generalization of the Arnoldi relation (3.16), in the sense that C and b do not require a specific structure. Matrix C is not necessarily an upper Hessemberg matrix, and vector b is an arbitrary vector that can be different from e_m. The bases that appear in the Krylov relation are not the same as the ones that appear in the Arnoldi relation, but they also generate a Krylov subspace.

The restart of the Krylov–Schur method is supported by the fact that as C has the form $C = \left[\begin{smallmatrix} C_1 & \star \\ 0 & C_2 \end{smallmatrix}\right]$, with $C_1 \in \mathbb{C}^{p\times p}$ and $C_2 \in \mathbb{C}^{(m-p)\times(m-p)}$, the relation (3.20) can be truncated. Once truncated, and a basis of dimension p is obtained, it can be extended again up to size m. The idea of the method is to maintain the dimension of the basis under some limits that allow to keep the desired target Ritz values and withdraw the less interesting ones. This restart technique follows the next steps:

1. Compute the Schur decomposition of C and sort it by keeping the most desired Ritz values in the first p positions of the diagonal of T,

$$C = Q_m T Q_m^*. \tag{3.21}$$

   A particular case appears when using real arithmetic, as the obtained matrix T is the real Schur form associated to C, with the eigenvalues in the diagonal or in 2×2 diagonal blocks, so p must be chosen ensuring that the 2×2 blocks of the diagonal are not truncated.

2. Write $Q_m = [Q_p, Q']$, and discard the columns associated to Q', to truncate the Krylov relation of order m into order p,

$$A\tilde{V}_p = \tilde{V}_p T_1 + v_{m+1}\tilde{b}^*, \tag{3.22}$$

   where $T_1 = Q_p^* C Q_p$, $\tilde{V}_p = V_m Q_p$ and $\tilde{b} = Q_p^* b$.

3. Restart of Arnoldi's iteration as in Algorithm 3.3, extending the Krylov subspace by means of $w = Av_{m+1}$

The Krylov–Schur method can be modified for the case of A being Hermitian in a similar way as Lanczos modifies Arnoldi. This specialization is known as the thick-restart Lanczos method [141].

Convergence of Krylov methods for eigenvalue computations is a non-trivial issue, that depends on separation of eigenvalues, among other things (see for instance [12]). If the eigenvalues of interest are exterior (that is, lying in the periphery of the spectrum), a restarted Krylov method will be able to retrieve them without problems. However, if the desired eigenvalues are in the interior of the spectrum, then it is necessary to apply an spectral transformation to the initial problem. A standard technique to compute interior eigenvalues is the shift-and-invert transformation. For the generalized eigenvalue problem (3.2) the Krylov method is then applied to the transformed problem

$$(A - \sigma B)^{-1} B x = \theta x, \tag{3.23}$$

where largest magnitude $\theta = (\lambda - \sigma)^{-1}$ correspond to eigenvalues λ closest to a given target value σ. Convergence in this case will require less iterations because the transformation also improves the separation, but the solver must implicitly handle the inverse of $A - \sigma B$ via direct linear solves.

3.3 Scientific computing software

The constant evolution of numerical software for solving eigenvalue problems has provided a great variety of software packages [59]. The different available libraries are specialized in performing serial or parallel computations, intended to solve dense and/or sparse eigenproblems in the standard (3.1) or the generalized (3.2) form, and they use a variety of direct and/or iterative methods.

We present now some of the most relevant libraries in the field, all of them publicly available in source form. As an example of libraries implementing direct methods for solving eigenvalues we can name ScaLAPACK [19], ELPA [92], EigenExa [48] and Elemental [107].

ScaLAPACK is a parallel library that implements a subset of the routines provided by LAPACK. It is based on MPI and relies on PBLAS (Parallel BLAS) for performing low-level operations. It provides parallel versions of the primary algorithms such as the QR iteration. It is developed mostly in Fortran 77 and released under a BSD license.

ELPA (Eigenvalue soLvers for Petascale Applications) is a parallel library that provides efficient and scalable direct eigensolvers for dense Hermitian matrices, addressing the standard and the generalized eigenvalue problems. It is based on ScaLAPACK's matrix layout, and replaces all parallel steps with subroutines of its own. ELPA provides parallelism by means of MPI, and its license is the LGPL.

EigenExa is a Fortran library oriented to be scalable and to provide high-performance eigenvalue solvers. It addresses the standard and generalized eigenvalue problems, using MPI for inter-node parallelism and OpenMP for intra-node threaded parallelism. The software is published with a BSD license.

Elemental is a library for direct linear algebra similar in functionality to ScaLAPACK, but unlike it, Elemental performs almost all computations using element-wise matrix distributions instead of block-wise ones. It is developed in C++ (directly used from C) with interfaces to Fortran 90, Python, and R. It is oriented to distributed-memory parallelism. Most of its code is released under a BSD License, although it also uses a variety of other free software licenses.

Within this chapter and Chapter 4 we will focus on the parallel solution of large and sparse eigenvalue problems by means of iterative methods. Libraries implementing such kind of methods are ARPACK [88], PRIMME [125], FEAST [105], Anasazi [14], and SLEPc [60].

ARPACK (ARnoldi PACKage) is a library for the computation of a few eigenpairs of a general large sparse matrix. As its name points out, it implements a variant of the Arnoldi algorithm with an implicit restart called Implicitly Restarted Arnoldi Method (IRAM). It is implemented in Fortran 77 and can solve both the standard and generalized eigenvalue problems, working with real or complex arithmetic, in single or double precision. It makes use of a reverse communication interface, that returns the control of the operations to the invoking program, indicating to it the next required operation. This feature allows it to work with any matrix storage type, but requires the user to implement matrix-vector products and linear solves. It has good efficiency, and given the maturity of the methods it is one of the most popular software libraries for computing eigenvalues. There also exists a parallel version of the library called PARPACK [93] that supports MPI. It uses a BSD license.

PRIMME (PReconditioned Iterative MultiMethod Eigensolver) is a parallel library for finding a few eigenpairs of an Hermitian matrix. The library provides a generic multimethod eigensolver, based on Davidson and Jacobi-Davidson, and supports preconditioning, as well as the computation of interior eigenvalues. It is implemented in C and provides Fortran 77, Python, MATLAB, and R interfaces. Its parallelism is based on MPI and uses a BSD license.

FEAST is a numerical parallel library for solving both the standard and the generalized eigenvalue problem, and obtaining all the eigenpairs within a given search interval. It is based on the FEAST algorithm, an innovative and stable numerical algorithm which deviates fundamentally from the traditional Krylov subspace based iterations or Jacobi–Davidson techniques. This algorithm takes its inspiration from the density-matrix representation and contour integration technique in quantum mechanics. The library is released under a BSD license and has been integrated into the Intel MKL library.

Anasazi is a subset of the Trilinos [61] library for solving large-scale eigenvalue problems. It implements several algorithms like a block oriented Krylov-Schur method, a generalized Davidson method, a Riemannian Trust-Region method, and LOBPCG method, among others. It is developed in C++, supports parallelism by means of MPI and is released under a LGPL license.

Next we describe in a more detailed way the two software libraries on top of which we have made all the developments that we present along the chapters. As this chapter well reflects, our work is focused in the field of eigenvalues and related areas. Within this field, SLEPc library is a reference in terms of numerical robustness and efficiency, with highly scalable parallel implementations of state of the field algorithms. It offers a wide range of solvers, allowing a high degree of parameter adjustment and is easily extensible. At the same time, SLEPc is developed over another key library in the linear algebra field as it is PETSc [15].

3.3.1 PETSc

The Portable, Extensible Toolkit for Scientific Computation (PETSc) is an object-oriented parallel framework for the numerical solution of partial differential equations.

PETSc uses the MPI paradigm for the coarse grain parallelization and is intended to build large-scale scientific applications and to run on distributed-memory high-performance computers. Its code is structured hierarchically in several modules, each one of them providing a single abstract interface and one or more implementations that use specific data structures. The data structures represent, for instance, vectors and matrices, and algorithmic objects (solvers) for different types of problems, including linear (and non-linear) systems of equations. Complex objects like solvers are composed of more basic objects in the hierarchy like vectors and matrices. Some of those structures are shown in Figure 3.1, where their associated subtypes

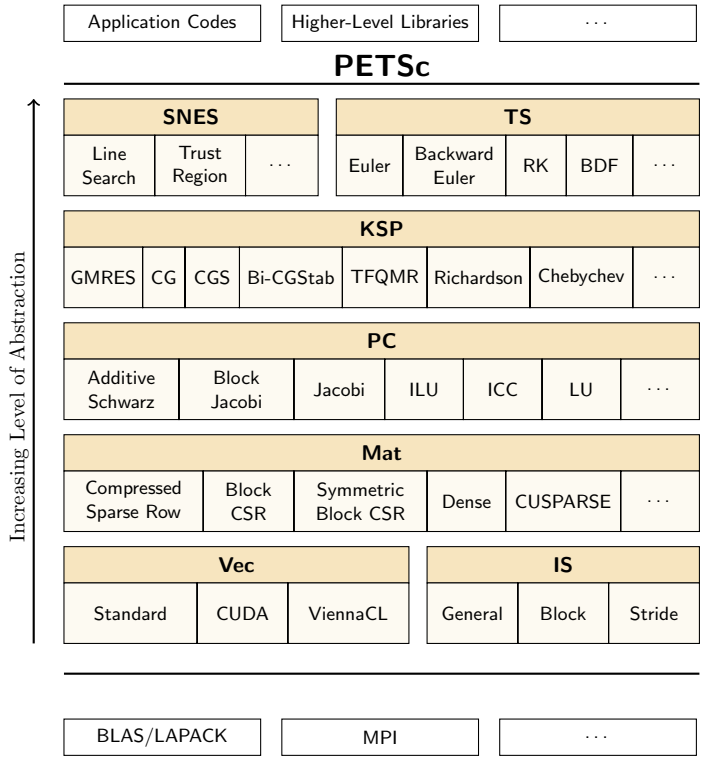

Figure 3.1. Components of PETSc.

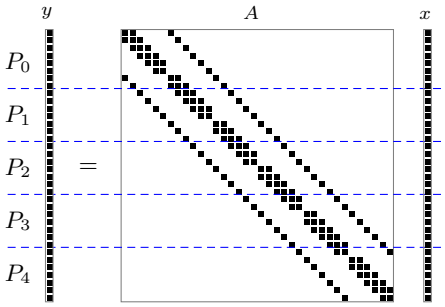

Figure 3.2. Parallel distribution of a sparse matrix and two vectors in PETSc.

also appear, corresponding with different implementations of the methods of a particular interface or with the storage format used. In general, the algorithm employed to perform a specific operation comes determined by the type of the object instantiated. Vector objects are dense, but matrices can be stored with many different representations including dense, several sparse formats, block oriented formats, and also allow the use of user-defined matrices called *shell matrices*[2], among others. In general, all those representations have a serial and a parallel subtype, that store the data on a single process or distribute it among several processes, respectively. Figure 3.2 shows the row-based distribution of the elements of a matrix among several processes.

PETSc is written in C, so it can be directly used from application codes written in C and C++, but also by means of Fortran and Python [32] bindings. The combination of C and MPI makes the library highly portable, allowing it to be used on almost any parallel machine. PETSc defines a neutral datatype called *scalar* that allows compiling the library to work with real or complex arithmetic, and specifying the precision to be used, being it single, double or quadruple. The default setup configures PETSc to work with real arithmetic in double precision.

The application code interacts with the interface of the objects provided without caring about the details of the underlying data structures and parallelism, allowing the users to ignore them and focus on the high-level problem to solve. PETSc provides calls to help in the distribution and management of parallel data, but as it employs MPI, the user can also make use of any MPI call to operate with the data.

A characteristic of PETSc is the possibility to interface with third-party libraries in order to provide additional functionality. PETSc can interface to multiple numerical libraries like BLAS, LAPACK or MUMPS [5], and to numerical environments like MATLAB. In addition, the library enables the runtime customization and/or extension of the data structures and algorithms provided.

Another interesting feature of PETSc is its built-in support for profiling the applications code. It may offer the users a summary of the different stages of the

[2]Shell matrices can handle the matrix implicitly or store the data allowing user-defined storage formats.

execution, in which performance factors like the time, the floating-point operations rate and the memory usage of the PETSc routines are reported, after being automatically logged at runtime. In the same way, the user-provided routines can also be profiled by means of this functionality.

During all the life of the library and due to its highly active development, a well-known characteristic of PETSc has been the continuous change of its own code, including its public interface, in order to enhance the solvers, and support new features and emerging architectures. Other feature of which we make use in this thesis is the support for GPU computing, officially available since version 3.9[3].

Early GPU support in PETSc [96] required the functionality of CUSP [33][4], for managing vectors on GPU. Vector operations and sparse matrix-vector products were performed through `VECCUSP`, a special vector class whose array is mirrored in the GPU, and `MATAIJCUSP`, a matrix class where data generated on the host is then copied to the device on demand. The computational effort is this way transferred to the graphics cards, transparently to the calling code. The AIJ part of the name of the matrix class indicates the format used to store the matrix in the CPU memory. This format used by PETSc is the same as the CSR format seen in Chapter 2. The layout in GPU memory does not necessarily corresponds with the one used in CPU memory, as `MATAIJCUSP` supports CSR, DIA and ELL formats on GPU.

One feature of PETSc is that besides automatically allocating memory for an object on its creation, it also allows the users to manually perform the memory allocation and passing a pointer to it when creating a new object. Due to the fact of being based on Thrust, vector objects created on GPU memory by means of CUSP do not allow to be instantiated in this way. This issue affects negatively to the usage of PETSc, limiting its flexibility. Within the course of this thesis we proceeded to create `VECCUDA`, a new vector type for the GPU, analogous to `VECCUSP`, that uses the memory management provided by the CUDA runtime and cuBLAS for performing vector operations. This new type does not have the limitations of `VECCUSP` and allows specifying the memory that the vector object will use.

Currently, the GPU support in PETSc is based on using cuBLAS and cuSPARSE[5] to perform vector operations and sparse matrix-vector products through `VECCUDA` and `MATAIJCUSPARSE`, vector and matrix classes with analogous functionality of the CUSP based classes. Matrices with type `MATAIJCUSPARSE` are stored on CPU memory using the AIJ format, and on GPU, CSR, ELL and HYB formats are supported.

The GPU model considered in PETSc uses MPI for communication between different processes, each of them having access to a single GPU. On a process, GPU memory is allocated on demand on the first usage, and the implementation includes mechanisms to guarantee the coherence of the mirrored data-structures on the host and the device. For instance, the underlying operations to the memory of the objects can have read, write, or read-write access. A flag is used to specify in which memory

[3]On previous versions GPU support was available on the development branch of PETSc.

[4]CUSP based GPU support was removed in version 3.9 of PETSc.

[5]PETSc also features OpenCL support for GPU hardware other than NVIDIA, but we do not consider this here.

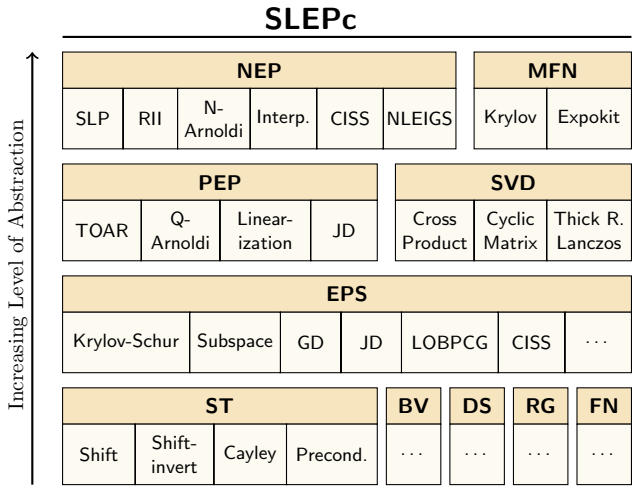

Figure 3.3. Components of SLEPc.

resides the newest data. Depending on the type of access required, the data may be copied from one memory to the other, and/or the flag may be modified[6].

Besides the transparent use of the GPU offered by PETSc objects, they also provide direct access to the allocated memory (being it on CPU or on GPU). This feature, together with the possibility of configuring PETSc to link with third party libraries, enables the use of other GPU-aware libraries such as MAGMA from user-provided routines.

3.3.2 SLEPc

The Scalable Library for Eigenvalue Problem Computations (SLEPc) follows the same object-oriented design of PETSc, and extends its functionality with methods to solve large-scale sparse eigenvalue and related problems. As PETSc, it is developed in C under a BSD license and can be used from C, C++, Fortran and Python applications.

It provides several modules to solve linear (EPS), polynomial (PEP), and non-linear (NEP) eigenvalue problems as well as other related problems like singular value decompositions (SVD), or the action of a matrix function on a vector (MFN). Figure 3.3 shows the hierarchy of objects that constitute the library. The main classes make use of other low-level classes that are described next.

`ST` provides the functionality for performing spectral transformations (changes in the variable associated with the eigenvalue). Some of the available transformations are *shift*, applying $\lambda = \theta + \sigma$ for a given σ, or *shift-and-invert* given by $\lambda = 1/\theta + \sigma$. Applying these transformations generates a new eigenvalue problem

[6]The flag can be set to "GPU not allocated", CPU, GPU or both memories.

with the same eigenvectors than the original one, and with the eigenvalues modified according to the transformation used. These transformations can be used together with eigenvalue solvers in order to improve its convergence.

Some of the modules to solve eigenvalue problems (`EPS` and `PEP`) provide an `ST` object that realizes the transformation of the initial problem given a specific spectral transformation. This encapsulation enables the eigensolvers to abstract from the spectral transformation used. For instance, when solving the standard problem with a matrix A, applying σ over the vector x is translated in $(A-\sigma I)^{-1}x$ when using the shift-and-invert transformation. By default, the transformed matrices are not explicitly built, and the inverse is not computed. Instead, the `ST` solves a linear system of equations by means of a PETSc's `KSP` object when necessary.

Parallelism within this class is given by the operations of the low-level parallel objects that it uses.

`BV` represents a vector basis and its associated operations. It provides data structures and routines to operate within sets of vectors, including functionality to perform the orthogonalization of the vectors. Most of SLEPc's solver classes contain a `BV` object to work with the subspace that they generate. There exist several subclasses depending on the type of storage used. The different subclasses are `svec`, that stores the vectors as a single PETSc's `Vec` object, `vecs`, that stores the vectors as a collection of `Vecs`, `mat`, that uses a PETSc's `Mat` object in which each column stores a vector, or `contiguous`, that stores the vectors contiguously in memory.

This class offers two levels of parallelism, task-level parallelism, as each process performs the computation of their local portion of the vectors, and thread-level parallelism, with parallel implementations of BLAS and LAPACK. It is optimized to perform its operations in an efficient way, minimizing the global communication [58] and prioritizing the use of block oriented operations with respect to vectorial ones, in the local computations.

`DS` provides the functionality to solve the projected problems of reduced dimension that most of the SLEPc solvers generate. There is a specific `DS` type for each one of the projected problems generated by the high-level classes.

The parallelism within this class is performed exclusively on the local node by means of parallel implementations of BLAS and LAPACK, as this computation is done redundantly by all the processes, given that all of them have the projected problem stored. As long as the size of the projected problem is not too large, the global performance is not affected by this step.

`RG` defines a region on the complex plane. Most of the solvers contain an `RG` object, that is used to delimit a region in which to look for eigenvalues. The different implementations define intervals, polygons and ellipses.

There is no parallelism associated with this class. The object is replicated on all the processes and its operations are performed redundantly without affecting the performance.

`FN` provides functionality to perform dense functions of matrices. The class includes some predefined functions that can be composed to obtain more complex expressions. They are used to specify a non-linear problem associated to a `NEP` object, or to indicate the matrix function of an `MFN` object. Some of the functionality of this class has been added in the context of this thesis, and more details about it are provided in Chapter 5.

The parallelism within this class is performed exclusively on the local node by means of parallel implementations of BLAS and LAPACK. The computation is done redundantly by all the processes, given that all of them have the same data.

SLEPc inherits all PETSc's functionality and characteristics, including the support for GPU computing. In the same way as PETSc does, SLEPc's solvers are data-structure neutral, meaning that the computation can be done with different sparse matrix storage formats.

It offers the users the possibility to specify many parameters such as the number of eigenpairs to compute, the convergence tolerance or the dimension of the built subspace, both programmatically and in run-time. And it is also extensible in the sense that it is possible to plug user-provided code to customize the solvers.

In this thesis we make use of the high-level modules `EPS` and `MFN`, that solve linear eigenvalue problems and the action of a matrix function on a vector, respectively. But in this chapter we focus exclusively on the solution of eigenvalue problems by means of `EPS`.

The default eigensolver in SLEPc is the Krylov-Schur method described in Section 3.2.1, which essentially consists in the Arnoldi recurrence enriched with an effective restart mechanism. The full algorithm involve the following operations:

1. Basis expansion. To obtain a new candidate vector for the Krylov subspace, a previous vector must be multiplied by A. In the generalized eigenproblem (3.2) the multiplication is by $B^{-1}A$ or, alternatively, by $(A-\sigma B)^{-1}B$ if shift-and-invert is used, and hence linear system solves are required.

2. Orthogonalization and normalization of the vectors. The new jth vector of the Krylov basis must be orthogonalized against the previous $j-1$ vectors. This can be done with the (iterated) modified or classical Gram-Schmidt procedure.

3. Solution of the projected eigenproblem. A small eigenvalue problem of size m must be solved at each restart, for matrix $H_m = V_m^* A V_m$.

4. Restart. The associated computation is $V_m Q_p$, $p < m$, where the columns of V span the Krylov subspace and Q contains the Schur vectors of H_m.

Table 3.1. List of cuBLAS routines employed in the `sveccuda` implementation of the `BV` object.

BLAS Routine	BLAS Level	Description	Number of `BV` functions using it
_scal	1	$x \leftarrow \alpha x$	2
_axpy	1	$y \leftarrow \alpha x + y$	1
_gemv	2	$y \leftarrow \alpha Ax + \beta y$	1
_gemm	3	$C \leftarrow \alpha AB + \beta C$	6

The operations with highest computational cost are the expansion of the subspace and the orthogonalization process, that in a parallel implementation also require communication between the processes.

The expansion of the subspace shown as a matrix-vector multiplication in step 5 of Algorithm 3.3, many times corresponds with the solution of a linear system of equations as it happens in the generalized case, where the operator is $B^{-1}A$ or $(A - \sigma B)^{-1}B$ in case of using the shift-and-invert spectral transformation. The sparse matrix-vector multiplication can be performed on GPU by using the appropriate `Mat` type. SLEPc's implementation of the Krylov–Schur method uses PETSc's `KSP` object to solve the linear system of equations.

The orthogonalization process is performed by SLEPc's `BV` object. One of the recent improvements made in the code of SLEPc has been the enhancement of the computational intensity of this object. As already commented, one of the implementations of `BV` (`svec`) uses a single PETSc's `Vec` object to store the vectors of the basis. SLEPc's code has been refactored to allow this `svec` implementation to operate with multiple vectors at the same time, using Level 2 and 3 routines of BLAS. Additionally, in the context of this thesis we have created the `sveccuda`[7] subtype to provide an implementation of `BV` that performs the operations on GPU. Table 3.1 shows the cuBLAS operations used by `sveccuda` and the number of internal routines that employ them. We can see how most of the internal routines can take advantage of BLAS Level 3 operations.

In SLEPc, the Krylov-Schur eigensolver includes the possibility of specifying the maximum dimension of the projected problem as a parameter (`mpd`), that enables the computation of a large number of eigenpairs in chunks, by locking eigenpairs converged at each restart.

3.4 Solving eigenproblems with GPUs

To illustrate the applicability of the SLEPc library and its use on GPU we next present our work concerned with numerical simulation in the context of molecular magnetism, where the goal is to analyze molecule-based magnetic materials with

[7]The implementation of `sveccuda` makes use of PETSc's `VECCUDA` type.

interesting properties that are important in applications such as high-density information storage. In particular, we focus on magnetic clusters, that is, molecular assemblies of a finite number of exchanged-coupled paramagnetic centers. These assemblies are midway between small molecular systems and the bulk state, being possible to model them as the former, with quantum mechanical principles, rather than with the simplifications required for the latter. This allows for deeper understanding of the magnetic exchange interaction. However, when analyzing clusters with a growing number of exchanged-coupled centers, the complexity soon becomes prohibitive due to the lack of translational symmetry within the cluster.

A flexible methodology for studying high nuclearity spin clusters is based on the use of the technique of irreducible tensor operators (ITO) [21, 121]. This approach enables the evaluation of eigenvalues and eigenvectors of the system, then deriving from them the magnetic susceptibility, the magnetization, as well as the inelastic neutron scattering spectra. This functionality is provided by the MAGPACK package [22], covering both anisotropic exchange interactions as well as the simpler isotropic case. MAGPACK is a set of serial Fortran codes, whose scope of applicability is limited to very small number of centers, due to the high computational cost associated with the creation of the Hamiltonian matrices followed by their diagonalization. The dimension of these matrices grows rapidly with the number of spin cluster basis functions.

In a previous work, E. Ramos et al. [109] reworked the MAGPACK codes in order to be able to cope with large-scale problems, with many spins. The developed codes are parallel, hence enabling the use of large supercomputers, and are based on carrying out a *partial* diagonalization of the system matrices by means of SLEPc. In this way, the computational load is shared across the various processors participating in the parallel computation, and in addition, only a modest percentage of the eigenvalues and eigenvectors is obtained, thus avoiding many superfluous calculations. Still, the computational requirements can be huge so we focus on improving the efficiency of the software as much as possible.

In particular, we work with PARISO, the parallel code for coping with isotropic spin clusters, although the use of GPUs could be easily extended to the anisotropic case. We have made two major improvements in this code. On the one hand, we have added a preprocessing step aiming at reducing the number of partial diagonalizations required during the computation. This step computes bounds for the spectrum of the submatrices in which the main Hamiltonian matrix decomposes, and estimates the number of eigenvalues to compute in each submatrix. With this strategy, the computation provides equally satisfactory results with a considerable decrease of the time of calculation, although we will discuss that the way in which the number of eigenvalues is estimated may not work in all cases. On the other hand, we have extended the parallelization paradigm in order to exploit graphics processing units. The use of GPUs represents a finer level of parallelism, to be added to the already mentioned coarse-grain parallelism, that provides an additional speedup factor that can significantly reduce the turnaround time of the computation.

3.4.1 The ParIso code

Consider a spin cluster composed of an arbitrary number of magnetic sites, N, with local spins. In order to obtain the set of spin cluster basis functions, the local spins are successively coupled,

$$\left|S_1 S_2 \left(\tilde{S}_2\right) S_3 \left(\tilde{S}_3\right) ....S_{N-1} \left(\tilde{S}_{N-1}\right) S_t\right\rangle = \left|\left(\tilde{S}\right) S_t\right\rangle \tag{3.24}$$

where $\tilde{S}_i$ refers to the intermediate spin values $S_1 + S_2 = \tilde{S}_2$, $\tilde{S}_2 + S_3 = \tilde{S}_3$, etc., $(\tilde{S})$ is the full set of $\tilde{S}_i$ ($N-1$ intermediate spin states) and S_t is the total spin [21]. The system matrix can be evaluated by applying the Hamiltonian to the created basis set, by means of the irreducible tensor operators technique. The advantage of this methodology is that it allows us to completely take into consideration all kinds of magnetic exchange interactions between the metal ions comprised in clusters of arbitrary size. This is done by expressing the contributions to the spin Hamiltonian (expressed in terms of the conventional spin operators) as a function of the generalized Hamiltonian (written in terms of ITO's).

The ParIso code computes the quantities mentioned above for isotropic systems, that is, it generates the spin functions of the system, calculates the energy matrix and obtains its eigenvalues and eigenvectors, all this in parallel. Isotropic systems are a special case where only the isotropic and biquadratic exchange terms are present in the spin Hamiltonian. These terms have the property of not mixing functions with different quantum number S and not breaking the degeneracy of levels with the same S and different M. This decouples the energy matrix into several submatrices, one per each different S quantum number. Taking into account the ITO technique it is possible to eliminate this M quantum number and reduce the size of each S submatrix by a factor of $2S+1$. No Zeeman terms are present in this case, during the diagonalization, but would be included after it.

Thus, in the case of isotropic systems, the energy matrix can be written as a block diagonal matrix,

$$A = \begin{bmatrix} A_1 & & & \\ & A_2 & & \\ & & \ddots & \\ & & & A_b \end{bmatrix}, \tag{3.25}$$

where each of the b blocks is a symmetric sparse matrix of different dimension. Finding the leftmost eigenvalues of A amounts to computing the leftmost eigenvalues of each of the blocks, A_i. Thus, the structure of the program ParIso is geared to this block structure, where one partial diagonalization is carried out per block.

The main steps of the computation are the following:

1. Setup of data containing the information of the cluster.
2. For each diagonal block, $i = 1, \ldots, b$, do:
   (2.1) Generation of starting spin functions.

(2.2) Evaluation of energy submatrix, A_i. All nonzero elements are computed and assembled into the matrix.

(2.3) Partial diagonalization of A_i. Given the eigenvalue relation $A_i x = \lambda x$, a subset of the spectrum is computed, corresponding to the leftmost eigenvalues.

3. Generation of final results.

3.4.2 Optimization of spin eigenanalysis

In the PARISO code, the individual blocks of matrix A in (3.25) are treated separately. The dimensions of these blocks vary widely, ranging from 1 to a hundred thousand or even more, and the percentage of nonzero elements of each (large) block is about 1–2%. The generation of the matrix is made submatrix by submatrix (each submatrix is a spin energy). Not all submatrices need to be in memory simultaneously, since after generating one submatrix it is possible to compute its partial diagonalization and then the matrix is no longer needed and can be destroyed. For all these reasons, it is possible to calculate much larger systems in the isotropic case than in the anisotropic one.

The optimization in PARISO that we introduce in this section tries to avoid the computation associated with those blocks that are not going to contribute significantly to the aggregated result. This allows a drastic reduction of the time necessary for the overall execution. The rationale is that Lanczos methods can provide robust bounds for the spectrum of each of the submatrices with a relatively small cost. Based on the location of the first and last eigenvalues of each block, we can discard some of the blocks if they lie outside the range of interest specified by the user.

The resulting algorithm performs the following steps:

1. In a first pass, the program traverses all the submatrices and, for each of them, it calculates the first and last eigenvalue, filling a table with the information obtained for all blocks.

2. From the table of minimum and maximum eigenvalues, the program determines a point of energetic cut depending on the value of population to be considered (provided as a user input parameter).

3. With the cut point and the dimension of each submatrix, the code determines how many eigenvalues are required in each of them. A number of zero implies that the corresponding submatrix can be discarded, since its energetic levels are outside the requested range.

4. Finally, a second pass computes the wanted eigenvalues, traversing only the submatrices that are going to provide useful information.

This process is illustrated in Figure 3.4 with an example. The horizontal lines represent the span of the spectrum of each of the matrix blocks (18 in this case). The dimensions of the blocks range from 1 (the last one) to 3150. The vertical line

Table 3.2. Systems used.

System	Size	Largest submatrix size
$7Mn^{2+}$	24017	3150
$8Mn^{2+}$	135954	16576
$9Mn^{2+}$	767394	88900
$9Mn^{2+} + 1Cu^{2+}$	1534788	177100

represents the energetic cut to be considered. In the plot we can easily see that this energetic cut leaves out 8 of the submatrices, which are the largest ones in this particular example. The vertical marks on the horizontal lines to the left of the energetic cut represent those eigenvalues that must actually be computed. Note that the plot does not show real eigenvalues, but an estimation based on the size of the submatrix and assuming a roughly uniform distribution of eigenvalues. This approximation is only valid for those systems with one spin state overstabilized from the rest, e.g. completely ferromagnetic systems with very high magnetic coupling which stabilize the high spin ground state or completely antiferromagnetic clusters with a very stabilized intermediate spin state. In this work, for validation we use the simple systems with this property shown in Table 3.2, the first one with ferromagnetic and anti-ferromagnetic interactions, and the rest with only ferromagnetic interactions.

