UNIVERSIDAD JAUME I DE CASTELLÓN
E. S. DE TECNOLOGÍA Y CIENCIAS EXPERIMENTALES

Matrix Computations on Graphics Processors and Clusters of GPUs

Castellón, May 2011

Presented by: Francisco Daniel Igual Peña
Supervised by: Gregorio Quintana Ortí
Rafael Mayo Gual

UNIVERSIDAD JAUME I DE CASTELLÓN
E. S. DE TECNOLOGÍA Y CIENCIAS EXPERIMENTALES

Matrix Computations on Graphics Processors and Clusters of GPUs

Francisco Daniel Igual Peña

Contents

List of Figures

List of Tables

Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.

Edsger Dijkstra
EWD896 - *On the nature of Computing Science*

Agradecimientos

Estos han sido cuatro años especialmente intensos. Decenas de aviones, países y ciudades han pasado por delante de mis ojos, y he tenido la increíble oportunidad de conocer a muchas personas realmente interesantes. Algunas me han hecho ver las cosas de un modo distinto, tanto a nivel profesional como personal. De entre todas ellas, quisiera expresar mi más sincero agradecimiento a aquellas que han jugado un papel decisivo en el desarrollo de esta tesis.

A mis directores Gregorio Quintana Ortí y Rafael Mayo Gual. A Greg, por ser una fuente infinita de conocimientos y un trabajador incansable. A Rafa, por respetar mi forma de trabajar y por animarme en los malos momentos.

A Enrique Quintana Ortí, que confió en mí desde el primer día, lo cual siempre le agradeceré.

Gracias a la Universidad Jaume I, Generalitat Valenciana, Ministerio de Ciencia e Innovación, Fundación Caixa Castelló/Bancaixa, Microsoft Research y Nvidia, por el apoyo económico prestado durante estos años, sin el cual este trabajo habría sido del todo imposible.

A todos mis compañeros en el grupo HPC&A, José, José Manuel, Sergio, Maribel, Juan Carlos, Germán, Merche, Alfredo, Asun, Manel y Toni. Gracias especialmente a Alberto, por enseñarme día a día lo que es realmente el espíritu científico, y por todas esas (en ocasiones interminables) discusiones filosóficas.

A todo el personal de administración y servicios de la UJI, y muy especialmente del Departamento de Ingeniería y Ciencia de los Computadores.

Mi más sincero agradecimiento a los compañeros de la Universidad de Málaga, Barcelona Supercomputing Center-Centro Nacional de Supercomputación, INESC-ID en Lisboa y de la Universidad de Texas en Austin, por brindarme la posibilidad de trabajar junto a ellos y por compartir sus conocimientos y amistad. Al personal del TACC (*Texas Advanced Computing Center*), por ofrecernos acceso exclusivo al increíble cluster LONGHORN.

Gracias al profesor Robert van de Geijn, por abrirme las puertas de su casa y hacerme sentir en la mía. A Manuel Ujaldón por sus consejos, su inagotable ilusión y su ayuda durante todo este tiempo.

A mis amigos en La Vall y en Cortes, y a toda mi familia. Lo creáis o no, aprecio profundamente cada pequeño momento que hemos compartido durante todo este tiempo. Me habéis ayudado mucho

más de lo que pensáis. A Javi y a Amparo. Y a Inma, por hacerme ver que las cosas fáciles no suelen valer la pena, por saber escucharme y por darme ese último empujón que tanto necesitaba.

A mis padres, Paco y Rosa, por darme la oportunidad de recibir la mejor educación, por respetar todas y cada una de mis decisiones y por su comprensión.

Y a Rosa Mari, la mejor hermana que cualquiera pueda imaginar.

Castellón de la Plana, mayo de 2011.

Financial support

Thanks to the University Jaume I, Generalitat Valenciana, Ministry of Science and Education and Fundación Caixa Castelló/Bancaixa for the economic support during these four years. To Microsoft Research and Nvidia for their interest in the research developed in this thesis, and their economic support.

Part I

Introduction

CHAPTER 1

Matrix computations on systems equipped with GPUs

GPUs (Graphics Processing Units) have become an appealing solution for High Performance Computing (HPC) in terms of performance and acquisition cost. As of today, many computational problems that five years ago were reserved to huge distributed-memory clusters can be solved in commodity workstations equipped with this type of hardware.

The evolution of the graphics hardware has also implied a revolution in the way GPUs are programmed. Novel programming models like, e.g., CUDA, OpenCL or Brook+, do not require anymore an advanced knowledge of intricate graphics-oriented APIs (Application Programming Interfaces), which were a major barrier to general-purpose computing only a few years ago.

This chapter motivates the usage of modern GPUs as an accelerating architecture for HPC. In particular, we take into account their weaknesses and strengths, in order to present the goals to be achieved in the framework of this thesis concerning the use of GPUs as an accelerating coprocessor for linear algebra operations. In addition, some common concepts necessary for the correct understanding of the document are introduced here.

The chapter is structured as follows. Section 1.1 motivates the usage of GPUs as an HPC device, and introduces the factors that have led to the emergence of this type of architecture as a feasible solution in this area. Section 1.2 summarizes the main motivation and goals of the thesis, and reviews the overall structure of the document. Section 1.3 describes the software and hardware infrastructure used for the experimental results in the rest of the document. Finally, Section 1.4 introduces some common concepts of the FLAME notation and methodology used through the rest of the document.

1.1. Introduction

1.1.1. The evolution of hardware for High Performance Computing

Several decades after Gordon Moore dictated his famous law [105], his predictions are still up-to-date. The exponential growth rate in the transistor density dictated by Moore's Law has been valid since 1965, and the trend is expected to continue until 2015 or even later.

Moore's Law states that the number of transistors that can be placed in an integrated circuit with an affordable cost roughly doubles every two years. Although the assumption is perfectly valid

nowadays, it is important to realize that Moore's Law actually addresses transistor density, not performance. Today, it seems clear that a larger number of transistors does not always yield higher performance: the manner in which transistors are used is what ultimately determines performance, as well as the type of applications that naturally adapt better to each architecture.

The strategies to exploit this exponential growth in the number of transistors to attain higher performance have dramatically changed during the last two decades. During the 90s (and the early 2000 decade), performance improvements were strongly rooted on (the increase of) processor frequency. With the technological limits still far, no problems were observed in the horizon. Execution time of sequential programs could be reduced for free (from the programmer's viewpoint) with the only effort of acquiring processors running at a higher pace (in some cases, this approach required to wait for the next generation of processors, just two years away). No major problems were devised from the software side beyond the development of sophisticated compilers or the exploitation of vector units, to name only two. In this sense, sequential programs were universally prevalent as performance requirements were satisfied by the continuous increase in the frequency of *single-core processors*.

However, in the mid 2000s, the frequency limit of the current technology was reached, and computer architects adopted a new strategy to continue exploiting Moore's Law and the increasing transistor density. *Multi-core* processors arose as the response of the industry to the frequency barrier in both desktop and HPC markets, as well as two other major barriers: memory latency and limited amount of instruction-level parallelism, or ILP. The approach aimed at keeping a relatively low and constant frequency, while increasing the number of processing units per die. The success of multi-core processors involved a second revolution from the software perspective. Sequential programs were no longer valid to exploit the novel parallel architectures, and concurrent implementations of the existing programs and libraries, together with novel programming paradigms, rapidly appeared as a response to the new architectures.

Parallel programming is not a novel discipline in HPC. Indeed, explicit concurrent programming has been practiced and improved for decades. However, the platforms where parallel programs have been historically executed were only justified for some elitist applications, where the demand of billions of FLOPS (floating-point arithmetic operations per second) was a must. With the popularization of multi-core chips, parallel programming has become a real necessity, not a matter for only a few specific performance-demanding applications.

The search for high performance did not stop with the advent of multi-core processors, and the next step in the process was the introduction of the so-called *many-core processors*, with dozens to hundreds of simple cores. This trend translated into the appearance of novel and revolutionary architectures, and the redesign of specific-purpose processors to adapt them to general-purpose computation. This was the case of the Graphics Processors, or GPUs.

A second design factor that will ultimately determine the design, development and success/failure of new architectures in the near future is power consumption. *Heterogeneous processors* and platforms, where specific parts are activated only when the application requires them, is a possible solution to the power consumption problem. In fact, fine-grained power consumption has already been introduced in new many-core chips, as an advance of what they will provide in the near future. This is the case, e.g., of the Intel SCC processor [81].

In the long run, processors specifically designed and built for a given application may be the only alternative to accomplish the performance/power trade-off. Unless the current technology experiences a dramatic change, *special-purpose processors* will possibly be the last response of industry to the ever growing high-performance demands.

The above discussion is captured in Figure 1.1, which illustrates the evolution of the performance of the architectures as Moore's Law has evolved through the years.

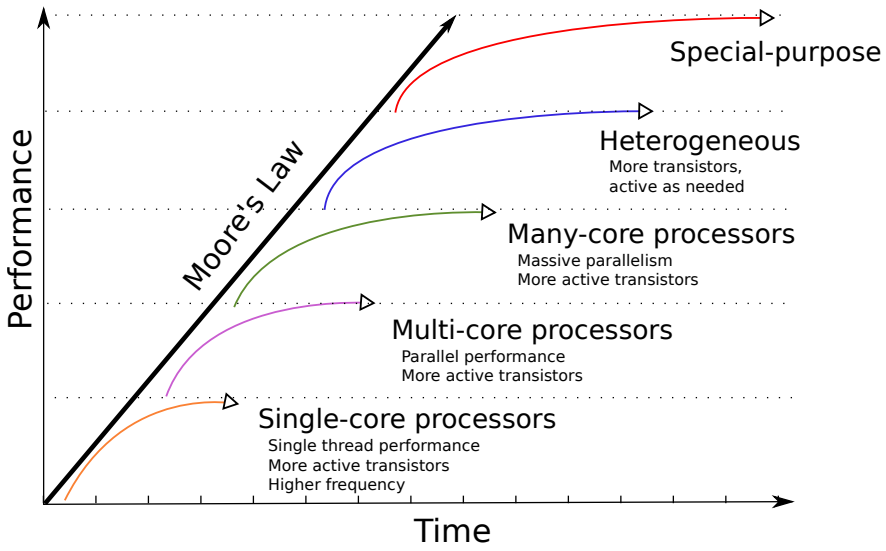


Figure 1.1: Evolution in performance of processor designs in response to Moore's Law. Figure based on a slide from P. Hofstee, IBM Austin.

As of today, multi-core and many-core processors, either with a general-purpose or designed specifically for a given task (e.g. GPUs), are a hot topic in HPC. In the near future, the integration of CPU and GPU (or even other type of processors) will likely lead to the design of heterogeneous processors, in which the ideas exposed in this dissertation will be of vital importance.

1.1.2. The GPU as a high-performance, general-purpose processor

As discussed in the previous section, two hardware solutions have appeared in response to the triple hurdles of frequency, memory latency, and limited ILP. Multi-core designs aim at maintaining performances by replicating relatively complex cores, keeping a reduced number of cores per chip. On the other side, many-core designs focus on execution throughput by replicating smaller cores at a higher number.

Following Moore's Law, the number of cores in modern GPUs have doubled with each new generation. Consider, for example, the last three generations of NVIDIA GPUs. The G80 architecture, released in 2006, featured a total of 128 cores. The GT200, introduced two years later, presented 240 cores (plus additional double precision units). The last generation of NVIDIA GPUs, Fermi (released in 2010), exhibits a total of up to 480 cores per chip. This impressive amount of processing units is translated into a remarkable floating-point performance increment since the GPU revolution in 2006. While current general-purpose processors exhibit performances in the order of a few hundreds of GFLOPS (10^9 FLOPS), modern GPUs can easily reach the TFLOPS (10^{12} FLOPS) barrier. However, these are theoretical peak performances, and they are not necessarily achievable by common applications. Indeed, not all types of applications fit well into the GPU architecture. To understand the gap in raw performance between GPUs and CPUs, we next dig into the fundamental design decisions behind each type of processor, and the way that the increasing number of transistors has been exploited.

General-purpose processors are designed to efficiently execute several sequential complex codes. Thus, a major part of the transistors are devoted to both control logic (to control out-of-order execution and handle branch prediction, for example) and cache memory (to exploit locality of reference and hide memory access latencies). However, these elements contribute little to boost

the raw arithmetic performance of the processor. On the other hand, the market to which GPUs are devoted dictates that attaining a high FLOPS rate is crucial. Thus, the design philosophy underlying graphics processors is dramatically different from that adopted by CPU manufacturers. In particular, the percentage of die area (transistors) devoted to control or on-chip memory in graphics processors is significantly more reduced than in a general-purpose device. Instead, a major part of the silicon is devoted to floating-point arithmetic units, as the ultimate goal is to optimize the floating-point throughput of the processor.

The amount of cores per chip and their complexity is not the only factor that differentiates both types of devices. Memory bandwidth is a second distinguishing feature between CPUs and GPUs. The high bandwidth demand required by graphics applications implies that GPU memories and interfaces are one or even two generations ahead from those used in general-purpose processors. As an example, the NVIDIA GTX480 chip has a peak memory bandwidth of 177 GB/s, whereas CPUs are not expected to deliver 50 Gb/s in the next 3 years [89].

Those characteristics ultimately determine the type of applications that a GPU can efficiently execute. Programs with a high degree of data parallelism (e.g., embarrassingly parallel), with no execution divergences (branches) and high arithmetic intensity (floating-point operations per memory transaction), are clear candidates to yield high performance on graphics processors. On the other side, inherently sequential codes, with complex data dependencies, or without evident data parallelism, will fit better to CPUs. In response to this, many scientific, engineering and industrial applications have been recently modified to port their most data-intensive parts to the GPU. This hybrid computation is the natural scenario for hybrid CPU-GPU systems, and will be addressed from many viewpoints in this dissertation.

Performance is the most appealing feature of modern GPUs. However, given the high demands of floating-point arithmetic operations per second of current applications, even the most modern GPUs cannot deliver enough performance. After NVIDIA released its first programmable GPU in 2006, the company soon realized it might not be enough for certain performance-demanding areas. The introduction of platforms with more than one GPU (multi-GPU systems) targeted this type of applications. However, because of the communication bottleneck between CPU and accelerators due to the chipset (and the PCIExpress bus), it is difficult to find platforms equipped with more than four GPUs. In response to this, the construction of clusters of GPUs with a few graphics processors per node and fast interconnection networks seems the trend in the search of future high-performance GPU-based systems. As a demonstration of this, three of the most powerful machines in the latest Top500 list (November 2010) are clusters with GPUs attached to its nodes [134]. These new heterogeneous HPC designs clearly justify the research on new techniques, methodologies, and implementations to exploit the potential that those machines can offer.

1.1.3. The programmability issue on novel graphics architectures

The hardware revolution introduced with the CUDA architecture entailed a similar revolution from the software perspective. For the first time, the main burden towards the standardization of GPGPU (general-purpose computing on graphics processors), the programmability issue, was seriously targeted.

The introduction of the CUDA software infrastructure is a significant advance in this field. However, many critical decisions are still in the programmer's hands. Many of these decisions involve a remarkable programming effort just to test their effectiveness. These key factors are still a barrier for the port of existing codes which yield high-performance.

Let us consider the three different GPU-based architectures mentioned at the end of the previous section to illustrate common difficulties in the development of GPGPU codes from the programmer's perspective.

On systems equipped with one single GPU, CUDA still exposes many parameters with high impact on performance. In addition, many of them are exclusively related to this specific type of architectures. Examples include explicit management of on-chip memory, aligned accesses to GPU memory, divergence control, correct adaptation to the SIMD programming paradigm, or convenient register usage, to name only a few. What is more important, many of these particularities usually vary from generation to generation of graphics processors. Thus, the programming effort invested to tune a particular routine for a specific GPU has to be repeated when a new processor appears.

In the framework of linear algebra, our aim is to demonstrate that a high-level approach can also be valid to attain high performance on this type of architectures. A low-level implementation approach on GPU computing requires a detailed knowledge of the graphics architecture and intricate programming techniques, usually reserved to a few experts like Kazushige Goto [99] on the CPU side. Application programmers usually do not have such deep programming abilities and architecture knowledge. Difficulty of programming and low code reuse are the main drawbacks of the low-level strategy.

Luckily, if a few, thoroughly tuned building blocks can be used to build new linear algebra routines, the programming effort can be focused exclusively on that reduced number of kernels. The rest of the implementations can then be directly built on top of them, and directly benefit from their high performance. We will demonstrate how this approach is perfectly valid in the framework of linear algebra implementations on GPUs. Similar ideas can be applied without major conceptual changes to other type of accelerated platforms.

Similar or even more challenging problems appear on multi-GPU systems. In this type of architectures, in addition to the programmability problems mentioned for single GPUs, programmers have to efficiently manage the bottleneck introduced by the shared PCIExpress bus, and the existence of separate memory spaces per GPU. Thus, a reduction in memory transfers is a must to achieve high performance. The impossibility of performing direct GPU to GPU communications adds an additional burden towards the achievement of high performance. Additionally, load balancing between all the execution resources in the system is also an important factor to deal with.

Putting all those parameters in the programmer's hands is a barrier if programmability and performance have to be addressed simultaneously. Ideally, a system with enough intelligence to deal with all of them would be a solution to ease programming while maintaining high performance rates. We will see how to develop and effectively use such a software system in the framework of linear algebra computations on current multi-GPU systems.

For accelerated distributed-memory architectures, another important question arises: Is it absolutely necessary to rebuild current linear algebra libraries from scratch? A positive answer would have a dramatic impact on existing non-accelerated user codes. We will demonstrate that, if the chosen library is well designed, an easy port to this class of accelerated architectures is possible. In particular, the impact on existing codes can be minimal, and the existence of accelerators attached to each node in the cluster can be transparent for the programmer.

1.2. About this document. Motivation and structure

1.2.1. State-of-the-art of linear algebra computation on GPUs

Since the emergence of the term "GPGPU" in 2001, linear algebra and GPU computing have been closely bound. The first efforts towards efficient linear algebra implementations on graphics processors began with the emergence of programmable graphics processors. Larsen et al. [94] introduced the first strategies for the efficient implementation of the matrix-matrix multiplication using graphics-oriented APIs. Fatahalian et al. [63] went one step further in 2004, analyzing and proposing novel optimized strategies towards efficient implementations of the same operation. Actually, this contribution was the origin of the growing interest of linear algebra implementations on the GPU. More complex routines were also studied before the formulation of the CUDA architecture. For example, the authors of [66] proposed the first factorizations implemented on the GPU using graphics-oriented APIs.