In our code it is also possible to specify an upper bound (`maxev`) for the number of eigenvalues to compute in any submatrix. This is useful for large problems where a given threshold would imply computing too many eigenvalues. As an example, in the problem of Figure 3.4 the maximum number of requested eigenvalues is 33, and setting `maxev`=25, for instance, would imply in practice shifting the vertical line a certain amount to the left.

The above procedure avoids lots of unnecessary computations, with the corresponding gain in the overall simulation time. The only drawback is that the submatrices that participate in the second pass must be regenerated since it is not possible to keep all blocks in memory simultaneously. Still, the new methodology is much faster than the former one, as will be shown below.

Apart from modifying the algorithm, we have also optimized the memory usage by (1) adjusting the dimension of the various arrays to fit the maximum submatrix size, and (2) converting real variables to integer ones whenever possible. These two actions have allowed a significant reduction of the memory footprint per MPI process, up to one third of the previous values. All in all, these improvements enable to cope with larger, more challenging problems.

Code validation

We have carried out a number of numerical experiments in order to validate the correctness of the parallel code, and to evaluate its performance. The computer

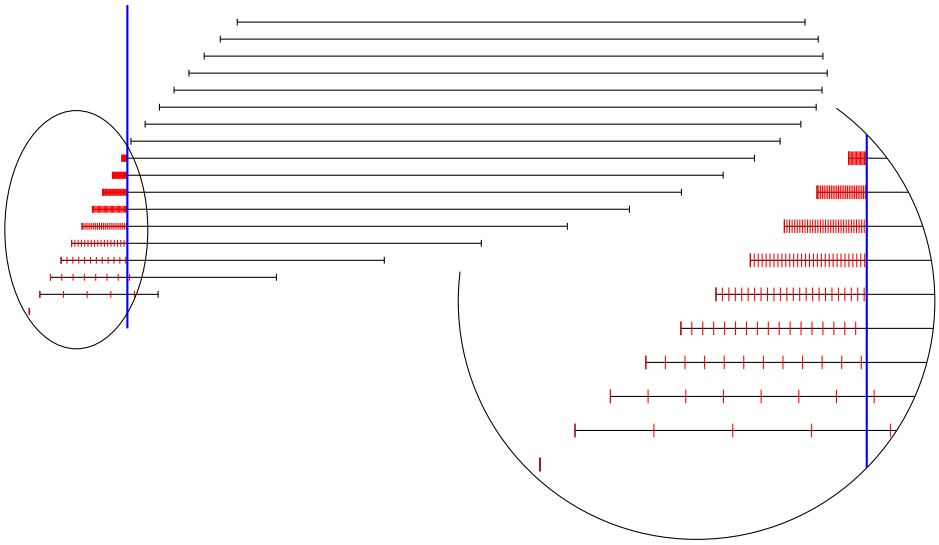

Figure 3.4. Example of energetic cut for a test problem with 18 submatrices. Only eigenvalues located to the left of the vertical line need to be computed. The number of eigenvalues to compute depends on the dimension of the submatrix. A zoom of the region of interest is shown on the right.

system used for the computational experiments in this section is Tirant[8], an IBM cluster consisting of 512 JS20 blade computing nodes, each of them with two 64-bit PowerPC 970FX processors running at 2.2 GHz, interconnected with a low latency Myrinet network.

For the validation of the code, we have used a small system ($7Mn^{2+}$) as a test case whose submatrices have the following dimensions: 1050, 1974, 2666, 3060, 3150, 2975, 2604, 2121, 1610, 1140, 750, 455, 252, 126, 56, 21, 6, and 1. This is the system used in Figure 3.4 to illustrate the algorithm, and its real data can be seen in Figure 3.5. The largest block is of size 3150 and the susceptibility was computed for a range of temperatures up to 300K. In this case, the cut value varies as a function of the population (`pobla`, a user input parameter), as shown in Table 3.3, and we have not considered setting the `maxev` parameter because the number of required eigenvalues is small. The vertical line depicted in Figure 3.5 corresponds to the energetic cut of `pobla`$=10^{-2}$. The total number of eigenvalues per spin of this system and the amount of them selected to compute after the energetic cut has been determined are shown on the left side of Table 3.4.

The computation time with one processor is also shown in Table 3.3. As a reference, the total time required in the case of computing all eigenvalues of all submatrices is 1917 seconds (this will be referred to as the *full computation*).

Figure 3.6 shows the susceptibility results (product χT against the temperature)

[8]Description of Tirant 2, decommissioned in 2018.

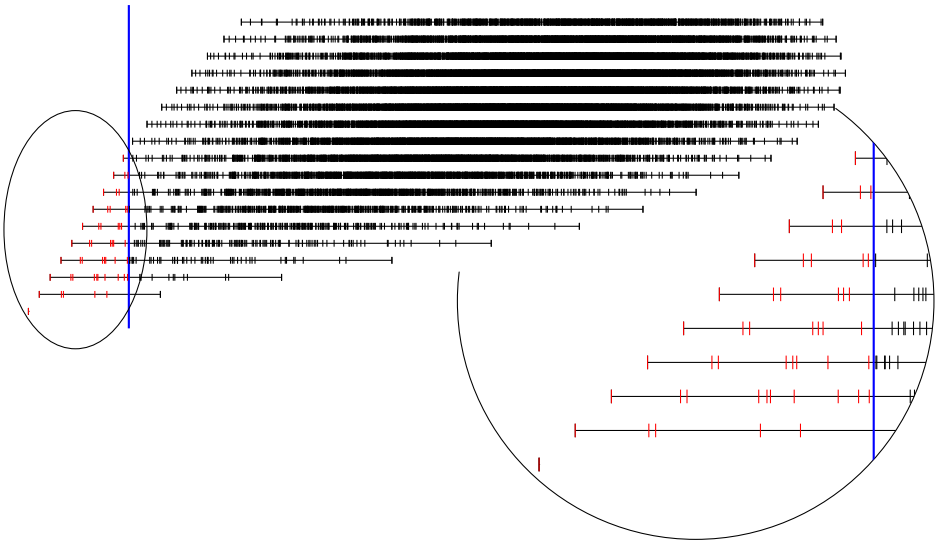

Figure 3.5. Full representation of the $7Mn^{2+}$ system, showing the energetic cut and the computed eigenvalues.

for the different values of the population, compared to the case of the full computation. For low temperatures, all lines match, while for higher temperatures the graphs diverge from the full computation, being more accurate for smaller values of `pobla`, as expected.

Performance analysis

In order to assess the performance of the code, both serially and in parallel, we have used a larger system ($8Mn^{2+}$), whose submatrices have dimensions: 2666, 7700, 11900, 14875, 16429, 16576, 15520, 13600, 11200, 8680, 6328, 4333, 2779, 1660, 916, 462, 210, 84, 28, 7, and 1. The largest block is of order 16576 and the temperature range of the simulation is 300K. In this case, the number of eigenvalues that are

Table 3.3. Value of energetic cut for different values of the population parameter in the $7Mn^{2+}$ system, and the corresponding computation time (in seconds) with one processor.

`pobla`	Energetic cut (cm^{-1})	Computation time (s)
10^{-1}	480.11	120
10^{-2}	960.23	133
10^{-3}	1440.35	217
10^{-4}	1920.47	313

Table 3.4. Total number of eigenvalues (n) for each of the spins and the selected values to be computed (k), for the systems $7Mn^{2+}$ (left) and $8Mn^{2+}$ (right) using `maxev`=1000 and `pobla`=10^{-2}.

	Eigenvalues	
Spin	n	k
0	1050	0
1	1974	0
2	2666	0
3	3060	0
4	3150	0
5	2975	0
6	2604	0
7	2121	0
8	1610	14
9	1140	28
10	750	33
11	455	30
12	252	24
13	126	18
14	56	12
15	21	8
16	6	5
17	1	1

	Eigenvalues	
Spin	n	k
0	2666	125
1	7700	372
2	11900	608
3	14875	807
4	16429	1000
5	16576	1000
6	15520	981
7	13600	896
8	11200	766
9	8680	614
10	6328	462
11	4333	86
12	2779	0
13	1660	0
14	916	0
15	462	0
16	210	0
17	84	0
18	28	0
19	7	0
20	1	0

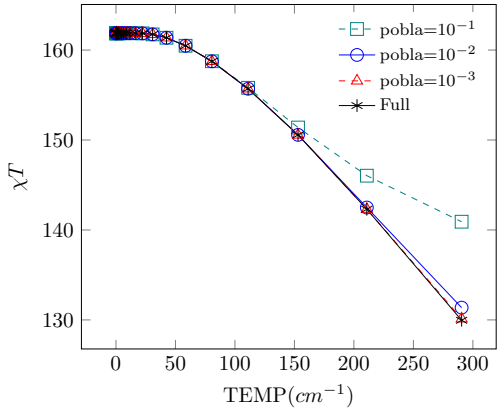

Figure 3.6. Susceptibility for different values of the population in the $7Mn^{2+}$ system.

Table 3.5. Value of energetic cut for different values of the `maxev` parameter in the $8Mn^{2+}$ system.

`maxev`	Energetic cut (K)
1000	1312.39
3000	3936.93
5000	6561.56

needed for a given energetic cut may be quite large, so we are interested in studying the influence of the `maxev` parameter. We have set this value to 1000, 3000 and 5000 eigenvalues, and it is the parameter that determines the energetic cut, see Table 3.5. The total and computed eigenvalues for a value of `maxev`=1000 can be seen on the right side of Table 3.4. If we set different values of `pobla` (10^{-1}, 10^{-2}, and 10^{-3} as in the previous example), we will hardly appreciate any noticeable differences in the results.

The sequential execution time corresponding to the full computation is 403000 seconds. Table 3.6 shows execution times in parallel for the three values of `maxev`. Even computing as many as 5000 eigenvalues, the sequential time has been reduced to one fourth of the original one. And with parallel computing we reduce the times even further, with a speedup factor of around 10 with 16 processors.

In order to assess the accuracy of the computed susceptibility, in Figure 3.7 we compare the plot obtained with the full computation against the results for the three values of `maxev` used. As expected, the larger the value of `maxev`, the closer to the full computation, but any of the values used provide a quite accurate susceptibility curve. Even for `maxev`=1000 the susceptibility curve only diverges at the end of the range used, and it would be necessary to double the temperature to appreciate the divergence. The plots correspond to a value of `pobla`=10^{-1}, but as mentioned

Table 3.6. Parallel execution time (in seconds) for the $8Mn^{2+}$ system with different values of `maxev` and increasing number of MPI processes.

	Processes				
`maxev`	1	2	4	8	16
1000	27351	17020	9543	5167	2810
3000	80172	51217	26932	14476	7926
5000	106384	62590	32909	17655	9690

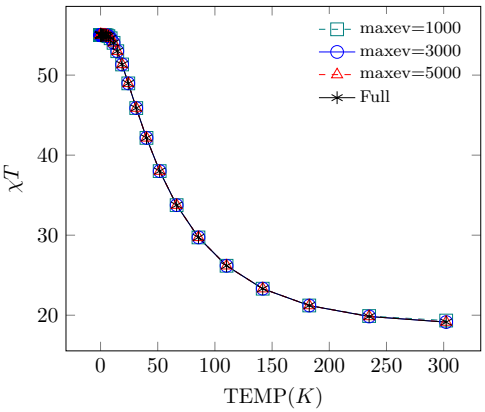

Figure 3.7. Susceptibility for different values of the `maxev` parameter in the $8Mn^{2+}$ system.

before, the graphs are essentially the same for 10^{-2} and 10^{-3}.

3.4.3 Acceleration with graphics processors

We now present a GPU-enabled implementation of PARISO that can operate also with multiple GPUs (one per MPI process). We remark that these new developments are independent of the optimization discussed in Section 3.4.2, and could also be integrated in the original PARISO code. PARISO can benefit from GPU computation mainly in two ways: when computing matrix coefficients and when solving the eigenproblems with SLEPc. The former is very appropriate for GPU computing, since matrix coefficients can be evaluated independently from each other. Regarding the latter, expected gains are modest since it corresponds to a sparse linear algebra computation.

Implementing sparse linear algebra operations on GPUs efficiently is difficult, and still a topic of active research. Usually, the most relevant operation is the sparse matrix-vector multiplication, that can be approached in different ways [89, 110]. In our case, we tried all different sparse storage formats available in PETSc with CUSP

Table 3.7. Different versions of PARISO with GPU support.

Version	Matrix created on	Eigencomputation on
CPU-`sbaij`	CPU	CPU
CPU-`aijcusp`	CPU	GPU
GPU-`sbaij`	GPU	CPU
GPU-`aijcusp`	GPU	GPU

and cuSPARSE, and the difference among them is not noticeable in our application, since the percentage of time devoted to this operation is just about 3–4% of the total computation.

Details of GPU implementation

Even though it is possible to use CUDA from Fortran with a non-free compiler, or by means of calling CUDA-C wrapper functions, in order to make use of the GPU computing power, we have ported the original Fortran code to C, as it enables a simpler development. The code uses CUDA in two ways, one by migrating parts of the application to run on the GPU as CUDA kernels, and another one by means of SLEPc, as it allows us to do the computation exclusively on the CPU or with the help of the GPU. We call 'GPU version' the one that uses CUDA kernels for the generation of the matrix (independently of the storage type used), even when the original CPU version can be instructed to use the GPU by means of the `aijcusp` matrix storage type.

With both versions and the different storage types it is possible to run the software in four main ways: CPU and GPU, each one with a matrix storage type that replicates the data into the GPU (`aijcusp`) or not (`sbaij`). Table 3.7 summarizes the four combinations. Note that the user can select one of them at run time by means of command-line arguments.

One of the benefits of working with symmetric matrices is that the storage can make use of that symmetry to halve the memory usage, as it is done in the code that runs exclusively on the CPU by means of the `sbaij` matrix storage type. In the case of the GPU, the type of matrix used by PETSc to store the data (`aijcusp`) does not take into account the symmetry and needs to allocate and fill both triangular parts (upper and lower) of the matrix.

The resulting code can be compiled to work with single and double precision arithmetic, but due to several values exceeding by far the single precision limits, some of the variables have been explicitly declared as double even when working with single precision. This mixed-precision approach allows us to reduce the memory requirements by sacrificing some accuracy in the results. There is also a reduction of computation time, but as will be shown later, the benefits of the single precision arithmetic in terms of performance are quite limited.

The migration has been done by selecting the most computationally demanding functions and moving them to run on the GPU. The selected functions are those re-

lated with the generation of the submatrices, and one of the functions that compute the final thermodynamic results. The cost of the generation of the magnetic susceptibility is negligible, but the time needed to compute the magnetization depends on the number of samples of the magnetic field intensity used. Even though this step is not very computationally demanding, as it is done exclusively by a single process, its relative time within the whole computation increases when increasing the number of processes and the total computation time is reduced. That is why porting it to the GPU was highly recommended.

In order to avoid heavy performance issues during the sparse matrices assembly, it is necessary that every process preallocates the memory corresponding to the part of the matrix that has been assigned to it [15]. For the preallocation to be done, it is necessary to specify how many diagonal and off-diagonal nonzero elements per row the matrix has (here, diagonal elements refer to matrix entries located in column indices matching the range of rows assigned to the current process, see shadowed blocks in Figure 3.8). Once the preallocation has been done, the process continues by setting the values of the elements and finally assembling the matrix. The matrix is ready to use once the assembly is finished.

Two functions are in charge of the generation and, in the GPU version, both compute all the elements of the matrix (not only the upper half), so we double the work to be done with respect to the previous CPU implementation. The first one counts the number of nonzero elements on the diagonal blocks and outside them, to do the preallocation, and the second one stores the values of these nonzero elements in memory. Both functions do almost the same work, but the amount of memory used by them differ significantly, as the counting of the elements needs only a small bi-dimensional array of integers to store the sum (with one row for each one of the matrix rows, and two columns, one for the nonzero diagonal elements and other for the nonzero off-diagonal ones), and the function that fills the matrix in with its values needs two bi-dimensional arrays of the full size of the matrix, one of floating point numbers and another one of integers where it records the column indices of the nonzero elements. The accounting of the elements is only done on the first pass, and stored to avoid repeating the computation on the second pass.

Each one of these two generation functions has been split in two, the main part runs as a CUDA kernel making use of a series of auxiliary functions that run as `__device__` functions, and the remaining work is done as a host wrapper function that allocates the memory on both CPU and GPU, calls the kernel with the appropriate arguments and copies the results to host memory. The function that computes the final results follows the same scheme of wrapper and kernel split, but it makes use of two kernels that need to be called serially.

The host memory used by the wrapper function is allocated with the `cudaHostAlloc` instruction to allocate page-locked memory. The copy of the values of the matrix and their accounting from the GPU is done with asynchronous instructions, but as the data is used immediately after it, the transfer can be considered synchronous with no concurrency of these data transfers and arithmetic operations.

As we saw in Chapter 2 (Section 2.4.3), inside the GPU we can find several types of memory with different sizes, latency accesses, scopes and lifetimes, that we

want to take advantage of. Within the generation functions, application lifetime data is copied once at the beginning of the program to `__constant__` memory, and block dependent data is copied to `__global__` memory in advance to the submatrix computation (of which a small array is accessed through texture cache).

The amount of memory in the device, limits the registers a thread can make use of, and the functions that need a large amount of registers limit the number of concurrent threads running on the device. The matrix generation functions result in computation bound kernels due to the high register use. Their launch cannot fully populate the symmetric multiprocessors of the GPU, so they are infra-utilized and the performance obtained is far from optimal.

Having said that, the computation of each different matrix element has no dependencies with the others, and this allows us to populate the GPU with any possible distribution of the grid and block dimensions and size. The launch of the kernels takes into account two different things, on the one hand, the size of the CUDA grid and blocks (number of blocks and number of threads per block respectively) per dimension, and on the other, the tile size (amount of work, measured in number of matrix elements, that a single thread has to compute) per dimension.

The work distribution scheme within the GPU device is the same for all the kernels. For the generation functions, for each of the dimensions of the matrix, two constants are defined, `BLOCK_SIZE` and `TILE_SIZE`, that are used to obtain the kernel call arguments. The calculus of these arguments begins by setting the number of blocks to one, and the number of threads per block to `BLOCK_SIZE`. Next, it is checked if the number of rows (or columns) is greater than the `BLOCK_SIZE` multiplied by the `TILE_SIZE`. If it is greater, the grid dimension (number of blocks) is increased to

```
dimGrid->x = (rows + ((BLOCK_SIZE_X * TILE_SIZE_X) - 1)) /
                                        (BLOCK_SIZE_X * TILE_SIZE_X),
```

if not, the block size is decreased to

```
dimBlock->x = (rows + (TILE_SIZE_X - 1)) / TILE_SIZE_X.
```

In the latter case, when the block size is reduced, it ends up not necessarily being a multiple of the warp size, and that means that we are working with a very small matrix, so the performance obtained from the use of the GPU is not going to be good.

Once the grid dimension is set, it is necessary to check that it does not exceed the limits of the device, and in that case, reduce its size to the maximum allowed value, and establish a counter in order to do several calls to the kernel. This is necessary in the case of very large matrices.

The current scheme creates blocks of one thread for the rows axis, and 64 threads for the columns axis, with tile sizes of one. Other work distributions have been tested with no better performance obtained. An analogous procedure is used for the kernels that compute the magnetization.

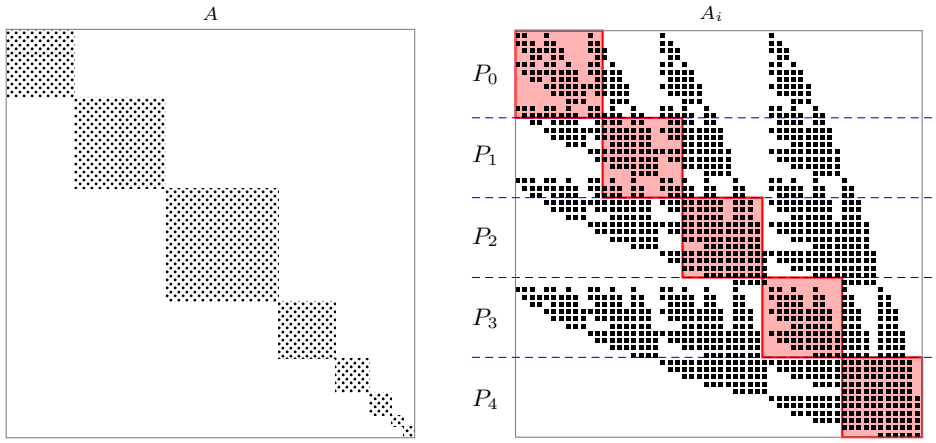

Figure 3.8. Example of an isotropic system expressed as a block diagonal matrix (left) and the partitioning of one of their symmetric sparse blocks between five MPI processes P_i (right).

Hybrid MPI-GPU approach for multi-GPU support

As mentioned in Section 3.3.1, the partitioning of the work between the different MPI processes done by PETSc uses a block-row distribution. Figure 3.8 shows how each one of the sparse blocks that form the block diagonal matrix are partitioned. The horizontal dashed lines delimit the P_i MPI process partition and the shadowed squares show the diagonal block of each process, formed by the columns whose indices correspond to the rows assigned to it.

The execution of the kernels is independent of MPI. Since the MPI version of the code evenly distributes the problem matrices across the processes, several CUDA devices can be used to individually accelerate the execution at each process, provided that the code is run on a cluster with GPUs available in all the nodes. If one node has more than one GPU, the processes select the CUDA device to be used based on their MPI rank, in the same way as PETSc does. The card identifier to be used is the reminder of the rank divided by the number of cards on the node. This simple formula allows to balance the number of processes per GPU, and for it to work, we present two options that can be used:

1. the rank used in the computation comes from a communicator that includes only the processes on the same node, or,

2. the rank used in the computation comes from the `MPI_COMM_WORLD` communicator. In this case, the processes on each node must be created consecutively.

This way, each process will use a different CUDA device (when available). To maximize the performance and to avoid overloading the devices, the MPI launch should take into account the problem size, the number of available nodes, and the number of devices per node.

Performance evaluation

In this section we analyze the performance obtained with the GPU version of the software and compare it with the CPU-only version. For this purpose, several runs have been done in the cluster Minotauro[9], where each node has two Intel Xeon E5649 processors at 2.53 GHz, 24 GB of main memory, and two NVIDIA Tesla M2090 GPU with 512 cores at 1.3 MHz and 6 GB of GDDR5. The nodes are interconnected with a low latency Infiniband network, and their operating system is RHEL 6.0 with GCC 4.6.1, MKL 11.1 and CUDA 7.0.

The value of the `pobla` and `maxev` parameters used in these runs has been set to 10^{-2} and 1000, respectively, and SLEPc's parameter `mpd` has been set to 50.

Three different cases have been used to evaluate the performance. We have started using the $8Mn^{2+}$ system of Section 3.4.2 to have a connection with the executions in the cluster Tirant. The matrix in this case has a size of 135954 and its largest block is of size 16576. The other two systems that we have used to complete the evaluation are $9Mn^{2+}$ and $9Mn^{2+} + 1Cu^{2+}$ with a size of 767394 and 1534788, and with their largest blocks being of size 88900 and 177100, respectively.

The $8Mn^{2+}$ system has been run with the CPU and with the GPU version in single and double precision arithmetic. At the same time, the GPU version has been run with two different matrix storage types: `sbaij` and `aijcusp`. The two largest systems have been run exclusively with the GPU version, with the `aijcusp` and `sbaij` matrix storage types, and with single and double precision arithmetic.

Figure 3.9 shows the execution times obtained with the $8Mn^{2+}$ system. In the figure we can see how the normal performance of the CPU-only version is improved by the two GPU runs. We can see that for a single process, both GPU runs reduce the time more than one order of magnitude with respect to CPU. For the multi-process runs, the GPU-`sbaij` run shows a similar speedup to the one obtained by the CPU version, maintaining the curves in parallel (up to 32 processes), while the `aijcusp` run does not reduce the time with the same rate or even increases it while increasing the number of processes. This kind of behaviour, where the performance decreases when the number of processes is increased, is due to the small size of the submatrices. The GPU-`sbaij` run shows clearly the great benefit provided by the kernels that generate the matrix with respect to its CPU counterpart, as both versions use a symmetric aware storage that reduces the memory and arithmetic operations (on the CPU). In the same curve it is possible to appreciate that the speedup is reduced when increasing the number of processes due to the higher cost of the inter-process communications and the infra-utilization of the GPU devices due to the reduction of the workload. The GPU-`aijcusp` run is even more affected by the problem size, as the overall performance decreases drastically compared with the `sbaij` versions due to two main drawbacks, it uses twice the memory needed by `sbaij` and it has to synchronize the data between the GPU and CPU besides between the processes, during the computation. As the SLEPc GPU implementation uses vector operations, the performance is directly dependent of the size of the problem. We can appreciate such dependency if we compare the results of the different test

[9]Description of Minotauro in 2015.

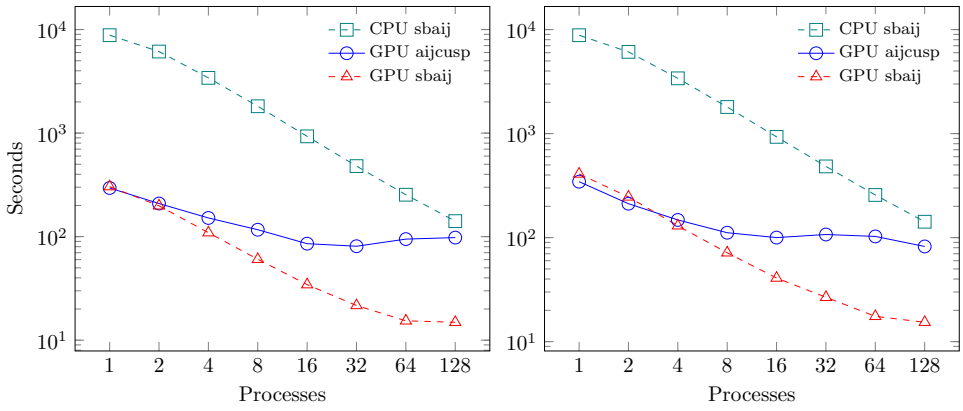

Figure 3.9. Total problem solve time for the $8Mn^{2+}$ system with single (left) and double precision arithmetic (right).

cases. While in Figure 3.9 we see that GPU-`aijcusp` starts with the smallest run times and quickly drops the performance, as we increase the size of the problem (Figures 3.10 and 3.11), we can see how it behaves much better and only reduces the performance with 128 processes. It is also noticeable how also the `sbaij` run improves with the increment of the size as its performance does not decrease so quickly when the number of processes is increased.

The figures show that the more the GPU devices are used, the better is the performance obtained. Both GPU runs clearly improve the CPU-`sbaij` run times, maintaining a similar performance and a good scalability and being able to solve a large system of 1.5 million elements in less than 230 seconds, with 128 GPUs and using double precision arithmetic. For such improvement to be possible it is necessary to maximize the use of the GPUs, as we can see how the performance decreases in all GPU runs when the processes do not have enough workload. The work done by the GPU needs to be enough to fully populate and use the device, as the performance depends directly on the usage of the device. The behaviour is the same for the kernel executions and for the eigenvalue computation.

3.5 Conclusions

In this chapter we have introduced the eigenvalue problem and the SLEPc library as the context within our work is done. SLEPc is able to exploit GPU accelerators in some parts of its computations by relying in PETSc capabilities, and enhance the performance on others by replacing BLAS routines of Level 1 with routines of Levels 2 and 3.

Within this chapter we have presented two main optimizations to ParIso, a program for simulation of isotropic molecular clusters with the ITO computational technique.

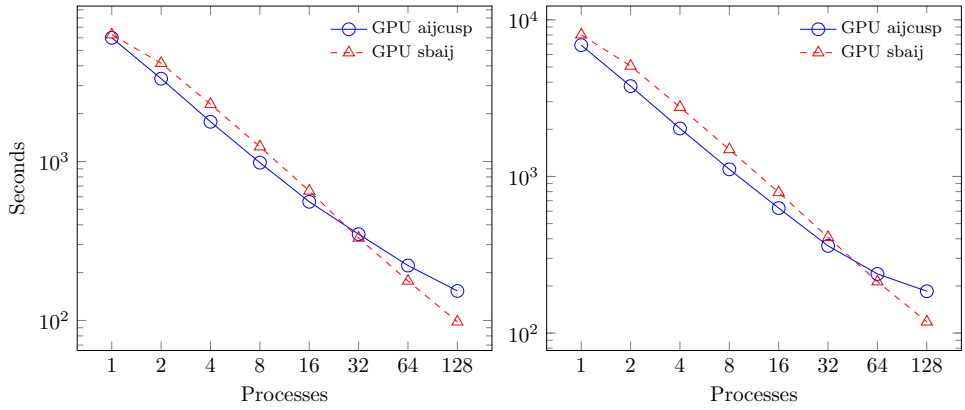

Figure 3.10. Total problem solve time for the $9Mn^{2+}$ system with single (left) and double precision arithmetic (right).

The first optimization consists in avoiding the computation that does not contribute significantly to the aggregate results. In our tests, this has allowed a drastic reduction of the execution time without losing validity in the results. However, our heuristics for determining the energetic cut (as well as to estimate the number of eigenvalues required in each block of the Hamiltonian matrix) assume a uniform distribution of eigenvalues. This assumption is valid only for systems with specific properties, as discussed in Section 3.4.2, so the method may not be appropriate for general systems.

The second major optimization is the implementation of a GPU-enabled version that can perform either the computation of matrix coefficients or the computation of the partial diagonalizations, or both, on high-performance graphics processing units. The performance gain is very significant, especially associated to the computation of the matrices. Regarding the efficiency of the diagonalization on the GPU, this step relies on the efficiency achieved by the SLEPc library. In the executions presented in this chapter, the `BV` object uses exclusively Level 1 operations.

With the multi-GPU version we are able to reduce the computation one order of magnitude with respect to the parallel MPI version running on CPUs. This will make it possible to solve much larger problems, those with real scientific interest, that would otherwise be impossible to address due to memory limitations or lack of computational power.

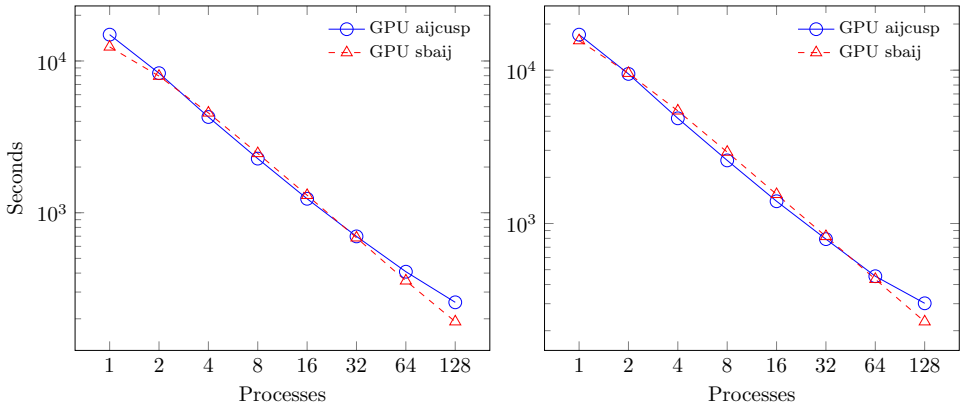

Figure 3.11. Total problem solve time for the $9Mn^{2+} + 1Cu^{2+}$ system with single (left) and double precision arithmetic (right).

Chapter 4

Block-tridiagonal eigenvalue problems

Let it be

A particular case of the eigenvalue problem that frequently appears in scientific and engineering problems involves a structured matrix with block-tridiagonal shape. Given a two-dimensional domain Ω partitioned in subdomains Ω_i like the one represented in Figure 4.1, where the points interconnected between two domains are only coupled to their closest neighbours in the mesh, then a specific domain Ω_i is only coupled to its predecessor Ω_{i-1} and to its successor Ω_{i+1}, and the corresponding matrix takes the form of a block-tridiagonal matrix.

A block-tridiagonal matrix is a square matrix with a tridiagonal pattern, in which the scalar elements are replaced with square submatrices (blocks). A block-

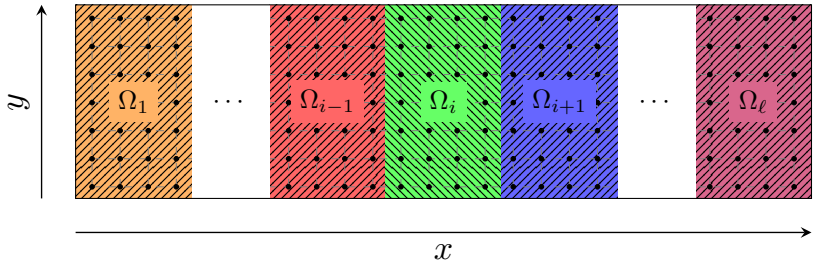

Figure 4.1. Schema of a 2D domain partitioned in subdomains Ω_i.

tridiagonal matrix

$$T = \begin{bmatrix} B_1 & C_1 & & & & & \\ A_2 & B_2 & C_2 & & & & \\ & A_3 & B_3 & C_3 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & A_{\ell-1} & B_{\ell-1} & C_{\ell-1} & \\ & & & & A_\ell & B_\ell \end{bmatrix}, \tag{4.1}$$

of order n, has ℓ blocks of size k that can be dense or sparse.

In this chapter we present the work done in order to extend SLEPc with parallel eigensolvers that specialize for this particular case, running on one or several GPUs. The solvers can also be used for dense banded matrices generated by applications like [1].

Algorithm 4.1. Arnoldi algorithm

```
1 for i = 1, 2, ... do
2 |   for j = 1, 2, ..., m do
3 |   |   Expansion of the basis of the subspace
4 |   |   Orthogonalization and normalization of the vectors
5 |   end
6 |   Solution of the projected eigenproblem of size m
7 |   Restart
8 end
```

In our parallel developments we assume that the blocks are dense and stored distributed by block-rows among the existing processes, but without splitting any of them across multiple processes. As already seen in Chapter 3, the Arnoldi algorithm with the Krylov–Schur restart can be summarized as shown in Algorithm 4.1. We assume that the size of the projected problem, m is very small compared to the dimension of the matrices, n. From the four main computational tasks of the algorithm, the dominant computational cost is the one associated with step 3. Steps 4 and 7, are currently done on the GPU as explained in Chapter 3 (Section 3.3.2), and the cost of step 6 is negligible compared to the rest and it is not worth performing it on the GPU.

For this, we next discuss in detail the operations related to the basis expansion, that vary depending on the requested eigenvalues. We present different strategies to parallelize the involved operations on the GPU and perform a study to evaluate their performance, comparing the results with a CPU implementation. Finally, we present a use case to compare the performance of our developments with a third party library.

4.1 Matrix-vector product

The most simple eigenvalue computation is the one aiming to obtain eigenvalues from the exterior of the spectrum. The basis expansion in this case is done by performing a matrix-vector multiplication. Given that our matrix is block-tridiagonal, the operations performed by the matrix-vector multiplication can be reduced to the ones involving the nonzero blocks. The matrix-vector product

$$y = Tv, \tag{4.2}$$

can be computed by blocks as

$$y_i = [A_i\ B_i\ C_i] \begin{bmatrix} v_{i-1} \\ v_i \\ v_{i+1} \end{bmatrix}, \qquad i = 2, \ldots, \ell - 1, \tag{4.3}$$

with analog expressions for the first and last block-row.

4.2 Shift and invert: Linear systems

If the desired eigenvalues are from the interior of the spectrum, a spectral transformation needs to be applied to the initial problem. One of the spectral transformations supported in SLEPc is the shift-and-invert transformation of (3.23), that in the case of the standard eigenvalue problem it reads

$$(T - \sigma I)^{-1} z = \theta z. \tag{4.4}$$

In a practical implementation the inverse of $(T-\sigma I)$ is never computed explicitly, instead, the basis expansion is performed by solving a linear system of equations

$$T_\sigma x = b, \tag{4.5}$$

where $T_\sigma = (T - \sigma I)$. Solving this linear system of equations could be approached with (Sca)LAPACK's general band factorization subroutine (`_gbsv`), but this is not available on the GPU. Hence, within this section, we center our attention on algorithms that operate specifically on the block-tridiagonal structure and are feasible to implement with CUDA. The (scalar) tridiagonal case was analyzed in [146], where the authors compare GPU implementations of several algorithms.

4.2.1 Thomas algorithm

Gaussian elimination on a tridiagonal system

$$\begin{bmatrix} d_1 & u_1 & & & \\ l_2 & d_2 & u_2 & & \\ & \ddots & \ddots & \ddots & \\ & & l_{n-1} & d_{n-1} & u_{n-1} \\ & & & l_n & d_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix} \tag{4.6}$$

can be specialized to use $O(n)$ operations instead of $O(n^3)$ and is sometimes referred to as the Thomas algorithm. In the forward elimination phase, the algorithm computes intermediate coefficients for the first row with $e_1 = u_1/d_1$ and $f_1 = b_1/d_1$. In the same way, intermediate coefficients for the next rows are computed with

$$e_i = \frac{u_i}{d_i - l_i e_{i-1}}, \tag{4.7}$$

for $i = 2, \ldots, n-1$, and

$$f_i = \frac{b_i - l_i f_{i-1}}{d_i - l_i e_{i-1}}, \tag{4.8}$$

for $i = 2, \ldots, n$. The backward substitution obtains x with $x_n = f_n$, and for $i = n-1, \ldots, 1$, it computes

$$x_i = f_i - e_i x_{i+1}. \tag{4.9}$$

The numerical stability of this algorithm cannot be guaranteed in general. Its applicability is limited to diagonally dominant or symmetric positive definite matrices, with which the method is stable. Otherwise, partial pivoting is necessary to make it stable.

This same algorithm can be applied to a block-tridiagonal system, which in the forward elimination phase computes

$$C_1 = B_1^{-1} C_1, \tag{4.10a}$$
$$b_1 = B_1^{-1} b_1, \tag{4.10b}$$

and for $i = 2, \ldots, \ell$ proceeds with

$$B_i = B_i - A_i C_{i-1}, \tag{4.11a}$$
$$C_i = B_i^{-1} C_i, \tag{4.11b}$$
$$b_i = B_i^{-1}(b_i - A_i b_{i-1}); \tag{4.11c}$$

the backward substitution starts with $x_\ell = b_\ell$, and runs

$$x_i = b_i - C_i x_{i+1}, \tag{4.12}$$

for $i = \ell - 1, \ldots, 1$.

Steps (4.11a)–(4.11b) perform a block LU factorization, that needs to be computed only once. Note that the factorization is destructive, but we assume that the original matrix T is no longer needed. The factorization can be accomplished with a few calls to BLAS' _gemm and LAPACK's _getrf/_getrs. With _getrf we compute the LU factorization with partial pivoting of the diagonal block B_i. We remark that since the pivoting is limited to the diagonal block, this algorithm is numerically less robust than a full LU factorization. Subsequent right-hand sides of the Arnoldi iterations only require steps (4.11c)–(4.12).

We mentioned Thomas algorithm and its block-oriented version just for reference here, because it is an inherently serial algorithm, with little opportunity of parallelism except for the computations within the blocks.

4.2.2 Block cyclic reduction

As can be found in [50, ch. 5], besides the classical Gaussian elimination there are several well-known algorithms to solve tridiagonal linear systems such as recursive doubling, cyclic reduction or parallel cyclic reduction. All these methods try to increase the number of concurrent tasks and therefore reduce the length of the critical path, although the cost in flops is increased. They can be extended to the block-tridiagonal case too.

One of the solvers that we have implemented is based on cyclic reduction [24] (also known as odd/even reduction or CORF). Cyclic reduction is a recursive algorithm to solve tridiagonal linear systems that has two main steps: a forward elimination reduces the number of rows it works with, and a backward substitution obtains the solution while undoing the recursion. It divides the rows in even-indexed and odd-indexed, and in the forward elimination it recursively eliminates the even-indexed rows in terms of the odd-indexed ones. Depending on the dimension of the matrix, the number of rows is approximately halved in each recursion (if the number of rows is a power of two, it is actually halved) until a single row is left. Assuming that no division by zero is encountered in any of these steps, with the last row, an equation with a single unknown is trivially solved. The backward substitution step progresses increasing the number of rows used in the same proportion as the forward elimination reduces them. It uses the current recursion level solution(s) to compute the adjacent even-indexed rows on the previous level until all the unknowns are solved.

As noted in [51,84], the cyclic reduction method has the property of being equivalent to Gaussian elimination without pivoting on the system $(PT_\sigma P^{T_\sigma})(Px) = Pb$, where P is a permutation matrix that places first the indices that are odd multiples of 2^0, then odd multiples of 2^1, and so on. Such link with Gaussian elimination supports the conclusion that cyclic reduction is numerically stable in the same cases where Gaussian elimination with diagonal pivots is. If T_σ is strictly diagonally dominant or symmetric positive definite, then no pivoting is necessary and cyclic reduction is stable.

A version of the algorithm that works with general block-tridiagonal matrices was proposed in [57] and more recently reworked in [66] as BCYCLIC, where block-rows are reduced cyclically instead of rows. In this case, the algorithm can progress as long as the diagonal blocks are non-singular, since it is necessary to compute their inverse, or handle them implicitly via factorization. Pivoting can be used when factorizing the diagonal blocks but this does not guarantee the overall stability of the algorithm. Some aspects of the numerical stability of the block cyclic reduction were analyzed in [57] and [143], where a study of the bounds of the forward error is conducted for the cases of (strictly) diagonally dominant matrices assuming that the matrix is block-column diagonally dominant.

During the forward elimination stage, the block cyclic reduction computes the inverse of the even-indexed diagonal blocks and a modified (hatted) version of the lower and upper blocks. In the same way, a modified version of the even-indexed blocks of the right-hand side (RHS) vector b is computed. In the first recursion, the

computed quantities are

$$\hat{B}_{2i} = B_{2i}^{-1}, \tag{4.13a}$$
$$\hat{A}_{2i} = \hat{B}_{2i} A_{2i}, \tag{4.13b}$$
$$\hat{C}_{2i} = \hat{B}_{2i} C_{2i}, \tag{4.13c}$$
$$\hat{b}_{2i} = \hat{B}_{2i} b_{2i}, \tag{4.13d}$$

for $i = 1, \ldots, \ell/2$. The respective modified version of the odd-indexed blocks is then computed by using the adjacent modified even-indexed blocks,

$$\hat{B}_{2i-1} = B_{2i-1} - A_{2i-1}\hat{C}_{2i-2} - C_{2i-1}\hat{A}_{2i}, \tag{4.14a}$$
$$\hat{A}_{2i-1} = -A_{2i-1}\hat{A}_{2i-2}, \tag{4.14b}$$
$$\hat{C}_{2i-1} = -C_{2i-1}\hat{C}_{2i}, \tag{4.14c}$$
$$\hat{b}_{2i-1} = b_{2i-1} - A_{2i-1}\hat{b}_{2i-2} - C_{2i-1}\hat{b}_{2i}. \tag{4.14d}$$

In subsequent recursion levels, the computation is analogous to (4.13)-(4.14), but for a matrix that has about half of the blocks with respect to the previous level (even blocks have been removed).

In an MPI implementation, if the matrix is distributed across several processes, prior to the computation of (4.14) it is necessary to ensure that $\hat{A}_{2i}$, $\hat{C}_{2i}$ and $\hat{b}_{2i}$ are accessible from the processes that own the adjacent odd-indexed block-rows, so communication may be necessary in this case. That occurs for every recursive step of the algorithm. The modified even-indexed blocks can overwrite the original ones, but for the odd-indexed blocks, in order to back-solve with successive right-hand sides, both the original lower and upper blocks and their modified versions must be retained producing a 66% increase of memory usage during this stage.

The backward substitution stage starts by solving the single block equation

$$x_1 = \hat{B}_1 \hat{b}_1, \tag{4.15}$$

where $\hat{B}_1$ and $\hat{b}_1$ correspond to quantities computed in the last recursion level. Once this final odd-indexed block, which is the first part of the solution, is obtained, the recursion tree is traversed in reverse order. The solution blocks from a certain recursion level are used to compute the adjacent even-indexed blocks on the previous level, with

$$x_{2i} = \hat{b}_{2i} - \hat{A}_{2i} x_{2i-1} - \hat{C}_{2i} x_{2i+1}. \tag{4.16}$$

Communication occurs in an analogous way as in the forward elimination, but in the opposite direction. The algorithm continues until all blocks are processed.

The computational cost is concentrated in the first 3 operations of (4.13) and (4.14). These operations represent the factorization itself, and can be amortized in the case of successive right-hand sides. This is what happens in the Arnoldi method, that needs to invoke the linear solver in each iteration of the eigensolver.

4.2.3 Spike

Spike [106] is an algorithm intended for the parallel solution of banded linear systems, with a possibly sparse band, that we have particularized for the block-tridiagonal structure. At the outset of the algorithm, matrix T_σ is partitioned and distributed evenly among the p available processes, and, as we mentioned before, we assume that none of the ℓ block-rows are split across different processes. Each process r organizes its local data in the form

$$T_r = \left[\begin{array}{c|c|c|c|c} 0 & \begin{matrix} D_r \\ 0 \end{matrix} & E_r & \begin{matrix} 0 \\ F_r \end{matrix} & 0 \end{array}\right], \quad r = 0, \ldots, p-1, \tag{4.17}$$

distinguishing between the diagonal portion E_r (that is, the column range corresponding to local rows), and the lower (D_r) and upper (F_r) blocks that represent the coupling with neighbouring processes. Note that the sizes of these three blocks differ, being the diagonal block larger than the lower and upper blocks. In our case, the size of D_r and F_r is k, the original block size of the block-tridiagonal matrix, whereas the diagonal block has an approximate size of n/p, being p the number of processes used.

The Spike algorithm has two main phases: factorization and post-processing. The factorization phase consists in computing the so-called *Spike* matrix, S, defined from the decomposition $T_\sigma = ES$, where $E = \mathrm{diag}(E_0, E_1, \ldots, E_{p-1})$. This amounts to multiplying the local matrix T_r by the inverse of the diagonal block E_r to get the local matrix S_r. In order to do that, the first step is to factorize the diagonal blocks E_r, and compute matrices V_r and W_r by solving the system

$$E_r \begin{bmatrix} V_r & W_r \end{bmatrix} = \left[\begin{array}{cc} \begin{bmatrix} 0 \\ F_r \end{bmatrix} & \begin{bmatrix} D_r \\ 0 \end{bmatrix} \end{array}\right] \tag{4.18}$$

on each process. We must emphasize that (4.18) is a system of linear equations with $2k$ right-hand sides, where the coefficient matrix is block-tridiagonal, and hence a solver that exploits this structure can be used. Furthermore, this computation can be performed independently by each process, without communication.

These V_r and W_r matrices are the spikes on a matrix S whose main block-diagonal is the identity matrix,

$$S = \begin{bmatrix} I & \hat{V}_0 & & & & & \\ \hat{W}_1 & I & \hat{V}_1 & & & & \\ & \hat{W}_2 & I & \hat{V}_2 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \hat{W}_{p-2} & I & \hat{V}_{p-2} & \\ & & & & \hat{W}_{p-1} & I & \end{bmatrix}, \tag{4.19}$$

where we use the notation $\hat{V}_r = \begin{bmatrix} V_r & 0 \end{bmatrix}$ and $\hat{W}_r = \begin{bmatrix} 0 & W_r \end{bmatrix}$.

For the post-processing phase, each process divides the local portion of the matrix S in top, middle and bottom parts, being the top and bottom parts of size k, and

the middle part of the same size as E_r minus $2k$. For instance, the upper spike is written as $V_r = \begin{bmatrix} V_r^{(t)} \\ V_r^{(m)} \\ V_r^{(b)} \end{bmatrix}$. Selecting only the top and bottom blocks of the spikes, a reduced matrix

$$\hat{S} = \begin{bmatrix} I & & V_0^{(t)} & & & & & \\ & I & V_0^{(b)} & & & & & \\ & W_1^{(t)} & I & & V_1^{(t)} & & & \\ & W_1^{(b)} & & I & V_1^{(b)} & & & \\ & & \ddots & \ddots & \ddots & & & \\ & & & W_{p-2}^{(t)} & I & & V_{p-2}^{(t)} & \\ & & & W_{p-2}^{(b)} & & I & V_{p-2}^{(b)} & \\ & & & & & W_{p-1}^{(t)} & I & \\ & & & & & W_{p-1}^{(b)} & & I \end{bmatrix} \tag{4.20}$$

is built, also with an identity matrix on its diagonal.

The first stage in the post-processing phase is the factorization of the reduced matrix $\hat{S}$. For the algorithm to work in parallel, the size of the original matrix has to be large enough to distribute at least two block-rows per process, so the number of processes has to be limited to allow this ratio when factorizing small matrices.

Once $\hat{S}$ has been factored, the next stage is to locally compute vector $g = E^{-1}b$ by solving the system

$$E_r g_r = b_r, \tag{4.21}$$

in a similar way as (4.18). Note that here b_r denotes the whole subvector of b stored locally in process r, not an individual block. Vector g_r is in turn partitioned in the same top, middle and bottom parts as the spikes. From the top and bottom parts of each process, a reduced vector $\hat{g}$ is formed and used as RHS to solve the reduced system

$$\hat{S}\hat{x} = \hat{g}, \tag{4.22}$$

where $\hat{x}$ refers to the top and bottom blocks of x on each process.

Finally, to fully solve the whole system, the middle parts of x can be obtained in parallel with

$$\begin{cases} x_0^{(m)} = g_0^{(m)} - V_0^{(m)}\hat{x}_1^{(t)}, \\ x_j^{(m)} = g_j^{(m)} - V_j^{(m)}\hat{x}_{j+1}^{(t)} - W_j^{(m)}\hat{x}_{j-1}^{(b)}, & j = 1, \ldots, p-2, \\ x_{p-1}^{(m)} = g_{p-1}^{(m)} - W_{p-1}^{(m)}\hat{x}_{p-2}^{(b)}. \end{cases} \tag{4.23}$$

For the case of diagonally dominant matrices, as noted in [106], it is interesting to consider that the value of the elements of the right spikes (V_r) decays in magnitude from bottom to top. The decay is greater, the more diagonally dominant the matrix is. As a consequence, the value of the elements of the top block of V_r can be expected

to be zero and can be discarded from the computation. The same assumption can be done for the bottom block of the left spikes W_r. Discarding these zero blocks, the extra reduced matrix

$$\bar{S} = \begin{bmatrix} I & V_0^{(b)} & & & & & & & \\ W_1^{(t)} & I & & & & & & & \\ & & I & V_1^{(b)} & & & & & \\ & & W_2^{(t)} & I & & & & & \\ & & \ddots & \ddots & \ddots & & & & \\ & & & & I & V_{p-3}^{(b)} & & & \\ & & & & W_{p-2}^{(t)} & I & & & \\ & & & & & & I & V_{p-2}^{(b)} \\ & & & & & & W_{p-1}^{(t)} & I \end{bmatrix} \tag{4.24}$$

can be built and factored as a block-diagonal matrix, instead of as a block-tridiagonal one. Data movement between processes is necessary to form the matrix $\bar{S}$, and all but one of the processes can work in parallel during the factorization without communication.

Polizzi and Sameh presented in [106] an optimization for the diagonally dominant case referred to as Truncated Spike that computes only the bottom block of V_r and the top block of W_r. It requires to perform one LU factorization and one UL factorization of the diagonal block E_r. Their experiments proved that this approach reduces the computing time on machines with arithmetic operations much faster than accesses to the memory. As the spikes are not fully computed, the post-processing stage differs from the general Spike algorithm. A study of the Truncated Spike [95] found that if the degree of diagonal dominance

$$d = \left(\max_i \left\{ \frac{\sum_{j \neq i} |a_{ij}|}{|a_{ii}|} \right\} \right)^{-1} \tag{4.25}$$

of the original matrix is not too close to 1 and if the partitions are sufficiently large, then the errors at every stage of the algorithm are small. A more recent work [94] discusses several strategies to approach the diagonally and non-diagonally dominant cases.

Inexact shift-and-invert

For the non-diagonally dominant case, the Truncated Spike can be used as a preconditioner of an iterative method to solve the linear system. In this case, the decay of the values in the spikes is considerably less prominent, and the Truncated Spike discards blocks with nonzero elements. The larger the block size and/or number of processes are, the greater the valid data discarded on this step, and the worse the approximated solution is. In such case, a large number of outer iterations is needed, dramatically increasing the communication and the computation, making

this algorithm not worthwhile to solve such kind of systems. Because of this, we have not considered this variant in the numerical experiments.

4.3 Block cyclic tridiagonal structures

Returning to our initial example of Figure 4.1, if periodic boundary conditions are imposed, then the domains Ω_1 and Ω_ℓ are coupled, and the resulting coefficient matrix is not purely block-tridiagonal, as in (4.1), but has additional nonzero blocks on the top-right and bottom-left corners

$$M = \begin{bmatrix} B_1 & C_1 & & & & A_1 \\ A_2 & B_2 & C_2 & & & \\ & A_3 & B_3 & C_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & A_{\ell-1} & B_{\ell-1} & C_{\ell-1} \\ C_\ell & & & & A_\ell & B_\ell \end{bmatrix}, \tag{4.26}$$

and hence algorithms like the block-cyclic reduction or Spike are not directly applicable.

4.3.1 Schur complement

The Schur complement [114, ch. 14] is a classical method that can be applied to the resolution of linear systems. Let M be a square matrix of dimension n partitioned as

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \tag{4.27}$$

on which we perform a block Gaussian elimination. The Schur complement of M relative to D is

$$S = M/D = A - BD^{-1}C. \tag{4.28}$$

Assuming that D is non-singular, it is possible to solve the system of linear equations

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} e \\ f \end{bmatrix} \tag{4.29}$$

by solving

$$Sx = e - BD^{-1}f, \tag{4.30}$$

and once this system is solved, the rest of the system can be solved with

$$Dy = f - Cx. \tag{4.31}$$

In our case M is a block cyclic tridiagonal matrix as (4.26), with ℓ block-rows, with blocks of size k ($n = \ell \cdot k$), and we take A, B, C and D to be of dimensions

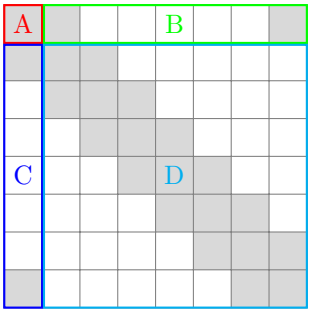

Figure 4.2. Schema of the partitioning of a block cyclic tridiagonal matrix into four smaller matrices to solve a system of linear equations by using the Schur complement.

$k \times k, k \times q, q \times k$ and $q \times q$, respectively (with $q = n - k$) as illustrated in Figure 4.2. With this partitioning, (4.30) is a system of small size, and D is a block-tridiagonal matrix that can be factored in parallel with the block-cyclic reduction or Spike algorithms.

4.4 Parallel implementations

We comment now the details of the implementations done of the block oriented methods described to expand the Krylov subspace and how they make use of GPUs. Given the block-tridiagonal matrix T of (4.1), we store it in memory as

$$\mathrm{rep}(T) = \begin{bmatrix} \square & B_1 & C_1 \\ A_2 & B_2 & C_2 \\ A_3 & B_3 & C_3 \\ \vdots & \vdots & \vdots \\ A_\ell & B_\ell & \square \end{bmatrix}, \tag{4.32}$$

where the $\square$ symbols indicate blocks with memory allocated but not being used, and all the blocks are stored contiguously as dense blocks.

In the MPI implementation, block-rows are distributed complete across processes, and hence only the first and last process have an unused block. Within our software, we have used different mathematical libraries that provide us with optimized parallel implementations of BLAS and LAPACK routines that we use to operate with the blocks of the matrices. Intel's MKL has been used for the CPU version of the software, providing thread parallelism. For the GPU version we have implementations that use cuBLAS and/or MAGMA libraries.

4.4.1 Matrix-vector product

The storage in memory mentioned in (4.32) allows us to use a single _gemv BLAS call per block-row to perform the matrix-vector product on CPU and GPU. Preceding

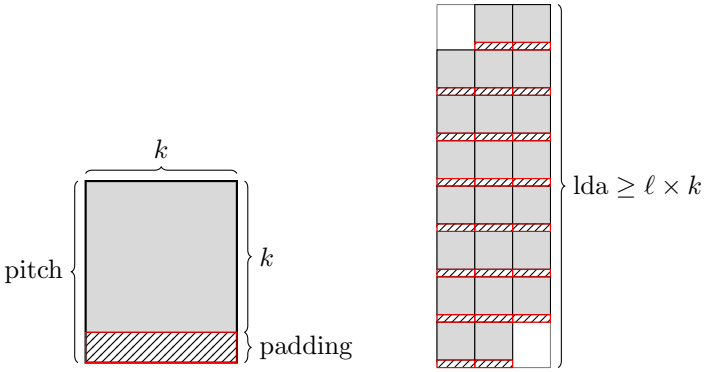

Figure 4.3. Schema of the memory allocated in the GPU for storing a matrix block of dimensions $k \times k$ (left), and a block tridiagonal matrix stored in 2D memory (right). The blocks are represented transposed to illustrate that we store the matrices using a column-major order.

the multiplication, in an MPI implementation, each process needs to receive from its neighbours the parts of the RHS vector of the size of a matrix block, necessary to perform the computation. We remark that when allocating memory for rep(T) on the GPU, this compact storage shape allows us to use 2D memory (pitched memory), to guarantee the alignment of the columns, which is important for coalesced memory accesses. Matrices in (CUDA) C are stored in a row-major order, and the `cudaMallocPitch` function arguments follow this memory layout providing alignment for the rows (the requested width), but we work with matrices organized in column-major order. Figure 4.3 shows how we store the matrix by blocks, allocating dense 2D blocks in memory and using them with a column-major layout. Apart from the (cu)BLAS version, we have also implemented a customized CUDA kernel that performs the whole matrix-vector computation with a single kernel invocation.

The kernel is invoked from the wrapper function shown in Figure 4.4. This host function selects the appropriate GPU card to be used based on the process rank in the MPI communicator, prepares the kernel execution, by configuring the sizes of the CUDA block and grid in a one dimensional shape, in which it distributes the rows of the matrix, and verifies the correct execution. When establishing the dimensions, it starts setting a fixed CUDA block size of 256, and (if necessary) halves it until it is less or equal to the matrix block size. The desired effect of this action is to increase the number of blocks, that are distributed across the streaming multiprocessors, increasing the card's occupancy. Starting with a block size of 256 ensures that the resulting number of threads in the block does not exceed the hardware maximum, and that is divisible by the warp size[1]. With this setting, the number of registers and the amount of shared memory used in the kernel is within the hardware limits.

[1]We do not expect to work with matrix block sizes smaller than the warp size.

```
#define CUDA_THREADS          256
#define COMPACT_BLOCK_COLUMNS 3

__host__ PetscErrorCode
blas_dgbmvn(PetscInt m, PetscInt n, const PetscScalar *A,
            PetscInt lda, const PetscScalar *x, PetscScalar *y)
{
  PetscInt       size, rank, first, last, cuda_threads, nblk;
  dim3           grid, threads;
  cudaError_t    cerr;

  set_cuda_device();
  MPI_Comm_size(PETSC_COMM_WORLD,&size);
  MPI_Comm_rank(PETSC_COMM_WORLD,&rank);

  first = last = 0;
  if (rank == 0) first = 1;
  if (rank == size - 1) last = 1;

  nblk = n / COMPACT_BLOCK_COLUMNS;
  cuda_threads = CUDA_THREADS;
  /* reduce the number of threads with small block size matrices */
  /* helps to distribute the work between more processors */
  while (cuda_threads > nblk) cuda_threads /= 2;
  threads.x = cuda_threads;

  /* cuda blocks per matrix block */
  grid.x = (PetscInt) PetscCeilReal((PetscReal) nblk / cuda_threads);
  grid.x *= (m / nblk);

  if (threads.x) {
    dgbmvn_kernel<<<grid, threads>>>(m, n, A, lda, x, y, first, last);
    cerr = cudaGetLastError(); CHKERRCUDA(cerr);
  }
  return 0;
}
```

Figure 4.4. Wrapper function for the matrix-vector kernel.

Since a single process uses a single GPU, in order to use all the available cards per node, the number of processes should be at least one per card. The selection of the card by the processes is done taking into account the number of available cards and the rank of the process, in the same way as it is done in Chapter 3 (Section 3.4.3), to attempt that each process uses a different CUDA card.

As the compact storage uses aligned memory by blocks, the matrix-vector product is computed by block-rows. The kernel, shown in Figure 4.5, receives the pointers to the data, the dimensions of the compacted storage, the leading dimension of the matrix, and two flags, that indicate if the process has the first and/or last rank in the MPI communicator.

At the beginning of the function, each thread computes several indexes that point out on which data to operate. All the threads in a CUDA block work with data on the same matrix block-row. The first thing computed is the number of CUDA

blocks necessary to operate within a matrix block-row. With this number and the CUDA block index, it is possible to know the index of the assigned local matrix block-row. The local prefix refers to the local portion of the matrix stored by the process. After that, the index of the row on the current matrix block-row, the local matrix row index, and the vector index are computed.

In order to benefit from coalesced memory operations, each thread computes all the elements in the matrix row that it has been assigned. This way, the load operation from global memory of the matrix elements is done jointly by all the threads in the same CUDA block. The advance of the computation through the columns is done in parallel by all the threads of the same CUDA block, in a loop, in batches of the size of the CUDA blocks, and without going beyond the last column. Within this loop, that has the CUDA block size as step, each thread of the block loads one element of the vector into a shared memory in a single step. This way, a consecutive part of the vector, of size equal to the number of threads, is stored in shared memory. In another inner loop each thread multiplies the consecutive elements of the matrix row times the corresponding elements of the vector stored in shared memory.

By default, in the main loop, all threads compute all columns of the storage between the first one and the largest multiple of the CUDA block dimension not greater than the last column. In the case of the threads computing the first matrix block, the column in which they begin is shifted to the right by one matrix block. And in the case of the threads computing the last matrix block, the last column computed is the largest multiple of the CUDA block dimension not greater than two matrix blocks. See again the storage of the matrix in Figure 4.3.

The main loop can leave columns of the matrix without being computed. As these remaining operations require additional checks that create divergent branches in the code flow, and slow down the computation, they are done separately. The process of storing the vector in shared memory and performing the multiplication is repeated with the remaining columns outside the loop. Finally, the result of each row is stored on the output vector.

4.4.2 Direct linear solvers

The algorithms employed to solve linear systems of equations when computing interior eigenvalues are implemented by means of BLAS and LAPACK routines. To adapt the algorithms in this section to solve multiple RHS on subsequent iterations of Arnoldi, practical implementations split them in two subroutines, factorization and solve. In all cases, the factorization of the matrix excludes the operations with the RHS vector(s), and those excluded steps are done in the solve subroutine. This allows us to invoke the factorization only once, and use the solve in each iteration of Arnoldi.

```
__global__ void
dgbmvn_kernel(PetscInt m, PetscInt n, const PetscScalar *A, PetscInt lda,
              const PetscScalar *x, PetscScalar *y, PetscInt first, PetscInt last)
{
  PetscInt               i, j, jbegin, jend, idx, yidx, nblk, block0, pitch;
  PetscInt               nmax, cblockspb, row;
  PetscScalar            res;
  __shared__ PetscScalar shx[CUDA_THREADS];

  nblk = n / COMPACT_BLOCK_COLUMNS;
  pitch = (lda / (m / nblk));
  /* cuda blocks per matrix block */
  cblockspb = (PetscInt) PetscCeilReal((PetscReal) nblk / blockDim.x);
  block0 = blockIdx.x / cblockspb; /* local matrix block index */
  /* row on current matrix block */
  row = (((blockIdx.x % cblockspb) * blockDim.x) + threadIdx.x);
  idx = block0 * pitch + row; /* local matrix row index */
  yidx = block0 * nblk + row; /* local vector index */
  res = 0.0;
  nmax = n;
  jbegin = 0;
  jend = ((n / blockDim.x) * blockDim.x);
  if (first && (block0 == 0)) {
    jbegin = nblk;
    jend = (((nblk * 2) / blockDim.x) * blockDim.x);
  }
  if (last && (block0 == ((lda / pitch) - 1))) {
    nmax= nblk * 2;
    jend = (((nmax) / blockDim.x) * blockDim.x);
  }
  A += idx + lda * jbegin;
  x += jbegin + (block0 * nblk) + threadIdx.x;
  for (i = 0; i < jend; i += blockDim.x) {
    shx[threadIdx.x] = x[i];
    __syncthreads();
    for (j = 0; j < blockDim.x; j++) {
      res += A[0] * shx[j];
      A += lda;
    }
    __syncthreads();
  }
  if (nmax > jend + jbegin) { /* remaining columns */
    if (threadIdx.x + jbegin + jend < nmax) shx[threadIdx.x] = x[jend];
    __syncthreads();
    if (row < nblk) {
      for (j = 0; j < (nmax - (jend + jbegin)); j++) {
        res += A[0] * shx[j];
        A += lda;
      }
    }
  }
  /* do not write beyond the matrix block size */
  if (row < nblk) y[yidx] = res;
}
```

Figure 4.5. Matrix-vector kernel.