With the introduction of the CUDA architecture in 2006, the number of works related with linear algebra on GPUs dramatically increased. Among them, Volkov and Demmel's [142] is one of the most referenced papers; there the authors thoroughly analyze the performance of modern GPUs, and design ad-hoc CUDA implementations to exploit all the potential of the graphics processor for some basic linear algebra implementations and the most common factorizations (Cholesky, LU with partial pivoting and QR).

Simultaneously with the evolution of the architectures towards systems with multiple GPUs, the scientific community also pursued the adaptation of the libraries to novel multi-GPU systems. In [139], the authors proposed one of the first implementations of the LU factorization with partial pivoting on a machine with two GPUs. The approach adopted in the paper was based on a column-cyclic layout of the matrix among the GPUs. This static approach has also been taken by other authors, and is one of the most recurrent solution mainly because its simplicity.

However, for this type of architectures, an alternative solution is to develop intelligent run-time systems that automatically extract the inherent parallelism existing in many dense linear algebra algorithms, while transparently dealing with data transfers between the different memory address spaces. In this line, the author of this thesis was one of the precursors of this approach in 2008 [39, 114]. Other studies have followed this preliminary proposal for linear algebra implementations. In [96], the authors propose scalable hybrid implementations of the Cholesky factorization for multi-core systems with several GPUs. This work was extended in [133] to cover a wider range of solvers in the framework of the MAGMA project [6]. The aim of the project is to develop a LAPACK-like implementations for novel architectures based on GPU and multi-GPU systems. However, in their latest release (version 1.0, December 2010), the MAGMA library does not provide multi-GPU support. Also, many of the approaches followed by the library for single-GPU and multi-GPU implementations were previously investigated and published independently in the framework of this thesis.

The solutions contributed by the MAGMA project, or other studies related to GPU computing for dense linear algebra, usually do not address one particular characteristic: *programmability*. Most of those pursue raw performance, without taking into account how easy is for the programmer to attain that performance. In fact, many of the implementations and improvements presented are ad-hoc designs for a specific architecture, and there is little guarantee that the implicit insights will remain valid for future architectures. Thus, each architectural change will in general require a deep redesign of the algorithms and implementations in these solutions, and a full reevaluation of the new codes. In addition, no systematic methodologies for the evaluation of key parameters in the new implementations were proposed in these works.

Few efforts have been made towards the transformation of the GPUs into a friendly environment for the library developer. To cover this field, FLAME (*Formal Linear Algebra Methods Environment*) [136, 75, 76, 28] is presented as an effective solution in the quest of high-performance, easy-to-develop dense linear algebra libraries. Remarkable results have been obtained from the application of the FLAME methodology to multi-core architectures [120, 114]. Many of the works developed within the framework of the FLAME project and GPU computing are in the context of this thesis. As an additional contribution of this thesis, APIs for fast and reliable code development have been implemented [21, 148] with the GPU as the underlying architecture in the framework of the FLAME project. Even *out-of-core* codes have also exploited the conjunction of both approaches [101, 100] in the linear algebra arena for large problems, which demonstrates the flexibility of the FLAME approach to adapt to novel architectures without traumatic changes from the programmer's perspective.

Precisely large-scale linear algebra problems have been historically solved on large computing clusters. With the appearance of GPUs, there is a good reason for equipping each one of the nodes of the cluster with a graphics processor. However, the porting of existing distributed-memory linear algebra libraries to clusters of GPUs is a novel and barely explored area, though GPU-based clusters are nowadays emerging as the top performance machines in HPC rankings [134]. Given the increasing interest in this type of architectures, new solutions are demanded in order to port well-known libraries such as ScaLAPACK [11]. Some works have demonstrated that ad-hoc implementations for linear algebra operations on distributed-memory machines with hardware accelerators can be very efficient [69, 68]. Nevertheless, once more no research is focused on a transparent port of existing libraries to these novel architectures.

In response to these deficiencies, this thesis covers the simultaneous exploitation of *programmability* and *performance* on single-GPU architectures, multi-GPU systems and clusters with GPUs.

1.2.2. Motivation and goals

Linear algebra operations lie at the bottom of the "food chain" for many scientific and engineering applications. The performance that can be attained for these operations is often the key factor that determines the global performance of applications. Also, the performance attained by some basic linear algebra operations is often employed as an illustrative example of the potential of a given architecture. Widely spread benchmarks, like LINPACK [56], base their functionality in the analysis of the performance attained by some well-known linear algebra methods.

The usage of linear algebra applications for exploring the capabilities of novel architectures is frequent. Thus, vendors usually devote considerable efforts to develop highly-tuned implementations of a few basic linear algebra operations to illustrate the capabilities of their new products.

In this dissertation we use linear algebra operations as the conducting thread for the investigation of the possibilities offered by novel architectures based on graphics processors in HPC. Three different architectures are targeted: systems with one GPU, systems with multiple GPUs, and clusters of GPUs. For each architecture, we propose a number of techniques to obtain *high-performance* linear algebra operations.

A second relevant factor considered in the development of the proposed solutions is *programmability*. This has become a key feature of today's implementations on multi-core and many-core architectures. Some efforts have been made in the past few years towards the solution of the programmability problem for GPUs, multi-core processors or specialized processors like the Cell B.E. [39, 109, 110].

To further advance in this area, as an additional contribution of our work, each technique and strategy proposed throughout our work will take into account programmability keeping an eye on reducing the development effort for the programmer while maintaining performance.

The main goal of this thesis is thus *to design, develop and evaluate programming strategies to improve the performance of existing dense linear algebra libraries on systems based on graphics processors*. This general goal is divided into the following specific goals:

- To design and develop blocked implementations for the main Level-3 BLAS based on a few highly tuned routine implementations, and to derive a set of algorithmic variants for each one of them.
- To develop a full set of algorithmic variants for common dense factorizations and to apply hybrid approaches to improve performance and accuracy.
- To evaluate new implementations for the Level-3 BLAS and LAPACK-level routines on systems equipped with one GPU, and to compare them with tuned implementations on general purpose processors and vendor-specific implementations on graphics processors (NVIDIA CUBLAS).
- To propose an efficient run-time system focused on systems with multiple GPUs which deals with data transfers and task scheduling transparently from the programmer's point of view. The run-time system will implement different data transfers policies with the goal of reducing the amount of data transfers.
- To evaluate the run-time system on a multi-GPU architecture for a representative set of BLAS and LAPACK-level routines, and to analyze the impact of the different alternatives on the performance of the solution.
- To select and modify a well-known dense linear algebra library for distributed-memory machines (PLAPACK) for clusters in which each node is equipped with one or more GPUs.
- To evaluate the library on a large GPU cluster, and to compare the attained performance with that of a purely CPU-based implementation.

These specific goals are developed under two common premises:

Programmability The codes must be easy to develop for the library programmer. Algorithmic variants must be easy to derive and systematically evaluate when necessary. If possible (e.g., in multi-GPU systems) work scheduling and data transfers must be transparent to the programmer. In general, the development of optimized routines cannot be traumatic for the programmer. In this sense, the goal is to show how *high-level approaches* help in exploring the capabilities of novel architectures, and optimally exploiting them.

Performance The target of our implementations is efficiency. That is the major reason for the usage of GPUs first, and systems with multiple GPUs and clusters of GPUs as their natural evolution. Our aim is to obtain efficient codes for dense linear algebra operations, as well as methodologies and techniques to develop future codes without major programming efforts.

Although these two concepts seem incompatible, this is not necessarily the case: a good development methodology, with systematic derivation and evaluation of different alternatives (in other

words, *programmability*) is the key to rapidly test all the possible parameters that influence the execution times of the codes (that is, *performance*).

On the hardware side, the aim is to address on three different architectures, representative of the state-of-the-art in GPU architectures. These systems comprise platforms with one GPU, systems with multiple GPUs, and clusters with one or more GPUs per node. We will often select a reduced set of operations to illustrate the techniques presented for each architecture. However, we believe that those routines are illustrative enough of the ideas and techniques introduced in the thesis. In fact, in general these ideas can be easily adapted to many other linear algebra operations without a significant effort or conceptual changes.

In summary, in this dissertation we will not pursue an exhaustive analysis of a full set of linear algebra routines. Instead, we will propose *methodologies*, *techniques* and also *implementations* that are general enough to be adapted to any linear algebra routine on each type of GPU-based architecture, with minor impact in the programmer's productivity.

1.2.3. Structure of the document

The manuscript is structured in four parts. Part I offers the basic concepts necessary to understand the rest of the document. In this part, Chapter 1 introduces the reasons underlying the usage of the GPU as a general-purpose computing architecture. In addition, it describes the motivation, main goals, and structure of the document. Chapter 2 describes the architecture of modern graphics processors and briefly introduces the evolution of this type of accelerators.

The three remaining parts naturally correspond to the work with each one of the three different classes of platforms targeted in our work: systems with one GPU, systems with more than one GPU, and clusters of GPUs.

Part II addresses the study of basic linear algebra subroutines (Chapter 3) and linear factorizations (Chapter 4) on systems equipped with a single GPU. Chapter 3 addresses the evaluation and tuning of Level-3 BLAS, comparing their performance with those for existing and optimized BLAS implementations on graphics processors (NVIDIA CUBLAS). The approach taken in this chapter casts BLAS operations in terms of a highly-tuned matrix-matrix multiplication implementation in NVIDIA CUBLAS, while exploring the performance of several algorithmic variants for each routine. Chapter 4 presents GPU implementations for the Cholesky and LU factorizations, and the reduction to tridiagonal form. The chapter also introduces hybrid algorithms in which CPU and GPU cooperate to improve performance, and revisits the iterative refinement technique to regain full accuracy while exploiting the capabilities of modern GPUs when operating in single precision.

Part III pursues the design and development of similar implementations on multi-GPU systems. The approach described in Chapter 5 moves the burden of task scheduling and data transfer management from the programmer. A run-time system keeps track of the availability of data on GPU memories while controlling which tasks have all dependencies solved and, therefore, are ready to be dispatched to the different processing units. The runtime operation and related mechanisms are illustrated for the Cholesky factorization, and experimental results for common BLAS-3 operations are also given in this chapter.

Part IV presents our programming solution for clusters with one or more GPUs attached to each node. The strategy ports a well-known dense linear algebra infrastructure (PLAPACK) to clusters of GPUs. As in the other two platform classes, the reasons underlying this decision are programmability and performance. Chapter 6 overviews the fundamentals of PLAPACK, and details the necessary changes to adapt its contents to a GPU-accelerated cluster. Results are reported on a large GPU cluster for two well-known linear algebra operations: the matrix-matrix multiplication and the Cholesky factorization.

Each chapter of this document presents the developed work as well as the experimental results attained for the corresponding architecture. In this sense, each part of the document is self-contained and can be read independently.

Finally, Chapter 7 presents the main conclusions from this research. In addition, it reports the main contributions of the thesis, the publications that have been generated, and the technological transfer activities derived from it. Finally, a few open research lines related to the work are discussed.

1.3. Description of the systems used in the experimental study

1.3.1. Performance metrics

The fundamental metric for performance evaluation (or efficiency) of an application is the *execution time*. However, codes with intensive floating-point arithmetic operations, as is the case of linear algebra operations, often employ other metrics to evaluate the pace at which these operations are performed. More precisely, the definition of *flop* is usually bound to a floating-point arithmetic operation. Thus, the execution speed of a linear algebra code is usually given in terms of MFLOPS (10^6 flops/s), GFLOPS (10^9 flops/s), or even TFLOPS (10^9 flops/s). Although the FLOPS rate is a metric derived from the execution time, the arithmetic processing speed (flops/sec) presents a clear advantage in the graphical representation of performance data. Specifically, as the problem size is increased, the execution time of codes for common dense linear algebra operations also increases proportionally (often at a cubic pace). However, the FLOPS rate is limited by the configuration and speed of the hardware (cycle time, amount of functional units, cache transfer rate, bus speed, etc.) Thus, the charts representing the FLOPS rate present an upper bound that makes them much easier to display and analyze.

Although there exist widely extended metrics such as acceleration or efficiency for parallel codes, such as the GPU implementations (also derived from the execution time), we advocate here for the homogeneity in the representations, and we will mostly measure parallel performance in terms of FLOPS. Nevertheless, whenever necessary, we will use other metrics to correctly illustrate parallel performance. In those specific cases, specific metrics will be introduced when necessary.

1.3.2. Hardware description

Three different systems have been used in the evaluation of the implementations presented in the following chapters. Those systems are representative of the different multi-core architectures present nowadays and, simultaneously, they illustrate how multiple hardware accelerators (in this case, GPUs) can be attached to a single system or a cluster of compute nodes to boost performance.

PECO is a cluster of four nodes interconnected using an Infiniband QDR network. Each node contains two Intel Xeon 5520 (Nehalem) Quadcore processors running at 2.27 Ghz, with 24 Gbytes of DDR2 RAM memory. Attached to the PCIExpress 2.0 bus of each node, there is a NVIDIA C1060 GPU with 4 Gbytes of DDR3 RAM memory. One of the nodes in this machine will be used for the evaluation stage of BLAS and LAPACK-level routines in Chapters 3 and 4.

TESLA2 is a shared memory multiprocessor equipped with the Intel Xeon technology. It is composed of two Intel Xeon 5440 (Harpertown) Quadcore processors running at 2.83 Ghz, with 16 Gbytes of DDR2 RAM memory. Attached to the PCIExpress 2.0 bus, there is a NVIDIA s1070 system consisting of four NVIDIA Tesla C1060 GPUs identical to those present in each

	PECO	TESLA2
Type of machine	Cluster of SMP	SMP
Number of nodes	4	-
Processor	Intel Xeon E5520	Intel Xeon E5440
Processor codename	Nehalem	Harpertown
Frequency	2.27 Ghz	2.83 Ghz
Number of cores	8	8
Available memory	24 Gbytes DDR2	16 Gbytes DDR2
Interconnection network	Infiniband QDR	-
GPU	NVIDIA Tesla C1060	NVIDIA Tesla S1070
Interconnection bus	PCIExpress 2.0	PCIExpress 2.0
Available memory	4 Gbytes DDR3	16 Gbytes DDR3

Table 1.1: Description of the hardware platforms used in the experiments. The features of PECO are referred to each node of the cluster.

node of PECO. This machine will be the base for the experimental process of the multi-GPU implementations described in Chapter 5.

LONGHORN is a distributed-memory machine composed of 256 nodes based on the Intel Xeon technology. It presents four Intel Xeon 5540 (Nehalem) Quadcore processors per node running at 2.13 Ghz, with 48 Gbytes of DDR2 RAM memory each. Attached to the PCIExpress 2.0 bus of each node, there are two NVIDIA Quadro FX5800 GPUs with 4 Gbytes of DDR3 RAM memory. Nodes are interconnected using a fast QDR Infiniband network. This machine will be the testbed for the experimental stage in Chapter 6.

A more detailed description of the hardware can be found in Tables 1.1 and 1.2. In the selection of those machines, we have chosen those graphics processors and general-purpose processors as close in generational time as possible. This makes it possible to fairly compare single-GPU, multi-GPU and distributed-memory implementations with minimal deviations.

1.3.3. Software description

The software elements used in our work can be divided in two different groups. The GPU-related software is selected to be as homogeneous as possible to allow a fair comparison of the different platforms. Note that many of the routines presented in the document employ the BLAS implementation (NVIDIA CUBLAS). This implies that experimental results will ultimately depend on the underlying BLAS performance. However, each technique introduced in this research is independent of the specific BLAS implementation. Thus, we expect that when future releases of the NVIDIA CUBLAS library (or other alternative implementations of the BLAS) appear, the performance attained by our codes will be improved in the same degree as the new BLAS implementation. In all the experiments, we employed CUDA 2.3 and CUBLAS 2.3. MKL 10.1 has been used for the CPU executions of the codes in all machines.

1.4. The FLAME algorithmic notation

Many of the algorithms included in this document are expressed using a notation developed by the FLAME (*Formal Linear Algebra Methods Environment*) project. The main contribution of

	Per Node	Per System
Type of machine	Cluster of SMP	
Number of nodes	256	
Processor	Intel Xeon E5440	
Processor codename	Nehalem	
Frequency	2.53 GHz	
Number of cores	8	2,048
Available memory	48 Gbytes	13.5 Tbytes
Interconnection network	Infiniband QDR	
Graphics system	128 NVIDIA Quadro Plex S4s	
GPU	2 x NVIDIA Quadro FX5800	512 x NVIDIA Quadro FX5800
Interconnection bus	PCI Express 2.0	
Available memory	8 Gbytes DDR3	2 Tbytes DDR3
Peak performance (SP)	161.6 GFLOPS	41.40 TFLOPS
Peak performance (DP)	80.8 GFLOPS	20.70 TFLOPS

Table 1.2: Detailed features of the LONGHORN cluster. Peak performance data only consider the raw performance delivered by the CPUs in the cluster, excluding the GPUs.

this project is the creation of a new methodology for the formal derivation of algorithms for linear algebra operations. In addition to this methodology, FLAME offers a set of programming interfaces (APIs) to easily transform algorithm into code, and a notation to specify linear algebra algorithms. The main advantages of the FLAME notation are its simplicity, concretion and high abstraction level.

Figure 1.2 shows an algorithm using the FLAME notation for the decomposition of a matrix A into the product of two triangular factors, $A = LU$, with L being unit lower triangular (all diagonal elements equal 1) and U upper triangular. In this case, the elements of the resulting matrices L and U are stored overwriting the elements of A.

Initially the matrix A is partitioned into four blocks $A_{TL}, A_{TR}, A_{BL}, A_{BR}$, with the first three blocks being empty. Thus, before the first iteration of the algorithm, the block A_{BR} contains all the elements of the matrix A.