Bcyclic

As a classical algorithm, the cyclic reduction has already been widely implemented for different programming paradigms and computer architectures. Studies of its performance on GPU against other solvers for tridiagonal matrices were carried out by Zhang et al. [146]. MPI-based implementations were studied for the block-tridiagonal case in [66] and [118]. The authors of the former combined the use of MPI parallelism over the block-rows with a threaded parallelism with OpenMP or GotoBLAS to perform the local operations. The effect of the block size in the performance was studied in [118]. A heterogeneous approach was implemented by Park and Perumalla [104], who use MPI, and the block arithmetic is done simultaneously on GPU with cuBLAS or MAGMA [132], and on multicore processors with ACML. Another single-GPU implementation for the block-tridiagonal case was done in [10], whose authors tested a variety of block and matrix sizes, showing that a better performance is obtained with systems with relatively large block sizes by better utilizing the available GPU threads. Recently, a comparison of the classical solvers, including Thomas algorithm, was addressed in [86], implementing them on CPU, many integrated cores (MIC) architecture, and GPU accelerators for the case of using a single node.

Algorithms 4.2 and 4.3 summarize the operations done on each of the two subroutines of the BCYCLIC algorithm with an iterative implementation. We remark that some (parts) of the operations of the algorithms only take place if the involved blocks exist on the process at the current level. In Algorithm 4.2, those operations correspond to steps 12, 14, 16, 17 and 18, and in Algorithm 4.3 they correspond to steps 12, 14, 18, 26 and 28.

Step 2 of Algorithms 4.2 and 4.3, which is detailed in Algorithm 4.4, obtains the lower and upper limit of the range of block-rows owned by the calling process. Algorithm 4.2 goes exclusively through the $\lceil \log_2(\ell) \rceil$ levels of the forward elimination, while Algorithm 4.3 performs the forward elimination and the backward substitution.

In Algorithm 4.2, there are two alternatives available to deal with the inverse of the diagonal blocks in step 10. One is to explicitly compute the inverse by means of LAPACK routines _getrf and _getri. It could seem inappropriate to do this due to the high cost of computing the inverse, but once it is computed, the rest of the steps of the algorithm can be done with optimized matrix-matrix and matrix-vector multiplications. The other alternative is to solve linear systems by using LAPACK routines _getrf and _getrs, that a priori seems the more reasonable way to go due to the cheaper cost of the operations, but as we will see in Section 4.5 the performance obtained with this alternative is seldom the best.

In steps 18 and 28 of Algorithm 4.3, parts of the solution vector are exchanged between processes. One block of the solution vector can be adjacent to multiple blocks at different levels of the backward substitution, and those blocks can belong to the same or different processes. To avoid repeating the same communication twice between the same pair of processes, the implementations of Algorithm 4.3 can store and take note of the parts of the solution vector already sent/received.

Algorithm 4.2. BCYCLIC factorization

For all processes (with rank r) **do in parallel**
  [low, high] = GetOwnershipRange(p,r,ℓ)
  for $j = 1 : \lceil \log_2(\ell) \rceil$ **do** `/* For each iteration level */`
    begin $= 2^{j-1} + 1$
    if begin $\leq$ high **then**
      while begin $<$ low **do** begin $=$ begin $+ 2^j$
      begin $=$ begin $-$ low
    end
    for $i =$ begin $: 2^j : \ell_r$ **do** `/* For each even block-row */`
      $\hat{B}_{2i} = B_{2i}^{-1}$
      $\hat{A}_{2i} = \hat{B}_{2i} A_{2i}$
      $\hat{C}_{2i} = \hat{B}_{2i} C_{2i}$
    end
    Send/receive $\hat{A}$ and $\hat{C}$ to/from adjacent block-rows
    for $i = 1 : 2^j : \ell_r$ **do** `/* For each odd block-row */`
      $\hat{A}_{2i-1} = -A_{2i-1}\hat{A}_{2i-2}$
      $\hat{B}_{2i-1} = B_{2i-1} - A_{2i-1}\hat{C}_{2i-2} - C_{2i-1}\hat{A}_{2i}$
      $\hat{C}_{2i-1} = -C_{2i-1}\hat{C}_{2i}$
    end
  end
  if $r == 0$ **then** $\hat{B}_1 = B_1^{-1}$
end

Algorithm 4.3. BCYCLIC solve

For all processes (with rank r) **do in parallel**
  [low, high] = GetOwnershipRange(r,ℓ,p)
  for $j = 1 : \lceil \log_2(\ell) \rceil$ **do** `/* For each iteration level */`
    begin $= 2^{j-1} + 1$
    if begin $\leq$ high **then**
      while begin $<$ low **do** begin = begin $+ 2^j$
      begin = begin $-$ low
    end
    for $i =$ begin $: 2^j : \ell_r$ **do** `/* For each even block-row */`
      $\hat{b}_{2i} = \hat{B}_{2i} b_{2i}$
    end
    Send/receive $\hat{b}$ to/from adjacent block-rows
    for $i = 1 : 2^j : \ell_r$ **do** `/* For each odd block-row */`
      $\hat{b}_{2i-1} = b_{2i-1} - A_{2i-1}\hat{b}_{2i-2} - C_{2i-1}\hat{b}_{2i}$
    end
  end
  if $r == 0$ **then** $x_1 = \hat{b}_1 = \hat{B}_1 b_1$
  Send/receive x_1 to/from adjacent block-rows
  for $j = \lceil \log_2(\ell) \rceil : -1 : 1$ **do** `/* For each iteration level */`
    begin $= 2^{j-1} + 1$;
    if begin $\leq$ high **then**
      while begin $<$ low **do** begin = begin $+ 2^j$
      begin = begin $-$ low
    end
    for $j =$ begin $: 2^j : \ell_r$ **do** `/* For each even block-row */`
      $x_{2i} = \hat{b}_{2i} - \hat{A}_{2i} x_{2i-1} - \hat{C}_{2i} x_{2i+1}$
    end
    Send/receive x to/from adjacent block-rows
  end
end

Algorithm 4.4. GetOwnershipRange

Input: number of processes: p, process identifier: r, number of block-rows: ℓ
Output: global index of the first block-row owned by the process: low, global index of the last block-row owned by the process: high

```
if r < (ℓ mod p) then
    low = r(⌊ℓ/p⌋ + 2)
    high = low + ⌊ℓ/p⌋ + 1
else
    low = r⌊ℓ/p⌋ + (ℓ mod p) + 1
    high = low + ⌊ℓ/p⌋
end
```

Table 4.1. BCYCLIC implementations with each of the mathematical libraries.

		Non-batched		Batched		
	Math library	_getri	_getrs	_getri_1	_getri_2	_getrs
CPU	MKL	✓	✓	-	-	-
GPU	cuBLAS	-	-	✓	✓	✓
	MAGMA	✓	✓	✓	✓	✓

Table 4.1 summarizes the BCYCLIC implementations we have put into practice. In them we have used the batched operations provided by the GPU libraries used, as they allow us to invoke a single function call to factorize, solve, invert or multiply all the even-indexed diagonal blocks owned by a process, per recursive step. In the case of the _gemm function, the batched variant is expected to perform well with small matrices only. So we have created two variants of the GPU version that computes the inverse of the diagonal blocks, one with normal _gemm in a loop (denoted as _getri_1) and another one with batched _gemm (denoted as _getri_2). We also made CPU-equivalent implementations, that run on GPU without batched operations, by making use of the MAGMA library.

Spike

The Spike algorithm is available in the Intel Spike library[2] with an MPI-based implementation. Another implementation, specific for tridiagonal systems, was developed in [27] for the (multi-)GPU case and later included in the cuSPARSE library. A modification of Spike for tridiagonal systems, that makes use of QR factorization without pivoting via Givens rotations, presented in [136] as g-Spike, safeguards the algorithm in case that the partitioning of the matrix results in at least one of the E_r diagonal blocks being singular. This work was later adapted to the Intel Xeon

[2] https://software.intel.com/en-us/articles/intel-adaptive-spike-based-solver/

Phi platform in [137]. An implementation of the Truncated Spike was used as a preconditioner to solve tridiagonal linear systems on GPU in [119] through the use of the CUSP library.

In our case, we focus on developing a fast implementation of the algorithm by making use of several GPUs to perform the block arithmetic managed by a multi-process MPI solution in a similar way as [104]. When using the Spike algorithm to solve a block-tridiagonal system, the band has to be large enough to cover all the entries of the original matrix and additional triangular zero borders are included in the computation. We avoid the extra work that this would entail by factorizing the main diagonal block exploiting its block-tridiagonal structure. To do that, our implementations of Spike make use of the BCYCLIC implementations to solve the linear systems involved.

The factorization subroutine of Spike is represented in Algorithm 4.5, in which we can remark that only step 5 requires communication.

Algorithm 4.5. Spike factorization

1 **For all** processes (with rank r) **do in parallel**
2 $\quad \tilde{E}_r = \text{bcyclic\_factorize}(E_r)$
3 $\quad [V_r, W_r] = \text{bc4spike\_solve}(\tilde{E}_r, F_r, D_r)$
4 $\quad$ Build $\hat{S}$ with top and bottom parts of V_r and W_r
5 $\quad \tilde{S} = \text{bcyclic\_factorize}(\hat{S})$
6 **end**

Step 2 factorizes the diagonal block (block-tridiagonal) E_r and step 3 solves the corresponding system (4.18) to compute the spikes. For step 3, a different version of Algorithm 4.3 was implemented to deal with multiple RHS and to send/receive multiple times the same block of the solution vector (in this case, the re-sending option was chosen to reduce the allocated memory by eliminating the buffer mechanism, as no communication exists on this step). Also, due to the high number of zeros in the RHS matrices of (4.18), another modification was introduced in the BCYCLIC solve used (bc4spike_solve) that saves time by skipping operations involving zero blocks on every recursion of the algorithm.

The second factorization needed in step 5 of Algorithm 4.5 is the factorization of the reduced matrix ($\hat{S}$), that is also done by means of BCYCLIC and, in this case, communication occurs between the processes as the matrix is distributed across them.

The solve subroutine of Spike is shown in Algorithm 4.6, in which the steps involving communication are 4, 5 and 6. Step 8 of Algorithm 4.6, that computes the middle part of the final solution on each process, corresponds to (4.23).

The memory requirements of our Spike implementations are higher than those of BCYCLIC, since in addition to the original matrix size and the 66% extra amount of memory used by BCYCLIC to perform the factorization of the local diagonal block E_r, more memory has to be allocated to build the reduced matrix, whose block size

Algorithm 4.6. Spike solve

For all processes (with rank r) **do in parallel**
  $g_r = \text{bcyclic\_solve}(\tilde{E}_r, b_r)$
  Build $\hat{g}$ with top and bottom parts of g_r
  $\hat{x} = \text{bcyclic\_solve}(\tilde{S}, \hat{g})$
  Send $\hat{x}_r^{(t)}$ and $\hat{x}_r^{(b)}$ to $r-1$ and $r+1$, respectively
  Receive $\hat{x}_{r+1}^{(t)}$ and $\hat{x}_{r-1}^{(b)}$
  $x^{(t)} = \hat{x}^{(t)}$
  $x_r^{(m)} = g_r^{(m)} - V_r^{(m)} \hat{x}_{r+1}^{(t)} - W_r^{(m)} \hat{x}_{r-1}^{(b)}$
  $x^{(b)} = \hat{x}^{(b)}$
end

Table 4.2. Operations used to factorize the two matrices and solve the systems on the Spike implementations.

	Spike		Reduced Spike	
	Matrix E_r	Matrix $\hat{S}$	Matrix E_r	Matrix $\bar{S}$
Factorization subroutine	Fact. bcyclic Solve bc4spike	Fact. bcyclic	Fact. bcyclic Solve bc4spike	Fact. _getrf
Solve subroutine	Solve bcyclic	Solve bcyclic	Solve bcyclic	Solve _getrs

is twice as large as the original block size, and it has its own 66% increase. Besides that, the auxiliary memory buffers used to send and receive blocks have to be of this new double block size.

For diagonally dominant matrices, a variant of the Truncated Spike algorithm that works with the extra reduced matrix $\bar{S}$ of (4.24) can be used. In the sequel, this method is referred to as reduced Spike. In our case, the implementation does not limit the solve to the first and last $k \times k$ blocks, but computes the full spikes (done via the BCYCLIC algorithm). The reduction in the computation that it provides is obtained through the use of the extra reduced system, as the blocks discarded should not contain any nonzero elements. Since the spikes are fully computed, the logic of the algorithm that processes the diagonal blocks E_r does not change with respect to the general Spike. The extra reduced matrix can be factored in parallel with a _getrf call on each process without any communication or additional storage.

The solve subroutine of both variants of the Spike algorithm differs in the second solve used with the reduced matrix. A summary of the different operations employed by both variants can be seen in Table 4.2.

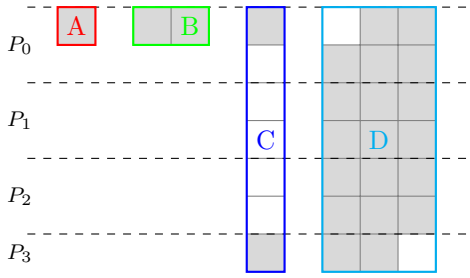

Figure 4.6. Distribution of the four sub-matrices of Figure 4.2 among several processes and its representation in memory.

Block cyclic tridiagonal

In our implementation of the Schur complement of Section 4.3.1, the submatrices A and B of (4.27) are stored on the first process, and C and D are partitioned among all the existing processes. Figure 4.6 illustrates this data distribution, where we can see how only the nonzero blocks of B are stored while the full size of C is allocated. D is a block-tridiagonal matrix that can be stored compacted as in (4.32), reducing the unused blocks to two.

When computing the Schur complement (4.28), the $D^{-1}C$ operation implies a linear solve with multiple right-hand sides, that can be computed in parallel, and since C has only two nonzero blocks on its edges, the computation involving the null blocks can be avoided during the solve stage of the block-cyclic reduction, reducing the processing time. The remaining operations to compute the Schur complement are only done by the first process, as it stores the first block-row of M that includes A and B. As B also has only two nonzero blocks, the matrix multiplication B times $D^{-1}C$ can be cheaply done by multiplying only these two blocks with the corresponding blocks of $D^{-1}C$; one of them is already stored in this first process and the other must be sent by the process with the highest rank.

In order to obtain x, the system (4.30) is solved in a similar way as (4.28) by computing $D^{-1}f$ in parallel with block-cyclic reduction. But this time no reduction in the operations is possible as f does not have any special zero pattern.

Once the first block of the solution is obtained, it must be sent from the first to the last process for them both to compute $f - Cx$, and after that, the block-cyclic reduction method can be used again to finally solve (4.31) in parallel.

4.5 Numerical experiments

After the development of the algorithms and with all the codes prepared for real and complex arithmetic, in both single and double precision, we performed some computational experiments aiming at assessing the performance of our software. Here we present the results of those experiments corresponding to real scalars using single and double precision.

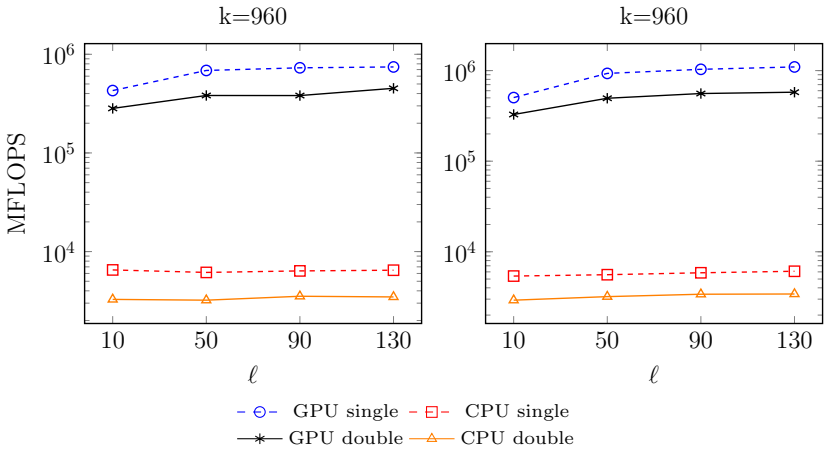

Figure 4.7. Performance of the matrix-vector product operation to compute the largest magnitude eigenvalue, for a fixed block size $k = 960$ and varying the number of blocks ℓ for both CPU and GPU on the Fermi (left) and Kepler (right) machine, in single and double precision arithmetic.

4.5.1 Single process executions

The matrix-vector product and the block cyclic reduction algorithm have been tested on single process computational experiments, conducted on random block-tridiagonal matrices of multiple sizes, where we have varied the number of blocks ℓ, and the block size k, up to reach the maximum storage space available on the GPU card. The four number of blocks used go from 10 to 130 with intervals of 40, and the block sizes vary between a minimum of 64 and a maximum of 960 with intervals of 128. The matrices used have been generated in all cases on the CPU to use the exact same matrix on both runs (CPU and GPU).

These tests have been run on two computer platforms:

Fermi 2 Intel Xeon E5649 processor (6 cores) at 2.53 GHz, 24 GB of main memory; 2 GPUs NVIDIA Tesla M2090, 512 cores and 6 GB GDDR per GPU. RHEL 6.0, with GCC 4.6.1 and MKL 11.1.

Kepler 2 Intel Core i7 3820 processor (2 cores) at 3,60 GHz with 16 GB of main memory; 2 GPUs NVIDIA Tesla K20c, with 2496 cores and 5 GB GDDR per GPU. CentOS 6.6, with GCC 4.4.7 and MKL 11.0.2.

In both platforms, the other software used is PETSc and SLEPc 3.6-dev, CUDA 7.0 and CUSP 0.5.0.

The matrix-vector product on CPU uses calls to BLAS' _gemv and is linked with MKL, and on GPU it uses the ad-hoc CUDA kernel of Figure 4.5. To assess their performance, we have computed the largest magnitude eigenvalue, for which the computation requires about 100 matrix-vector products.

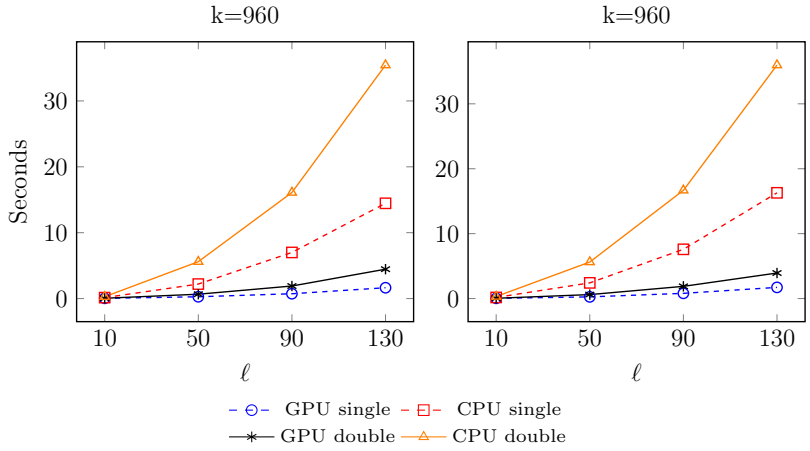

Figure 4.8. Total eigensolve operation time to compute the largest magnitude eigenvalue, for a fixed block size $k = 960$ and varying the number of blocks ℓ for both CPU and GPU on the Fermi (left) and Kepler (right) machine, in double precision arithmetic.

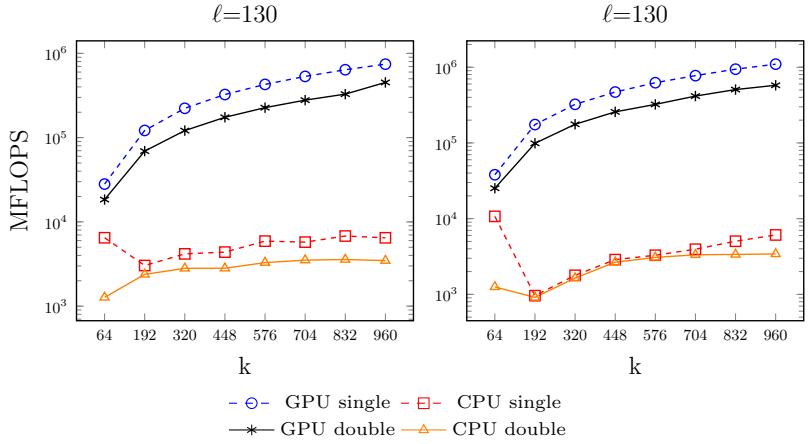

Figure 4.9. Performance of the matrix-vector product operation to compute the largest magnitude eigenvalue, for a fixed number of blocks $\ell = 130$ and varying the block size k for both CPU and GPU on the Fermi (left) and Kepler (right) machine, in single and double precision arithmetic.

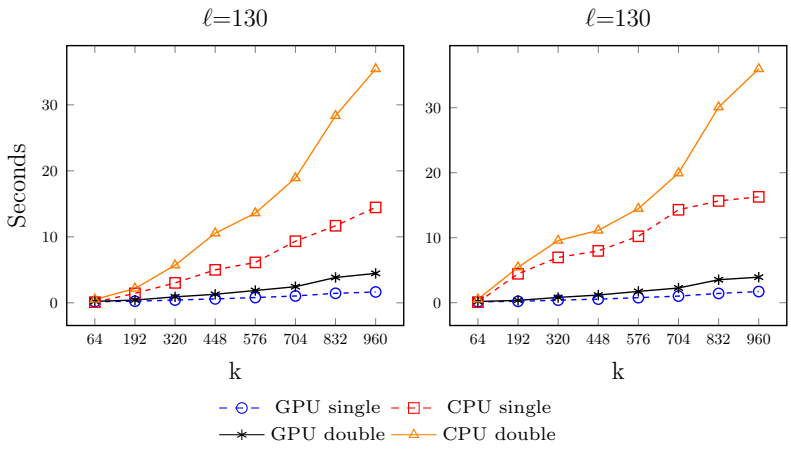

Figure 4.10. Total eigensolve operation time to compute the largest magnitude eigenvalue, for a fixed number of blocks $\ell = 130$ and varying the block size k for both CPU and GPU on the Fermi (left) and Kepler (right) machine, in double precision arithmetic.

Figures 4.7 and 4.9 show the Mflop/s rate obtained with the matrix-vector product operation, and Figures 4.8 and 4.10 show the total eigensolve operation time, when computing the largest magnitude eigenvalue. With a large block size (Figure 4.7), we can see that the performance does not depend too much on the number of blocks. In contrast, when we fix the number of blocks (Figure 4.9) the performance is significantly lower for small block sizes. In any case, the benefit of using the GPU is evident since we are able to reach about 1 Tflop/s with large block sizes. The GPU runs obtain speedups up to 9.0 (single) and 8.2 (double) on Fermi, with little variation when modifying the block size, and up to 20.8 (single) and 15.2 (double) on Kepler, with respect to the CPU runs. On Kepler, although the greatest performance is obtained with large block sizes, the speedup is reduced when increasing them.

For assessing the performance of the shift-and-invert computation using the block oriented cyclic reduction algorithm, we have computed one eigenvalue closest to the origin ($\sigma = 0$). The implementation of the block cyclic reduction used on these runs avoids to explicitly compute the inverse of the diagonal blocks B_i and solves a system of linear equations with the batched version of the _getrs routine provided by cuBLAS. Results are shown in Figures 4.11–4.14. In this case, the GPU version does not beat the CPU computation (using as many threads as computational cores in MKL operations). Nevertheless we can appreciate the sensitiveness of the GPU to the data size, as its performance improves when the number of blocks is increased (for a large block size), as well as when the block size is increased, while the CPU is only affected by the block size. The reported Mflop/s rates correspond to the LU factorization, whereas the triangular solves only achieve a performance around 2.5-4

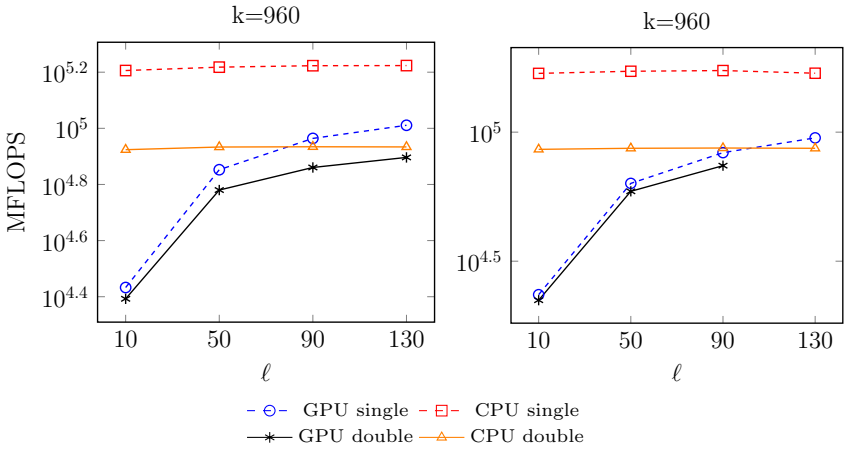

Figure 4.11. Performance of the factorization operation to compute the eigenvalue closest to the origin, for a fixed block size $k = 960$ and varying the number of blocks ℓ for both CPU and GPU on the Fermi (left) and Kepler (right) machine, in double precision arithmetic.

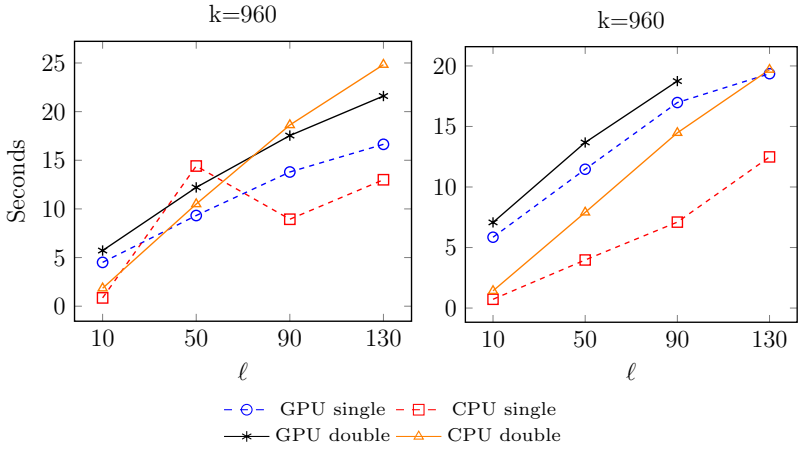

Figure 4.12. Total eigensolve operation time to compute the eigenvalue closest to the origin, for a fixed block size $k = 960$ and varying the number of blocks ℓ for both CPU and GPU on the Fermi (left) and Kepler (right) machine, in double precision arithmetic.

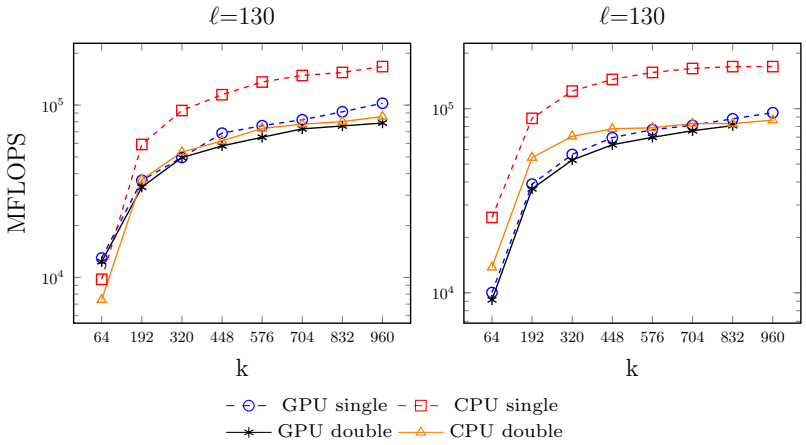

Figure 4.13. Performance of factorization to compute the eigenvalue closest to the origin, for a fixed number of blocks $\ell = 130$ and varying the block size k for both CPU and GPU on the Fermi (left) and Kepler (right) machine, in double precision arithmetic.

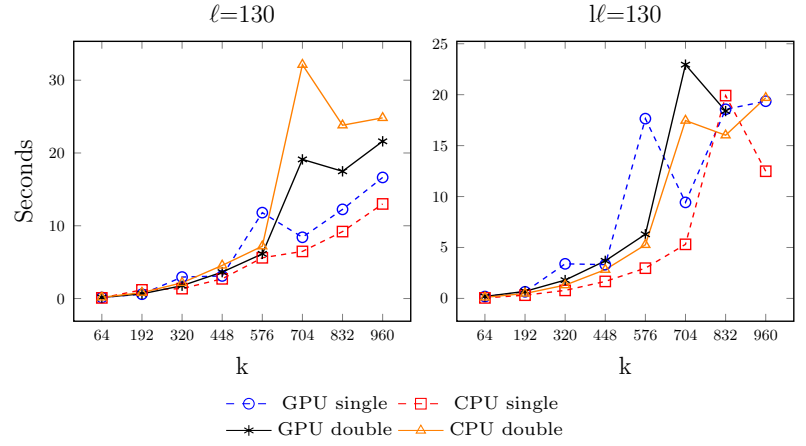

Figure 4.14. Total eigensolve operation time to compute the eigenvalue closest to the origin, for a fixed number of blocks $\ell = 130$ and varying the block size k for both CPU and GPU on the Fermi (left) and Kepler (right) machine, in double precision arithmetic.

Gflop/s, both on CPU and GPU. The performance of the single precision tests is presented, but they provided inaccurate results. The time shown in Figures 4.12 and 4.14 corresponds to the total eigensolve operation, using the same number of restarts on both GPU and CPU runs. All in all, the performance is much worse than in the matrix-vector product case, as expected.

4.5.2 Multi-process executions

The problem dimension of scientific applications has increased over time, and made necessary to use multi-process solutions. In this section we test our software with multi-process executions, as we are especially interested in evaluating the scalability for an increasing number of MPI processes (and GPUs), and also in knowing how the matrix block size impacts the performance, as the communication is directly affected by it. Another question we want to answer with the tests is which of the implemented variants performs best. We focus now on the time needed to perform the computation, rather than on the flop/s rate, and to strengthen the results obtained, on these executions we have moved the random generated matrices off the scene and employ matrices from real applications.

Block-tridiagonal case

For the scalability studies, we consider an application coming from astrophysics, where the matrices are banded (with a dense band) and can be generated for any matrix size with arbitrary bandwidth. The integral operator $T : X \to X$, arising from a transfer problem in stellar atmospheres [1], is defined by

$$(T\varphi)(\tau) = \frac{\varpi}{2} \int_0^{\tau^\star} \int_1^\infty \frac{e^{-|\tau-\tau'|\mu}}{\mu} d\mu\varphi(\tau')d\tau, \quad \tau \in [0, \tau^\star], \tag{4.33}$$

which depends on the albedo, $\varpi \in [0, 1]$, and the optical thickness of the stellar atmosphere, $\tau^\star$. We are interested in the eigenvalue problem $T\varphi = \lambda\varphi$ with $\lambda \in \mathbb{C}$ and $\varphi \in X$. This problem can be solved via discretization, that is, by projection onto a finite dimensional subspace X_n, resulting in an algebraic eigenvalue problem $A_n x_n = \theta_n x_n$ of dimension n, where A_n is the restriction of the projected operator to X_n. Further details can be found in [135].

Due to the exponential decay, the matrix A_n has a banded structure, with a bandwidth depending on the ratio between the matrix size, n, and the parameter $\tau^\star$,

$$b_w = \left\lfloor n \left(1 - \exp\left(-\frac{n}{t_c \tau^\star}\right)\right)\right\rfloor, \tag{4.34}$$

where $t_c = \max(n/100, 5)$. We always choose $\tau^\star$ in such a way that the band is contained within a block-tridiagonal structure, that we compute as dense.

As before, the problem size n is defined by the number of block-rows, ℓ, and by the block size, k. The strong and weak scaling analyses have been obtained with a batch of experiments on which the number of processes vary in powers of two

Table 4.3. Block sizes and number of block-rows used for the strong and weak scaling experiments. The column of the weak scaling shows the number of rows used with 128 processes, that is halved with the number of processes.

	Number of block-rows	
Block size	Strong scaling	Weak scaling
64	4800	51200
96	3200	38400
128	2400	25600
256	1200	12800
384	800	9600
512	600	6400
640	480	5600
768	400	4800
896	343	4000
1024	300	3200

from 1 to 128, and ten different block sizes that vary from 64 up to 1024 have been tested. Table 4.3 details the block sizes used and the number of block-rows with each block size. When computing the weak scaling, as the problem size per process is maintained fixed, the number of block-rows per process depends exclusively on the block size (for a given matrix size). For the strong scaling, the number of block-rows depends on the number of processes used, on the block size and on the process index, as the total matrix size does not change.

In these tests we are interested in computing eigenvalues closest to the albedo parameter. As we know that all the eigenvalues are smaller than the albedo, we use it as shift with the shift-and-invert technique operating on matrix $(A_n - \varpi I)^{-1}$, where $\varpi = 0.75$ in our runs. In all cases, the relative residual norm of the computed eigenpairs, $\|Az - \theta z\|/\|\theta z\|$, is always below the requested tolerance, `tol` $= 10^{-8}$. We have set the eigensolver to restart with a basis size of 16. In most runs, all eigenvalues converge without needing to restart, and hence 16 linear solves are performed per each factorization.

The executions have been carried out on Minotauro[3], on a cluster equipped with four GPUs per node. The cluster is formed by 39 servers with two Intel Xeon E5-2630 v3 processors and 128 GB of RAM, interconnected with FDR Infiniband cards at 56 Gb/s in a switched fabric network topology, and two NVIDIA K80 cards. Each K80 has two Kepler GPUs with 2496 cores and 12 GB of GDDR memory per GPU.

The servers run RedHat Linux 6.7 as operating system, and our software has been compiled with gcc 4.6.1 using PETSc and SLEPc 3.7-dev, and linked with the Intel MKL 11.3.2, NVIDIA CUDA 7.5 and MAGMA 1.7.0 libraries. The MPI version used for the inter-process communication is the cluster's manufacturer bullxmpi 1.2.9.1.

[3]Minotauro experienced a partial upgrade in March 2016. We here use exclusively the new cluster.

Since our codes use a single GPU per process, the number of processes per node has been limited to four in order to fully utilize the servers without oversubscribing the GPU cards with more than one process when running the GPU executions. In the case of the CPU runs, the same limit of four processes per node has been used to have the same communication overhead, and in this case the number of threads has been set also to four, to allow the software to use all the computational cores (one thread per core) with the four processes.

We have prepared three sets of experiments to evaluate the weak scaling of the different software versions, which compute 5 eigenvalues. In the following figures we show the factorization time and the aggregated time of the multiple solves of the Arnoldi algorithm needed to obtain five eigenvalues working in double precision arithmetic for the smallest and the largest block sizes used.

The first set shows the time needed with the non-batched versions of BCYCLIC for both approaches: explicitly computing the inverse of the diagonal block (_getri) or solving linear systems (_getrs). The initial highlight is the better performance of the CPU executions with small block sizes and the better performance on the GPU with large block sizes. With small block sizes, the kernels executed on the GPU do not allow the devices to obtain their maximum performance.

In Figure 4.15 it is clear how explicitly computing the inverse takes more time during the factorization stage, and less in the solving stage, as could be expected. For larger block sizes the differences between the _getri and _getrs versions disappear in both stages, and the _getri times turn to be slightly smaller in the factorization while maintain the better performance during the solve, as can be seen in Figure 4.16.

The factorization subroutine that uses _getri to compute the inverse of the diagonal blocks on the CPU requires a block size larger than 512 to be faster than the one using _getrs, while the GPU executions start to be faster with a block size larger than 128.

The second set of experiments shows exclusively executions on the GPU. We carry out a comparison of the five implementations that compute the inverse: the one already used in the first set (_getri) and two more batched implementations (_getri_1 and _getri_2) per library used. Both batched versions share the same solve stage, so no different results should be seen in it.

Figure 4.17 allows us to see how for a small block size, any of the batched versions performs faster than the non-batched one during the factorization. As expected, the batched versions allow the software to gain performance by computing the blocks in parallel. The solve results show a significant difference between the cuBLAS and MAGMA libraries due to their inherent performance, being cuBLAS faster in this stage.

The executions with a large block size seen in Figure 4.18 do not show the difference in time between the batched and non-batched implementations that occurs in the factorization with small sizes, as in this case, the large block size is enough to fulfill the massive parallel processing of the GPU. This occurs with block sizes larger than 512.

It is noticeable in Figure 4.18 how the cuBLAS implementations need considerably more time to perform the factorization while they are the fastest during the

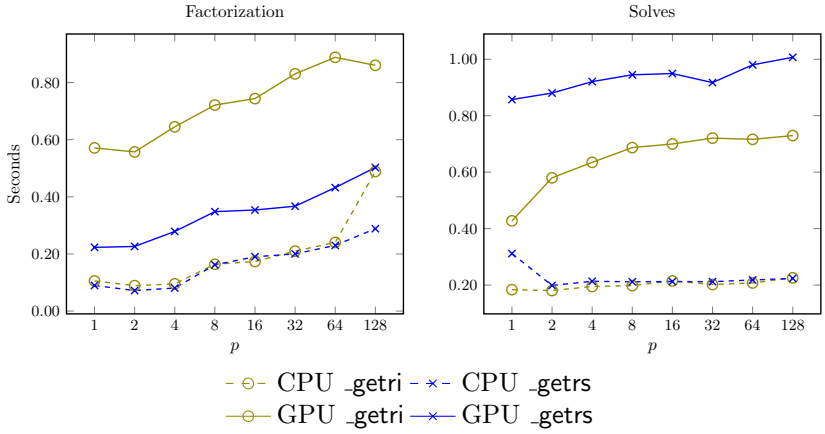

Figure 4.15. Weak scaling for the BCYCLIC algorithm running on CPU and on GPU with the non-batched version with $k = 64$ and $\ell = p \cdot 400$, where p is the number of MPI processes.

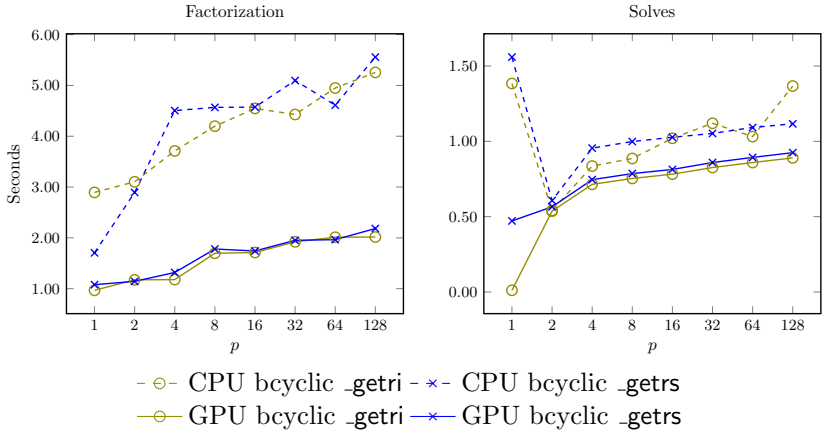

Figure 4.16. Weak scaling for the BCYCLIC algorithm running on CPU and on GPU with the non-batched version with $k = 1024$ and $\ell = p \cdot 25$, where p is the number of MPI processes.

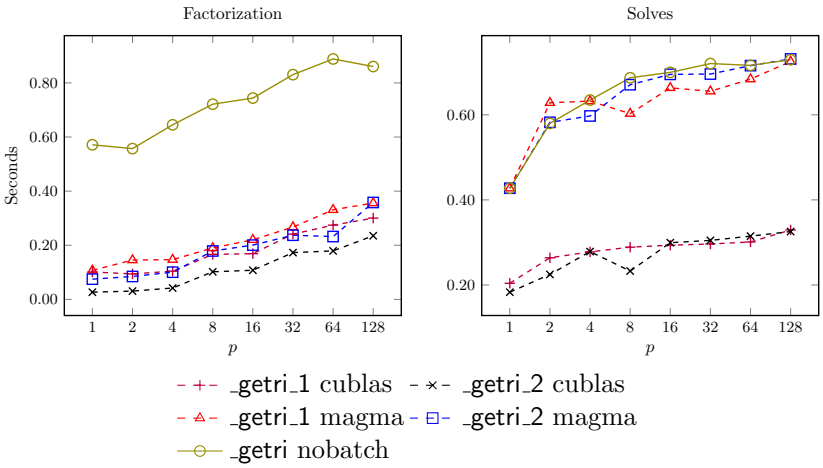

Figure 4.17. Weak scaling for the BCYCLIC algorithm running on GPU with batched and non-batched versions with $k = 64$ and $\ell = p \cdot 400$, where p is the number of MPI processes.

solve stage. These findings made us build a combined version of the software to benefit from the best of each libraries. This hybrid implementation selects the routines to use in the factorization based on the block size of the matrix. More precisely, cuBLAS _getri_2 is used with block sizes up to 128 and MAGMA _getri_1 for larger block sizes. The solve stage is managed by cuBLAS regardless of the block size.

Finally, the third set compares the performance of the BCYCLIC and the Spike algorithms running on the GPU and on the CPU. For the GPU versions, both algorithms use the combined implementation originated from the results of the previous set of experiments. For the sake of simplicity, an exception to the 'select the fastest functions' rule has been made, since for the case of using a block size smaller than 128 and no more than 4 processes the _getrs variant was slightly faster. Note that some executions of the Spike implementation with large block sizes could not run on the GPU due to memory constraints.

Figure 4.19 shows the behaviour of the algorithms with a small block size. Spike scales better than BCYCLIC in the factorization stage for this size and for sizes no larger than 128. That is evident in both the GPU and CPU executions. It starts being slower than BCYCLIC, but as soon as the number of processes is increased, it is able to obtain smaller times, needing a larger number of processes when executed on the GPU.

On the other hand, BCYCLIC scales worse due to the relatively higher cost of communication with respect to the poor computing performance. We can see how the time increases noticeable when using several nodes (more than 4 processes). A block size larger than 128 is required for the BCYCLIC algorithm to perform better than Spike for any number of processes used during the factorization stage.

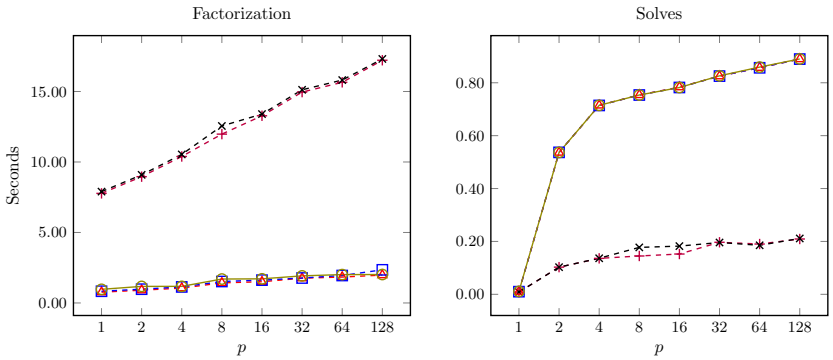

Figure 4.18. Weak scaling for the BCYCLIC algorithm running on GPU with batched and non-batched versions with $k = 1024$ and $\ell = p \cdot 25$, where p is the number of MPI processes. Consult the legend in Figure 4.17.

With all of the block sizes tested, the BCYCLIC algorithm has always performed faster than Spike in the solve stage. For small block sizes, the results highlight the better CPU performance with small BLAS-2 operations and the benefit of the absence of GPU-CPU data copies. From block sizes larger than 128, the payload of calling the kernels is worth the performance obtained with the GPU. On the CPU executions, the time gap between the two algorithms tends to increase when the block size grows, while in the GPU executions it tends to decrease.

If the block size is increased up to 1024 as can be seen in Figure 4.20, the differences between the two algorithms and the two platforms are more prominent.

The different times obtained when varying the block size and the number of processes can be seen in more detail in Tables 4.4 and 4.5. The first one shows the total eigenproblem times obtained for all the different block sizes when using 128 processes. This table does not intend to contrast the performance obtained with different block sizes, as their computational cost differs and are not comparable in that sense. It allows us to compare the behaviour of the implementations for a specific block size. Spike is clearly faster with small block sizes as well as the executions on CPU. Once the block size exceeds 128, the executions on GPU with BCYCLIC obtain the smallest times. The speedup obtained with the GPU implementation with respect to the CPU one is increased when increasing the block size. For such amount of processes, with the largest block size, the GPU speedup of the BCYCLIC is a modest 2.8.

Table 4.5 shows the total eigenproblem times for all the different number of processes when using the largest block size. When increasing the problem size by a factor of 128, the time increasing factor for the fastest implementation (BCYCLIC on GPU) is 3.3, whereas the same algorithm scales slightly better on CPU with a factor of 2.0. Even not being very close to a perfect scaling, these factors are reasonably good and contrast with the Spike factors, that double them. The GPU versions obtain speedups of 4.8 (Spike) and 4.7 (BCYCLIC) with respect to the CPU

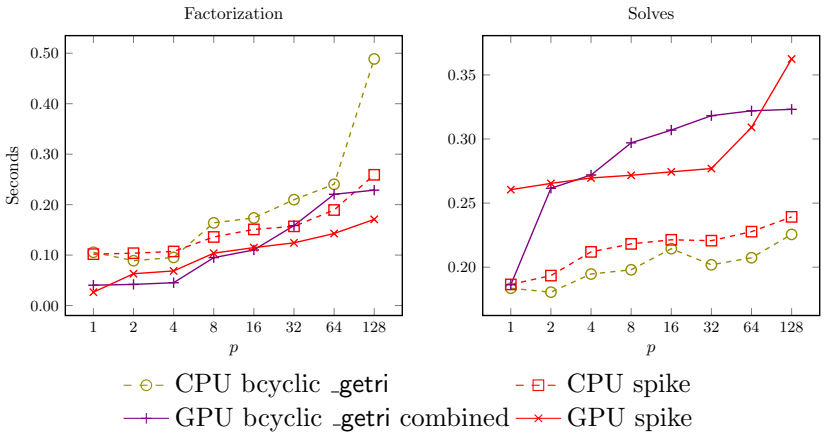

Figure 4.19. Weak scaling for the BCYCLIC and the Spike algorithms running on CPU and on GPU with $k = 64$ and $\ell = p \cdot 400$, where p is the number of MPI processes.

when executing with one process, but those values are reduced when increasing the number of processes.

The total eigenproblem time is also represented in Figure 4.21, in which we can highlight the case of using a block size of 640, where on CPU, the Spike algorithm performed faster than BCYCLIC against the normal behaviour, due to a drop in the performance of BCYCLIC with block sizes between 512 and 1024. From 640 and up to 1024 the BCYCLIC algorithm progressively recovers performance and obtains smaller times with larger block sizes.

The experiments to measure the strong scaling compute 1 eigenvalue of diagonally dominant matrices, and compare the performance of BCYCLIC and Spike with the reduced Spike, used as a direct solver. As matrix A_n coming from (4.33) is not diagonally dominant in general, we have forced it to be artificially diagonally dominant, so the reduced Spike method can be employed.

The figures represent the results obtained when measuring the strong scaling for a fixed matrix size of 307200[4]. For the case of one process, this size is larger than the one used in weak scaling tests, in order to have enough workload with a reasonable number of processes.

Since for the strong scaling tests the time needed to complete the solve stage does not vary significantly between the algorithms, we present the total eigenvalue problem solve operation time in the figures, that have an almost direct correspondence with the time needed to perform the factorization stage and at the same time provide us with a more global view.

Figure 4.22 shows the strong scaling results for three different block sizes. Again, for small block sizes where the GPU has a large overhead launching a lot of small

[4]The actual size of the matrix when using a block size of 896 is 307328.

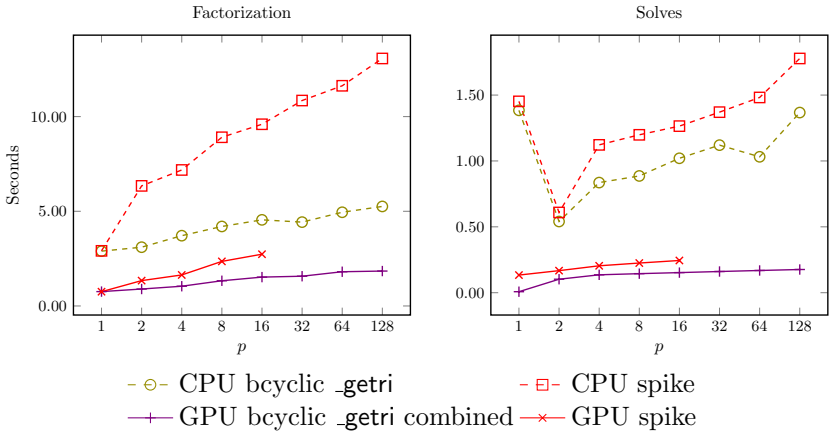

Figure 4.20. Weak scaling for the BCYCLIC and the Spike algorithms running on CPU and on GPU with $k = 1024$ and $\ell = p \cdot 25$, where p is the number of MPI processes. The executions on GPU with the Spike algorithm with more than 16 processes could not be done due to memory constraints.

Table 4.4. Total eigenproblem time obtained with the weak scaling tests for 128 processes.

	CPU		GPU	
	Spike	BCYCLIC	Spike	BCYCLIC
Block size	Seconds	Seconds	Seconds	Seconds
64	**0.67**	1.18	0.68	0.85
96	0.96	1.42	**0.75**	0.81
128	0.85	1.04	**0.75**	0.76
256	1.34	1.88		**1.13**
384	3.04	2.50		**1.06**
512	5.12	3.91		**1.90**
640	7.83	10.46		**6.91**
768	14.05	9.62		**6.03**
896	17.06	8.76		**3.55**
1024	22.08	8.42		**3.01**

Table 4.5. Total eigenproblem time obtained with the weak scaling tests for a block size $k = 1024$.

	CPU		GPU	
	Spike	BCYCLIC	Spike	BCYCLIC
Processes	Seconds	Seconds	Seconds	Seconds
1	4.37	4.29	**0.91**	0.91
2	8.07	3.70	1.68	**1.12**
4	10.22	4.78	2.32	**1.38**
8	13.17	5.57	3.76	**1.85**
16	14.91	6.36	4.55	**2.21**
32	17.22	6.50		**2.39**
64	19.12	7.37		**2.89**
128	22.08	8.42		**3.01**

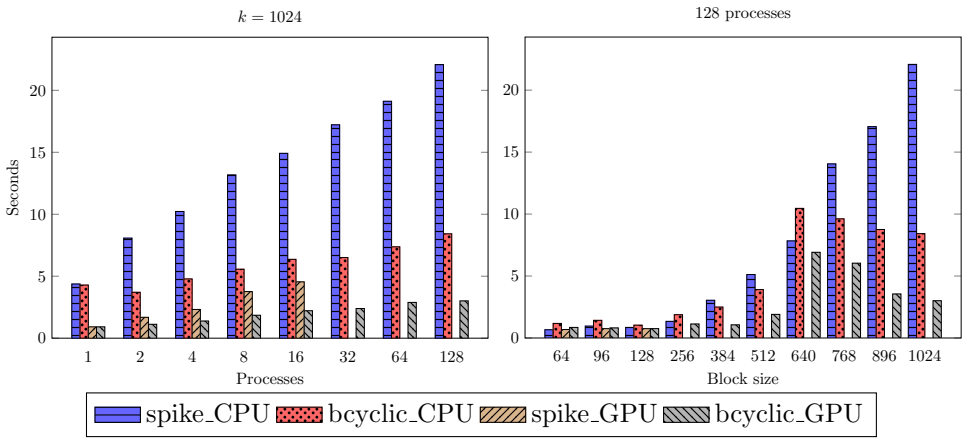

Figure 4.21. Total eigenproblem time obtained with the weak scaling tests for a block size $k = 1024$ (left) and for 128 processes (right).

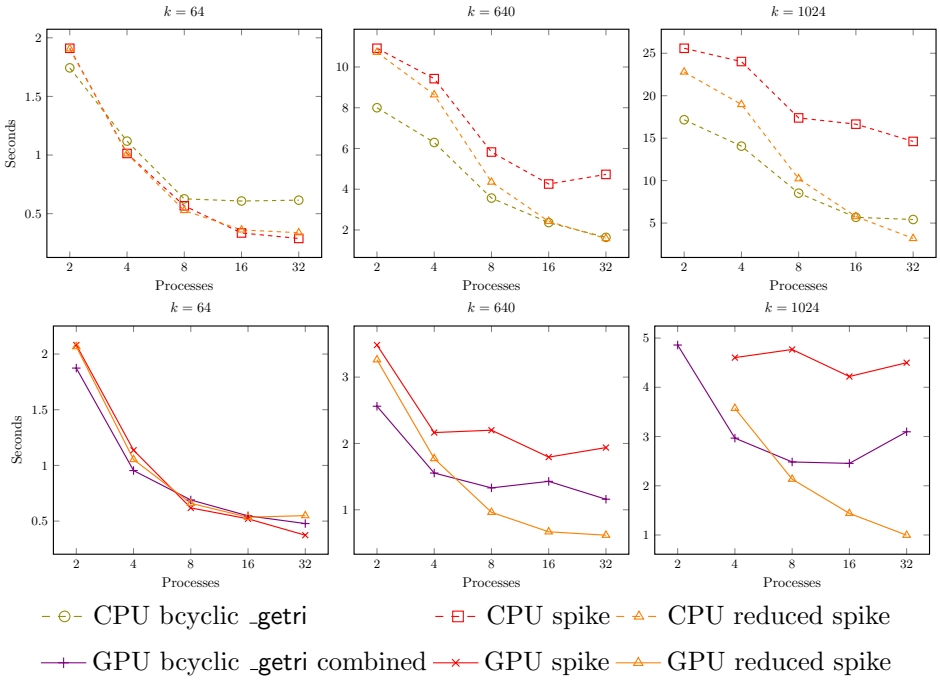

Figure 4.22. Strong scaling for the BCYCLIC, the Spike and the reduced Spike algorithms running on CPU and on GPU with a total matrix dimension of 307200 and different block sizes k.

kernels, the CPU times tend to be smaller. And when the block size is increased, the same algorithm performs faster on GPU. For a block size of 64, the three algorithms scale well up to 8 processes, and from that point on the performance of BCYCLIC decays due to a higher ratio between communication and computation time. Spike achieves a better scalability than the other two algorithms.

The middle and right plots in Figure 4.22 show that the scalability of the algorithms with larger block sizes is not good. Spike turns into the slowest of the algorithms and the one with the worst scalability. On the other side, the reduced Spike benefits of a larger block size scaling up to a larger number of processes where the BCYCLIC algorithm performance starts to decay. As in the weak scaling, the speedup given by the GPU with respect to the CPU is reduced when incrementing the number of processes, and increased when increasing the block size of the matrix. With the largest block size used and four processes, the GPU versions give speedups of 4.7 (BCYCLIC), 5.2 (Spike), and 5.3 (Reduced Spike) with respect to the CPU executions. Those values are reduced to 1.8 (BCYCLIC), 3.3 (Spike), and 3.2 (reduced Spike) when running with 32 processes.

Block cyclic tridiagonal case

The block cyclic tridiagonal case has been studied with an application to provide an accurate picture of the electronic structure of large systems. This kind of applications allows the design of new devices that are essential for many emerging technologies.

In order to understand the electronic and optical properties of semiconductor hetero and nanostructures, it is imperative to know the allowed energy levels for electrons and their quantum state, completely determined by their corresponding wavefunctions. These quantities are provided by the Schrödinger equation, a linear partial differential equation (PDE) which, in its single particle (an electron from now on) time-independent version, reads

$$\hat{H}\psi(\mathbf{r}) = \left[-\frac{\hbar^2}{2m_0}\nabla^2 + V(\mathbf{r})\right]\psi(\mathbf{r}) = E\psi(\mathbf{r}), \tag{4.35}$$

where $\hat{H}$ is the Hamiltonian operator, $\hbar$ is the reduced Planck constant, m_0 is the free electron mass, $V(\mathbf{r})$ is the microscopic potential energy affecting the electron, and $\psi(\mathbf{r})$ and E are the sought electron wavefunction and energy, respectively. When the unknown $\psi(\mathbf{r})$ is expanded as a linear combination of unknown coefficients c_i times known basis functions $\phi_i(\mathbf{r})$,

$$\psi(\mathbf{r}) = \sum_i c_i\phi_i(\mathbf{r}), \tag{4.36}$$

the Schrödinger PDE is transformed into an algebraic (maybe generalized) eigenvalue problem.

In a semiconductor heterostructure, the matrix resulting from the expansion in (4.36) using empirical tight-binding methods (ETB) [122], will have the following characteristics:

- Each atom or primitive cell will contribute with N_b basis functions to the wavefunction, and with an $N_b \times N_b$ block to the matrix A representing $\hat{H}$.
- The topology of the non-vanishing interactions between the atomic-like orbitals will determine which blocks in A will have non-zero entries.

If, in addition, the semiconductor heterostructure has translational symmetry along the x, y coordinates and varies along the z axis (i.e. is effectively one-dimensional), the obtained matrix A will be block-tridiagonal, with the possibility that non-zero blocks appear in the upper-right and lower-left corners if periodic boundary conditions are imposed (block cyclic tridiagonal structure).

A set of experiments to measure the performance of the software have been conducted in Tirant[5]. Two matrices arising from the ETB parametrization [74] of a quantum cascade laser structure [20], and a 4/2 GaAs/AlAs superlattice with fluctuations in the layer widths have been used in the tests. Their dimensions are 82,400 (qcl) and 143,840 (anderson), respectively, and both have a small block size

[5]See hardware description in Chapter 3 (Section 3.4.2).

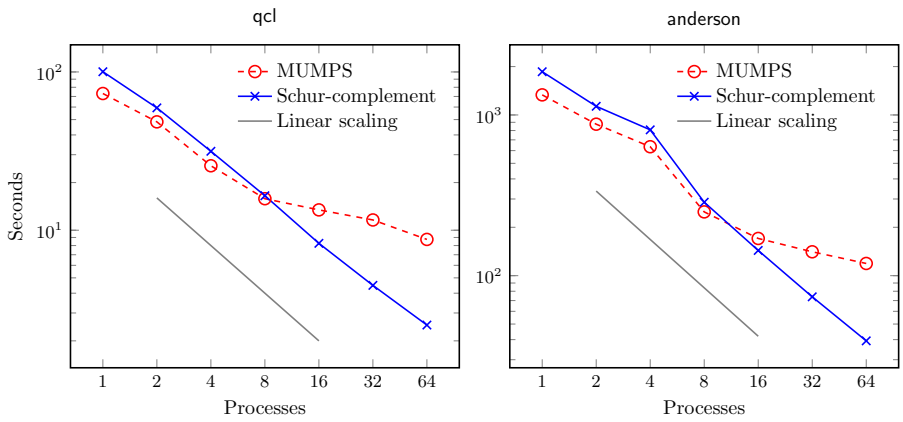

Figure 4.23. Total eigenproblem solve time to obtain 40 eigenvalues closest to 1.5 for the qcl and anderson matrices using 4 processes per node.

of 20. These experiments are done exclusively on CPU because the small block size of the matrices does not make the problem suitable for running it on GPU.

The servers run SuSE Linux Enterprise Server 10 as operating system, and our software has been compiled in complex arithmetic and double precision with gcc 4.6.1 using PETSc and SLEPc 3.7.1, MUMPS 5.0.1-p1, and MPICH2 1.0.8p1 as the inter-process communication library.

As the nodes have four computational cores, a limit of four processes per node have been used during the executions.

In the experiments we compare the performance of a state of the field library such as MUMPS with our implementation of the Schur complement that uses the block-cyclic reduction algorithm to solve the block-tridiagonal system that involves D of (4.27). We should remark that MUMPS stores (and works with) the full matrix (in blocked sparse format) while our software reduces the operations to the nonzero blocks (using dense storage), and except in the case of the sub-matrix C of (4.27), it initially reduces the memory footprint to them. In both cases, we have instructed SLEPc to obtain 40 eigenvalues closest to the target value 1.5 eV (lowest energy states in the conduction band) with the shift-and-invert technique and a default tolerance of 10^{-8}.

The relevant command-line options used in these experiments are:

- MUMPS: `-matload_block_size 20 -eps_nev 40 -eps_target 1.5 -st_type sinvert -st_pc_factor_mat_solver_package mumps`
- Schur complement: `-eps_nev 40 -eps_target 1.5 -st_type shell`

Additionally, for the runs with matrix anderson, the value of the `-eps_ncv` parameter has been increased and set to 128 in order to reduce the number of restarts and achieve a better performance.

Results of the executions can be seen in Figure 4.23. The plots show that the eigensolver scales linearly (up to 64 MPI processes) when using our custom linear solver based on Schur complement with block-cyclic reduction. In contrast, the scalability of MUMPS is more limited, and performance degrades with 16 processes or more. This can be attributed to the fact that MUMPS is based on a classical scheme of factorization followed by triangular solves, where the latter operation is inherently sequential and results in bad scalability. The cyclic reduction scheme rearranges the operations in such a way that there is more opportunity for a larger degree of parallelism.

4.6 Conclusions

In this chapter we have addressed the particular case of solving eigenproblems of large-scale block-tridiagonal and block cyclic tridiagonal matrices. We have developed a set of codes for computing a few eigenpairs of such matrices via Krylov methods. The codes are integrated in the SLEPc/PETSc framework, with an MPI-CUDA programming style, and allow to use many processors/GPUs to address very large-scale problems.

We have focused on the optimization of the basis expansion of the Arnoldi algorithm for which we have developed a high-performance matrix-vector kernel, that is able to fully exploit the computing power of the available GPUs when computing exterior eigenvalues. This solution significantly reduces the computation time when comparing with a multi-threaded CPU version.

For the case of interior eigenvalues, the developing effort has been concentrated on the scalable solution of block-tridiagonal linear systems on the GPU. In the case of the band oriented Spike algorithm, we have adapted it to explicitly work with block-tridiagonal matrices by means of using the block cyclic reduction underneath it.

We have performed a study of the scalability of the algorithms used to see how they react to the changes in the block size of the matrix. The performance analysis allows us to draw several conclusions. In general, BCYCLIC performs better than Spike, but Spike scales better when using small block sizes. All GPU implementations have shown to be faster than the CPU counterparts, except for small block sizes. In terms of scalability, we can state that for sufficiently large block sizes, the codes scale well for up to 128 MPI processes (GPUs). Our implementations can use either cuBLAS or MAGMA, or a combination of the two. The best performance has been obtained with the mixed implementation.

Another conclusion is that, for a large block size, BCYCLIC is able to solve larger problem sizes with respect to Spike, because Spike has larger memory requirements. In the case of diagonally dominant block-tridiagonal matrices, the reduced Spike method achieves better scalability.

We have also addressed the case of coefficient matrices with block cyclic tridiagonal structure and have analyzed the scalability of the linear solver when used within an iterative eigensolver in the context of electronic structure calculations. In

this type of applications, it is important to obtain the solution very fast, especially when the process involves a self-consistency loop that requires solving an eigenvalue problem in each iteration. Our results illustrate that exploiting the matrix structure in the solver may provide some advantage, such as a better scalability, although it implies a higher development effort compared to using general-purpose numerical libraries.