For each iteration, the block A_{BR} is divided, as shown in Figure 1.3, into four blocks: $\alpha_{11}, a_{21}, a_{12}^T$ and A_{22}, and thus being partitioned into 9 blocks). The elements of the L and U factors corresponding to the first three blocks are calculated during the current iteration, while A_{22} is updated accordingly. At the end of the current iteration, the 2×2 partition of the matrix is recovered, and A_{BR} corresponds to the sub-matrix A_{22}. Proceeding in this manner, the operations of the next iteration will be applied again to the recently created block A_{BR}.

The algorithm is completed when the block A_{BR} is empty, that is, when the number of rows of the block A_{TL} equals the number of rows of the whole input matrix A and, therefore, $A_{TL} = A$. Using the FLAME notation, the loop iterates while $m(A_{TL}) < m(A)$, where $m(\cdot)$ is the number of rows of a matrix (or the number of elements of a column vector). Similarly, $n(\cdot)$ will be used to denote the number of columns in a matrix (or the number of elements in a row vector).

It is quite usual to formulate an *algorithm by blocks* for a given operation. These algorithms are easily illustrated using the same notation, in which it is only necessary to include an additional parameter: the *algorithmic block size*. This is a scalar dimension that determines the value of the blocks involved in the algorithms and is usually denoted by the character b.

Algorithm: $A := \mathrm{LU}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$

where A_{TL} is 0×0

while $m(A_{TL}) < m(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

where α_{11} is a scalar

$$\begin{array}{lcl} \alpha_{11} & := & v_{11} = \alpha_{11} \\ a_{12}^T & := & v_{12}^T = a_{12}^T \\ a_{21} & := & l_{21} = a_{21}/v_{11} \\ A_{22} & := & A_{22} - l_{21} \cdot v_{12}^T \end{array}$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

endwhile

Figure 1.2: Algorithm for the LU factorizaction without pivoting using the FLAME notation.

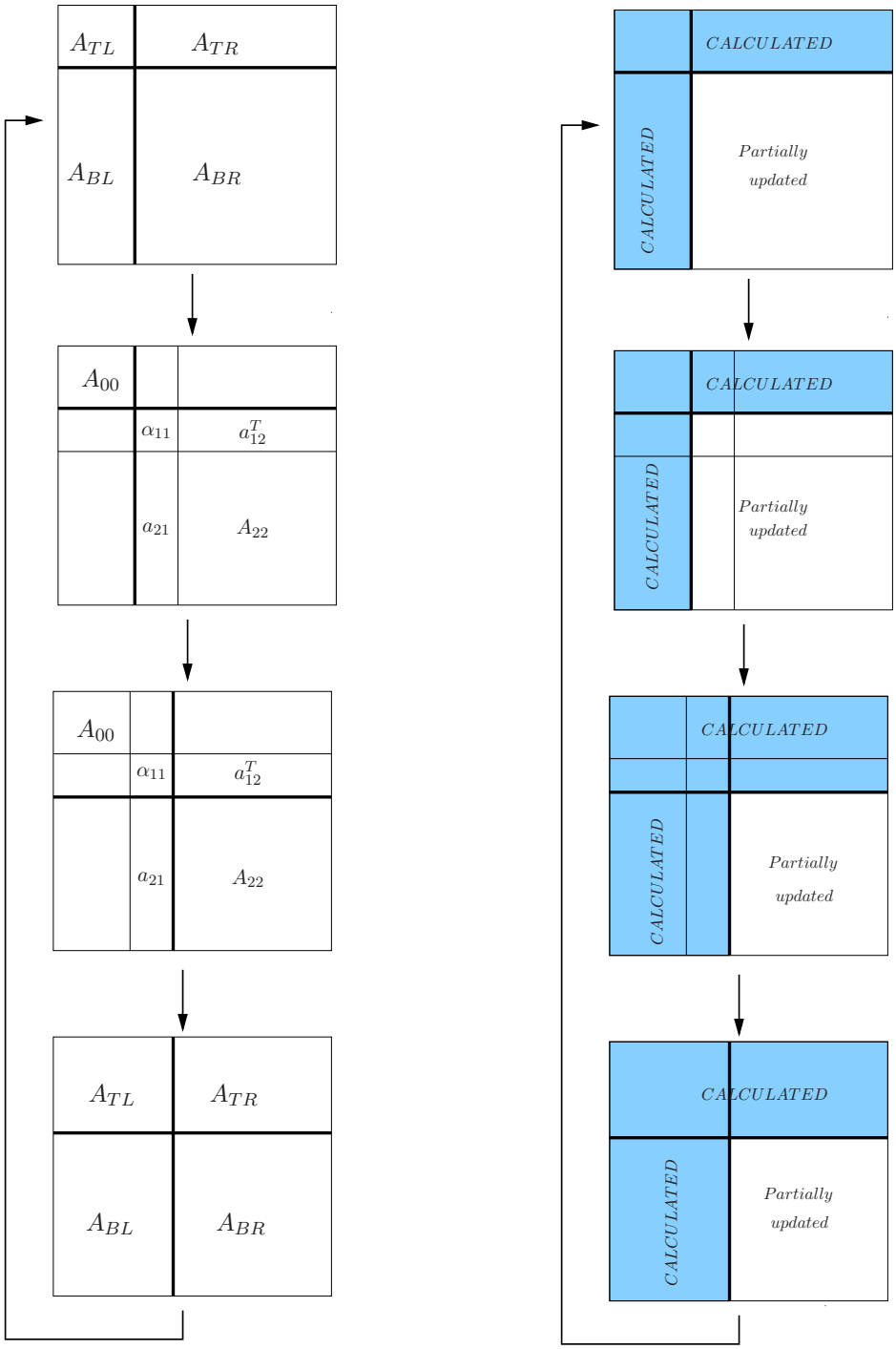


Figure 1.3: Matrix partitions applied to matrix A during the algorithm of Figure 1.2.

The FLAME notation is useful as it allows the algorithms to be expressed with a high level of abstraction. These algorithms have to be transformed into a *routine* or *implementation*, with a higher level of concretion. For example, the algorithm in Figure 1.2 does not specify how the operations inside the loop are implemented. Thus, the operation $A_{21} := A_{21}/v_{11}$ could be executed by an invocation to the routine SCAL in BLAS-1; alternatively, this operation could be calculated using a simple loop. Therefore, it is important to distinguish between *algorithm* and *routine*. In the following chapters we will show how the FLAME/C API allows an almost straightforward translation of the algorithms into high performance C codes.

CHAPTER 2

The architecture of modern graphics processors

Programmable graphics processors (GPUs) have emerged as a low-cost, high-performance solution for general-purpose computations. To understand the nature of the architecture of the graphics processors, it is necessary to introduce the graphics concepts that deliver their high performance, and the evolution of this type of processors through the years.

This introduction to GPU architecture basically follows a high-level approach, much like the rest of the material presented in this dissertation. This overview of the architecture of modern GPUs does not aim at providing a detailed low-level description; the goal is instead to understand the origins of the architecture, the reason underlying some of the features of modern graphics processors, and the justification for the type of algorithms that best fit to the architecture. Additionally, many of these particularities justify some of the decisions taken and techniques used in our research. Thus, no low-level details will be exposed in this section other than those that are strictly necessary to understand the rest of the document. A deeper exposition of the architecture can be easily found in the literature [1, 89, 3].

The chapter is structured as follows. Section 2.1 introduces the basic concepts underlying the graphics transformations performed in the graphics pipeline implemented by commodity graphics processors. Section 2.2 describes the novelties introduced by the unified architecture to implement the graphics pipeline in modern graphics processors. Section 2.3 explains the core ideas of the unified architecture through the description of an illustrative implementation, the NVIDIA TESLA architecture. Section 2.4 details the basic architecture of an usual hybrid system equipped with one or more graphics processors. Section 2.5 reports the main capabilities of older and modern GPUs regarding data accuracy. Section 2.6 introduces the successive hardware evolutions since the first unified architecture. Section 2.7 lists the main implications of the architectural details reported through the the chapter on the decisions taken in the rest of the dissertation.

2.1. The graphics pipeline

Although the work in this thesis is focused on the use of GPUs for general-purpose computations, the particularities of the hardware make it necessary to introduce some graphical concepts. The main goal is to understand the origin of the massively multi-threading available in today's GPUs.

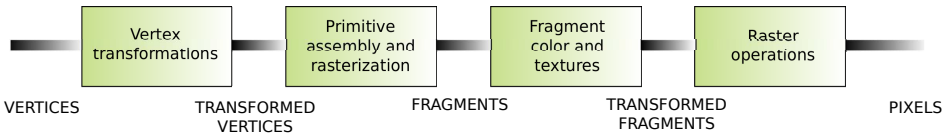


Figure 2.1: Schematic representation of the graphics pipeline.

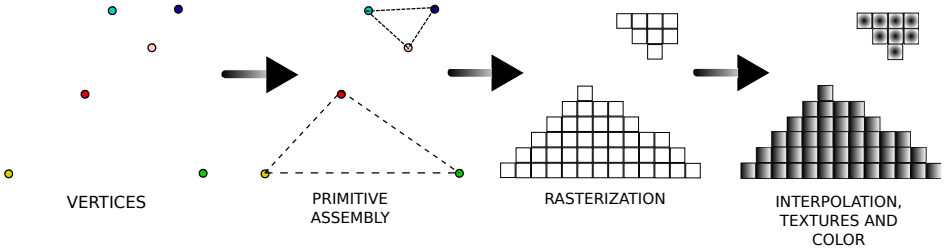


Figure 2.2: Operands of each stage of the graphics pipeline implemented by modern GPUs.

Traditionally, the way GPUs work is represented as a *pipeline* consisting of by specialized stages, executed in a in a pre-established sequential order. Each one of these stages receives its input from the former stage, and sends its output to the following one. A higher degree of concurrency is available within some of the stages.

Figure 2.1 shows an schematic pipeline used by current GPUs. Note that this logical sequence of stages can be implemented using different graphics architectures. Old GPUs employed separated and specialized execution units for each stage, adapting the graphics pipeline to a sequential architecture. Last generation GPUs implement a unified architecture, in which specialized execution units disappear, and every stage of the pipeline can be potentially executed in every execution unit inside the processor. However, the logical sequential pipeline remains.

In the graphics pipeline, the graphics application sends a sequence of vertexes, grouped in geometric primitives (polygons, lines and points) that are treated sequentially through the following four different stages (see Figure 2.2):

Vertex transformation

Vertex transformation is the first stage of the graphics pipeline. Basically, a series of mathematical operations is performed on each vertex provided by the application (transformation of the position of the vertex into an on-screen position, coordinate generation for the application of textures and color assignation to each vertex,...).

Primitive assembly and Rasterization

The transformed vertices generated from the previous stage are grouped into geometric primitives using the information gathered from the application. As a result, a sequence of triangles, lines or points is obtained. Those points are then passed to a subsequent sub-stage called *rasterization*. The rasterization is the process by which the set of pixels "covered" by a primitive is determined.

It is important to correctly define the concept of *fragment*, as it is transformed into a concept of critical importance for general-purpose computations. A pixel location and information regarding its color, specular color and one or more sets of texture coordinates are associated to each fragment.

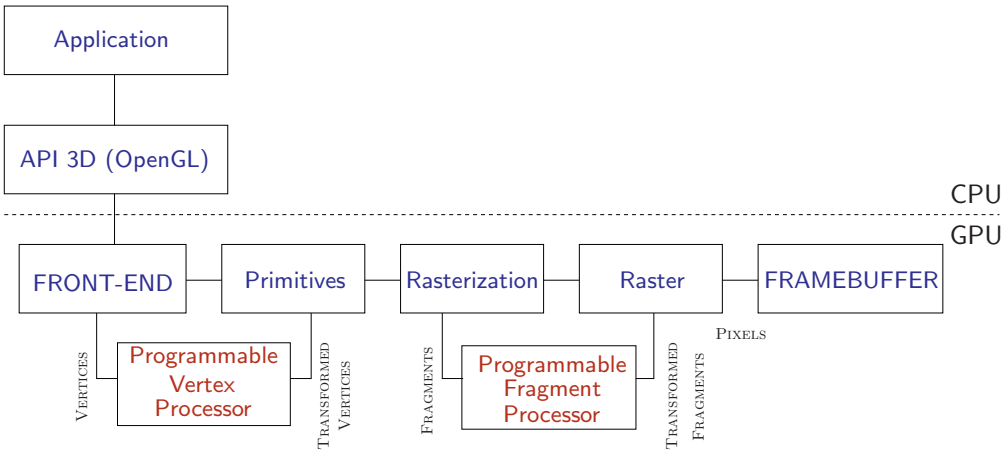


Figure 2.3: Detailed graphics pipeline with programmable stages (in red).

In particular, it is possible to view a fragment as a "potential pixel": if the fragment successfully passes the rest of the pipeline stages, the pixel information will be updated as a result.

Interpolation, Textures and Colors

Once the rasterization stage is complete, and a number of fragments have been extracted from it, each fragment is transformed by interpolation and texture operations (that will result in the most interesting phase for general-purpose computation) and the final color value is determined. In addition to the final coloring of the fragment, this stage is also responsible of discarding a given fragment to prevent its value from being updated in memory; thus, this stage produces either one or zero output fragments for each input fragment.

Last stages

In the final stages of the pipeline, the *raster* operations process each fragment with a set of tests to evaluate its graphical properties. These tests determine the values that the new pixel will receive from the original fragment. If any of these tests fails, the corresponding pixel is discarded. If all tests succeed, the result of the process is written to memory (usually referred as the *framebuffer*).

2.1.1. Programmable pipeline stages

Until the late 1990s, the way GPUs implemented the logical graphics pipeline consisted on a fixed-function physical pipeline that was configurable, but not fully programmable in any of its parts. The revolution started in 2001 with the introduction of the NVIDIA Geforce 3 series. This was the first graphics processor that provided some level of programmability in the shaders, and thus it offered the programmer the possibility of personalizing the behavior of some of the stages of the pipeline. NVIDIA Geforce 3 followed the Microsoft DirectX 8 guidelines [19], which required that compliant hardware featured both programmable vertex and pixel processing units.

Nevertheless, the novel generations of GPUs still exhibited separated types of processing units (also called *shaders*) for the vertex and the pixel processing, with different functionality and programming capabilities (see Figure 2.3). In general, the fragment processors were employed for GPGPU programming. First, because they were more versatile than the vertex processors. Second, because the number of fragment processors was usually larger than that of vertex processors.

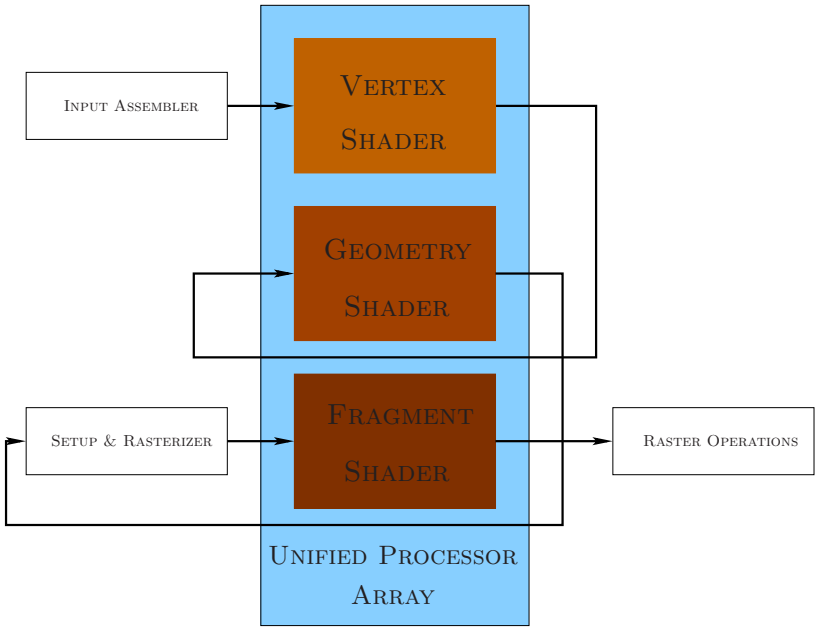


Figure 2.4: Cyclic approach of the graphics pipeline in the unified architectures.

The key contribution of this evolved architecture was the introduction of programmable stages, which in fact became the kernel of current graphics architectures, with plenty of fully-programmable processing units; this then led to the transformation of GPUs into a feasible target for general-purpose computation with the appearance of the programmable units in the graphics pipeline implementation.

In addition to the hardware update, the introduction of new APIs for programming the GPU entailed a renewed interest in GPGPU. Between those APIs, the most successful ones were Cg [64] and HLSL [124], jointly developed by Nvidia and Microsoft.

2.2. The Nvidia G80 as an example of the CUDA architecture

The mapping of this logical programmable pipeline onto the physical processor is what ultimately transformed the GPU computing scenario. In 2006, a novel architectural design was introduced by GPU vendors based on the idea of unified vertex and pixel processors. In this approach, there is no distinction between the units that perform the tasks for the vertex and the pixel processing. From this generation of GPUs on, all programming stages were performed by the same functional units, without taking into account the nature of the calculation to be done.

From the graphics perspective, the aim of this transformation was to reduce the unbalance that frequently occurred between vertex and pixel processing. Due to this unbalance, many of the functional units inside the GPU were basically idle for significant periods of time. In the unified architecture, there is only one type of processing unit, capable of executing both vertex and pixel operations. Thus, the sequential pipeline is transformed into a cyclic one, in which data recirculates through the processor. Data produced by one stage is used as an input to subsequent stages, using the same computational resources, but with a reconfigured behavior. Figure 2.4 illustrates this novel view of the graphics pipeline.

The implication for the GPGPU field rapidly became notorious. The unified architecture implies more arithmetic units at the disposal of the programmer, each of them with an improved functionality.

Together with the evolution of the processors towards a unified architecture, the software also experienced a drastic revolution with the introduction of the CUDA architecture and the CUDA programming paradigm, both by NVIDIA [1]. The aim of CUDA is to define an implementation of the unified architecture focused on performance and programmability.

The first GPU that implemented the unified architecture following the CUDA guidelines was the NVIDIA G80. Despite being a design from 2006, it is still the base of the newest generations of GPUs from NVIDIA, and no major changes have been introduced in the architecture. Actually, the NVIDIA G80 implemented the directives of Microsoft DirectX 10 [97], which dictated the fusion of the functionality of vertex and pixel shaders, and the addition of geometry shaders, with no real use for GPGPU. Although it was the first GPU complying those requirements, since then other companies have adopted this unified architecture.