Although the performance analysis were carried out in the context of Krylov eigensolvers, the conclusions could be applied to other applications where a sequence of linear systems with block (cyclic) tridiagonal matrices must be solved, since in our tests almost all the computation is associated with the factorization and linear solves. The solvers can also be used with banded matrices, in which case the off-diagonal blocks are triangular (although we have not exploited this fact).

Chapter 5

Matrix functions

Mel de romer

The area of matrix functions has developed significantly in the last years, both in formalizing the relevant theoretical concepts and in developing algorithms for their computation [64, 65]. Matrix functions can be found in many scientific computing applications, especially the exponential and the square root, but also other less frequent functions such as the logarithm, or trigonometric functions.

Matrix functions can be understood in several ways. For this, it is necessary to state to which of them we are referring to. Throughout this chapter we will interpret matrix functions as those that map $\mathbb{C}^{n\times n}$ to $\mathbb{C}^{n\times n}$ and are defined in terms of a scalar function f. For instance, the matrix exponential $\exp(A)$ is defined in terms of the scalar exponential function $\exp(z)$ in the sense discussed below, not in other senses such as element-wise evaluation. Note that functions of a matrix like the determinant, that yield a scalar result instead of mapping $\mathbb{C}^{n\times n}$ to $\mathbb{C}^{n\times n}$, or a function such as $X \mapsto AX^2 + BX + C$ where A, B, C are also matrices are not included in the definition and not considered in this chapter.

There are several equivalent definitions for such matrix functions. One definition is based on the Jordan canonical form of the matrix, $A = ZJZ^{-1}$. Then

$$f(A) := Z \operatorname{diag}\left(f(J_1(\lambda_{i_1})), \ldots, f(J_p(\lambda_{i_p}))\right) Z^{-1}, \tag{5.1}$$

where $J_k(\lambda_{i_k})$ is the Jordan block corresponding to the eigenvalue λ_{i_k}, and $f(J_k(\lambda_{i_k}))$ can be defined in terms of the successive derivatives of f evaluated on λ_{i_k}

$$f(J_k(\lambda_{i_k})) = \begin{bmatrix} f(\lambda_k) & f'(\lambda_k) & \ldots & \frac{f^{(m_k-1)}(\lambda_k)}{(m_k-1)!} \\ & f(\lambda_k) & \ddots & \vdots \\ & & \ddots & f'(\lambda_k) \\ & & & f(\lambda_k) \end{bmatrix}. \tag{5.2}$$

Alternatively, matrix functions can be defined in terms of a polynomial

$$f(A) := r(A), \tag{5.3}$$

where r interpolates f in the Hermite sense at the eigenvalues λ_{i_k}. And also by means of the Cauchy integral formula

$$f(A) = \frac{1}{2\pi i} \int_\Gamma f(z)(zI - A)^{-1} \mathrm{d}z, \tag{5.4}$$

assuming f is analytic in the considered domain Γ. All these definitions give rise to different computational methods.

As in other numerical linear algebra problems, such as linear systems or eigensystems, there are two broad classes of methods that we can generically refer to as dense and sparse. For dense matrices of relatively small size we can afford to explicitly compute the matrix

$$F = f(A) \tag{5.5}$$

with dense methods. The computational cost of these methods depends cubically on n, the matrix size. For sparse matrices, whose size is usually much larger, it is not possible to explicitly compute (or even store) the matrix F of (5.5), so the alternative is the implicit application of the matrix function to a given vector,

$$y = f(A)b. \tag{5.6}$$

The cost in this case is less than cubic, but this cost is accounted for every different vector b to which the matrix function is applied.

In this chapter we discuss CPU and GPU implementations for both the dense and the sparse case. Along the next sections we review the available types of methods to perform the dense computation of matrix functions, and present some of them to compute matrix square roots and exponentials that can be efficiently implemented on GPU. For the sparse case of (5.6) we will center our attention on Krylov methods for performing its computation in a iterative way, where A is typically large and sparse, or is available implicitly by means of a matrix-vector multiplication subroutine. We will also point out how Krylov methods are related with the dense computation, and the impact that the implementation of the latter may have on the performance of the sparse computation. Finally, we show some results comparing the performance of the selected methods with executions on CPU and on GPU.

5.1 Dense methods for matrix functions

There are many different methods for the dense computation of $f(A)$. Here we present three classes of methods that are relevant for our purposes:

Similarity transformation. Computing $f(A)$ via (5.1) is not viable due to the difficulty of obtaining the Jordan form in a numerically stable way. An alternative is to compute the Schur form (3.6), $A = QTQ^*$, with Q orthogonal (or

unitary in the complex case), and then evaluate the matrix function of the (quasi-)triangular matrix T,

$$f(A) = Qf(T)Q^*. \tag{5.7}$$

In the case that A is symmetric (or Hermitian), it is even simpler because T is then diagonal. The latter case will be referred to as diagonalization.

Rational approximation. Approximating $f(A)$ can be tackled by replicating the procedures used on scalar functions. One of the main rational approximations are based on Padé approximants.

Matrix iterations. Some matrix functions like matrix roots or the sign function are amenable to be computed with iterations derived from Newton's method.

Methods based on similarity transformation are appealing since they usually require less floating-point operations than methods of the other types. However, that same characteristic makes them less likely to take advantage of the GPU.

Previous works have compared different algorithms for matrix functions, for instance [25] analyzes Matlab implementations for the matrix exponential. Our intention is not simply to carry out a comparison, something inescapably in the developing process, but to provide robust and efficient implementations for both CPU and GPU, that can be used from sequential or parallel codes written in C, C++ or Fortran. We have implemented four methods for the matrix square root and three for the matrix exponential. All of them have been implemented for both CPU and GPU, except the Schur method for the square root that is only available for CPU.

5.1.1 Square root

A square root of A is any matrix satisfying the matrix equation

$$F^2 = A. \tag{5.8}$$

If A has no eigenvalues on $\mathbb{R}^-$, the closed negative real axis, there is a unique *principal* square root, denoted as $A^{1/2}$ or $\sqrt{A}$, whose eigenvalues have all positive real part. The methods discussed below compute the principal square root.

As mentioned above, an interesting strategy is to first reduce A to the Schur form, $A = QTQ^*$. The Schur-Parlett method [65] uses a recurrence to evaluate $f(T)$ exploiting the (quasi-)triangular structure of T. Implementing the Parlett recurrence in a numerically stable way is tricky. Fortunately, in the case of the matrix square root the method can be simplified, as the Parlett recurrence is not necessary, and hence it is simply called Schur method [62]. In this method the diagonal elements are determined as $\sqrt{t_{ii}}$ and the off-diagonal ones can be obtained from the equation $X^2 = T$. Once X, the square root of T, has been obtained, a final backtransform step is necessary to recover F.

We have implemented a blocked variant of this method as described in [34] that runs on CPU, see Algorithm 5.1. Although when solving on CPU, the _gemm calls

are the most time-consuming operations, this method is not appropriate for implementation on GPU, due to the need of reduction to the Schur form and the steps after it, that involve using level 2 BLAS to solve Sylvester equations.

Algorithm 5.1. Blocked Schur method for the square root

```
1 Compute (real) Schur decomposition A = QTQ* ;
2 for j = 1, 2, ..., n_blk do
3 |   Evaluate X_jj = T_jj^{1/2} ;
4 |   for i = j − 1, ..., 1 do
5 |   |   Solve Sylvester eq. X_ii X_ij − X_ij X_jj = T_ij − Σ_{k=j+1}^{i−1} X_ik X_kj
6 |   end
7 end
8 Backtransform F = QXQ*
```

The other methods that we have considered to compute the matrix square root are based on matrix iterations. Iterative methods are interesting to implement as they are easily built with common matrix operations, and are particularly rich in matrix-matrix products. Those characteristics make them a suitable choice for GPU computing.

The Newton method is the basis of many existing iterative methods. When applied to the matrix equation of (5.8), it gives the recurrence

$$F_{k+1} = \frac{1}{2}(F_k + F_k^{-1}A), \quad F_0 = A. \tag{5.9}$$

For A having no eigenvalues on $\mathbb{R}^-$, the recurrence converges quadratically to the principal square root $A^{1/2}$ for F_0 sufficiently close to it. Nevertheless, this iteration is numerically unstable and is not useful for practical computation. It is necessary to rewrite the iteration in a different way to make it stable. Several variants of the Newton method that stabilize the iteration have been developed, and they often scale the F_k terms to reduce the steps until the quadratic convergence starts.

One variant of the Newton method is the product form [29] of the Denman–Beavers iteration [35], with a scaling factor μ_k:

$$\mu_k = |\det(M_k)|^{-1/(2n)}, \tag{5.10a}$$

$$M_{k+1} = \frac{1}{2}\left(I + \frac{\mu_k^2 M_k + \mu_k^{-2} M_k^{-1}}{2}\right), \qquad M_0 = A, \tag{5.10b}$$

$$F_{k+1} = \frac{1}{2}\mu_k F_k (I + \mu_k^{-2} M_k^{-1}), \qquad F_0 = A. \tag{5.10c}$$

Contrary to the recurrence (5.9), that solves a multiple right-hand side linear system per iteration, (5.10) requires to compute a matrix inverse and, in addition, obtaining the scaling factor μ_k also requires computing the determinant of M_k.

Following Higham's reference implementation[1], the scaling can be stopped when $\|F_k - F_{k-1}\|/\|F_k\| < 10^{-2}$. In our implementation, the determinant is computed from the LU factorization of M_k (_getrf). However, for the computational experiments in this section we have turned off the scaling in this method, as the determinant may suffer from overflow or underflow in finite precision arithmetic when the matrix is quite large. The scaling factor μ_k may accelerate the initial convergence, but the results obtained without scaling are equally accurate. Alternative scaling factors discussed in [64, Ch. 5] in the context of the matrix sign function are not suitable for the case of the square root.

Another method for computing the square root is Newton–Schulz [117], from the Padé family of iterations. It usually needs more iterations to converge, but as it is an inverse-free iterative method, it does not rely on having an efficient implementation of the matrix inverse, having the matrix-matrix product as its main operation. It consists of two coupled recurrences,

$$X_{k+1} = \frac{1}{2} X_k (3I - Z_k X_k), \qquad X_0 = B, \tag{5.11a}$$

$$Z_{k+1} = \frac{1}{2} (3I - Z_k X_k) Z_k, \qquad Z_0 = I. \tag{5.11b}$$

The initial guess $B = A/\|I-A\|_\ell$ is a scaled version of A, so scaling must be undone after convergence, $F = \sqrt{\|I - A\|_\ell} X$. The condition $\|I - A\|_\ell < 1$ is sufficient for the method to converge for $\ell = 1, 2$ or ∞. The scaling of the initial matrix may help the convergence of the method, although it does not guarantee it.

Recently, a cubically convergent iterative method for computing the square root has been developed by Sadeghi [116]. It starts with $B = A$ (or $B = A/\|A\|$ in case $\rho(A) > 1$), and builds two coupled recurrences

$$X_{k+1} = X_k \cdot \left(\frac{5}{16} I_n + \frac{1}{16} M_k (15 I_n - 5 M_k + M_k^2) \right), \qquad X_0 = I, \tag{5.12a}$$

$$M_{k+1} = M_k \cdot \left(\frac{5}{16} I_n + \frac{1}{16} M_k (15 I_n - 5 M_k + M_k^2) \right)^{-2}, \qquad M_0 = B, \tag{5.12b}$$

being $F = \sqrt{\|A\|} X$ when the initial guess has been scaled or $F = X$ otherwise. This method, like (5.10), requires computing an inverse in each iteration. But as it converges faster, the number of iterations needed to compute the square root is expected to be smaller.

In our implementations, we use the Frobenius norm for scaling as well as for convergence tests.

[1] The Matrix Function Toolbox, http://www.ma.man.ac.uk/~higham/mftoolbox/

5.1.2 Sign

Assuming that A is non-singular and has no eigenvalues on the imaginary axis, the matrix sign function $\text{sign}(A)$ can be defined. Let

$$A = Z \begin{bmatrix} J_n & \\ & J_p \end{bmatrix} Z^{-1} \tag{5.13}$$

be the Jordan canonical form of A, where J_n and J_p contain the Jordan blocks of eigenvalues with negative and positive real parts, respectively, being n and p their respective dimensions. Then the sign function of A is defined as

$$\text{sign}(A) = Z \begin{bmatrix} -I_n & \\ & I_p \end{bmatrix} Z^{-1}. \tag{5.14}$$

A different representation of the matrix sign function, introduced in [63], shows its link with the matrix square root:

$$\text{sign}(A) = A(A^2)^{-1/2}. \tag{5.15}$$

Although there are specific Newton-type iterations for $\text{sign}(A)$ [64, Ch. 5], here we discuss its computation using square root solvers. Since A is assumed to be non-singular, $(A^2)^{-1/2}$ of (5.15) can be obtained. Some of the methods for the square root mentioned above can be used to obtain the inverse square root. In particular, in Denman–Beavers (5.10) the sequence F_k converges to $A^{-1/2}$ if the iteration starts with $F_0 = I$, and in Newton–Schulz (5.11) the sequence Z_k converges to $B^{-1/2}$ so both the square root and its inverse are obtained simultaneously. If using another method to compute the square root, like Schur or Sadeghi, the inverse square root can be obtained by additionally solving a system of linear equations with multiple right-hand sides

$$AF = A^{1/2}, \tag{5.16}$$

where $F = A^{-1/2}$.

5.1.3 Exponential

The matrix exponential is a function of major importance due to its connection with solving linear differential equations [45], and due to this, this matrix function has been widely studied and many methods for computing it have been proposed [97]. For the matrix exponential, we use a rational function $r(A)$ to approximate $\exp(A)$. This rational function is chosen in a similar way as in scalar approximation theory, although in the case of matrix functions there is no guarantee that the approximation will be good (depending on the spectral properties of A) [65].

The first method that we have implemented is a rational approximation based on Padé approximants of order $[p/p]$ with $p = 6$, combined with scaling and squaring. The notation $[\cdot/\cdot]$ indicates the polynomial degree of the numerator and denominator, which are equal in this case (diagonal Padé approximant). The evaluation of the

numerator and denominator is done following the Horner scheme, to reduce the number of required matrix-matrix multiplications. The last step for computing the rational function is done with a linear solve with multiple right-hand sides. The scaling and squaring technique consists in determining the minimal integer $s \geq 0$ such that $\|A/2^s\|$ is smaller than a certain constant, then scale the matrix as $A/2^s$ prior to the computation of the rational matrix function. In that case, a post-processing is required to form F^{2^s}, which is done with s additional matrix-matrix multiplications.

The above technique is very close to one of the methods used in Expokit [120]. A more recent work [3] suggests using higher degree Padé approximants (up to degree 13), but rearranging the computation in such a way that the number of required matrix products is much smaller. This approach (included in the latest versions of Matlab) is more accurate in some cases, and also avoids choosing a too large value of s (overscaling). Our second implementation follows this approach, and will be referred to as Higham.

A third method, presented by Güttel and Nakatsukasa in [55], also based on Padé approximations, tries to reduce its cost by using a subdiagonal Padé approximant of low degree (such as [3/4]), and also uses a small scaling and squaring factor to avoid potential instability caused by overscaling. Before starting, the method shifts the matrix

$$A_\sigma = A - \sigma I, \tag{5.17}$$

being σ the real part of the rightmost eigenvalue of A, so that all the eigenvalues of A_σ are in the left half plane. It is supposed to perform well with an estimation of such eigenvalue, assuming that the rightmost eigenvalues do not have widely varying imaginary parts. If the matrix is stable, with no eigenvalues in the right half plane, the shift is not required. A drawback of this method is that it employs complex arithmetic even when A is real, so its performance with real matrices is not expected to be good when compared with methods that do the computation with real scalars. This last implementation is referred to as Güttel–Nakatsukasa.

5.2 Krylov methods for matrix functions

Krylov methods, as in the case of computing eigenvalues, basically amount to building an Arnoldi decomposition and evaluating the matrix function on the computed Hessenberg matrix. In an MPI parallel implementation of this method, the former can be implemented efficiently, but the latter is a dense matrix computation, that must be done redundantly by all MPI processes. This redundant computation is in the critical path, so it may hinder performance if the size of the Hessenberg matrix is not very small, as it may happen when using restart techniques. In this scenario, having GPU implementations of the methods of Section 5.1 is very interesting as it may help to improve the overall scalability of parallel Krylov solvers.

Other authors have considered the topic of GPU calculation of sparse matrix functions, particularly the case of computing $\exp(tA)b$ with iterative methods [43, 139]. Here, we cover both the exponential and the square root for the sparse case. In

this section, we discuss the specific method that we have considered for this problem, in the context of the SLEPc library.

5.2.1 Restarted Arnoldi

Krylov methods [64, Ch. 13] are appropriate for the case of large and sparse A, although they are not the only alternative (see, e.g. [25]). They approximate the result vector $y = f(A)b$ of (5.6) by an element of the Krylov subspace

$$\mathcal{K}_m(A,b) \equiv \text{span}\left\{b, Ab, A^2b, \ldots, A^{m-1}b\right\}, \tag{5.18}$$

without explicitly building the matrix $f(A)$. The computation comprises two parts. The first part is to build an orthonormal basis V_m of the Krylov subspace by means of the Arnoldi method. The computed quantities satisfy the relation

$$AV_m = V_m H_m + h_{m+1,m} v_{m+1} e_m^*, \tag{5.19}$$

where H_m is an $m \times m$ upper Hessenberg matrix. The second part is to compute the approximation of y as

$$y^{(0)} = \beta V_m f(H_m) e_1, \tag{5.20}$$

where $\beta = \|b\|_2$, and e_1 is the first coordinate vector. In this way, the problem of computing the function of a large matrix A of order n is reduced to computing the function of a small matrix H_m of order m, with $m \ll n$.

In an MPI parallel implementation of this method, the main ingredients of the first part are the parallel sparse matrix-vector product and the orthogonalization of the v_j vectors (the columns of V_m). These two operations can be implemented efficiently and in a scalable way, provided that A has an appropriate sparse pattern. The second part, the evaluation of $f(H_m)$ as a dense matrix computation, must be done redundantly by all MPI processes. Although this computation is in the critical path, it is usually negligible whenever m, the size of the Hessenberg matrix H_m, is much smaller than the size of A. However, this is not always the case, as we now discuss.

The value of the m parameter is difficult to choose. If m is too small the Krylov subspace will not contain enough information to build an accurate approximation. And if m is too large, the memory requirements for storing V_m (as well as the associated computational cost) will be prohibitive. In a practical implementation, a *restarted* variant of the method must be used, where m is prescribed to a fixed value and when the subspace reaches this size, a restart is carried out by keeping part of the data computed so far and discarding unnecessary information. We use the Eiermann-Ernst restart [39], in which only the last basis vector v_{m+1} is kept (to continue the Arnoldi recurrence), along with the matrix H_m that is *glued* together with the previous ones. More precisely, the approximation of y is improved at each restart by an additive correction. After k restarts, the approximation is updated as $y^{(k)} = y^{(k-1)} + c^{(k)}$, where the correction is

$$c^{(k)} = \beta V_m [0, I_m] f(H_{km}) e_1. \tag{5.21}$$

The upper Hessenberg matrix H_{km} is obtained by extending the one from previous restarts,

$$H_{km} = \begin{bmatrix} H_{(k-1)m} & 0 \\ h_{m+1,m}^{(k-1)} e_1 e_{(k-1)m}^T & H_m^{(k)} \end{bmatrix}, \tag{5.22}$$

where $H_m^{(k)}$ is the matrix computed by Arnoldi in the kth restart. The stopping criterion can be based on the norm of the correction: $\|c^{(k)}\| < \beta \cdot \text{tol}$. We use a value $\text{tol} = 10^{-8}$ in the tests of Section 5.3.2.

Note that the matrix of (5.22) has increasing size $k \cdot m$, where k is the number of restarts. In practical applications, this size can often become as high as a few thousands. Note also that this matrix is not symmetric even if A is symmetric.

5.3 Numerical experiments

The methods that we have considered in this chapter are not new, but we are interested in furnishing SLEPc with efficient implementations to solve the function of a matrix (5.5) and the action of a matrix function on a vector (5.6). In the case of dense functions, the implementations are coded as a sequence of function calls to libraries such as BLAS, relying on their high efficiency either on the CPU or on the GPU. For solving sparse functions we rely on the efficient implementation of the parallel matrix-vector multiplication in PETSc, as well as in our implementations of methods for computing dense matrix functions. Both of them are amenable to run on CPU or GPU. Our implementations can operate with either real or complex scalars, in single or double precision arithmetic, but we restrict the experiments to double precision only.

Matrix functions are not as common in applications as other linear algebra problems such as linear systems or eigenvalue problems. For the evaluation of the solvers we have used matrices from well-known collections and from some sample applications with the aim of showing that different matrix functions appear in different contexts. Before presenting each set of results, we provide a description of the corresponding applications from which the matrices arise. Note that for the selected applications it is sufficient to compute the action of the matrix function on a vector, so sparse methods should be employed on them. Still, we also use these matrices as use cases to test the dense algorithms, since they are more representative of real situations than, e.g. random matrices.

5.3.1 Computational evaluation of dense solvers

We have analyzed the performance of the dense solvers implementations by conducting several tests on the Minotauro supercomputer presented in Chapters 3 and 4. We comment again the characteristics of the two types of nodes of the clusters that compose Minotauro:

Fermi 2 Intel Xeon E5649 processors (6 cores per processor) at 2.53 GHz with 24 GB of main memory; 2 GPUs NVIDIA Tesla M2090, with 512 cores and 6 GB

GDDR per GPU.

Kepler 2 Intel Xeon E5-2630 v3 processors (8 cores per processor) at 2.4 GHz with 128 GB of main memory; 2 NVIDIA K80 cards (2 GPUs each), with 2496 cores and 12 GB GDDR per GPU.

On both platforms, the codes are compiled with GCC 5.1.0, and linked against PETSc 3.8, SLEPc 3.8, CUDA 8.0, MAGMA 2.2.0 and MKL 11.3.2. The algorithm's implementations are built using BLAS and LAPACK operations as main computational blocks. In general, the GPU implementations make use of cuBLAS to perform the BLAS operations, and MAGMA for the LAPACK routines. Some auxiliary kernels are used for simple, non computationally intensive operations, like setting or modifying the diagonal elements of a matrix.

Running on two platforms with consecutive generations of GPUs, allows us not just to see the gain obtained with their use, but also the evolution of such cards compared with their coetaneous generations of CPUs.

When using sparse matrices during the experiments of this section, they are stored and treated as dense. As PETSc does not have a type for dense matrices on the GPU, we have to manage the memory and the transfer of the data between CPU and GPU. We make use of PETSc's logging functionality to measure the time and flops rate of the solvers. All the tables in this section show the total elapsed time needed to compute the respective (dense) matrix function and the achieved performance in gigaflops per second. The time includes, in all cases, the memory allocation (and deallocation) of the auxiliary variables needed, and in the case of the GPU implementations, also the copies to and from the device memory. The reported times are the minimum from three independent executions working in double precision arithmetic. All the experiments with dense solvers consist in a CPU execution, using as many threads as available cores (12 threads in Fermi and 16 threads in Kepler), and a GPU execution using only one of the available GPUs. The size of the matrices used in the tests is limited by the memory available in the GPU of the Fermi platform (6 GB).

Sample applications (1)

Data assimilation Numerical weather prediction relies on modern data assimilation techniques for merging satellite observations (of order 10^5 or more) with model forecasts in order to improve the initial conditions and hence obtain more accurate results. The EnSRF method [140] is a variant of Ensemble Kalman Filter used with deterministic observations that includes a matrix square root to account for the uncertainty of the unperturbed ensemble observations.

For the ensemble mean x_m and ensemble perturbations X_A, the square-root observation filter can be written as

$$x_m^{(a)} = x_m^{(f)} + K(y - HX), \tag{5.23a}$$

$$X_A^{(a)} = X_A^{(f)} + \tilde{K}(0 - HA), \tag{5.23b}$$

where the superscripts (a) and (f) denote the analysis and the previous forecast, respectively. Vector y contains the observations. The traditional Kalman gain is

$$K = C_{x,y} D^{-1}, \tag{5.24}$$

with $D = C_{y,y} + R$, where $C_{x,y}$ and $C_{y,y}$ are covariance matrices and R the observation error covariance. The correction from using unperturbed observations is

$$\tilde{K} = C_{x,y} D^{-1/2} (\sqrt{D} + \sqrt{R})^{-1}, \tag{5.25}$$

which simplifies to

$$\tilde{K} = C_{x,y} (D + \sqrt{D})^{-1}. \tag{5.26}$$

Further details can be found in [127].

Stability of dynamical systems In control theory and many other contexts, it is important to determine if a dynamical system is stable or not, and this question can often be formulated as a problem involving eigenvalues of matrices. In this context, a matrix A is considered to be a stable matrix if all its eigenvalues lie in the open left half plane, $\mathrm{Re}[\lambda_i] < 0$ for all i. As suggested in [64, §2.5], the matrix sign function can be used to count the number of eigenvalues of a matrix located in a particular region of the complex plane, and determine if a system is stable or unstable. The matrix sign function is related to the matrix square root, as discussed in Section 5.1.2.

Dense square root experiments

Two matrices are considered for the matrix square root experiments. The first one is ensrf7864, a symmetric positive-definite matrix from the data assimilation application, with dimension 7864, for which the square root is computed.

The second one is rdb5000, a sparse non-symmetric matrix belonging to the NEP collection [11], with dimension 5000, which is used to compute the matrix inverse square root, as part of the computation of the matrix sign function for determining the stability of a dynamical system.

Apart from the performance data, the tables for the square root tests (Tables 5.1 and 5.2) also show the relative error of the computed solution, $\|F^2 - A\|_F / \|A\|_F$ for the square root and $\|F^2 A - I\|_F / \|A\|_F$ for the inverse square root. According to Higham [64, §6.1], the best relative error we can expect for the numerical computation of the matrix square root is of order $\alpha(F)u$, where u is the unit round-off and $\alpha(F) = \frac{\|F\|^2}{\|A\|}$. We have checked that in the considered matrices this quantity is of order 10^1, and hence the maximum expected accuracy in double precision would be around 10^{-15}. The tables also show the number of required iterations in the case of iterative methods. Figure 5.1 summarizes the square root and inverse square root executions, on which the fastest method is Denman–Beavers running on GPU.

Table 5.1 shows the results for the square root. Given that the matrix is symmetric, $f(A)$ can be computed as $Q\,\mathrm{diag}(f(\lambda_i))Q^*$. This diagonalization method is

Table 5.1. Results for computing the matrix square root of the ensrf7864 matrix. Time expressed in seconds. GF/s indicates gigaflops per second, Iter indicates iterations done, and Error is computed as $\|F^2 - A\|_F / \|A\|_F$.

		CPU				GPU			
Platform	Algorithm	Time	GF/s	Iter.	Error	Time	GF/s	Iter.	Error
Fermi	Diagonalization	196.9	27	-	$1.5 \cdot 10^{-14}$	-	-	-	-
	Denman–Beavers	208.3	93	10	$4.2 \cdot 10^{-14}$	56.6	344	10	$9.3 \cdot 10^{-14}$
	Newton–Schulz	489.4	101	17	$7.4 \cdot 10^{-15}$	127.0	391	17	$3.0 \cdot 10^{-14}$
	Sadeghi	346.5	101	6	$3.9 \cdot 10^{-15}$	93.7	374	6	$2.4 \cdot 10^{-14}$
Kepler	Diagonalization	37.3	143	-	$1.2 \cdot 10^{-14}$	-	-	-	-
	Denman–Beavers	74.4	262	10	$4.4 \cdot 10^{-14}$	25.8	754	10	$9.1 \cdot 10^{-14}$
	Newton–Schulz	135.0	367	17	$6.5 \cdot 10^{-15}$	49.6	1001	17	$3.0 \cdot 10^{-14}$
	Sadeghi	88.2	397	6	$3.9 \cdot 10^{-15}$	38.6	907	6	$2.4 \cdot 10^{-14}$

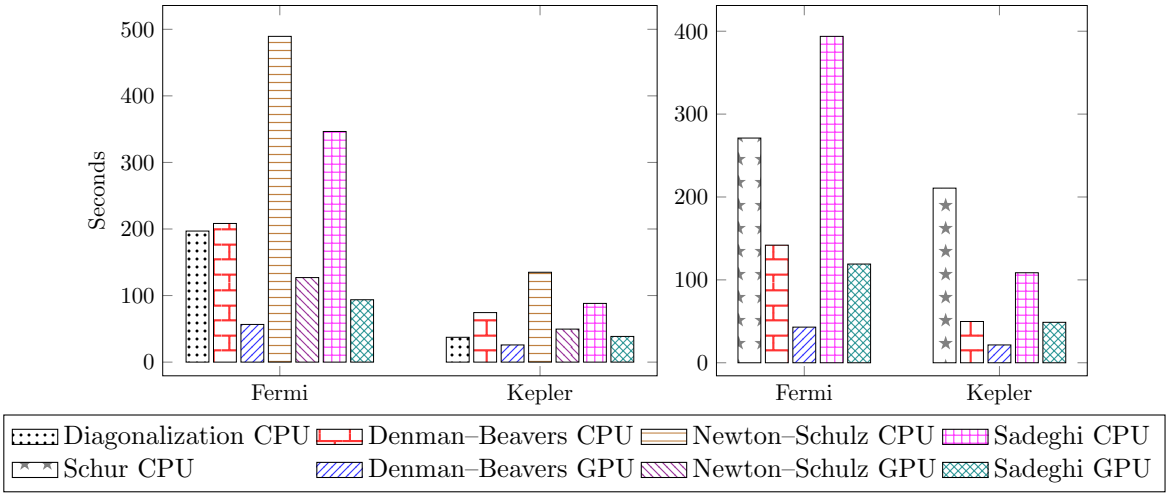

Figure 5.1. Time results for computing the matrix square root (left) and the matrix inverse square root (right).

Table 5.2. Results for computing the matrix inverse square root of the rdb5000 matrix. Time expressed in seconds. GF/s indicates gigaflops per second, Iter indicates iterations done, and Error is computed as $\|F^2A - I\|_F / \|A\|_F$.

		CPU				GPU			
Platform	Algorithm	Time	GF/s	Iter.	Error	Time	GF/s	Iter.	Error
Fermi	Schur	271.1	26	-	$2.3 \cdot 10^{-12}$	-	-	-	-
	Denman–Beavers	141.9	85	12	$6.6 \cdot 10^{-16}$	43.0	279	12	$1.6 \cdot 10^{-15}$
	Sadeghi	393.8	101	13	$6.9 \cdot 10^{-13}$	119.1	333	13	$1.6 \cdot 10^{-12}$
Kepler	Schur	210.7	33.6	-	$2.0 \cdot 10^{-12}$	-	-	-	-
	Denman–Beavers	49.8	241.0	12	$5.9 \cdot 10^{-16}$	21.5	558	12	$1.6 \cdot 10^{-15}$
	Sadeghi	108.6	365.3	13	$7.1 \cdot 10^{-13}$	48.8	814	13	$1.6 \cdot 10^{-12}$

the fastest one for the CPU runs, and the ranking between methods is the same on both platforms, but with notable differences with respect to their relative execution time.

The relative behaviour of the methods does not differ substantially between CPU and GPU executions. On the Fermi platform, the GPU runs obtain speedups ranging from 3.6 to 3.8 with respect to the CPU ones, and a speedup of 3.5 is obtained if comparing the fastest method on GPU (Denman–Beavers) with the fastest one on CPU (diagonalization). On the Kepler platform, the speedups achieved with the GPU runs are smaller, going from 2.3 to 2.9. And the comparison between the fastest methods on CPU and GPU gives a speedup of only 1.4.

Although Newton–Schulz is the method that attains the highest Gflop/s rate in almost all the cases due to the matrix-matrix operations, it is not competitive in terms of execution time because of the larger number of required iterations. Neither is Sadeghi despite its cubic convergence that allows it to terminate in only six iterations.

Table 5.2 shows the results of the inverse square root with the non-symmetric matrix. The implementation of the blocked Schur method employed here uses a block size of 64. The Newton–Schulz method does not converge with this matrix, so no results are shown for it. The poor error obtained with Schur and Sadeghi comes from the final linear solve to obtain the inverse, not from computing the matrix square root. This additional step also implies increasing the time needed to obtain the inverse square root. Denman–Beavers is the fastest method on CPU and GPU on both platforms, achieving speedups of 3.3 on Fermi and 2.3 on Kepler.

Sample applications (2)

Time evolution in quantum problems The time-dependent Schrödinger equation

$$i\frac{\partial}{\partial t}\Psi(t) = H(t)\Psi(t) \tag{5.27}$$

can be solved numerically with a time-stepping scheme, where at time t_n

$$\Psi(t_n) = e^{-iH(t_n)\Delta t}\,\Psi(t_{n-1}) \tag{5.28}$$

with $\Delta t = t_n - t_{n-1}$. See [91] for an example that uses the matrix exponential in SLEPc for the simulation of quantum systems.

Here we will use a very simple problem where the Hamiltonian H is constant in all time steps, consisting in N spins in a uniform transverse field, with disordered potential. The dimension of the Hamiltonian matrix is 2^N.

We emphasize that complex arithmetic is required to compute $\exp(-iH\Delta t)$, even if H is real.

Migration modeling In the last years, population genomic datasets have been collected, making it possible to use these data to infer demographic histories of populations, e.g. under models of migration and divergence. The method presented in [75] relies on a Markov chain representation, and uses the matrix exponential to obtain probability distributions at different times. More precisely, if M is the transition matrix, the vector $\pi(t)$ of probabilities of being in each state of the Markov chain at time t is

$$\pi(t) = \pi(0)\,e^{tM}. \tag{5.29}$$

Dense exponential experiments

Four matrices are considered for the matrix exponential tests. The first two belong to the Harwell-Boeing collection [38] and are commonly used as benchmarks: orani678, nonsymmetric sparse matrix of order 2529, and bcspwr10, symmetric sparse matrix of order 5300. The other two are te12, a complex matrix from the time evolution application, with dimension 4096, and imclam55, a matrix from the migration modelling application, sparse, nonsymmetric of dimension 6770.

Tables 5.3 and 5.4, and Figures 5.2 and 5.3 show the results for each of the three methods implemented to compute the matrix exponential function. Tables 5.3 and 5.4 also show the absolute error of the computed solution, $\|F - F_m\|_F$, taking as a reference the computation done in Matlab, F_m, that uses the algorithm described in [3].

When using Güttel–Nakatsukasa, computing the eigenvalues of A to shift the matrix implies adding too much overhead to the method. For the executions, we have disabled that computation and eliminated the shift. Also, since most of the operations in this method are performed in complex arithmetic in the case of real matrices, the flop count discerns between real and complex operations.

The results of the executions on the Fermi platform are displayed in Table 5.3 and Figure 5.2. They show that Higham's method (Padé up to degree 13) is faster than basic Padé and Güttel–Nakatsukasa, but it is slightly less efficient in terms of computational intensity. The high number of Gflop/s achieved by Güttel–Nakatsukasa with real matrices, comes from unconditionally using complex arithmetic. The speedup obtained with the GPU on this platform is smaller than in the square root case, ranging from 1.3 to 3.0 with the fastest method. Higham's method always attains the

Table 5.3. Results for the matrix exponential running on the Fermi platform. Time expressed in seconds. GF/s indicates gigaflops per second.

		CPU			GPU		
Matrix	Algorithm	Time	GF/s	$\|F - F_m\|_F$	Time	GF/s	$\|F - F_m\|_F$
orani678	Higham	3.1	67	$4.1 \cdot 10^{-12}$	2.5	83	$4.1 \cdot 10^{-12}$
	Padé	5.0	80	$4.1 \cdot 10^{-12}$	2.3	219	$6.6 \cdot 10^{-12}$
	Güttel–Nakatsukasa†	15.5	97	$3.9 \cdot 10^{-12}$	5.7	260	$4.7 \cdot 10^{-12}$
bcspwr10	Higham	27.2	80	$2.7 \cdot 10^{-12}$	11.7	187	$3.7 \cdot 10^{-12}$
	Padé	49.9	74	$4.7 \cdot 10^{-12}$	16.5	295	$1.8 \cdot 10^{-10}$
	Güttel–Nakatsukasa†	129.8	104	$8.4 \cdot 10^{-9}$	34.6	390	$8.4 \cdot 10^{-9}$
te12	Higham	55.3	83	$1.2 \cdot 10^{-13}$	18.5	248	$3.9 \cdot 10^{-13}$
	Padé	71.9	94	$4.3 \cdot 10^{-13}$	22.5	399	$1.1 \cdot 10^{-10}$
	Güttel–Nakatsukasa†	77.6	102	$3.3 \cdot 10^{-9}$	22.4	359	$3.3 \cdot 10^{-9}$
imclam55	Higham	76.8	92	$5.0 \cdot 10^{-14}$	26.9	262	$5.1 \cdot 10^{-14}$
	Padé	105.8	96	$1.6 \cdot 10^{-13}$	36.0	351	$1.7 \cdot 10^{-12}$
	Güttel–Nakatsukasa†	264.6	106	$1.6 \cdot 10^{-12}$	72.3	401	$1.4 \cdot 10^{-12}$

†Without computing the eigenvalues and shifting the matrix

smallest error on all the executions, because Matlab's reference solution implements the same algorithm.

Table 5.4 and Figure 5.3 contain the results for the tests on the Kepler platform. The results corresponding to the CPU runs maintain the same ranking as on Fermi, but the increase of computational intensity of Padé is remarkable. It is noticeable how the time ratio between Higham and Padé is reduced on this newer platform. Güttel–Nakatsukasa returns the poorest results even when working with complex arithmetic, where it can be competitive.

The GPU executions obtain the fastest times, but the speedups with respect to the CPU are smaller on this platform. The best speedup of 2.1 is obtained with Padé when working with the largest matrix. Contrary to the CPU results, Padé is always faster than Higham on GPU, benefiting from a higher computational intensity. The higher number of Gflop/s obtained by Padé comes from the smaller relative weight of the _gesv routine with respect to higher number of matrix-matrix multiplications.

5.3.2 Computational evaluation of sparse solvers

The sparse tests were conducted on two clusters of the Minotauro supercomputer composed by nodes with the same characteristics as the platforms of Section 5.3.1, interconnected through an Infiniband network. In the experiments of this section, all the executions launch a single process per node, using 12 (Fermi) or 16 (Kepler) threads per process on the CPU runs, and a single GPU card per process on the

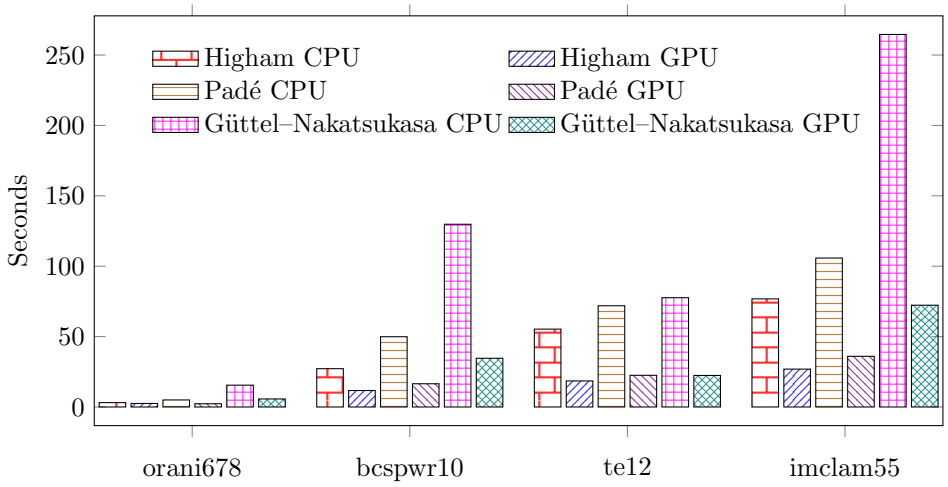

Figure 5.2. Time results for computing the matrix exponential on Fermi.

Table 5.4. Results for the matrix exponential running on the Kepler platform. Time expressed in seconds. GF/s indicates gigaflops per second.

		CPU			GPU		
Matrix	Algorithm	Time	GF/s	$\|F - F_m\|_F$	Time	GF/s	$\|F - F_m\|_F$
orani678	Higham	1.7	122	$4.1 \cdot 10^{-12}$	2.3	90	$4.1 \cdot 10^{-12}$
	Padé	1.8	225	$4.1 \cdot 10^{-12}$	1.5	324	$6.6 \cdot 10^{-12}$
	Güttel–Nakatsukasa†	4.9	310	$3.9 \cdot 10^{-12}$	2.9	505	$4.7 \cdot 10^{-12}$
bcspwr10	Higham	10.3	212	$2.6 \cdot 10^{-12}$	7.8	280	$4.1 \cdot 10^{-12}$
	Padé	10.9	337	$4.9 \cdot 10^{-12}$	6.3	773	$1.8 \cdot 10^{-10}$
	Güttel–Nakatsukasa†	30.1	449	$8.4 \cdot 10^{-9}$	17.9	753	$8.4 \cdot 10^{-9}$
te12	Higham	15.3	300	$1.4 \cdot 10^{-13}$	11.9	386	$3.8 \cdot 10^{-13}$
	Padé	16.0	423	$3.4 \cdot 10^{-13}$	11.0	818	$1.1 \cdot 10^{-10}$
	Güttel–Nakatsukasa†	19.2	410	$3.3 \cdot 10^{-9}$	13.7	588	$3.3 \cdot 10^{-9}$
imclam55	Higham	22.9	307	$4.3 \cdot 10^{-14}$	13.6	518	$5.1 \cdot 10^{-14}$
	Padé	26.1	388	$1.3 \cdot 10^{-13}$	12.5	1012	$1.7 \cdot 10^{-12}$
	Güttel–Nakatsukasa†	59.2	475	$1.6 \cdot 10^{-12}$	31.0	934	$1.4 \cdot 10^{-12}$

†Without computing the eigenvalues and shifting the matrix

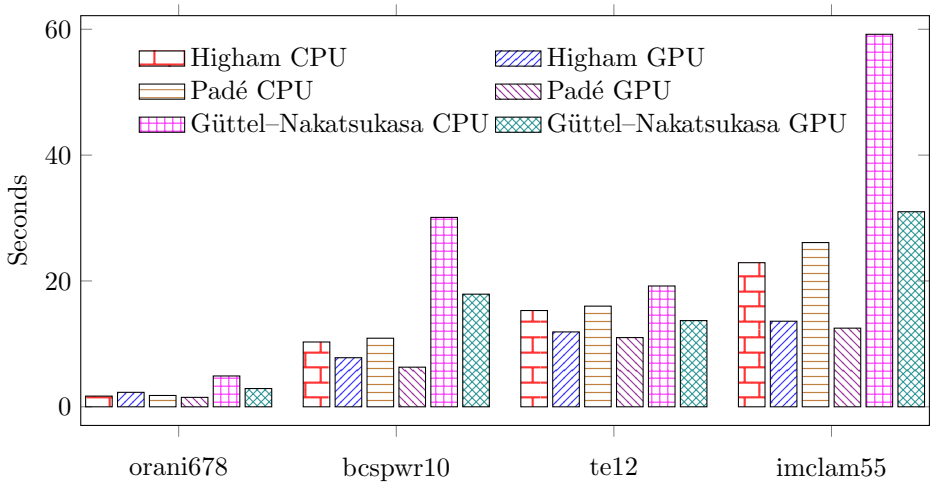

Figure 5.3. Time results for computing the matrix exponential on Kepler.

GPU runs. The reported times are the minimum from three independent executions working in double precision arithmetic. The performance of the different steps of the computation may vary a lot depending on the problem solved and the parameters used. Here we break down the total computation time into the main computational units to analyze their behaviour. These units are: the matrix-vector product used to expand the basis of the Krylov subspace, referred to as MatVec; the orthogonalization and normalization of the vectors, referred to as Orthog; and the computation of the projected dense problem, $f(H_{km})$.

In this section, we present performance results for the Krylov solver using two test cases, with the exponential and square root functions respectively.

Sparse exponential experiments

Advection diffusion equation The application employed in the tests of the exponential function, also used in [26], corresponds to the discretization of the advection diffusion equation

$$\partial_t u = \varepsilon \Delta u + c \nabla u, \tag{5.30}$$

on the domain $\Omega = [0,1]^2$ with homogeneous Dirichlet boundary conditions. A standard 5-point finite-difference discretization with grid size $h = \frac{1}{N+1}$ in both spatial directions results in a sparse matrix of order $n = N^2$ with at most five nonzero elements per row. The Péclet number is defined as the advection diffusion ratio, scaled by h, $\mathrm{Pe} = \frac{ch}{2\varepsilon}$, allowing to control the normality of the matrix. In this case, we are interested in computing $u_k = \exp(\Delta t\, A) u_{k-1}$ for a few values of k using a constant time step, Δt. The computation can be done with the restarted Arnoldi method described in Section 5.2.1, together with the algorithms of Section 5.1.3 for the explicit evaluation of $\exp(H_{km})$ at each restart.

Table 5.5. Results for the advection diffusion problem. Time expressed in seconds. The MatVec and Orthog columns show the time needed in the matrix-vector product, and in the orthogonalization and normalization of the basis vectors, respectively.

			CPU Time			GPU Time		
Platform	Processes	Algorithm	MatVec	Orthog	exp(H_{km})	MatVec	Orthog	exp(H_{km})
Fermi	1	Higham	136.9	554.9	61.8	10.1	160.4	11.0
		Padé	135.0	557.6	65.2	10.1	160.5	6.9
	16	Higham	23.9	78.9	63.5	9.1	24.6	12.2
		Padé	19.3	71.9	67.1	5.2	15.0	7.5
Kepler	1	Higham	103.7	473.3	6.1	8.1	78.0	4.4
		Padé	112.8	473.1	6.4	8.1	78.1	3.5
	16	Higham	8.7	19.4	7.5	4.4	10.5	4.9
		Padé	8.1	19.1	7.6	3.9	9.3	4.4

Figure 5.4 and Table 5.5 show the results when computing $u_k = \exp(\Delta t\, A)u_{k-1}$ for five repetitions with $\Delta t = 10^{-4}$. The parameters that we have used to generate the advection diffusion discretization matrix are Pe $= 0.5$, $\varepsilon = 1$, and $N = 1735$ (hence the size of A is roughly 3 million). The initial vector used is the one given by the discretization of the initial state $u_0(x, y) = 256 \cdot x^2(1-x)^2y^2(1-y)^2$. The Arnoldi restart parameter used in this case is $m = 30$, with which the solver needed 23 restarts on each time step.

On Section 5.3.1 we saw that on the Kepler platform, none of the algorithms to compute the dense exponential was faster on both CPU and GPU, so with this problem we present results using Higham (faster on CPU) and Padé (faster on GPU). On Fermi, MatVec, Orthog and exp(H_{km}) are able to achieve good speedups of 13.4, 3.5 and 9.5 respectively, with the single process execution on the GPU runs (comparing to CPU). The overall speedup obtained with a single process reaches only 4.3, as the dominant time-consuming operation is the orthogonalization, that attains the smallest speedup. As the dense solver is not executed in parallel, its runtime should not vary in the multi-process execution, although on actual runs there can indeed be some variation due to idle times produced by process synchronization. On the multi-process execution, the speedup of MatVec is reduced to 3.8 and the speedup of Orthog improves up to 4.8. These speedups, together with the reduced relative weight of the orthogonalization when executing with multiple processes, increase the maximum speedup obtained with the GPU for the whole computation up to 5.8 when using 16 processes.

On Kepler, the single process execution on GPU obtains speedups of 13.9 and 6.1 for MatVec and Orthog, with respect to the CPU runs. The dense exponential performance is reduced to a speedup of 1.8 when running on GPU. The whole computation speedup for GPU runs compared with the CPU ones, achieve 6.6 with a single process, and a more limited speedup of 2.1 on the multi-process execution.

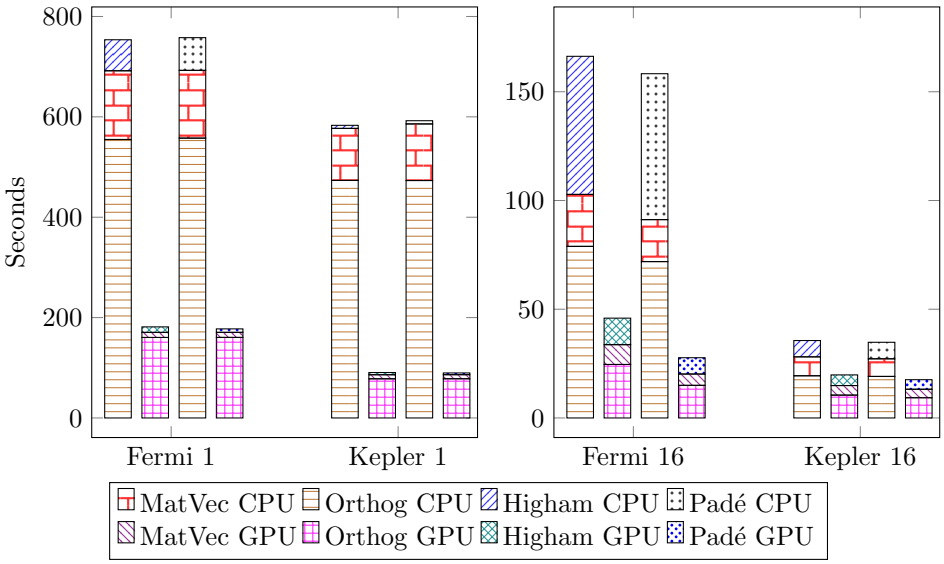

Figure 5.4. Time results for the advection diffusion problem.

Sparse square root experiments

Ensemble Square-Root Kalman Filter The application used in the tests of the square root function is the data assimilation application of Section 5.3.1. Note that in Section 5.1 we used the same application, but a different matrix, to analyze the square root function, and now we deal with the complete function with a matrix of dimension 17733 and a sparsity of 40%, much denser than in the advection diffusion case. In this application, we are interested in solving linear systems of the form $(D + \sqrt{D})x = b$, which is equivalent to computing $x = f(D)b$, with

$$f(D) = (D + \sqrt{D})^{-1}. \tag{5.31}$$

SLEPc allows certain flexibility in the definition of functions, by combining two simpler functions. In our case, we define $f(\cdot)$ as the reciprocal of another function, which in turn is defined as the sum of two functions (the identity and the square root). All these sub-functions can be evaluated easily, except for the matrix square root that needs the algorithms of Section 5.1.1. These evaluations will operate on the Hessenberg matrix in every restart of the Arnoldi method.

Figure 5.5 and Table 5.6 show the results obtained for computing $f(D)b$ for 30 different right-hand sides b with the EnSRF matrix function of (5.31), and using Denman–Beavers (5.10) as the method to compute $\sqrt{H_{km}}$. In this experiment we have used an Arnoldi restart parameter of $m = 150$, with which the solver required only three restarts in all cases.

On Fermi, the single process execution running on GPU achieves speedups of 16.2 and 2.7 for MatVec and Orthog, with respect to the CPU, while the dense function

Table 5.6. Results for the EnSRF matrix function using Denman–Beavers to compute $\sqrt{H_{km}}$. Time expressed in seconds. The MatVec and Orthog columns show the time needed in the matrix-vector product, and in the orthogonalization and normalization of the basis vectors, respectively.

		CPU Time			GPU Time		
Platform	Processes	MatVec	Orthog	$\sqrt{H_{km}}$	MatVec	Orthog	$\sqrt{H_{km}}$
Fermi	1	3459.8	67.6	49.0	212.8	25.0	43.2
	16	269.1	124.4	55.0	29.2	18.1	43.6
Kepler	1	2453.2	21.5	15.4	135.2	13.0	15.3
	16	192.1	58.9	28.8	20.7	14.4	19.0

obtains almost the same time on CPU and GPU. The overall obtained speedup of 12.7 when running on GPU stems from the MatVec operation. The multi-process execution reduces the speedup of MatVec to 9.2, and Orthog speedup increases to 6.8, reducing the total speedup down to 4.8 for the GPU with respect to the CPU run. The runs on GPU on the Kepler platform, with a single process, obtain better speedup in MatVec, the dominant operation, reaching 18.1. This improvement of MatVec helps to accelerate the whole computation more than in the case of the Fermi platform, with a speedup of 15.2 when running a single process on GPU, with respect to CPU. Using several processes on Kepler also improves the gain obtained on Fermi, reaching a speedup of 5 on the whole computation with respect to the CPU runs.

Addendum Here, we would like to take the opportunity to add an aside to comment about the improvement in the time and memory usage of the data assimilation application with the matrix function approach.

In a previous work, Steward et al. [126] proposed a parallel method to compute the Ensemble Square-Root Kalman Filter (EnSRF) equations by computing a matrix function via eigenpairs. Unlike other methods that assimilate the observations in a given window sequentially, the proposed solution does not depend on the order of observations, allowing to assimilate all the observations in the window independently at the same time. With our work [127] we have enabled the optimization of this methodology by switching from computing the inverse and square root portion of the EnSRF equations into computing the action of a matrix function on a vector. Matrix-vector products are then used to build a Krylov subspace and the matrix function is applied to a projected problem as described in Section 5.2.

We briefly present the results of a set of executions carried out in the xjet supercomputer. Xjet is composed by 812 nodes interconnected with an FDR Infiniband network. Each node has two Intel Haswell processors (12 cores per processor) running at 2.3 GHz and 64 GB of main memory.

The modifications were incorporated to the HEDAS application [2] and tested

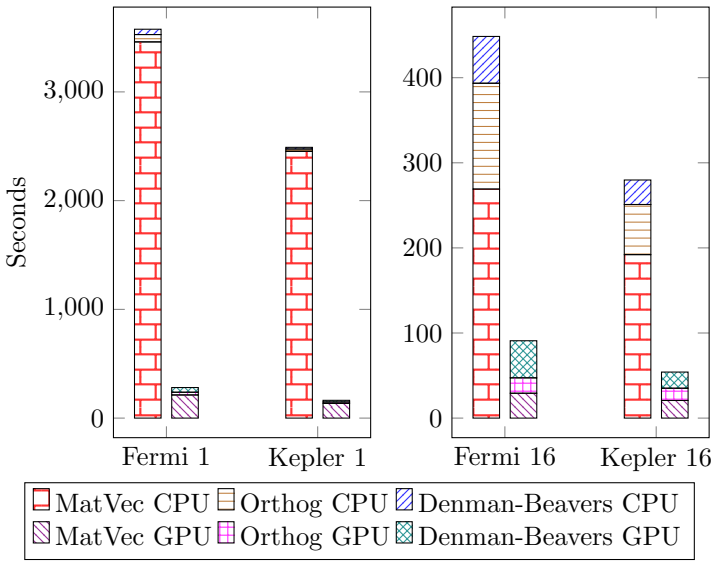

Figure 5.5. Time results for the EnSRF matrix function using Denman–Beavers to compute $\sqrt{H_{km}}$.

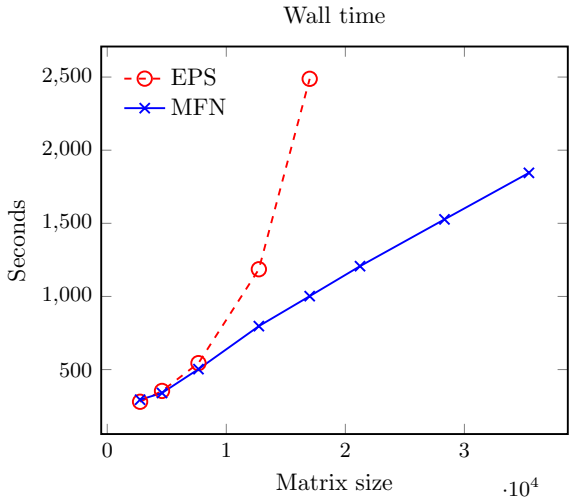

Figure 5.6. Total computation time for the EPS and MFN methods as a function of the matrix size with 386 processes, in double precision arithmetic.

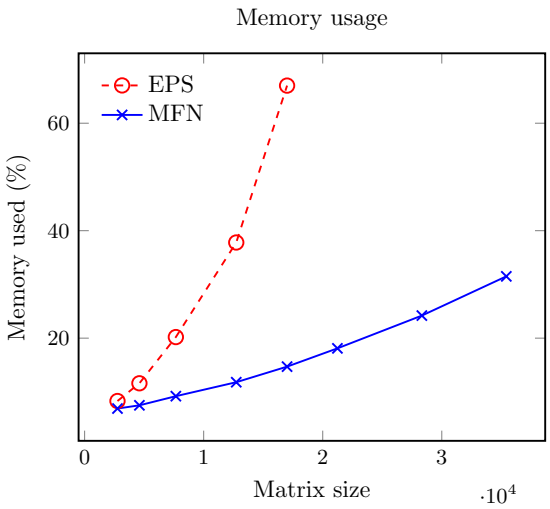

Figure 5.7. Memory usage for the EPS and MFN methods as a function of the matrix size with 386 processes, in double precision arithmetic.

with 384 processes, running on the same number of computational cores. The matrix function approach provides a numerical solution analogous to the eigenproblem based method, that has proven error bounds. We show in Figure 5.6 the execution time as a function of the matrix size (number of observations to assimilate) for both methods of solving the EnSKF equations, EPS which computes the eigenvalues of the forward observation covariance matrix, and MFN that performs the computation of the action of a matrix function on a vector with Krylov methods. Times above 2500 seconds are not shown. The EPS solution scales approximately as n^3, while MFN seems to scale linearly. We remark that the time of the computation is directly related with the sparsity of the matrix and these matrices are relative dense.

Figure 5.7 shows the total memory footprint for the same executions. The eigenproblem method yields the same cubic increase with respect to the matrix size, and the matrix function increase is more than linear. Again, the memory scaling is directly related with the sparsity of the matrix, where a sparse matrix should provide linear scaling and a dense one quadratic scaling.

5.4 Conclusions

In this chapter we have studied the suitability of several dense methods ($f(A)$) to compute the matrix square root, the inverse square root, and the matrix exponential functions on GPU platforms. The simplicity of iterative methods entitle them to be implemented on GPU with common linear algebra operations and allow the acceleration of their computation with respect to a CPU implementation. In our tests, we have been able to obtain up to a gain factor of 3.8 when comparing GPU

implementations against multi-threaded CPU implementations, for matrices with dimension of a few thousands. We have also observed that the increasing number of available cores on newer CPUs make them closer in efficiency to its contemporary GPUs.

The obtained acceleration can be useful to reduce the cost present in the critical path of parallel Krylov methods for the computation of the action of a matrix function on a vector ($f(A)b$), when executed on clusters of GPUs, provided that the size of the Hessenberg matrix is not too small, that is, the computation associated with this matrix takes a non-negligible percentage of the overall computation time. The GPU implementations of the dense methods will also be helpful for explicitly computing $f(A)$ in some applications.

We have also analyzed the overall performance on GPU platforms of the restarted Arnoldi method for matrix functions, obtaining speedups of 5.8 in a multi-process execution. We have seen how the computation is divided in three main steps that have more or less relevance depending on the characteristics of the initial matrix. A parallel high performance implementation of the matrix-vector product is essential to provide fast results. The most time-consuming operation is the orthogonalization of the vectors of the Krylov basis. Each restart of Arnoldi increases the size of the dense problem, and it may entail a considerable cost with many restarts. We have shown how all three operations can be done on GPU, and the improvement that it provides.

Finally, we have taken the opportunity to present the results of implementing our methods on an application, and compare the matrix function solution with an eigenvalue oriented method. The matrix function solution scales better as a function of the matrix size, both in time and in memory usage, while achieving similar numerical results and maintaining the feature of assimilating the observations within the assimilation window in parallel.

Chapter 6

Conclusions

No lusco e fusco

With this chapter we conclude the exposition of these years of work. As the culmination of our activity we summarize here the most relevant conclusions of the fulfilled targets in this thesis.

Our work has focused on creating multi-process parallel implementations of large sparse eigenvalue solvers and functions of matrices that make use of graphics processing units to accelerate the computations. Each of our implementations has subsequently been used in scientific applications to demonstrate the improvement achieved and its applicability.

In Chapter 3 we introduced the eigenvalue problem and the most common methods for solving it. We also presented SLEPc and PETSc libraries and offered a description of how they can make use of GPUs in a multi-process parallel implementation that combines MPI and CUDA. We raised the problematic encountered with the initial approach to GPU computing in PETSc when we found some specific lacks of capabilities in the GPU support based on CUSP, that restrained our developments. The constraints were originated in some decisions in the design of Thrust, the C++ library on top of which CUSP is developed. This situation made us work to extend PETSc functionality with a more complete support for GPU operations basing this support in the CUDA runtime, in cuBLAS, and in cuSPARSE.

Nevertheless, with the initial version of the GPU support based on CUSP, we were able to obtain very rewarding results with the PARISO application. The speedup of the eigenvalue solve when running on the GPU in a transparent way, in addition to the speedup of the generation of the matrix with our implementation of a specific kernel, resulted in a good scalability. The multi-process trials showed that the GPU version is far superior than the previous one.

The implementations and tests performed in Chapter 4 were those who required most of our time, out of all the work that we have done. In this chapter we have worked in obtaining a few eigenpairs from block-tridiagonal and block cyclic tridi-

agonal matrices by means of Krylov methods. We have considered the blocks of the matrices as dense blocks in order to use BLAS routines in the computations. The matrix storage format used has helped us to simplify the implementations, and also allowed the improvement of the performance by using coalesced access to the GPU memory by the CUDA threads. In the case of computing eigenvalues on the exterior of the spectrum we experimented with BLAS based solutions and eventually ended up implementing a better performing kernel to do the matrix product. In the case of computing interior eigenvalues by means of the shift-and-invert technique, we implemented and made a comparison of several algorithms for solving linear systems of equations with such structure. Our GPU implementations have proven to be very competitive when comparing them with CPU implementations for a large enough matrix block size. With small block sizes, it is difficult to recover from the overhead of launching a kernel to the GPU. In both cases, exterior and interior eigenvalues, our software can make use of several GPUs with an MPI-GPU approach, scaling up to very large problem sizes. Finally, we compared our multi-process solution for solving linear systems of equations with a state of the field library obtaining a better scalability.

In Chapter 5 we have implemented different matrix function solvers in the SLEPc library, extending the usability of the `FN` and `MFN` modules. The methods have been validated with a collection of test cases coming from scientific applications, and have proven to provide robust numerical results and achieve a good performance. They are included in the public release of SLEPc, and are already being used in different scientific computing applications.

Throughout this years of work the SLEPc library has experienced changes and improvements. The main contributions of this thesis have been the extension of the computing capabilities of the library by means of the use of graphics processing units as accelerators, and the inclusion of new algorithms that allow to perform operations that were not available before.

Publications and research projects

Publications

Together with the presented developments, the contents of this thesis have been published in several scientific journals and presented in international conferences. In particular, Chapters 3, 4 and 5 gave rise to the following articles and proceedings:

[80] A. Lamas Daviña, E. Ramos, and J. E. Roman. Optimized analysis of isotropic high-nuclearity spin clusters with GPU acceleration. Comput. Phys. Commun., 209:70–78, 2016.

[81] A. Lamas Daviña and J. E. Roman. GPU implementation of Krylov solvers for block-tridiagonal eigenvalue problems. In R. Wyrzykowski et al., editors, Parallel Processing and Applied Mathematics-PPAM 2015, Part I, volume 9573 of Lect. Notes Comp. Sci., pages 182–191. Springer, 2016.

[82] A. Lamas Daviña and J. E. Roman. Parallel MPI-GPU linear system solvers specific for block-tridiagonal matrices. In Libro de comunicaciones definitivas presentadas en CEDYA+CMA2017, pages 674–681. UP4Sciences, 2017.

[83] A. Lamas Daviña and J. E. Roman. MPI-CUDA parallel linear solvers for block-tridiagonal matrices in the context of SLEPc's eigensolvers. Parallel Comput., 74:118–135, 2018.