2.3. The architecture of modern graphics processors

GPUs with unified architecture are built as a parallel array of programmable processors. Vertex, geometry and pixel shaders are merged, offering general-purpose computation on the same type of processors, unlike previous generations of GPUs. This programmable array collaborate with other fixed-function processing units that are devoted exclusively to graphics computing. Compared with multi-core CPUs, GPUs have a completely different perspective from the design point of view. These differences are mostly translated into a larger number of transistors devoted to computation, and less to on-chip caches and other functionality.

2.3.1. General architecture overview. Nvidia Tesla

A unified GPU processor array contains a number of processor cores, usually organized into groups or clusters acting as multi-threaded multiprocessors. Figure 2.5 shows the usual architecture of a unified GPU. Each processing unit is known as a *Streaming Processor* (SP). SPs are organized as a set of clusters usually referred to as *Streaming Multiprocessors* (SM). Each SP is highly multi-threaded, and handles thread deployment, execution and destruction exclusively in hardware. This provides an execution environment which can deal with thousands of threads without major overheads. The array is connected via a high-speed network to a partitioned memory space.

The basic Tesla architecture appeared in 2006, codenamed TESLA. The first GPU that implemented this architecture was the NVIDIA 8800. In this architecture, the processor array was composed of 128 SPs, grouped in clusters (SMs) of 8 SPs each, and connected with four 64-bit-wide DRAM partitions. In addition, each SM had two special function units (SFUs), instruction and data caches, a multi-threaded instruction unit, and a small shared memory of 16 Kbytes shared by all the SPs of the cluster. Two SMs share a texture unit in each texture/processor cluster (TPC). Originally, an array of eight TPCs composed the so-called *Streaming Processor Array*, or SPA, which in fact is the unified core which executes all graphics shader programs as well as, in our case, general-purpose programs.

This type of architectures are scalable in two different directions. First, the number of SMs in the chip can be increased further. Second, the number of SPs in each cluster can be enlarged, keeping constant the number of SMs. Section 2.6 discusses these possibilities and the solutions adopted by the latest generations of GPUs in order to deliver higher levels of performance.

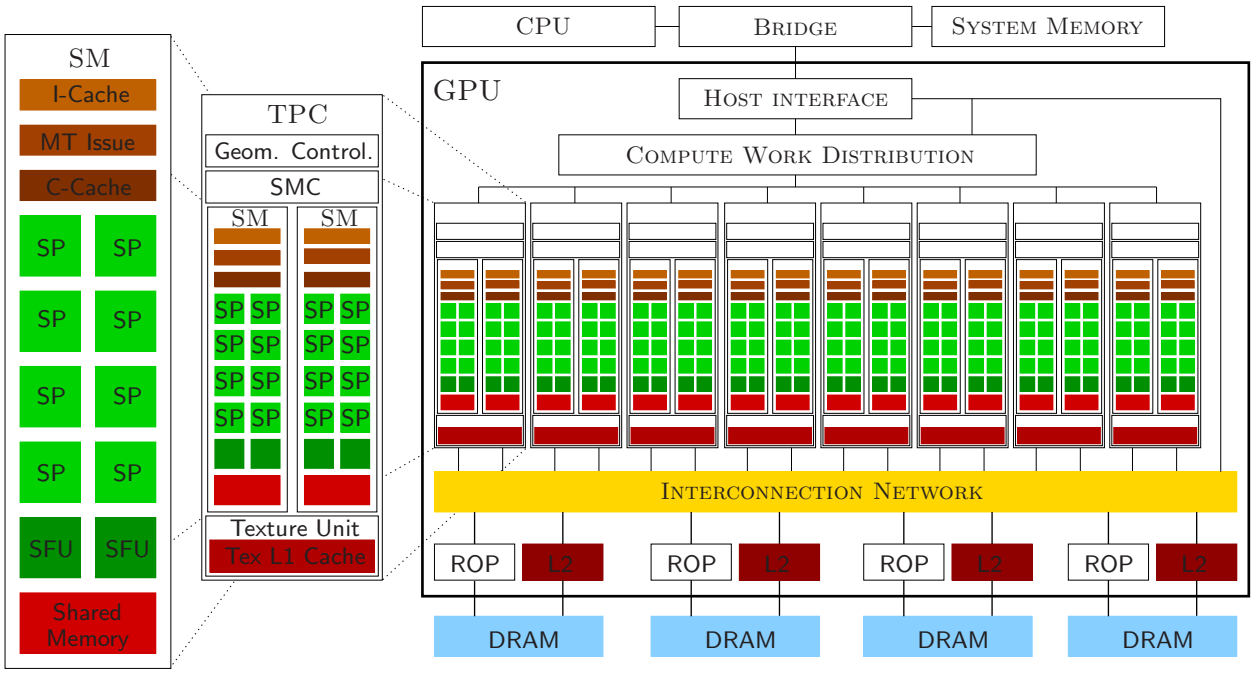


Figure 2.5: Unified architecture implementation on the NVIDIA TESLA series.

Single-Instruction Multiple-Threads (SIMT paradigm)

To efficiently manage the vast number of threads that execute a kernel, the multiprocessor exhibits a *single-instruction multiple-thread (SIMT)* architecture. The SIMT architecture concurrently applies one instruction to multiple independent threads during the parallel execution of a kernel. The multiprocessor is in charge of the creation, management, scheduling and execution of threads grouped in the so-called *warps*. A warp is a set of parallel threads executing the same instruction together in an SIMT architecture. In practice, the warp is the minimum scheduling unit of threads to SMs. At each scheduling step, a group of threads are bundled and issued to an SM, in which they logically execute in parallel. A correct management of divergence and memory access patterns within the threads in a warp is a key factor in the final performance of current GPU implementations.

The Streaming Processor Array (SPA)

We next describe in detail the architecture of the NVIDIA 8800 GPU to illustrate a real implementation of the unified architecture. As shown in Figure 2.5, the NVIDIA 8800 implementation has up to 128 SPs organized as 16 SMs. A group of two SPs share a common texture unit; these three elements define a texture cluster. An array of eight texture clusters define the SPA (Streaming Processor Array), the real kernel of a GPU with unified architecture.

The host interface unit communicates the GPU with the CPU via a PCI-Express and performs context switching in hardware. The work distribution units are in charge of dispatching vertices, pixels or, in case of GPGPU computing, compute thread arrays (*warps*) to the available TPCs in the array. Thus, the TPCs execute vertex and pixel shaders, and also general-purpose computing programs. Graphical output data is sent to specialized hardware units after TPC processing, for

example, the ROP units through the interconnection network. This network also routes texture data loaded from the SPA to DRAM and vice-versa via an L2 cache.

Streaming Multiprocessor (SM)

The *Streaming Multiprocessor* is the basic execution unit of the SPA. The SM is a unified graphics and general-purpose multiprocessor which can execute vertex, geometry, and pixel shaders, together with parallel computing programs. Each SM contains eight SPs, two SFUs, a multi-threaded fetch and issue instruction unit, an instruction cache, a constant cache (read-only cache), and 16 Kbytes of on-chip shared memory.

To deal with the execution of hundreds of parallel threads, the SM is hardware multi-threaded. The management and execution of up to 768 concurrent threads is performed in hardware with basically *zero-scheduling overhead*. This negligible penalty is absolutely necessary to deal with such a large pool of parallel threads and one of the distinctive features of graphics processors with unified architecture.

In the SIMT model previously described, the instruction fetch and issue unit of each SM is shared across 32 threads. The SM schedules and executes warps from a pool of ready warps. An issued warp instruction runs as four sets of 8 thread on four cycles. At each cycle, ready warps are qualified as prepared to be issued using a scoreboard [79]. The instruction scheduler assigns priorities to each warp and selects the first warp in the list for execution during the next issue cycle. This priority is based on the type of the warp (vertex, geometry, pixel or parallel computing), instruction type and other factors to assure a load balance among different warp types that execute in the SM.

In practice, the SM execute groups of cooperative thread arrays (also referred as CTAs); logically, CTAs are multiple concurrent warps which can communicate using a fast, on-chip shared memory region.

Instruction set

Unlike previous GPU architectures, in which the instruction set was designed to basically support vector instructions (to process four color channels per pixel), threads execute exclusively scalar instructions. In other words, the unified architecture is basically a scalar architecture. Only texture operations remain to be vector-based. This instruction set is supported by the Streaming Processor implementation, basically a scalar processor without vector capabilities.

The instruction set is register-based, and includes floating-point and integer arithmetic, logical, flow control, texture and load/store instructions. The load/store instructions can access three different memory-spaces:

- Local memory for per-thread, private and temporary data.
- Shared memory for low-latency, per-CTA data that is shared by the threads within a CTA.
- Global memory for data shared by all threads that participate in the computations.

In addition, operations for fast barrier synchronization within the threads in a CTA are available.

Streaming Processor (SP)

Each Streaming Processor contains both integer and floating-point arithmetic units to execute most the operations needed by graphics and general-purpose programs. Two key factors characterize the architecture of an SP. First, it is highly hardware multi-threaded, supporting up to 64

simultaneous threads. Second, each SP presents a large multi-thread register file. Each SP core has a file of 1024 general-purpose 32-bit registers, which are partitioned among the assigned threads mapped for execution in the core.

From the software point of view, this architecture clearly determines the nature of the parallel executions. The vast amount of threads and wide register file requires fine-grained executions with massive multi-threading to exploit the architecture features. CUDA programs often need a small amount of registers (typically between 8 and 32), which ultimately limits the number of threads that will execute a kernel program.

The multiprocessor executes texture fetch instructions and memory load, store and atomic operations concurrently with instructions on the SPs. Shared-memory access (explained later) employs low-latency interconnections between the SPs and shared memory banks inside each SM.

2.3.2. Memory subsystem

The second key factor that ultimately determines the performance of graphics (and general-purpose) applications is the graphics memory subsystem. From the general-purpose computation perspective, the features of the memory subsystem define the applications that fit better to the GPU, and which algorithms and implementations are well-suited for this class of architecture.

The graphics-oriented nature of the GPUs dictates why the graphics memories have historically been developed at a highest pace compared with central memories. Graphics applications are data-intensive, with high data traffic demands. Consider, for example, the NVIDIA Geforce 8800 described above. From the graphics point of view, it can process up to 32 pixels per cycle, running at 600 MHz [80]. Typically, each pixel requires a color read and a color write, plus a depth read and a depth write for a 4-byte pixel (four color channels). In order to generate a pixel's color, usually two or even three texels (elements of texture), of four bytes each, are read from texture memory. In a typical case, this demands 28 bytes times 32 pixels = 896 bytes per clock, which is an considerable bandwidth rate that the memory subsystem must provide.

GPU memories usually satisfy a number of features:

- Width, offering a large set of pins to transfer data to and from the chip, and even to perform intra-memory transfers.
- Speed, to maximize the data rate by using aggressive signaling techniques.
- Usage of explicitly managed memory hierarchies. A large fraction of the transistors in a GPU are devoted to computing power. However, there exist strategic memory spaces (shared memory per SM) or caches (texture cache) that can be exploited by the programmer in order to boost performance.
- GPUs are designed to exploit every idle cycle to transfer data to and from global memory. GPUs do not aim to specifically minimize memory latency, but to hide it by increasing the throughput or utilization efficiency (for example, by increasing the number of concurrent threads in the system).

Off-chip memory spaces

DRAM chips present some characteristics that must be taken into account in the design of GPUs. DRAM chips are internally built as a set of banks organized by rows (typically around 16,384, and each row with a number of cells (or bits, typically 8,192). Current DRAM limitations require that both GPU architectures and run-time implementations consider these particularities.

As an example, the activation of a row in a DRAM bank usually takes dozens of cycles, but once activated, the bits in the row are randomly accessible in a few clock cycles.

In addition, the graphics pipeline presents several sources of data traffic and requests, with high heterogeneity and poor spatial locality. Usually, the GPU memory controller deals with this independent data traffic generators by waiting until enough traffic is generated for a given DRAM row before activating it and transferring the corresponding data. This type of strategies usually have a negative impact on latency, but improve the usage of the bus.

DRAM is usually deployed as multiple memory partitions, each one with an independent memory controller. Usually, addresses are interleaved across all memory partitions in order to optimize the load balance in memory accesses.

The CUDA architecture exposes different memory spaces that allow the developer to optimally exploit the computational power of unified GPUs. The following description presents the different memory spaces available in the NVIDIA TESLA GPUs, although similar approaches have been taken by other GPU manufacturers.

Global and local memory

Global memory is stored in DRAM partitions. It is meant for the communication among threads that belong to different CTAs, so it is not local to any physical SM. Sequential consistency is not guaranteed in the access to global memory by different threads. Threads view a relaxed ordering model: within a thread, the order of memory reads and writes to the same address is effectively preserved; the order of accesses to different addresses is not guaranteed to prevail. In order to obtain a strict memory ordering among threads in the same CTA, explicit barrier synchronizations are available. There is also a special thread instruction in the Instruction Set Architecture (`membar`) that provides a memory barrier to commit previous memory transactions and make them visible to the remaining threads. Atomic global memory operations are also available for threads that cooperate via shared memory addresses.

Local memory is also allocated in external DRAM. It is private memory visible only to a single thread. From the architecture perspective, it is than the large register file available per thread, so it supports large allocations of memory. Note that the total amount of local memory equals the local memory allocated per thread times the total number of active threads.

Local and global memory load/store instructions *coalesce* individual parallel thread requests from the same warp into a single memory block request if the addresses fall in the same block, and meet some alignment criteria specific to each architecture. Satisfying these restrictions improves memory bandwidth in a remarkable way. This is a very common optimization in CUDA programs, and usually reduces the impact introduced by the usage of external DRAM memories.

Constant and texture memories

Constant memory is also stored in DRAM. It is a read-only memory space, cached in the SM. Its main purpose is to broadcast scalar values to all the threads in a warp, an operation that is very useful for graphics codes.

Texture memory is designed to hold large-arrays of read-only data. Textures can be seen as one-dimensional, two-dimensional or three-dimensional data structures. Texture transfers are performed using special instructions, referencing the name of the texture and the coordinates to be extracted. Texture fetches are also cached in a streaming cache hierarchy, specifically designed to optimize texture fetches from the large amount of active threads present in modern GPUs. In

some sense, texture memory can be used, with restrictions, as a way of caching global memory in a transparent way for the programmer.

On-chip memory spaces

Due to the high penalty introduced by the access to DRAM, on-chip SRAM memories play a key role in the final performance attained by general-purpose applications. We describe on-chip memory spaces dividing them into shared memory and caches.

Shared memory

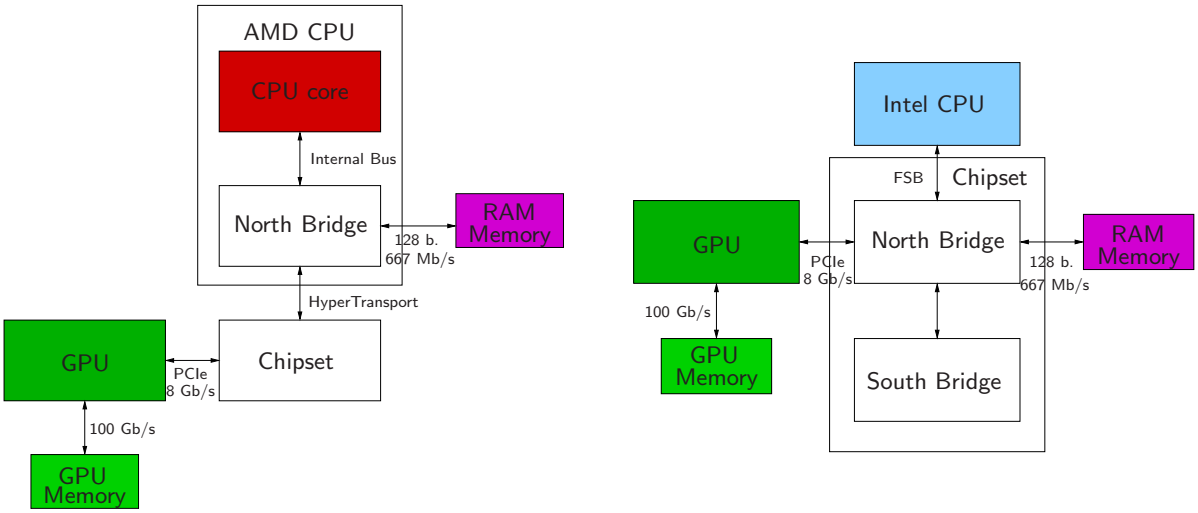
Shared memory resides on chip (SRAM) and is only visible to the threads in the same CTA. It is dynamically allocated and only occupies space since the creation of the CTA until its destruction. As it resides on chip, shared memory traffic does not interfere with global memory traffic, and does not share its bandwidth limitations. In practice, it is useful to build very high-bandwidth memory structures on chip that support the high-demanding read/write needs of each SM. In fact, shared memories are usually exploited as small caches (usually known as *scratchpad* memories) whose integrity and coherence is managed by software. As threads in each CTA can potentially generate different shared memory addresses at each instruction, shared memory is usually divided in banks that are independently addressable. In the NVIDIA 8800, shared memory is divided into 16 independent blocks; in general, this quantity is large enough to deliver a high throughput unless pathological cases of unique addressing are given. Avoiding bank conflicts is another important aspect to attain high performance in GPU computing.

Caches

Graphics routines often involve working with large data sets, usually in the order of Mbytes, to generate one single frame. The graphics-oriented nature of the GPU makes it unpractical to build caches on chip which are large enough to hold a significant fraction of the necessary data for the computation close to the processing unit. Modern GPUs offer texture caches that store read-only data fetched from DRAM. From the GPGPU perspective, these caches can be exploited to avoid unnecessary global memory access for certain data structures. As for graphics workloads, these caches are for read-only memory chunks. The latest models of GPUs from NVIDIA (Fermi) include for the first time L1 caches mapped in the same SRAM space devoted to shared memory. However, the management of these memory spaces is performed by hardware, without the programmer's intervention, which simplifies the development process and has a direct impact on performance.