[79] A. Lamas Daviña, X. Cartoixà, and J. E. Roman. Scalable block-tridiagonal eigensolvers in the context of electronic structure calculations. In S. Bassini, M. Danelutto, P. Dazzi, G. Joubert, and F. Peters, editors, Parallel Computing is Everywhere, volume 32 of Advances in Parallel Computing, pages 117–126. IOS Press, 2018.

[127] J. Steward, J. E. Roman, A. Lamas Daviña, and A. Aksoy. Parallel direct solution of the covariance-localized ensemble square-root Kalman filter equations with matrix functions. Mon. Weather Rev., 146(9):2819–2836, 2018.

Research projects

This thesis has been developed under the FPU program (FPU2013-06655) granted by the Spanish Ministry of Education, Culture and Sport, and in the framework of the following research projects:

Extending the SLEPc library for matrix polynomials, matrix functions and matrix equations in emerging computing platforms (TIN2013-41049-P) (2014-2016), granted by the Spanish Ministry of Economy and Competitiveness.

Highly scalable eigensolvers in the context of the SLEPc library (TIN2016-75985-P) (2017-2019), granted by the Spanish National Research Agency.

Open research lines

Many things have remained to be done on each of the developments made. Some of them are just small ideas, and other, complex things that would entail a time that we did not have. We here name some possible future courses of action that we think feasible and would improve the software capabilities.

The PARISO software of Chapter 3 may be extended in such a way that not only the first and last eigenvalue of each block is computed in the first phase, but also a rough approximation of how all eigenvalues are distributed within that range. This information, usually known as density of states (DOS), is difficult to obtain, but recent efforts are trying to do this cost-effectively [90].

When we solve the block (cyclic) tridiagonal matrices of Chapter 4, we treat the blocks of the matrix as dense, therefore as an evident future work remains to take profit of their sparsity. It can be made by using any of the sparse matrix storage formats available on cuBLAS and MAGMA. Other attractive possibility is

allocating the blocks directly as PETSc matrix types, as it would allow a transparent management of the memory transfers between CPU and GPU, and the operations with them could be easily migrated. Another possible research direction would be to expand the software to work with tridiagonal blocks that appear in some fast Poisson solvers.

With respect to the implementation of matrix functions of Chapter 5, we will continue adding more functionality to the matrix function solvers in SLEPc, as it is a very interesting field with an active development of new algorithms. In particular, for the dense case, we could study the inclusion of trigonometric functions like the sine and cosine, that appear in the solution of second order differential equations. For the sparse case, alternative restart schemes [40, 46] have been proposed recently that may be more effective than the Eiermann-Ernst scheme that we currently use.

Talking about a more low level aspect of the implementations, the lack of a matrix dense type that works on the GPU (a possible MatCUDADense type) in PETSc could be addressed. This feature would benefit both SLEPc and user codes by enabling the transparent run of dense operations on the GPU. Also, the inclusion of MAGMA solvers in PETSc for performing operations on the GPU would certainly accelerate LAPACK operations. In more general terms, the use of other accelerators could be studied.

Bibliography

[1] M. Ahues, F. D. d'Almeida, A. Largillier, O. Titaud, and P. Vasconcelos. An L^1 refined projection approximate solution of the radiation transfer equation in stellar atmospheres. *J. Comput. Appl. Math.*, 140:13–26, 2002. (Cited on pages 72 and 98.)

[2] A. Aksoy, S. Lorsolo, T. Vukicevic, K. J. Sellwood, S. D. Aberson, and F. Zhang. The HWRF Hurricane Ensemble Data Assimilation System (HEDAS) for High-Resolution Data: The Impact of Airborne Doppler Radar Observations in an OSSE. *Mon. Weather Rev.*, 140(6):1843–1862, 2012. (Cited on page 132.)

[3] A. H. Al-Mohy and N. J. Higham. A new scaling and squaring algorithm for the matrix exponential. *SIAM J. Matrix Anal. Appl.*, 31(3):970–989, 2010. (Cited on pages 119 and 126.)

[4] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485. ACM, 1967. (Cited on page 34.)

[5] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng.*, 184(2–4):501–520, 2000. (Cited on page 48.)

[6] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999. (Cited on page 19.)

[7] C. Andrew, C. F. F., L. Rainald, and W. John. Running unstructured grid-based CFD solvers on modern graphics hardware. *Int. J. Numer. Methods Flu.*, 66(2):221–229, 2011. (Cited on page 3.)

[8] H. Anzt, S. Tomov, and J. Dongarra. Implementing a sparse matrix vector product for the SELL-C/SELL-C-σ formats on NVIDIA GPUs. Technical Report ut-eecs-14-727, University of Tennessee, March 2014. (Cited on page 32.)

[9] W. E. Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quart. Appl. Math.*, 9:17–29, 1951. (Cited on page 41.)

[10] B. Baghapour, V. Esfahanian, M. Torabzadeh, and H. M. Darian. A discontinuous Galerkin method with block cyclic reduction solver for simulating compressible flows on GPUs. *Int. J. Comput. Math.*, 92(1):110–131, 2015. (Cited on page 86.)

[11] Z. Bai, D. Day, J. Demmel, and J. Dongarra. A test matrix collection for non-Hermitian eigenvalue problems (release 1.0). available at `https://sparse.tamu.edu`, 1996. (Cited on page 123.)

[12] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, editors. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2000. (Cited on pages 2, 37, and 44.)

[13] Z. Bai and G. W. Stewart. Algorithm 776: SRRIT: A FORTRAN subroutine to calculate the dominant invariant subspace of a nonsymmetric matrix. *ACM Trans. Math. Software*, 23(4):494–513, 1997. (Cited on page 40.)

[14] C. G. Baker, U. L. Hetmaniuk, R. B. Lehoucq, and H. K. Thornquist. Anasazi software for the numerical solution of large-scale eigenvalue problems. *ACM Trans. Math. Software*, 36(3):13:1–13:23, 2009. (Cited on page 45.)

[15] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. Gropp, D. Karpeyev, D. Kaushik, M. Knepley, D. May, L. C. McInnes, R. Mills, T. Munson, K. Rupp, P. Sanan, B. Smith, S. Zampini, H. Zhang, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.9, Argonne National Laboratory, 2018. (Cited on pages 46 and 64.)

[16] M. M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs using compile-time and run-time strategies, 2008. (Cited on page 3.)

[17] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11. ACM, 2009. (Cited on page 32.)

[18] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C. C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - processor: A 64-core SoC with mesh interconnect. In *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, pages 88–598, 2008. (Cited on page 23.)

[19] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997. (Cited on pages 19 and 45.)

[20] S. Blaser, M. Rochat, L. Ajili, M. Beck, J. Faist, H. Beere, G. Davies, E. Linfield, and D. Ritchie. Terahertz interminiband emission and magneto-transport measurements from a quantum cascade chirped superlattice. *Physica E*, 13(2–4):854–857, 2002. (Cited on page 108.)

[21] J. J. Borras-Almenar, J. M. Clemente-Juan, E. Coronado, and B. S. Tsukerblat. High-nuclearity magnetic clusters: Generalized spin Hamiltonian and its use for the calculation of the energy levels, bulk magnetic properties, and inelastic neutron scattering spectra. *Inorg. Chem.*, 38:6081–6088, 1999. (Cited on pages 54 and 55.)

[22] J. J. Borras-Almenar, J. M. Clemente-Juan, E. Coronado, and B. S. Tsukerblat. MAGPACK: A package to calculate the energy levels, bulk magnetic properties, and inelastic neutron scattering spectra of high nuclearity spin clusters. *J. Comput. Chem.*, 22(9):985–991, 2001. (Cited on page 54.)

[23] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, 35(1):38–53, 2009. (Cited on page 19.)

[24] B. L. Buzbee, G. H. Golub, and C. W. Nielson. On direct methods for solving Poisson's equations. *SIAM J. Numer. Anal.*, 7(4):627–656, 1970. (Cited on page 75.)

[25] M. Caliari, P. Kandolf, A. Ostermann, and S. Rainer. Comparison of software for computing the action of the matrix exponential. *BIT Numer. Math.*, 54(1):113–128, 2014. (Cited on pages 115 and 120.)

[26] M. Caliari, P. Kandolf, A. Ostermann, and S. Rainer. The Leja method revisited: Backward error analysis for the matrix exponential. *SIAM J. Sci. Comput.*, 38(3):A1639–A1661, 2016. (Cited on page 129.)

[27] L.-W. Chang, J. A. Stratton, H.-S. Kim, and W.-M. W. Hwu. A scalable, numerically stable, high-performance tridiagonal solver using GPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 27:1–27:11, Nov. 2012. (Cited on page 89.)

[28] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP Portable Shared Memory Parallel Programming*. The MIT Press, 2008. (Cited on page 13.)

[29] S. H. Cheng, N. J. Higham, C. S. Kenney, and A. J. Laub. Approximating the logarithm of a matrix to specified accuracy. *SIAM J. Matrix Anal. Appl.*, 22(4):1112–1125, 2001. (Cited on page 116.)

[30] CUDA C Programming Guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`. Accessed on 17 May 2018. (Cited on page 28.)

[31] J. J. M. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numer. Math.*, 36(2):177–195, 1980. (Cited on page 39.)

[32] L. D. Dalcin, R. R. Paz, P. A. Kler, and A. Cosimo. Parallel distributed computing using Python. *Adv. Water Resour.*, 34(9):1124–1139, 2011. (Cited on page 48.)

[33] S. Dalton, N. Bell, L. Olson, and M. Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2015. Version 0.5.1, available at `https://cusplibrary.github.io/`. (Cited on pages 32 and 49.)

[34] E. Deadman, N. J. Higham, and R. Ralha. Blocked Schur algorithms for computing the matrix square root. In P. Manninen and P. Öster, editors, *Applied Parallel and Scientific Computing, PARA 2012*, pages 171–182, 2013. (Cited on page 115.)

[35] E. D. Denman and A. N. Beavers, Jr. The matrix sign function and computations in systems. *Appl. Math. Comput.*, 2(1):63–94, 1976. (Cited on page 116.)

[36] R. Dolbeau. Theoretical peak flops per instruction set: a tutorial. *J. Supercomput.*, 74(3):1341–1377, 2018. (Cited on page 36.)

[37] J. Dongarra, C. Moler, J. Bunch, and G. Stewart. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1979. (Cited on page 19.)

[38] I. S. Duff, R. G. Grimes, and J. G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (release I). available at `https://sparse.tamu.edu`, 1992. (Cited on page 126.)

[39] M. Eiermann and O. G. Ernst. A restarted Krylov subspace method for the evaluation of matrix functions. *SIAM J. Numer. Anal.*, 44(6):2481–2504, 2006. (Cited on page 120.)

[40] M. Eiermann, O. G. Ernst, and S. Güttel. Deflated restarting for matrix functions. *SIAM J. Matrix Anal. Appl.*, 32(2):621–641, 2011. (Cited on page 140.)

[41] N. England. A system for interactive modeling of physical curved surface objects. *SIGGRAPH Comput. Graph.*, 12(3):336–340, 1978. (Cited on page 24.)

[42] N. England. A graphics system architecture for interactive application-specific display functions. *IEEE Computer Graphics and Applications*, 6(1):60–70, 1986. (Cited on page 24.)

[43] M. E. Farquhar, T. J. Moroney, Q. Yang, and I. W. Turner. GPU accelerated algorithms for computing matrix function vector products with applications to exponential integrators and fractional diffusion. *SIAM J. Sci. Comput.*, 38(3):C127–C149, 2016. (Cited on page 119.)

[44] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, 1972. (Cited on page 11.)

[45] R. A. Frazer, W. J. Duncan, and A. R. Collar. *Elementary Matrices: And Some Applications to Dynamics and Differential Equations*. Cambridge University Press, 1938. (Cited on page 118.)

[46] A. Frommer, K. Lund, M. Schweitzer, and D. B. Szyld. The Radau–Lanczos method for matrix functions. *SIAM J. Matrix Anal. Appl.*, 38(3):710–732, 2017. (Cited on page 140.)

[47] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, W. Zhao, X. Yin, C. Hou, C. Zhang, W. Ge, J. Zhang, Y. Wang, C. Zhou, and G. Yang. The Sunway Taihulight supercomputer: system and applications. *Sci. China Infor. Sci.*, 59(7):072001, 2016. (Cited on page 23.)

[48] T. Fukaya and T. Imamura. Performance evaluation of the Eigen Exa eigensolver on Oakleaf-FX: Tridiagonalization versus pentadiagonalization. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 960–969, 2015. (Cited on page 45.)

[49] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In D. Kranzlmüller, P. Kacsuk, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer Berlin Heidelberg, 2004. (Cited on page 21.)

[50] E. Gallopoulos, B. Philippe, and A. H. Sameh. *Parallelism in Matrix Computations*. Springer, Dordrecht, 2016. (Cited on page 75.)

[51] W. Gander and G. H. Golub. Cyclic Reduction: history and applications. In R. J. P. Franklin T. Luk, editor, *Proceedings of the Workshop on Scientific Computing*, pages 73–85. Springer, Mar. 1997. (Cited on page 75.)

[52] M. Gates, A. Haidar, and J. Dongarra. Accelerating computation of eigenvectors in the nonsymmetric eigenvalue problem. Technical Report Working Note 286, LAPACK, 2014. (Cited on page 4.)

[53] G. H. Golub and H. A. van der Vorst. Eigenvalue computation in the 20th century. *J. Comput. Appl. Math.*, 123(1-2):35–65, 2000. (Cited on page 37.)

[54] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006. (Cited on page 10.)

[55] S. Güttel and Y. Nakatsukasa. Scaled and squared subdiagonal Padé approximation for the matrix exponential. *SIAM J. Matrix Anal. Appl.*, 37(1):145–170, 2016. (Cited on page 119.)

[56] A. Haidar, H. Ltaief, and J. Dongarra. Toward a high performance tile divide and conquer algorithm for the dense symmetric eigenvalue problem. *SIAM J. Sci. Comput.*, 34(6):C249–C274, 2012. (Cited on page 4.)

[57] D. Heller. Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems. *SIAM J. Numer. Anal.*, 13(4):484–496, 1976. (Cited on page 75.)

[58] V. Hernandez, J. E. Roman, and A. Tomas. Parallel Arnoldi eigensolvers with enhanced scalability via global communications rearrangement. *Parallel Comput.*, 33(7–8):521–540, 2007. (Cited on page 51.)

[59] V. Hernandez, J. E. Roman, A. Tomas, and V. Vidal. A survey of software for sparse eigenvalue problems. Technical Report STR-6, Universitat Politècnica de València, 2006. (Cited on page 44.)

[60] V. Hernandez, J. E. Roman, and V. Vidal. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Trans. Math. Software*, 31(3):351–362, 2005. (Cited on pages 2 and 45.)

[61] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Software*, 31(3):397–423, 2005. (Cited on page 46.)

[62] N. J. Higham. Computing real square roots of a real matrix. *Linear Algebra Appl.*, 88–89:405–430, 1987. (Cited on page 115.)

[63] N. J. Higham. The matrix sign decomposition and its relation to the polar decomposition. *Linear Algebra Appl.*, 212-213:3–20, 1994. (Cited on page 118.)

[64] N. J. Higham. *Functions of Matrices: Theory and Computation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2008. (Cited on pages 113, 117, 118, 120, and 123.)

[65] N. J. Higham and A. H. Al-Mohy. Computing matrix functions. *Acta Numerica*, 19:159–208, 2010. (Cited on pages 113, 115, and 118.)

[66] S. P. Hirshman, K. S. Perumalla, V. E. Lynch, and R. Sanchez. BCYCLIC: A parallel block tridiagonal matrix cyclic solver. *J. Comput. Phys.*, 229(18):6392–6404, 2010. (Cited on pages 75 and 86.)

[67] J. D. Hogg. A fast dense triangular solve in CUDA. *SIAM J. Sci. Comput.*, 35(3):303–C322, 2013. (Cited on page 4.)

[68] C. Hong, A. Sukumaran-Rajam, B. Bandyopadhyay, J. Kim, S. E. Kurt, I. Nisa, S. Sabhlok, Ü. V. Çatalyürek, S. Parthasarathy, and P. Sadayappan. Efficient sparse-matrix multi-vector product on GPUs. In *Proceedings of the*

27th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '18, pages 66–79. ACM, 2018. (Cited on page 32.)

[69] J. F. Hughes, A. van Dam, M. McGuire, D. F. Sklar, J. D. Foley, S. K. Feiner, and K. Akeley. *Computer Graphics: Principles and Practice*. Addison-Wesley Professional, Boston, MA, USA, third edition, 2013. (Cited on page 23.)

[70] IEEE and The Open Group. POSIX.1-2017, IEEE std 1003.1-2017 (revision of IEEE std 1003.1-2008), The Open Group Base Specifications issue 7, 2018. (Cited on page 13.)

[71] Intel Math Kernel Library (MKL) developer reference (C). `https://software.intel.com/en-us/mkl-developer-reference-c/`. Accessed on 17 May 2018. (Cited on page 19.)

[72] H. Iwai. CMOS technology after reaching the scale limit. In *Junction Technology, 2008. IWJT '08. Extended Abstracts - 2008 8th International workshop on*, pages 1–2, May 2008. (Cited on page 8.)

[73] J. D. Jakub Kurzak, David A. Bader. *Scientific Computing with Multicore and Accelerators*. CRC Press, 2010. (Cited on page 2.)

[74] J. M. Jancu, R. Scholz, F. Beltram, and F. Bassani. Empirical spds* tight-binding calculation for cubic semiconductors: General method and material parameters. *Phys. Rev. B*, 57(11):6493–6507, 1998. (Cited on page 108.)

[75] A. D. Kern and J. Hey. Exact calculation of the joint allele frequency spectrum for generalized isolation with migration models. *Genetics*, 207(1):241–253, 2017. (Cited on page 126.)

[76] J. Kessenich, G. Sellers, and D. Shreiner. *The OpenGL Programming Guide*. Addison-Wesley Professional, Boston, MA, USA, 9th edition, 2017. (Cited on page 24.)

[77] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. Bishop. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM J. Sci. Comput.*, 36(5):C401–C423, 2014. (Cited on page 32.)

[78] V. Kysenko, K. Rupp, O. Marchenko, S. Selberherr, and A. Anisimov. GPU-accelerated non-negative matrix factorization for text mining. In G. Bouma, A. Ittoo, E. Métais, and H. Wortmann, editors, *Natural Language Processing and Information Systems*, pages 158–163, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. (Cited on page 4.)

[79] A. Lamas Daviña, X. Cartoixà, and J. E. Roman. Scalable block-tridiagonal eigensolvers in the context of electronic structure calculations. In S. Bassini, M. Danelutto, P. Dazzi, G. Joubert, and F. Peters, editors, *Parallel Computing is Everywhere*, volume 32 of *Advances in Parallel Computing*, pages 117–126. IOS Press, 2018. (Cited on page 139.)

[80] A. Lamas Daviña, E. Ramos, and J. E. Roman. Optimized analysis of isotropic high-nuclearity spin clusters with GPU acceleration. *Comput. Phys. Commun.*, 209:70–78, 2016. (Cited on page 138.)

[81] A. Lamas Daviña and J. E. Roman. GPU implementation of Krylov solvers for block-tridiagonal eigenvalue problems. In R. Wyrzykowski et al., editors, *Parallel Processing and Applied Mathematics–PPAM 2015, Part I*, volume 9573 of *Lect. Notes Comp. Sci.*, pages 182–191. Springer, 2016. (Cited on page 138.)

[82] A. Lamas Daviña and J. E. Roman. Parallel MPI-GPU linear system solvers specific for block-tridiagonal matrices. In *Libro de comunicaciones definitivas presentadas en CEDYA+CMA2017*, pages 674–681. UP4Sciences, 2017. (Cited on page 139.)

[83] A. Lamas Daviña and J. E. Roman. MPI-CUDA parallel linear solvers for block-tridiagonal matrices in the context of SLEPc's eigensolvers. *Parallel Comput.*, 74:118 – 135, 2018. (Cited on page 139.)

[84] J. J. Lambiotte, Jr. and R. G. Voigt. The solution of tridiagonal linear systems on the CDC STAR 100 computer. *ACM Trans. Math. Software*, 1(4):308–329, 1975. (Cited on page 75.)

[85] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 55–55. ACM, 2001. (Cited on page 24.)

[86] E. László, M. Giles, and J. Appleyard. Manycore algorithms for batch scalar and block tridiagonal solvers. *ACM Trans. Math. Software*, 42(4):31:1–31:36, 2016. (Cited on page 86.)

[87] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Software*, 5(3):308–323, 1979. (Cited on page 18.)

[88] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK Users' Guide, Solution of Large-Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1998. (Cited on page 45.)

[89] R. Li and Y. Saad. GPU-accelerated preconditioned iterative linear solvers. *J. Supercomput.*, 63(2):443–466, 2013. (Cited on pages 4 and 62.)

[90] L. Lin. Randomized estimation of spectral densities of large matrices made accurate. *Numer. Math.*, 136(1):183–213, 2017. (Cited on page 139.)

[91] D. J. Luitz and Y. B. Lev. Information propagation in isolated quantum systems. *Phys. Rev. B*, 96:020406, 2017. (Cited on page 126.)

[92] A. Marek, V. Blum, R. Johanni, V. Havu, B. Lang, T. Auckenthaler, A. Heinecke, H.-J. Bungartz, and H. Lederer. The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science. *J. Phys.: Cond. Matter*, 26(21):213201, 2014. (Cited on page 45.)

[93] K. J. Maschhoff and D. C. Sorensen. PARPACK: An efficient portable large scale eigenvalue package for distributed memory parallel architectures. *Lect. Notes Comp. Sci.*, 1184:478–486, 1996. (Cited on page 45.)

[94] K. Mendiratta and E. Polizzi. A threaded SPIKE algorithm for solving general banded systems. *Parallel Comput.*, 37(12):733–741, 2011. (Cited on page 79.)

[95] C. C. K. Mikkelsen and M. Manguoglu. Analysis of the truncated SPIKE algorithm. *SIAM J. Matrix Anal. Appl.*, 30(4):1500–1519, 2009. (Cited on page 79.)

[96] V. Minden, B. Smith, and M. G. Knepley. Preliminary implementation of PETSc using GPUs. In D. A. Yuen et al., editors, *GPU solutions to multi-scale problems in science and engineering*, pages 131–140. Springer, 2013. (Cited on pages 3 and 49.)

[97] C. Moler and C. F. V. Loan. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM Rev.*, 45(1):3–49, 2003. (Cited on page 118.)

[98] MPI Forum. MPI: a message-passing interface standard. *Int. J. Supercomp. Applic. High Perf. Comp.*, 8(3/4):159–416, 1994. (Cited on page 14.)

[99] T. H. Myer and I. E. Sutherland. On the design of display processors. *Commun. ACM*, 11(6):410–414, 1968. (Cited on page 21.)

[100] S. G. Narendra and A. Chandrakasan. *Leakage in nanometer CMOS technologies*. Springer US, New York, NY, 2006. (Cited on page 9.)

[101] NVIDIA. CUBLAS Library V9.2. Technical Report DU-06702-001_v9.2, NVIDIA Corporation, 2018. (Cited on page 31.)

[102] C. C. Paige. Accuracy and effectiveness of the Lanczos algorithm for the symmetric eigenproblem. *Linear Algebra Appl.*, 34:235–258, 1980. (Cited on page 42.)

[103] M. Palesi and M. Daneshtalab. *Routing Algorithms in Networks-on-Chip*. Springer, 2013. (Cited on page 10.)

[104] A. J. Park and K. S. Perumalla. Efficient heterogeneous execution on large multicore and accelerator platforms: Case study using a block tridiagonal solver. *J. Parallel and Distrib. Comput.*, 73(12):1578–1591, 2013. (Cited on pages 86 and 90.)

[105] E. Polizzi. Density-matrix-based algorithm for solving eigenvalue problems. *Physical Review B*, 79(11):115112, 2009. (Cited on page 45.)

[106] E. Polizzi and A. H. Sameh. A parallel hybrid banded system solver: the SPIKE algorithm. *Parallel Comput.*, 32(2):177–194, 2006. (Cited on pages 77, 78, and 79.)

[107] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Softw.*, 39(2):13:1–13:24, 2013. (Cited on page 45.)

[108] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Comput. Archit. News*, 42(3):13–24, 2014. (Cited on page 22.)

[109] E. Ramos, J. E. Roman, S. Cardona-Serra, and J. M. Clemente-Juan. Parallel implementation of the MAGPACK package for the analysis of high-nuclearity spin clusters. *Comput. Phys. Commun.*, 181(12):1929–1940, 2010. (Cited on page 54.)

[110] I. Reguly and M. Giles. Efficient sparse matrix-vector multiplication on cache-based GPUs. In *Innovative Parallel Computing (InPar)*, pages 1–12, 2012. (Cited on page 62.)

[111] J. E. Roman and P. B. Vasconcelos. Harnessing GPU power from high-level libraries: eigenvalues of integral operators with SLEPc. In *International Conference on Computational Science*, volume 18 of *Procedia Comp. Sci.*, pages 2591–2594. Elsevier, 2013. (Cited on page 3.)

[112] K. Rupp, A. Jüngel, and T. Grasser. Facing the multicore-challenge ii. chapter A GPU-Accelerated Parallel Preconditioner for the Solution of the Boltzmann Transport Equation for Semiconductors, pages 147–157. Springer-Verlag, Berlin, Heidelberg, 2012. (Cited on page 4.)

[113] K. Rupp and B. Smith. On level scheduling for incomplete lu factorization preconditioners on accelerators, 2013. (Cited on page 4.)

[114] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM Publications, 2nd edition, 2003. (Cited on page 80.)

[115] Y. Saad. *Numerical Methods for Large Eigenvalue Problems, Revised Edition*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2011. (Cited on page 37.)

[116] A. Sadeghi. Approximating the principal matrix square root using some novel third-order iterative methods. *Ain Shams Engineering Journal*, 2016. In press, DOI: 10.1016/j.asej.2016.06.004. (Cited on page 117.)

[117] G. Schulz. Iterative Berechung der reziproken Matrix. *Z. Angew. Math. Mech.*, 13(1):57–59, 1933. (Cited on page 117.)

[118] S. K. Seal, K. S. Perumalla, and S. P. Hirshman. Revisiting parallel cyclic reduction and parallel prefix-based algorithms for block tridiagonal systems of equations. *J. Parallel and Distrib. Comput.*, 73(2):273–280, 2013. (Cited on page 86.)

[119] R. Serban, D. Melanz, A. Li, I. Stanciulescu, P. Jayakumar, and D. Negrut. A GPU-based preconditioned Newton–Krylov solver for flexible multibody dynamics. *Int. J. Numer. Methods Eng.*, 102(9):1585–1604, 2015. (Cited on page 90.)

[120] R. B. Sidje. Expokit: a software package for computing matrix exponentials. *ACM Trans. Math. Software*, 24(1):130–156, 1998. (Cited on page 119.)

[121] B. L. Silver. *Irreducible Tensor Methods. An Introduction for Chemists.* Academic Press, London, 1988. (Cited on page 54.)

[122] J. C. Slater and G. F. Koster. Simplified LCAO method for the periodic potential problem. *Phys. Rev.*, 94(6):1498–1524, 1954. (Cited on page 108.)

[123] B. T. Smith, J. M. Boyle, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines: EISPACK Guide.* Springer, New York, NY, USA, second edition, 1970. (Cited on page 19.)

[124] D. C. Sorensen. Implicit application of polynomial filters in a k-step Arnoldi method. *SIAM J. Matrix Anal. Appl.*, 13:357–385, 1992. (Cited on page 43.)

[125] A. Stathopoulos and J. R. McCombs. PRIMME: PReconditioned Iterative MultiMethod Eigensolver: Methods and software description. *ACM Trans. Math. Software*, 37(2):21:1–21:30, 2010. (Cited on page 45.)

[126] J. Steward, A. Aksoy, and Z. Haddad. Parallel direct solution of the Ensemble Square-Root Kalman Filter equations with observation principal components. *J. Atmos. Ocean. Tech.*, 34(9):1867–1884, 2017. (Cited on page 132.)

[127] J. Steward, J. E. Roman, A. Lamas Daviña, and A. Aksoy. Parallel direct solution of the covariance-localized Ensemble Square-Root Kalman Filter equations with matrix functions. *Mon. Weather Rev.*, 146(9):2819–2836, 2018. (Cited on pages 123, 132, and 139.)

[128] G. W. Stewart. A Krylov–Schur algorithm for large eigenproblems. *SIAM J. Matrix Anal. Appl.*, 23(3):601–614, 2001. (Cited on page 43.)

[129] G. W. Stewart. *Matrix Algorithms. Volume II: Eigensystems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001. (Cited on page 37.)

[130] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw microprocessor: a computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002. (Cited on page 23.)

[131] The TOP500 Supercomputer Sites. `https://www.top500.org/`. Accessed on 23 Mar 2018. (Cited on pages 11 and 23.)

[132] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Comput.*, 36(5-6):232–240, 2010. (Cited on pages 32 and 86.)

[133] A. Vajda. *Programming Many-Core Chips*. Springer, Boston, MA, 2011. (Cited on page 10.)

[134] T. Van Hook. Real-time shaded NC milling display. *SIGGRAPH Comput. Graph.*, 20(4):15–20, 1986. (Cited on page 24.)

[135] P. B. Vasconcelos, O. Marques, and J. E. Roman. Parallel eigensolvers for a discretized radiative transfer problem. In J. M. L. M. Palma et al., editors, *High Performance Computing for Computational Science–VECPAR 2008*, volume 5336 of *Lect. Notes Comp. Sci.*, pages 336–348. Springer, 2008. (Cited on page 98.)

[136] I. Venetis, A. Kouris, A. Sobczyk, E. Gallopoulos, and A. Sameh. A direct tridiagonal solver based on Givens rotations for GPU architectures. *Parallel Comput.*, 49:101–116, 2015. (Cited on page 89.)

[137] I. E. Venetis, A. Sobczyk, A. Kouris, A. Nakos, N. Nikoloutsakos, and E. Gallopoulos. A general tridiagonal solver for coprocessors: Adapting g-Spike for the Intel Xeon Phi. In G. R. Joubert et al., editors, *Parallel Computing: On the Road to Exascale*, pages 371–380. IOS Press, 2015. (Cited on page 90.)

[138] C. Vomel, S. Tomov, and J. Dongarra. Divide and conquer on hybrid GPU-accelerated multicore systems. *SIAM J. Sci. Comput.*, 34(2):C70–C82, 2012. (Cited on page 4.)

[139] S.-H. Weng, Q. Chen, N. Wong, and C.-K. Cheng. Circuit simulation via matrix exponential method for stiffness handling and parallel processing. In *Proceedings of IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)*, pages 407–414, 2012. (Cited on page 119.)

[140] J. S. Whitaker and T. M. Hamill. Ensemble data assimilation without perturbed observations. *Mon. Weather Rev.*, 130(7):1913–1924, 2002. (Cited on page 122.)

[141] K. Wu and H. Simon. Thick-restart Lanczos method for large symmetric eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 22(2):602–616, 2000. (Cited on page 44.)

[142] Z. Xianyi, W. Qian, and Z. Yunquan. Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 684–691, 2012. (Cited on page 19.)

[143] P. Yalamov and V. Pavlov. Stability of the block cyclic reduction. *Linear Algebra Appl.*, 249(1):341–358, 1996. (Cited on page 75.)

[144] I. Yamazaki, T. Dong, R. Solcà, S. Tomov, J. Dongarra, and T. Schulthess. Tridiagonalization of a dense symmetric matrix on multiple GPUs and its application to symmetric eigenvalue problems. 26:2652–2666, 2014. (Cited on page 4.)

[145] C. D. Yu and W. Wang. Performance models and workload distribution algorithms for optimizing a hybrid CPU-GPU multifrontal solver. *Comput. Math. Appl.*, 67(7):1421–1437, 2014. (Cited on page 4.)

[146] Y. Zhang, J. Cohen, and J. D. Owens. Fast tridiagonal solvers on the GPU. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 127–136, 2010. (Cited on pages 73 and 86.)

[147] F. Zheng, H.-L. Li, H. Lv, F. Guo, X.-H. Xu, and X.-H. Xie. Cooperative computing techniques for a deeply fused and heterogeneous many-core processor architecture. *J. Comput. Sci. Tech.*, 30(1):145–162, 2015. (Cited on page 23.)