2.4. The GPU as a part of a hybrid system

GPUs do not work as an isolated device, but as a component of a hybrid architecture in conjunction with the CPU, chipset, system memory and, possibly, other graphics processors. Figure 2.6 illustrates two common configurations of current hybrid CPU-GPU systems. Although future trends advocate for the integration of CPU and GPU in the same die, current architectures are based on discrete GPUs, usually attached to the rest of the system through a high speed bus. The schema in the left of the figure shows a typical configuration for an AMD-based architecture. In this case, the North Bridge is integrated into the die, and the GPU connects directly to the chipset through the PCIExpress Gen2 bus. The organization in the right of the figure shows the distribution of elements in an Intel-based architecture. Observe how the GPU is attached directly to the North



(a) AMD configuration **(b)** Intel configuration

Figure 2.6: Examples of two contemporary architectures for hybrid CPU-GPU systems. (a) with an AMD CPU; (b) with an Intel CPU.

Bridge via a 16-lane PCIExpress Gen2 link. This interface provides a peak of 16 Gbytes/s transfer (a peak of 8 Gbytes/s in each direction). In multi-GPU systems, the PCIExpress bus is shared by all the graphics processors.

Hybrid systems present more than one memory address space. More specifically, there is an independent memory address space per GPU, plus the one bound to the CPU system memory. This implies a constant movement of data through the PCIExpress bus. These transfers are critical as the number of GPUs is increased, since data transfers grow then, and effective bandwidth is reduced.

Figure 2.6 provides information about the bandwidth in the interconnections between the elements in the architectures. Note how the closer buses are to the CPU, the faster they are. As a remarkable exception, GPU memory is fastest than system memory, due to the graphics design requirements. The bandwidth rates shown in the figure can vary for specific system configurations, but are an illustrative example of the transfer rates available in current GPU-based desktop and HPC architectures.

The PCIExpress bus

PCIExpress (formerly known as 3GIO, Third Generation I/O Interface) was introduced in response to the necessity of increasing bandwidths in system buses and memory interfaces during the early 2000s, to keep pace with the processor. By that time, the PCI bus had become a real bottleneck for the growing bandwidth demands from the processors and I/O devices. The parallel approach of the older PCI bus dramatically limited the future of this type of interconnection. The PCI bus was close to its practical limits of performance: it could not be scaled up in frequency or down in voltage, as its synchronous data transfer was dramatically limited by signal skew. This constraint finally derived in a wide variety of interconnection links adapted to the application necessities (AGP for graphics, USB for external devices interconnection, ATA for disk interfaces, PCI for other devices,...). In addition, a common architecture must deal with concurrent data transfers at increasing rates. It was no acceptable that all data transfers are treated in the same way.

PCIExpress specification	Base frequency (Ghz)	Interconnect Bandwidth (Gbits/s)	Bandwidth Lane/direction (Mbytes/s)	Total Bandwidth for x16 link (Gbytes/s)
PCIe 1.x	2.5	2	~ 250	~ 8
PCIe 2.x	5.0	4	~ 500	~ 16
PCIe 3.0	8.0	8	~ 1000	~ 32

Table 2.1: Summary of the bit rate and approximate bandwidths for the various generations of the PCIe architecture

A new standard that offered both higher bandwidth and enough flexibility to deal with different bandwidth demands was necessary.

The design of the PCIExpress specification presented some basic requirements:

- Flexibility to adapt to a wide variety of bandwidth-demanding devices, from network interfaces (Gigabit Ethernet and InfiniBand) to graphics devices. PCIExpress was the natural substitute of the AGP port for the connection of graphics devices.
- Performance and scalability by adding additional lanes to the bus, higher bandwidth per pin, low overhead in communications and low latency.
- Support for a variety of connection types: chip-to-chip, board-to-board via connector, docking station, and mobile devices. This ability makes it possible to physically separate devices from hosts, as in the Tesla s1070 multi-GPU system used in this work.
- Compatibility with former PCI software model.
- Advanced features as hot-plugging, power management, error handling, QoS, etc.

A basic PCIExpress link consists of two pairs of signals: a transmit pair and a receive pair. An 8b/10b data encoding scheme is used to attain high data rates. The initial frequency is 2.5 GHz, attaining a peak bandwidth of 250 Mbytes/s per direction. The initial plans were to increase this rate to 10 GHz, which is the upper limit for transmitting signals on copper). Note the serial approach in contrast to the parallel design of the original PCI bus. The bandwidth of a link can be linearly scaled with the addition of new signal pairs to form new lanes. The physical layer of the initial specification supported 1x, 2x, 4x, 8x, 16x and 32x lane widths. Compatibility between devices is negotiated during initialization. The PCIExpress architecture can be eventually improved by upgrading speed and implementing more advanced encoding techniques. This is the case of PCIExpress 2.0 or Gen2, that doubles the base frequency to 5 GHz to attain the same transfer rate using half of the lanes of its predecessor, as the base bandwidth is doubled to 500 Mbytes/s per direction. Future PCIExpress 3.0 will deliver about 1 GByte/s of bandwidth per lane direction, by increasing the base frequency and eliminating the overhead of the 8b/10b encoding scheme, using a more efficient 128b/130b coding scheme. Table 2.1 summarizes the evolution in performance of the PCIExpress specifications, including the eventual PCIExpress 3.0.

2.5. Arithmetic precision. Accuracy and performance

Accuracy was one of the historical problems in GPU computing. Together with the evolution of the graphics architectures, the fixed-point arithmetic of GPUs has evolved from 16-bit, 24-bit,

32-bit to the current single precision (32-bit) IEEE 754-compliant floating-point arithmetic. In addition, recent GPUs also provide for double-precision (64-bit) IEEE 754-compliant floating-point operations. Even though double precision is not necessary in the type of computations the GPUs are designed for, this capability has been added in order to satisfy the demands of many scientific applications.

Single precision floating-point operations supported by the GPU cores include addition, multiplication, multiply-add, minimum, maximum, compare, set predicate, and conversions between integer and floating-point numbers [80]. Most modern GPUs implement floating-point addition and multiplication that are fully compatible with the IEEE 754 standard for single precision floating-point numbers, including `NaN` and infinity values. The most common operation offered by GPUs is often the multiply-add instruction (`madd`). In practice, this operation performs a floating-point multiplication with truncation [80], followed by a floating-point addition with round-to-nearest-even. This operation is thus composed of two floating-point operations, and can be performed in only one cycle. No intervention of the instruction scheduler is needed to dispatch two different operations. However, the computation is not fused and the product is truncated before the addition is performed.

Special functions such as cosine, sine, binary exponential, binary logarithm, reciprocal and reciprocal square root are available and executed in the SFUs. Although the IEEE 754 standard requires exact-rounding for these operations, many GPU application do not require such an exact compliance. In this domain, higher throughput is preferred to exact accuracy. However, software libraries (CUDA) provide both a full accuracy function and a fast function with the native SFU accuracy.

The floating-point addition and multiplication units are fully pipelined. Although they are also fully pipelined, special operations (executed in the SFUs) yield a throughput smaller than that of floating-point addition and multiplication. A ratio 1/4 is common when comparing both types of operations. This factor, however, is higher than that offered by the CPUs for common special operations like division or square root, even though CPUs deliver higher accuracy.

Double-precision capabilities have been added to the latest generations of GPUs. These processors support 64-bit IEEE 754 operations in hardware. These operations include addition, multiplication and conversion between different floating-point and integer formats. These three operations are performed in hardware in a FMA (fused multiply-add) unit. Unlike for single-precision floating-point, the FMA unit in modern GPUs performs a fused-multiply-add operation without truncation of the result after the multiplication. Thus, accuracy is kept in intermediate operations. This unit also enables the formulation of accurate divisions and square roots in software, without needing a special function unit for double-precision arithmetic.

2.6. Present and future of GPU architectures

In the last four years, the number of transistors in the successive generations of NVIDIA hardware has multiplied by four, close to the improvement rate dictated by Moore's law. In addition, the number of cores (SPs) per GPU has roughly doubled every two years. However, none of the newest hardware developments has been revolutionary; instead, they can be viewed just as the natural evolution of the original G80 implementation with unified architecture. In this section, we will review the main changes introduced by the GT200 and the recently introduced Fermi microarchitectures, remarking the main differences with the original unified implementation.

Table 2.2 summarizes the main differences between those three instances of the unified architecture. Only those features with a relevant impact on general-purpose computing have been listed

in the table. Each generation follows a different scalability approach. The GT200 increases the number of SMs in the chip, maintaining the number of SPs per multiprocessor constant. On the other hand, the Fermi architecture improves the amount of SPs per multiprocessor. These are the differential features between both evolutions.

The GT200 was the first evolution of the G80 architecture in response to the increase in the number of transistors since the first appearance of the unified architecture. The main improvements were the introduction of double-precision support in hardware and the increase in the number of multiprocessors in the chip (from 16 to 30). No other significant changes were introduced with the new processor. Still, the peak performance of the GPU was doubled in single precision. Double-precision arithmetic was 8 times slower than single precision.

Fermi [3] represents the major modification in the unified architecture since its introduction in 2006. Many of the novelties introduced by this microarchitecture have a direct impact on or exclusively target general-purpose computations and, more specifically, in scientific computing.

The main improvements in the Fermi architecture appear in the new design and capabilities of the SPA. Each Streaming Multiprocessor features 32 SPs (both the G80 and GT200 featured 8 SPs per multiprocessor). The amount of shared memory per SM increases accordingly to 64 Kbytes. A major change in this generation is the introduction of an L1 cache, mapped to the same SRAM memory as the shared space. The on-chip memory can be configured to act as 48 Kbytes of user-managed shared memory and 16 Kbytes of L1 cache, or as 16 Kbytes of shared memory and 48 Kbytes of L1 cache.

The number of load/store units per SM grows to 16 units, allowing source and destination addresses to be calculated by 16 threads per clock cycle. The register file is further enlarged, up to 32,768 32-bit registers per SM. The number of SFU units increases accordingly to 4 per SM.

Each SP in the Fermi architecture provides a fully pipelined integer arithmetic logic unit (ALU) and floating-point unit (FPU). The architecture implements the fused multiply-add (FMA) instruction for both single and double precision, unlike previous implementations that only implemented MAD for single precision, loosing accuracy in the results. In addition, the integer ALU is now optimized for 32 and 64-bit operations; previous implementations were based on 24-bit accuracy, needing software emulation to perform integer arithmetic.

The improvement in double-precision performance between the GT200 and Fermi is dramatic. In the former, the ratio double-single precision was 1/8. This ratio has been reduced to 1/2 in Fermi, much in the line of modern CPUs. Other features related with HPC, but strictly necessary in graphics computations, include the support for ECC memory and the execution of concurrent kernels in the same SPA.

2.7. Conclusions and implications on GPU computing

Many of the architectural details exposed in this chapter have a direct implication on the design decisions and techniques presented in this dissertation. The following are representative examples of these implications:

- Modern GPUs have evolved into complex architectures in order to satisfy the strict requirements of current graphical applications. Additionally, they follow a radically different approach in their design compared with general-purpose processors. Although novel programming paradigms have facilitated the development task, the programmer still needs to be aware of many of the architectural details. Explicit management of on-chip memories (shared memory), memory access patters (global memory coalescing and elimination of shared memory

	G80	GT200	Fermi
Year	2006	2008	2010
Transistors	681 million	1.4 billion	3.0 billion
Total SPs	128	240	512
DP Capabilities	-	30 FMA ops/clock	256 FMA ops/clock
SP Capabilities	128 MADD ops/clock	240 MADD ops/clock	512 FMA ops/clock
Total SFUs per SM	2	2	4
Warp schedulers per SM	1	1	2
Shared Memory per SM	16 Kbytes	16 Kbytes	48 or 16 Kbytes
L1 Cache per SM	-	-	16 or 48 Kbytes
L2 Cache	-	-	768 Kbytes
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Addr. Width	32-bit	32-bit	64-bit

Table 2.2: Summary of the main features of the three generations of unified GPUs by NVIDIA

bank conflicts) or divergence control are some examples of programming decisions that the developer must face. Multi-GPU systems present additional problems such as the management of multiple memory address spaces.

Thus, the design, implementation and validation of high-level approaches that hide these details and abstract the programmer from them is an important step towards the fast development of high-performance codes. We introduce such approaches in the framework of single-GPU systems (Chapter 3), multi-GPU systems (Chapter 5), and clusters of GPUs (Chapter 6).

- The bottleneck introduced by the PCIExpress bus is more important as the number of GPUs in the system increases. In this type of architectures, a strategy to reduce the number of data transfers is mandatory if high performance is required.

  Therefore, the development of run-time systems that carefully orchestrate data transfers between different memory spaces is a necessary approach to reduce data transfers, and to hide them from the programmer. The run-time system exposed in Chapter 5 provides a solution to these requirements.

- GPUs are processors targeted to the gaming market. Thus, some of their capabilities are not perfectly suited to the HPC arena. The ratio performance/precision is one of them. Although double-precision floating-point arithmetic is possible in modern GPUs, the difference in performance compared with single-precision floating-point arithmetic is remarkable. Strategies that combine the performance of GPUs in single-precision and the accuracy of double-precision are welcome.

  In this dissertation, we apply an iterative-refinement approach (Chapter 4, Section 4.6) as a successful solution for this problem. This solution combines precision and accuracy for the solution of linear systems, although similar guidelines can be applied to other linear algebra operations.

- The particularities of modern GPUs make them suitable only for certain types of applications. Only those applications that exhibit high arithmetic intensity, a large degree of data parallelism, and few divergent paths fit well on these architectures.

  In the framework of this dissertation, we have selected those dense linear algebra algorithms that best adapt to the GPU (for example, Level-3 BLAS in Chapter 3).

However, in situations in which certain parts of an algorithm are not well suited to the GPU architecture, we propose *hybrid algorithms*, in which the strengths of the different execution units in the system are exploited in different parts of the algorithm. Some examples of these strategies are shown in Chapter 4 for common linear algebra operations.

The experiments reported in this dissertation will employ NVIDIA hardware. However, it is important to note that *all* techniques exposed and methodology introduced in our work can be potentially adapted to other similar graphics architectures. In general, only a reduced set of optimized BLAS routines are necessary to port the insights gained from our study to other kinds of graphics devices.

Moreover, many of the topics covered in this dissertation are not exclusively applicable to graphics architectures, but also to any general accelerator-based architecture similar to that described for GPUs in this chapter. That is one of the main contributions of the usage of a high-level approach for the development of accelerated linear algebra implementations.

Part II

Matrix computations on single-GPU systems

CHAPTER 3

BLAS on single-GPU architectures

The *Basic Linear Algebra Subprograms* (BLAS) are the fundamental building blocks for the development of complex dense linear algebra applications. In this chapter, the implementation of the Level-3 BLAS specification from NVIDIA (CUBLAS) is evaluated. The major contribution though is the design and evaluation of a new, faster implementation of the main routines in Level 3 BLAS. The aim of these new implementations is twofold: First, to improve the performance of the existing BLAS implementations for graphics processors. Second, to illustrate a methodology to systematically evaluate a number of parameters that become crucial to attain high performance. To achieve these goals, a set of *algorithmic variants* that take benefit from a reduced number of existing high-performance BLAS kernels is presented, together with a detailed evaluation of the performance of those new implementations.

As a result, our new implementations attain remarkable speedups compared to those in NVIDIA CUBLAS. Furthermore, they show a homogeneous performance for *all* Level-3 BLAS routines. In addition, we demonstrate how, by systematically applying a set of high-level methodologies, it is possible to obtain high-performance implementations for *all* Level-3 BLAS routines for graphics processors without the necessity of any low-level coding effort. These homogeneous performance rates differ from those attained with the NVIDIA CUBLAS implementation, which only reaches high performance for a selected group of BLAS-3 routines (namely, the general matrix-matrix multiplication for a restricted set of particular cases).

Although the conclusions extracted from the evaluation of these alternative implementations can be directly applied to low-level programming codes, the developed routines are based on an existing BLAS implementation for graphics processors, improving *portability* and *programmability*. Given the large impact of the performance of the Level-3 BLAS implementations on higher-level linear algebra libraries, and the potential performance of the graphics processors on routines with high arithmetic intensity, our optimizations will exclusively address this BLAS level.

The chapter is divided as follows. Section 3.1 describes the basic concepts and nomenclature behind the BLAS specification. Section 3.2 shows a full evaluation of the Level-3 BLAS routine implementations in NVIDIA CUBLAS, comparing the results with those attained with a highly tuned library in a current general-purpose multi-core processor. Section 3.3 presents a variety of techniques to tune the performance of those implementations; the attained performance results are reported in Section 3.4. Section 3.5 summarizes the main contributions of the chapter.

All experiments presented through the chapter were carried out using a single GPU (Tesla C1060) on peco. The specific hardware and software details of the experimental setup were presented in Section 1.3.2.

3.1. BLAS: Basic Linear Algebra Subprograms

Fundamental dense linear algebra problems, such as the solution of systems of linear equations or eigenvalue problems, arise in a wide variety of applications in science and engineering. Chemical simulations, automatic control or integrated circuit design are just some examples of application areas in which linear algebra operations usually conform the computationally most expensive part.

A set of basic operations frequently appear during the solution of these problems, such as the scalar product of two vectors, the solution of a triangular linear system or the product of two matrices. This set of basic linear algebra routines is grouped under the name BLAS *(Basic Linear Algebra Subprograms)*. In fact, the development of BLAS was a joint effort of experts from diverse areas, so that the final specification covered the basic requirements that appear in many fields of science.

The BLAS specification was originally a trade-off between functionality and simplicity. The number of routines and their parameters were designed to be reasonable in number; but, simultaneously the functionality was intended to be as rich as possible, covering those routines that often appear in complex problems. An illustrative example of the flexibility of the specification is the representation of a vector: the elements of the vector do not have to be stored contiguously in memory; instead, the corresponding routines provide a parameter to define the physical separation between two logically consecutive elements in the vector.

Since the first definition of the specification, BLAS has been of great relevance in the solution of linear algebra problems. Its reliability, flexibility and the efficiency of the existing implementations allowed the emergence of other libraries that made internal usage of the BLAS routines. In addition to reliability and efficiency, there are other advantages that make the use of BLAS so appealing:

- Code legibility: the names of the routines denote their internal functionality. This standardization makes code simpler and more readable.
- Portability: by providing a well-defined specification, the migration of the codes to other machines is straightforward. Given a tuned implementation for the target machine, the ported implementations will stay optimized and the resulting codes will remain highly efficient.
- Documentation: there is a rich documentation available for each BLAS routine.

There is a generic implementation of the BLAS available since its original definition [106]. This reference implementation offers the full functionality of the specification, but without any further optimization specific to a particular hardware. However, the real value of BLAS lies with the tuned implementations developed for different hardware architectures. Since the publication of the specification, the development of tuned implementations adapted to each hardware architecture has been a task for either processor manufacturers or the scientific community. Today, the quality of vendor-specific implementations are usually employed as demonstrations of the full potential of an specific processor. There exist proprietary implementations from AMD (ACML) [4], Intel (MKL) [85], IBM (ESSL) [62] or SUN (SUN Performance Library) [111] for general-purpose architectures. Vendor-specific implementations are emerging for novel specific-purpose architectures, such as Nvidia Cublas for the Nvidia graphics processors, or CSXL for the ClearSpeed

boards. In general, the code for each routine in those implementations is designed to optimally exploit the resources of a given architecture. Independent third-party implementations, such as GotoBLAS [71] or ATLAS [145] also pursue the optimizations of the implementations on general-purpose processors; as of today, no independent implementations of BLAS for graphics processors exist that improve the performance of the full NVIDIA CUBLAS implementation, although partial solutions have been proposed for specific routines [142].

BLAS are usually implemented using C and Fortran but, in many cases, the use of assembly language allows fine-grained optimizations to extract the full potential of the architecture. In specific-purpose architectures, the use of *ad-hoc* languages is a must. As an example, the internal NVIDIA CUBLAS implementation makes heavy use of the CUDA language and specific tuning techniques that are exclusive of this kind of hardware.

3.1.1. BLAS levels

The development of BLAS is closely bound to the development and evolution of the hardware architectures. At the beginning of the BLAS development, in early 1970s, the most common HPC computers were based on vector processors. With those architectures in mind, BLAS was initially designed as a group of basic operations on vectors (Level-1 BLAS, or BLAS-1). The ultimate goal of the BLAS specification was to motivate the hardware vendors to develop fully optimized versions following the standard specification.

In 1987, the BLAS specification was improved with a set of routines for matrix-vector operations, usually known as Level-2 BLAS (or BLAS-2). Both the number of operations and the amount of data involved on these routines are of quadratic order. As for the first BLAS routines, a generic BLAS-2 implementation was published after the specification. One of the most remarkable observations of these early implementations was the column-wise access to the data in the matrices. Efficient implementations of BLAS-2 can reduce the number of memory accesses by exploiting data reuse on registers inside the processor.

The increase in the gap between processor and main memory speeds [129] implied the appearance of architectures with multiple levels of cache memory, and thus a hierarchical organization of the system memory. With the popularization of those architectures, it was accepted that the libraries built on top of BLAS-1 and BLAS-2 routines would never attain high performances when ported to the new architectures: the main bottleneck of BLAS-1 and BLAS-2 codes is memory. While on these routines the ratio between the number of operations and the number of memory accesses is $O(1)$, the ratio between the processor speed and the memory speed was much higher in those emerging architectures. The main implication of this speed gap is that memory needs more time to feed the processor with data than that needed by the processor to process them. Thus, the performance of the BLAS-1 and BLAS-2 routines is mainly limited by the speed at which the memory subsystem can provide data to the execution path.

The third level of BLAS (Level-3 BLAS or BLAS-3) was defined in 1989 in response to the problems exposed above. The specification proposed a set of operations featuring a cubic number of floating-point operations on a quadratic amount of data. This unbalance between the number of calculations and memory accesses allows a better exploitation of the principle of locality in architectures with hierarchical memories, if carefully designed algorithms are applied. In practice, this hides the memory latency, and offers performances close to the peak performance delivered by the processor. From the algorithmic viewpoint, these higher performances are attained by developing *algorithms by blocks*. These algorithms partition the matrix into sub-matrices (or blocks), grouping memory accesses and increasing the locality of reference. By exploiting this property, the possibility of finding data in higher (and faster) levels of the memory hierarchy increases, and thus the memory

access penalty is reduced. The size and organization of cache memories determines the optimal block size. Thus, exhaustive experimental searches are usually carried out to find those optimal values. Inside each matrix, the column-wise access was also recommended by the specification [57].

To sum up, the BLAS are divided into three levels:

- Level 1: The number of operations and the amount of data increase linearly with the size of the problem.
- Level 2: The number of operations and the amount of data increase quadratically with the size of the problem.
- Level 3: The number of operations increases cubically with the size of the problem, while the amount of data increases only quadratically.

From the performance perspective, the main reason of this classification is the ratio between the number of operations and the amount of data involved on them. This ratio is critical in architectures with a hierarchical memory system, widely extended nowadays.

With the emergence of modern shared memory multi-core and many-core architectures, the parallel BLAS implementations have received further attention, and big efforts have been devoted to adapt the implementations to these architectures. Usually, computations are parallelized by implementing a multi-threaded code, and distributing the operations among the available processing units. In these implementations, the advantage of using Level-3 BLAS routines, is even more dramatic, as the memory bandwidth becomes a stronger bottleneck when shared by several processors.

From the performance viewpoint of the parallel implementations, we can conclude that:

- The performances of the BLAS-1 and BLAS-2 routines are dramatically limited by the pace at which memory can feed the processor with data.
- The Level-3 BLAS is more efficient, as for each memory access more calculations can be performed, attaining performances near the peak of the processor and near-optimal speedups in many operations. In addition, BLAS-3 routines can deliver higher parallel efficiencies, with a better adaptation to multi-core and many-core architectures.

3.1.2. Naming conventions

The concise, descriptive and homogeneous nomenclature of the BLAS routines proposed in the specification is one of its main advantages. The names of the BLAS routines consist of four to six characters. The main goal in the naming convention BLAS is to provide enough information on the functionality of the routine. In general, for the BLAS-2 and BLAS-3 routines, their names present the form XYYZZZ, where:

- X: Denotes the data type involved in the operation. Possible values are:
  - S: single-precision floating-point number.
  - D: double-precision floating-point number.
  - C: simple-precision complex number.
  - Z: double-precision complex number.

- YY: Denotes the type of matrix involved in the operation; the most common values are:
  - GE: general matrix.
  - SY: symmetric matrix.
  - TR: triangular matrix.
  - GB: general band matrix.
  - SB: symmetric band matrix.
- ZZZ: Denotes the type of operation performed by the routine. With a length of two or three characters, common examples are MM for a matrix-matrix product, MV for a matrix-vector product or RK for a rank-k update.

BLAS-1 routines do not follow the above scheme. As the operands for these operations are exclusively vectors, the only necessary information is the operation to be performed. In general, the name of these routines only informs about the data type (single or double precision, real or complex data), and the operation to be performed.

3.1.3. Storage schemes

The BLAS implementations usually deal with three different storage types:

- Canonical column-wise storage: The matrix is stored by columns as is usual, for example, in Fortran. Consecutive columns are stored contiguously in memory. This scheme is common in operations involving dense general matrices.
- Band storage: Used for band matrices, it only stores elements inside the band.
- Symmetric band storage: Used for symmetric or Hermitian band matrices, it only stores elements inside the band of the upper or lower triangular part of the matrix.

3.1.4. Overview of the Level-3 BLAS operations

The Level-3 BLAS targets matrix-matrix operations. The functionality of the Level-3 BLAS was designed to be limited. For example, no routines for matrix factorizations are included, as they are supported by higher-level libraries, such as LAPACK, which implements blocked algorithms for those purposes making use of Level-3 BLAS whenever possible. Instead, the Level-3 BLAS are intended to be a set of *basic* matrix algebra operations from which the developer is capable of implementing more complex routines.

The routines offered by Level-3 BLAS in real arithmetic support the following functionality:

- Matrix-matrix products (routines `xGEMM` for general matrices, and `xSYMM` for symmetric matrices):

$$\begin{array}{rcll} C & := & \alpha AB & + \ \beta C \\ C & := & \alpha A^T B & + \ \beta C \\ C & := & \alpha AB^T & + \ \beta C \\ C & := & \alpha A^T B^T & + \ \beta C \end{array}$$

- Rank-k and rank-$2k$ updates of a symmetric matrix C (routines `xSYRK` and `xSYR2K`, respectively):

$$\begin{array}{lcl} C & := & \alpha AA + \beta C \\ C & := & \alpha A^T A + \beta C \\ C & := & \alpha AB^T + \alpha BA^T + \beta C \\ C & := & \alpha A^T B + \alpha B^T A + \beta C \end{array}$$

- Multiplication of a general matrix C by a triangular matrix T (routine `xTRMM`):

$$\begin{array}{lcl} C & := & \alpha TC \\ C & := & \alpha T^T C \\ C & := & \alpha CT \\ C & := & \alpha CT^T \end{array}$$

- Solution of triangular systems of linear equations with multiple right-hand sides (routines `xTRSM`):

$$\begin{array}{lcl} B & := & \alpha T^{-1} B \\ B & := & \alpha T^{-T} B \\ B & := & \alpha BT^{-1} \\ B & := & \alpha BT^{-T} \end{array}$$

In the definitions above, α and β are scalars, matrices A, B and C are dense general matrices (symmetric in the corresponding routines), and T is an upper or lower triangular matrix. Analogous routines provide support for double precision and real/complex arithmetic.

Table 3.1 summarizes the Level-3 BLAS routines operating in real arithmetic. The names of the routines follow the conventions of the rest of the BLAS specification, with the first character denoting the data type of the matrix (`S` for single precision, `D` for double precision), second and third character denoting the type of matrix involved (`GE`, `SY` or `TR` for general, symmetric or triangular matrices, respectively) and the rest of the character denoting the type of operation (`MM` for matrix-matrix product, `RK` and `R2K` for rank-k and rank-$2k$ updates of a symmetric matrix, respectively, and `SM` for the solution of a system of linear equations for a matrix of right-hand sides).

To support the whole set of operations for each routine, all BLAS share a convention for their arguments Table 3.2 summarizes the available arguments for the Level-3 BLAS just described. For each routine, the parameters follow necessarily this order:

1. Arguments specifying options.

   The available option arguments are SIDE, TRANSA, TRANSB, TRANS, UPLO and DIAG, and their functionality and possible values are:

Routine	Operation	Comments	*flops*
xGEMM	$C := \alpha\ op(A)\ op(B) + \beta C$	$op(X) = X, X^T, X^H, C$ is $m \times n$	$2mnk$
xSYMM	$C := \alpha AB + \beta C$ $C := \alpha BA + \beta C$	C is $m \times n$, $A = A^T$	$2m^2n$ $2mn^2$
xSYRK	$C := \alpha AA^T + \beta C$ $C := \alpha A^TA + \beta C$	$C = C^T$ is $n \times n$	n^2k
xSYR2K	$C := \alpha AB^T + \alpha BA^T + \beta C$ $C := \alpha A^TB + \alpha B^TA + \beta C$	$C = C^T$ is $n \times n$	$2n^2k$
xTRMM	$C := \alpha\ op(A)C$ $C := \alpha\ Cop(A)$	$op(A) = A, A^T, A^H, C$ is $m \times n$	nm^2 mn^2
xTRSM	$C := \alpha\ op(A^{-1})C$ $C := \alpha\ Cop(A^{-1})$	$op(A) = A, A^T, A^H, C$ is $m \times n$	nm^2 mn^2

Table 3.1: Functionality and number of floating point operations of the studied BLAS routines

Option	Value	Meaning
SIDE	L	Multiply general matrix by symmetric/triangular matrix on the left
	R	Multiply general matrix by symmetric/triangular matrix on the right
TRANSX	N	Operate with the matrix
	T	Operate with the transpose of the matrix
UPLO	U	Reference only the upper triangle
	L	Reference only the lower triangle
DIAG	U	Matrix is unit triangular
	N	Matrix is nonunit triangular

2. Arguments defining matrix sizes.

   The sizes of the matrices are determined by the arguments M, N and K. The input-output matrices (B for the TR routines, C otherwise) is always $m \times n$ if rectangular, and $n \times n$ if square.

3. Input scalar.

   The scalars affecting input and input-output matrices receive the names `alpha` and `beta`, respectively.

4. Description of input matrices.

   The description of the matrices consists of the array name (A, B or C followed by the leading dimension of the array, LDA, LDB and LDC, respectively).

5. Input scalar (related to the input-output matrix, if available).

6. Description of the input-output matrix.

3.1.5. BLAS on Graphics Processors: NVIDIA CUBLAS

Since the emergence of the *GPGPU* concept in early 2000s, linear algebra problems have been solved by the scientific community in this class of hardware, often attaining remarkable performance results. Krüger and Westermann [90] developed a framework to deal with linear algebra

Routine	Parameters
GEMM	TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC
SYMM	SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA, C, LDC
SYRK	UPLO, TRANS, N, K, ALPHA, A, LDA, BETA, C, LDC
SYR2K	UPLO, TRANS, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC
TRMM	SIDE, UPLO, TRANS, DIAG, M, N, ALPHA, A, LDA, C, LDC
TRSM	SIDE, UPLO, TRANS, DIAG, M, N, ALPHA, A, LDA, C, LDC

Table 3.2: BLAS-3 parameters.

problems and data structures. They proposed the first building blocks to develop a wide variety of linear algebra codes running on the GPU, including BLAS implementations, and investigated the impact of the usage of ad-hoc data representations on the performance of dense, banded and sparse linear algebra routines. In [63] Fatahalian et al. conducted the first performance analysis of the matrix-matrix multiplication on programmable graphics processors. The first efforts towards the optimization and tuning of existing implementations were performed by Jiang et al. [86]; they designed a system that dynamically chose between a set of optimized kernels depending on several problem parameters. The optimizations implemented on those kernels are exclusive for the graphics processor, and very different to those tuning techniques often used on general-purpose processors.

Even though the attained performances were promising, the absence of general-purpose tools and APIs to easily develop codes for GPUs limited not only the final performance of the resulting routines, but also the number of applications that could be developed, due to the intricate programming models available and the the graphics-oriented concepts that had to be mastered.

In the mid 2000s, the emergence of CUDA, a new paradigm for general-purpose computations on graphics processors, brought the necessary tools to exploit the potential performance of the hardware. However, the key factor that attracted the attention of the scientific community was the introduction of NVIDIA CUBLAS, an implementation of the BLAS specification optimized for NVIDIA GPUs.

NVIDIA CUBLAS implements almost the full set of the three BLAS levels, providing a compatible interface with existing BLAS implementations. Thus, the porting of existing BLAS-based linear algebra codes is almost straightforward for programmers, with independence from their knowledge about GPU programming. The main difference between the BLAS implementations for general-purpose processors and NVIDIA CUBLAS is derived directly from the existence of two separate memory spaces: the system memory space and the GPU memory space. As GPUs can only operate with data allocated on GPU memory, the use of the NVIDIA CUBLAS library implies the creation of data structures on the GPU memory space, the transfer of data between memory spaces, the invocation of the appropriate NVIDIA CUBLAS routines, and the transfer of data back to main memory. To meet these requirements, NVIDIA CUBLAS provides a set of auxiliary functions for the creation and destruction of data layouts on the GPU memory, and for transferring data from/to main memory.

3.2. Evaluation of the performance of Level-3 NVIDIA CUBLAS

Tuned implementations of the BLAS-3 specification are of wide appeal, as they allow a fast acceleration of more complex codes based on this set of routines. The performance and scalability of the parallel Level-3 BLAS implementations has regained even more importance with the advent

of modern many-core architectures, such as graphics processors. These architectures are an example of the rapid increase in the ratio between the amount of processing capabilities (up to 240 cores in modern graphics processors) and the pace at which memory can provide them with data. Thus, despite the impressive peak computational power of these novel architectures, a careful choice of the algorithm becomes critical to attain remarkable speedups [82]. In this sense, the Level-3 BLAS are a clear candidate to achieve high performance on the graphics processor, as they exhibit a good ratio between computation and memory accesses. This section presents a detailed evaluation of the implementation of the routines in the Level-3 NVIDIA CUBLAS, comparing their performance with that attained for a tuned implementation of BLAS for modern general-purpose multi-core processors.

3.2.1. Evaluation of the performance of NVIDIA CUBLAS

The evaluation of the performance of the Level-3 routines of a specific BLAS implementation is a common way of extracting conclusions about both architecture capabilities and implementation particularities. In the case of NVIDIA CUBLAS, a detailed evaluation of the GEMM implementation will be useful to further optimize the rest of the BLAS-3 operations. In addition, the routines for general matrix-matrix multiplication are usually highly tuned in many BLAS implementations, and therefore its performance is illustrative of the potential of the underlying architecture.

The performance of the implementations of the rest of the BLAS-3 routines is also useful as their optimization may have a big impact on higher-level libraries that utilize them, e.g., LAPACK. Furthermore, a comparison between current implementations of BLAS for graphics processors and general-purpose processors is a fair way of extracting conclusions about the performance and specific capabilities of both architectures.

The procedure for the evaluation of the Level-3 NVIDIA CUBLAS covers the following cases and details:

- Evaluation of the GEMM implementation in NVIDIA CUBLAS for square and rectangular matrices and all combinations of transposed/no transposed operators.
- Evaluation of the rest of the BLAS-3 routines in NVIDIA CUBLAS for square and rectangular matrices and the most representative cases.
- Comparison of the attained performances with optimized versions of BLAS-3 for general-purpose processors.
- Evaluation of the capabilities of modern graphics processors for single and double-precision real arithmetic.

Performance results presented next count the number of *flops* as a measure unit. A flop is a single floating-point arithmetic operation. The number of *flops* required for each evaluated BLAS-3 routine is given in Table 3.1. To quantify the performance rate at which a given calculation is performed, the number of *flops* per second is measured. Thus, the term FLOPS represents the number of floating-point arithmetic operations per second attained by a certain implementation. Modern processors and GPUs often achieve GFLOPS (10^9 *flops* per second) or even TFLOPS (10^{12} *flops* per second). The maximum performance in the plots (y axis) represents the peak performance of the architecture (e.g., 933 GFLOPS in single precision and 78 GFLOPS in double precision for the TESLA C1060 of PECO).

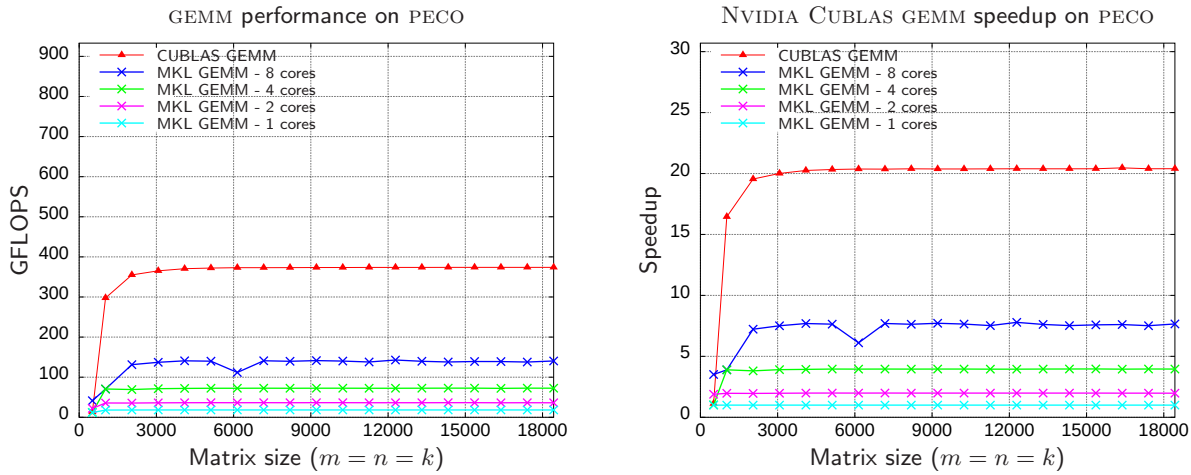


Figure 3.1: Performance of NVIDIA CUBLAS GEMM and Intel MKL implementation on 1, 2, 4 and 8 cores for square matrices (left). Speedups (right) consider the serial MKL implementation as a reference.

Evaluation of the BLAS-3. The GEMM implementation.

In this section, we evaluate the NVIDIA CUBLAS implementation of the matrix-matrix product, comparing its results with those of a high-performance BLAS implementation for a general-purpose processor. The performances and observed behaviors will be useful as a reference for further optimizations in the rest of BLAS-3 operations, and as an explanation for the observations made hereafter.

Figure 3.1 (left) compares the performances of the GEMM implementation in NVIDIA CUBLAS and the implementation of the matrix-matrix product routine in Intel MKL for 1, 2, 4 and 8 Intel Xeon cores. The specific operation in this case is $C := C + AB$. In the figure, performance is reported in terms of GFLOPS for matrices of increasing size. Matrices are square ($m = n = k$), and their dimensions are integer multiples of 1,024; the maximum matrix size tested is determined by the total amount of available GPU memory (adding the case where $n = 512$ to observe the performance of the implementations for small matrices). Single-precison real arithmetic was used in the experiments. In the case of the GPU implementations, data is transferred to the GPU memory prior to the execution of the corresponding routine, so the cost of data transfers between main memory and GPU memory is not included in the measurements. The speedups represented in the right plot of the figure are calculated taking the performance of Intel MKL for one Xeon core as a reference. The attained speedup for the NVIDIA CUBLAS implementation is about 21× compared with the serial BLAS on the CPU, and 2.5× compared with the multi-threaded MKL executed on 8 Intel Xeon cores.

Figure 3.2 reports the efficiency of the GEMM implementation calculated as the percentage of the peak performance of the corresponding architecture attained by the GEMM implementation. As pointed out in previous sections, the performance of BLAS-3 implementations is usually considered to be representative of the attainable performance of an architecture. Despite being a tuned implementation of the BLAS specification for the NVIDIA graphics processors, the performance of the routines of the Level-3 implementation in NVIDIA CUBLAS are usually far from the peak performance of the architecture.

On a GPU with NVIDIA CUBLAS, the sustained performance achieved by the GEMM implementation (373.4 GFLOPS) represents roughly a 40% of the peak performance of the architecture.

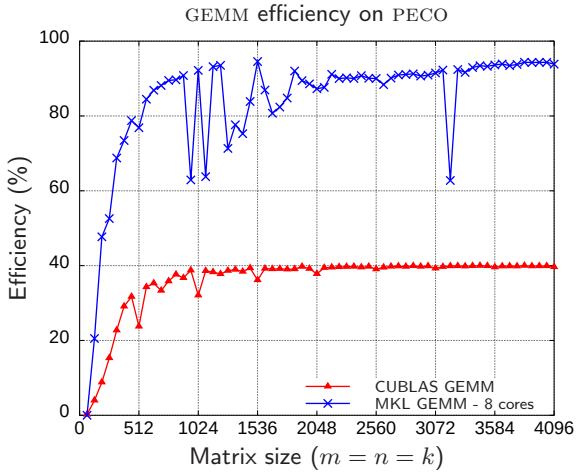


Figure 3.2: Efficiency of the GEMM implementation of NVIDIA CUBLAS and Intel MKL on 8 cores for square matrices.

The efficiency of the the tuned GEMM implementation on a multi-core processor with Intel MKL is around a 90% of the architecture peak performance (135.38 GFLOPS for a peak performance of 149.12 GFLOPS using the 8 cores of the system).

By comparing the GEMM implementations in NVIDIA CUBLAS and Intel MKL, several conclusions related to raw performance and efficiency can be extracted. First, modern GPUs demonstrate their potential in arithmetic-intensive operations (as in Level-3 BLAS), attaining remarkable speedups compared with the corresponding sequential or parallel versions for general-purpose processors. Second, the efficiency delivered by the GEMM implementation on the CPU is near optimal, while the NVIDIA CUBLAS implementation yields a GFLOPS rate far from the peak performance of the architecture. This characteristic is not due to a bad design or a suboptimal implementation of the NVIDIA CUBLAS routine; it is a typical behavior of general-purpose algorithms implemented on the GPU. The explanation is directly related to the increasing difference between the computing capabilities of modern GPUs and the rate at which data can be provided to the processing units or cores. This observation, usually known as *the memory wall* [147], is critical and determines the maximum performance that can be attained for a given operation [82]. This observation is not new; in fact, it was (successfully) predicted by John von Neumann in 1945 [143]:

> *" [. . .] This result deserves to be noted. It shows in a most striking way where the real difficulty, the main bottleneck, of an automatic very high speed computing device lies: At the memory."*

With the advent of multi-core and many-core processors (such as modern GPUs), the interest of this observation has been renewed. In practice, this involves a deeper analysis of which kind of algorithms are candidates to hide the memory wall and thus attain high performance on novel architectures. For algorithms that present high arithmetic intensity (that is, they are rich in arithmetic operations per memory transaction) the memory wall does not play such an important role. This is the case for graphics applications, for which the GPUs are designed. As the arithmetic intensity decreases, the memory wall becomes more determinant, reducing the maximum performance that can be attained.

To gain more thorough comprehension of the routines in the Level-3 NVIDIA CUBLAS, the implementations are evaluated with different values for two parameters. The first parameter is the storage layout of matrices in memory, which can be used to specify whether the matrices are

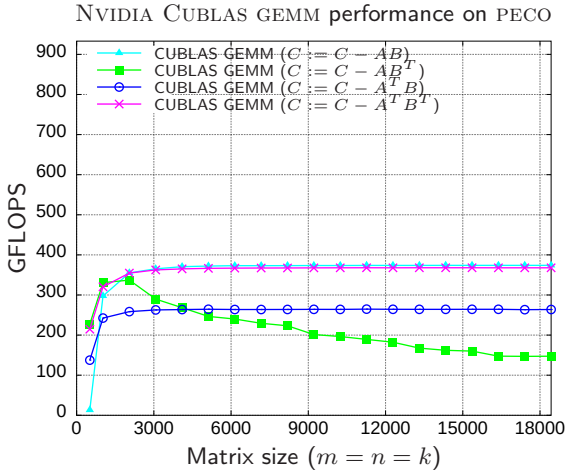


Figure 3.3: Performance of the GEMM routine of NVIDIA CUBLAS for square matrices.

transposed or not in memory. The second parameter is the dimension of the matrices involved in the operation.

Figure 3.3 shows the performance of the GEMM routine in NVIDIA CUBLAS for square matrices, for the four possible combinations of the parameters TRANSA and TRANSB: $C := C - AB$, $C := C - A^T B$, $C := C - AB^T$, and $C := C - A^T B^T$.

The behavior of the four cases of the GEMM implementation shows different levels of optimization of this routine depending on the layout in memory of its operands. More precisely, the cases in which A and B are both transposed or not transposed in memory are optimal. The implementation is specially poor for the case where the first operand A is not transposed while B is transposed, with performance rapidly decreasing as the dimension of the matrices increases.

The second key factor that affects the performance of BLAS is the dimension of the operands involved in the operation (matrices in the case of BLAS-3 routines). These sizes are given by the parameters M, N and K in the routines (see Table 3.2).

To analyze the impact of this factor in the final performance of NVIDIA CUBLAS, an evaluation of the GEMM routine for rectangular matrices has been done, fixing one of the dimensions is fixed on a small values and while the other two dimensions grow. Depending on the specific dimensions of the operands A and B, a set of three different tests have been performed, as detailed in Table 3.3. The performance results for these special shapes of the operands are important as they often appear in many situations in dense linear algebra routines, including BLAS operations (see Section 3.3) or more complex operations including those in LAPACK (see Chapter 4).

Figure 3.4 reports the performance results from this experiment. One of the dimensions is fixed to a small value (up to 128, as larger values did not offer substantial differences in performance) in each plot so that it corresponds to a different row in Table 3.3.

From the attained results, the dimensions m (number of rows of matrices A and C) and k (number of columns of matrix A and number of rows of matrix B) are the most critical factors for the performance of the operation. In addition, the performance of the routine improves as the value of the varying dimension increases. These values for the dimensions lead to two main insights:

- The shape of the operands in the GEMM routine is a key factor in the performance of the implementation. The implementation offers its optimal performance for relatively large values of the smallest dimension tested (values $m = n = k = 128$ in the figure). For blocked

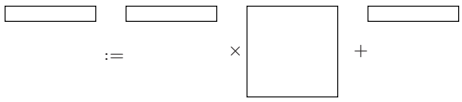
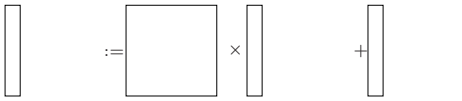
m	n	k	Operation
small	large	large	
small	large	large	
small	large	large	

Table 3.3: Shapes of the operands involved in the evaluation of the non-square matrices for the GEMM routine.

algorithms that make use of GEMM (or other BLAS routines), this will be a lower bound of the block sizes to be tested.

- From the observation of the shapes of the operands in Table 3.3, a common specific matrix shape involving a small m dimension (first row in the table) and a small k (third row in the table) offers a particular poor performance in NVIDIA CUBLAS; in both cases, a matrix with a reduced number of rows is used (A and C in the first case, only B in the second case). The GEMM implementation is specially inefficient when operating with matrices with this shape; in particular as two matrices with few rows are involved in the first case, performance results attained are poorer for it; for the third case, with only one matrix with few rows referenced, performance results are significantly better. When no matrices with a small number of rows are involved (second row of Table 3.3), the implementation is considerably more efficient.

Evaluation of SYMM, SYRK, SYR2K, TRMM and TRSM

Table 3.1 shows the performances attained for the remaining routines from the Level-3 BLAS. One representative case for each routine has been selected. The concrete cases are as follows:

SYRK: $C := C - AA^T$, where only the lower triangular part of C is stored and updated.

SYMM: $C := C + AB$, where only the lower triangular part of A is referenced.

SYR2K: $C := C - AB^T - BA^T$, where only the lower triangular part of C is stored and updated.

TRMM: $C := C + AB$, where A is upper triangular.

TRSM: $XA^T = B$, where A is lower triangular.

The performance attained for these routines is shown in Figure 3.5. The plot on the left reports the performance of the routines SYRK, SYR2K and SYMM. The behavior of the first two routines is similar, attaining a peak performance of 141 GFLOPS for the largest matrix sizes. This performance is far from the performance offered by the GEMM implementation (see Figure 3.3), where the NVIDIA CUBLAS implementation delivered 375 GFLOPS. Although the SYMM routine

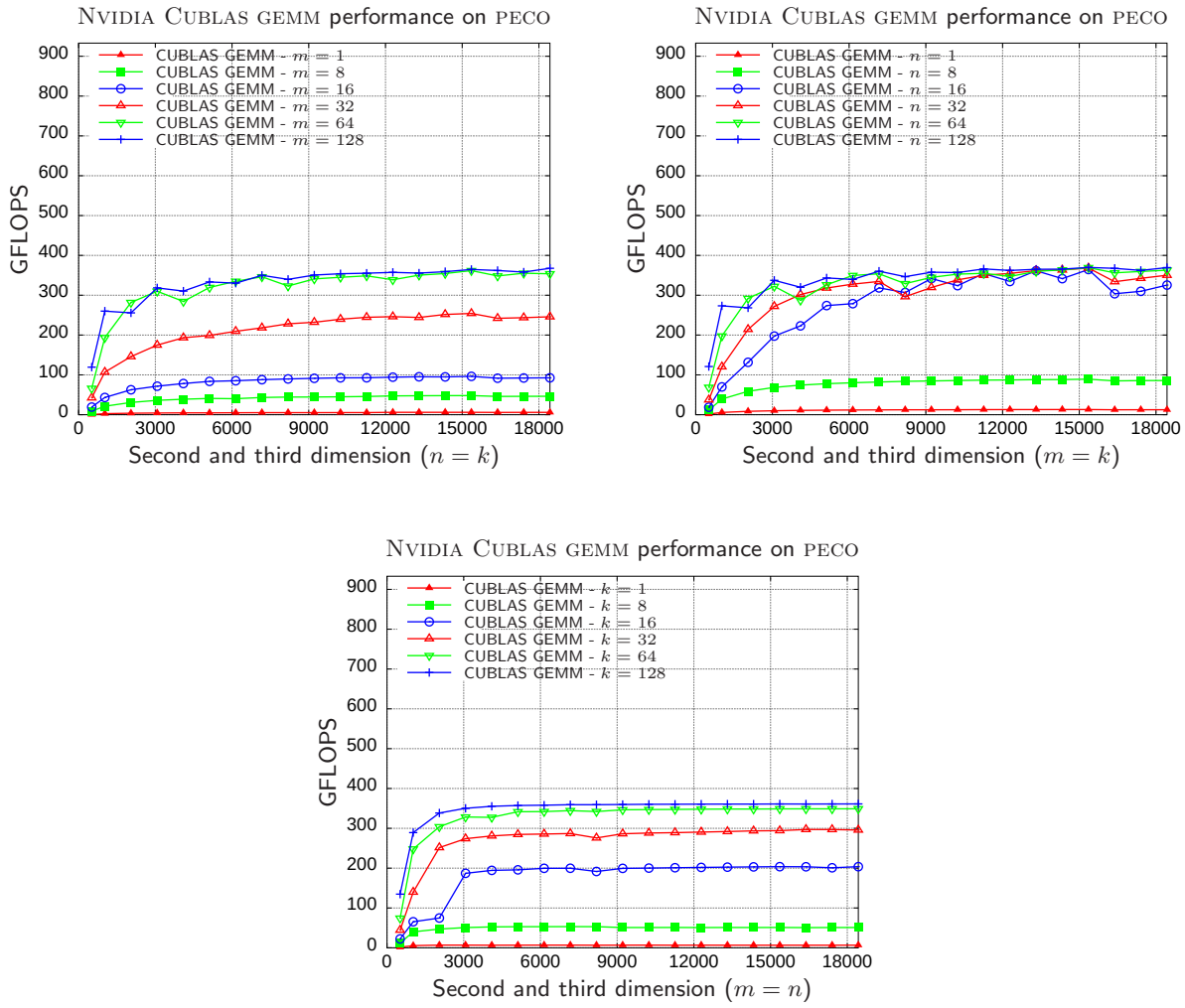


Figure 3.4: Performance of the GEMM implementation in NVIDIA CUBLAS for rectangular matrices. Top-left: m dimension is fixed. Top-right: n dimension is fixed. Bottom-center: k dimension is fixed.

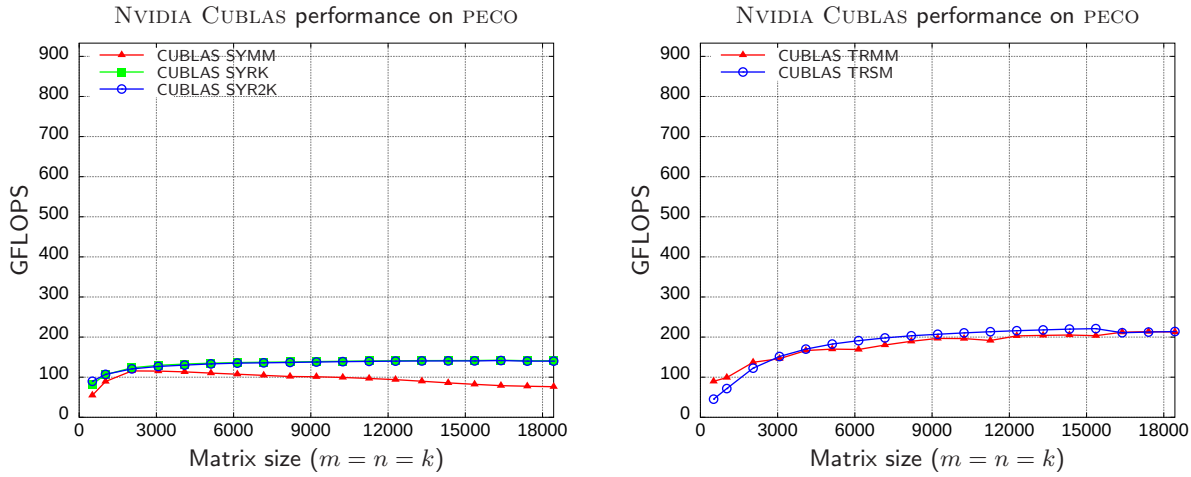


Figure 3.5: Performance of the Level-3 routines in NVIDIA CUBLAS for square matrices. Left: SYMM, SYRK and SYR2K. Right: TRMM and TRSM.

provides a functionality similar to that of GEMM, it yields very poor performance as the dimension of the matrices increases.

The results for the TRMM and the TRSM routines are given in the right-hand plot of the figure, and exhibit a similar behavior for both operations, attaining slightly better results than those of SYRK, SYR2K and SYMM (212 GFLOPS in both cases). Those results are also far from those obtained with GEMM.

There are some important remarks that can be extracted from the evaluation of the main BLAS-3 routines in NVIDIA CUBLAS:

- The Level-3 BLAS routines offer important speedups when are implemented in many-core processors such as GPUs compared with the corresponding tuned implementations executed in general-purpose processors (see Figure 3.1 for the GEMM implementation. Similar qualitative results have been observed for the remaining routines in Level-3 BLAS).

- The observed performance of the implementations in NVIDIA CUBLAS is far from the theoretical peak performance of the architecture, attaining roughly a 40% of efficiency for GEMM. This behavior differs from the efficiency of the tuned implementations for general-purpose processors, in which the efficiency is near optimal (see Figure 3.2).

- Even being an optimized library, the degree of efficiency of the different routines in Level-3 NVIDIA CUBLAS is not homogeneous. Although the general matrix-matrix multiplication offers high performance, the efficiency attained by the rest of routines is much lower than that of GEMM (see Figures 3.3 and 3.5).

- Matrix shapes play an important role in the performance of the NVIDIA CUBLAS implementations (see Figure 3.4). Whether this impact is related to the architecture or the specific implementation is not clear at this point, and will be determined by the improvements introduced in Section 3.3.

3.2.2. Influence of data transfers

Like other heterogeneous architectures based on hardware accelerators, GPU-based platforms consist of two (logically and physically) separate memory spaces. Therefore, data to be processed on the GPU must be previously transferred to GPU memory before the computation commences, and results must be retrieved back to main memory once the computation is completed.

The performance results for the NVIDIA CUBLAS routines shown in previous experiments do not include the time devoted to data transfers between memory spaces, as those transfers are not always needed (e.g., the data can be already stored in GPU memory when the kernel is invoked). However, given the reduced bandwidth of the PCI-Express bus, it is important to evaluate the impact that data transfers have on the overall computation procedure.

The usage of *page-locked* memory in the host side (also known as *pinned* memory) increases the performance of the data transfers through the PCI-Express bus by associating a given virtual memory page with a real page frame, so that it will never be paged out, and no page faults will occur. This technique is specially interesting for time-critical and real-time processes. The impact of the usage of *pinned* memory on the data transfer rate can be observed by comparing the transfer rates in Figure 3.6 (see left vertical axes). The use of *page-locked* memory is mandatory to measure the *real* peak transfer rates that can be attained by the PCI-Express bus. Figure 3.6 evaluates the impact of data transfers for a specific NVIDIA CUBLAS operation: in this case, the matrix-matrix multiplication using single-precision arithmetic of the form $C := C + AB$. For this operation, the transfer of the three input matrices, A, B and C is performed before the invocation of the NVIDIA CUBLAS kernel; upon completion, the output matrix C is transferred back to main memory. The experimental results compare the performance of a basic data transfer scheme and that of an optimized transfer scenario in which *page-locked* memory is used.

The PCI-Express bus presents an asymmetric behavior; transferring data from main memory to GPU memory delivers a higher bandwidth than the other way round. This difference is specially relevant when using *page-locked* memory; in this case, the sustained peak bandwidth when transferring data to GPU memory is 6 Gbytes per second, a rate that can be viewed as the real sustainable bandwidth that the architecture can offer to transfer data from main memory. When transferring data from GPU memory, the sustained bandwidth is reduced to 4 Gbytes/s. In the case of *non page-locked* memory, the difference between both transfer directions is reduced: 2 Gbytes/s for uploading data to the GPU and 1.5 Gbytes/s for the inverse process.

In general for operations in which the amount of data transfer is $O(n^2)$ while the amount of calculations is $O(n^3)$, the transfer stages do not have an important impact on the total necessary computation time provided matrices are large enough.

Figure 3.6 also illustrates the percentage of time devoted to data transfers compared with the total time for the operation (see scale of the right y axis). The percentage of time devoted to calculation is much higher than that consumed by transfers as the size of the input matrices grows. As an example, consider the results for non-pinned memory for a matrix-matrix multiplication of dimension $m = n = k = 2000$ in Figure 3.6. For this matrix size, the overhead of data transfers is very significant, taking a 60% of the total execution time. On the other hand, consider the same operation for matrices of dimension $m = n = k = 18000$: this overhead is reduced to a 8% of the overall time. These values are reduced using pinned memory, presenting a relative cost of a 28% and 4%, respectively.

Thus, in memory-bounded algorithms (e.g. BLAS-2 routines, that involve $O(n^2)$ data for $O(n^2)$ computations, or BLAS-3 routines involving small data structures), data transfers are a non-negligible overhead source that must be taken into account.

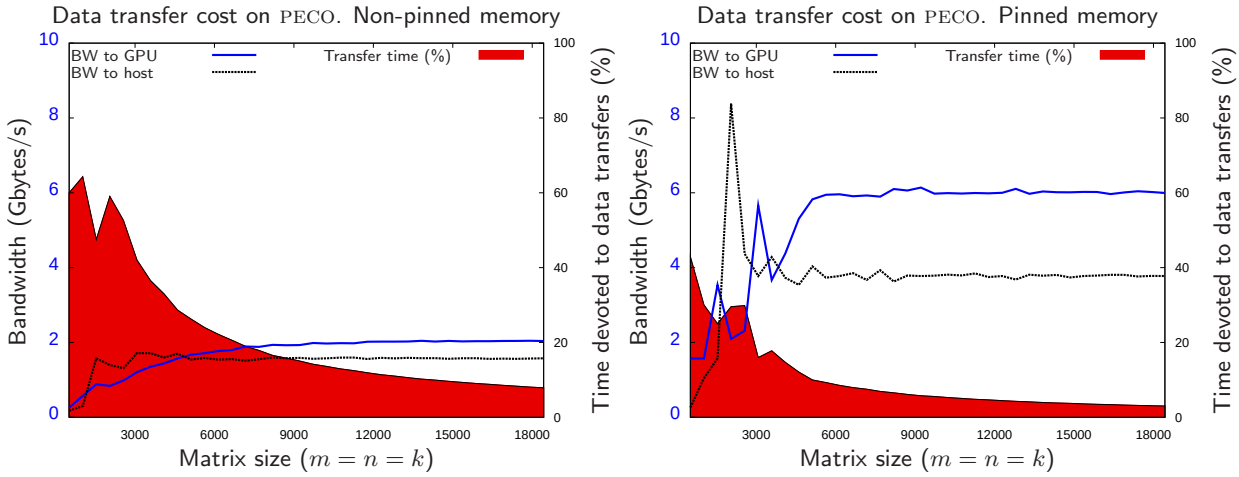


Figure 3.6: Percentage of time devoted to data transfers for the matrix-matrix multiplication. The plots include the effective bandwidth offered by the PCI-Express bus using *non-pinned* memory and *pinned* memory (left and right, respectively).

3.3. Improvements in the performance of Level-3 NVIDIA CUBLAS

After analyzing the previous results, we conclude that the routine implementations in NVIDIA CUBLAS exhibit heterogeneous performances. This irregular behavior depends not only on the particular operation (compare the performance of different BLAS-3 routines in Figure 3.5), but also on each specific case of each routine (see the behavior of each case for the GEMM routine in NVIDIA CUBLAS on Figure 3.3).

This irregular behavior justifies one of the contributions of the work presented in this thesis. Low-level programming requires a deep knowledge of the underlying architectural details and ad-hoc programming languages. Very often, these details are closed to the scientific community; in other cases, even the hardware vendors are not able to fully exploit them for every implementation.

Developing a low-level approach from the initial stages of the library development is not always the correct choice. Several key decisions can have critical consequences on performance, and very frequently they are difficult to evaluate when programming at low level, driving to a loss of efficiency in the programming effort.

In this section, we demonstrate how, by applying some high-level methodologies for the derivation and evaluation of dense linear algebra algorithms, it is possible to extract key insights that are crucial to deliver high-performance codes. Moreover, we also show that these codes are highly competitive from the performance perspective, attaining better performance results than those obtained with a low-level programming methodology.

Our implementations improve the performance of the NVIDIA CUBLAS routines by applying two main techniques. First, for each BLAS-3 routine we cast a major part of the operations in terms of the high performance GEMM routine. Second, we systematically derive a full set of blocked algorithmic variants and evaluate them to find out the optimal one for each combination of parameter values and routine. Other parameters, such as the block size, are also systematically studied for each algorithmic variant. The result is a full evaluation and a high-performance implementation of the main routines in the Level-3 BLAS. The attained results clearly improve those in NVIDIA CUBLAS for the tested routines.

3.3.1. GEMM-based programming for the Level-3 BLAS

Improving the performance of the BLAS usually involves a deep knowledge of the internals of the target architecture, in this case the graphics architecture provided by NVIDIA, and a hard coding effort. This deep knowledge is often translated into a small set of complex and efficient codes adapted to the underlying architecture [142]. An illustrative example of this development complexity, as exposed in Section 3.2.1, is the NVIDIA CUBLAS implementation. While the GEMM implementation in NVIDIA CUBLAS delivers high performance, the rest of the Level-3 NVIDIA CUBLAS routines are sub-optimal, offering worse efficiency, far from that attained for the general matrix-matrix multiplication. This performance analysis gives an idea of the necessary programming effort in the development of tuned codes for the GPU.

However, there is a different way for developing high performance kernels for the BLAS. Following this alternative approach, the new implementations of the BLAS are based on a small set of tuned codes, or *inner kernels*, taking benefit from their high performance to boost the performance of the new codes. This approach has been successfully applied in other high performance BLAS implementations, such as *GotoBLAS* [72] for general-purpose processors, attaining remarkable speedups without affecting the development effort. As a novelty, we apply a similar technique for the development of a full set of optimized dense linear algebra routines for modern graphics processors.

Besides increasing performance, the main advantage of the usage of tuned inner kernels is programmability. With this approach, the low-level architectural details have already been exploited in the inner-kernel implementation, and in consequence the programmer does not have to deal with them in the new codes. In addition, any further optimizations of the inner kernel will automatically translate into an improvement in the performance of higher-level codes, without any additional development effort.

Given the performance results observed in Section 3.2.1, in our case it will be convenient to use the GEMM implementation in NVIDIA CUBLAS as the building block for developing the rest of the tuned BLAS. In other words, the main goal is to derive algorithms for the BLAS-3 routines that reformulate a big part of its operations in terms of the high-performance GEMM implementation in NVIDIA CUBLAS.

This technique was first applied for general-purpose architectures by Kågström, Per Ling and Van Loan [88], but the GEMM-based concept can be ported to any architecture as long as there exists a tuned implementation of the general matrix-matrix multiplication routine, to build a complete and tuned BLAS-3 implementation.

Building a BLAS-3 based on this guidelines involves some remarkable advantages:

- The GEMM routine is usually illustrative of the peak performance of the architecture, so manufacturers often devote big efforts to tune it, delivering near optimal implementations for their own architectures. Although the efficiency of the GEMM implementation in NVIDIA CUBLAS, is far from the theoretical peak of the architecture, this routine is considered to be a near-optimal implementation that exploits all the capabilities of the GPU for general-purpose applications, and thus it is considered to reflect the practical peak performance of the architecture for this type of implementations [142]. Our GEMM-based implementations will inherit this high performance for each BLAS-3 routine.

- With this approach, the optimizing effort can be focused on just one routine (in this case, the matrix-matrix multiplication). The potential benefits of the optimization work will be immediately effective for the rest of the routines. Similarly, with the advent of new architectures,

or even with future modifications of the current ones, only one routine should be adapted to the new hardware in order to get a global boost in performance.

- Although this is a high-level approach, the insights gained from the evaluation of new routines (for example, optimal block sizes or algorithmic variants) can be directly applied to lower levels of abstraction (assembly code or CUDA code) for future native implementations.

- The developed algorithms are portable to different architectures if a tuned optimization of the matrix-matrix multiplication exists for those architectures.

Modularity and performance are two key goals in the development of linear algebra algorithms. Usually, the derivation of *blocked algorithms* is the first step towards the optimization of the performance of the implementation, improving data reuse and exploitation of the memory hierarchy in general-purpose architectures [70]. In addition to the performance boost, the usage of blocked algorithms also drives to modular algorithms, in which the operations performed on each block can be reformulated in terms of different BLAS-3 routines.

For example, the general matrix-matrix multiplication can be reformulated in terms of operations with sub-matrices of the original data. Figure 3.7 shows a possible algorithm for the matrix-matrix multiplication. At the beginning of the iteration, C_0 and B_0 have already been computed and used, respectively. In the current iteration, the next panel of matrix C is updated performing the operation $C_1 := C_1 + AB_1$. Then, the preparation for the next iteration shifts C_1 and B_1 to the next blocks of data, increasing the number of columns in C_0 and B_0, since they contain more processed data at this point. This visual description, together with the three main stages involved in the algorithm (block definition, operations on blocks and data repartitioning), drives to the algorithm given in Figure 3.8 in FLAME notation.

As an illustrative implementation of the GEMM-based approach, we follow a similar block-oriented procedure as shown in Figure 3.7 for the blocked SYRK (symmetric rank-k update) implementation in BLAS-3. The conclusions and procedure applied to SYRK can be easily extended to the rest of BLAS-3.

The SYRK routine performs one of the two following rank-k updates:

$$\begin{aligned} C &:= \alpha AA^T + \beta C, \\ C &:= \alpha A^T A + \beta C, \end{aligned}$$

where C is $n \times n$, symmetric and stored as an upper or lower triangular matrix. The matrix A is $n \times k$ or $k \times n$, respectively.

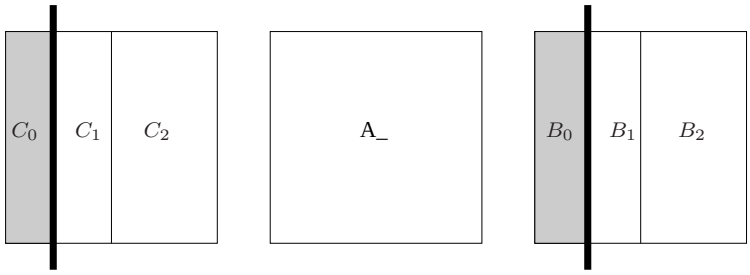
For each iteration of the blocked algorithm (see Figure 3.9), the update of the current block of columns of C involves two different tasks operating on two different blocks of C:

1. A diagonal block of the symmetric matrix C: $C_{11} := C_{11} + A_1 A_1^T$.

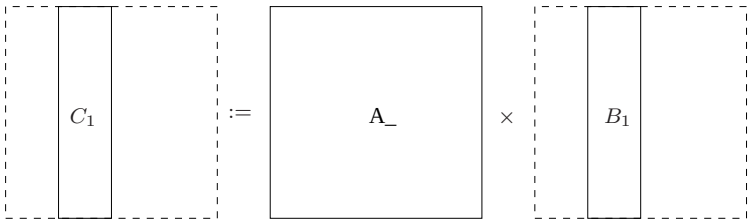
2. An off-diagonal block of C: $C_{21} := C_{21} + A_2 A_1^T$.

The operations involving the off-diagonal block of C can be carried out using the general matrix-matrix multiplication routine in BLAS (GEMM). Attending to the structure of the diagonal blocks, only those blocks must be updated using the rank-k update operation in BLAS (SYRK). Thus, in a major part of the iterations, most of the computation can be cast in terms of the highly tuned general matrix-matrix multiplication routine, and only certain blocks (in this case, the diagonal ones) need to be updated using the SYRK operation.

Step 1: Partitioning before iteration.



Step 2: Computation in iteration.



Step 3: Progress of partitioning in preparation for the next iteration.

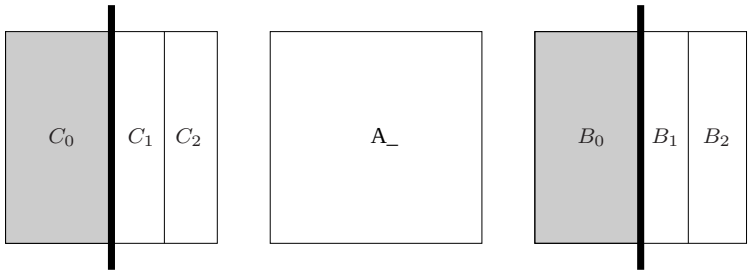


Figure 3.7: A visualization of the algorithm for matrix-panel variant of GEMM.

Algorithm: GEMM_MP(A, B, C)

Partition $B \rightarrow \left(\begin{array}{c|c} B_L & B_R \end{array} \right)$, $C \rightarrow \left(\begin{array}{c|c} C_L & C_R \end{array} \right)$
where B_L has 0 columns, C_L has 0 columns
while $n(B_L) < n(B)$ **do**
Determine block size b
Repartition

$$\left(\begin{array}{c|c} B_L & B_R \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array} \right),$$
$$\left(\begin{array}{c|c} C_L & C_R \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array} \right)$$

where B_1 has b columns, C_1 has b columns

$C_1 := C_1 + AB_1$

Continue with

$$\left(\begin{array}{c|c} B_L & B_R \end{array} \right) \leftarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array} \right),$$
$$\left(\begin{array}{c|c} C_L & C_R \end{array} \right) \leftarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array} \right)$$

endwhile

Figure 3.8: Matrix-panel variant of the algorithm for computing the matrix-matrix multiplication.

The total number of flops for the SYRK routine is $kn(n+1)$ [12]. Defining a block size b, and considering the dimension n as a multiple of b, the number of flops to update each diagonal block of C (with dimension $b \times b$) in the algorithm in Figure 3.9 is:

$$\texttt{fl}_{\texttt{diag}} = kb(b+1) \approx kb^2 \text{ flops.} \tag{3.1}$$

As there are $\frac{n}{b}$ diagonal blocks in C, the total number of flops cast in terms of the SYRK routine in the whole procedure is:

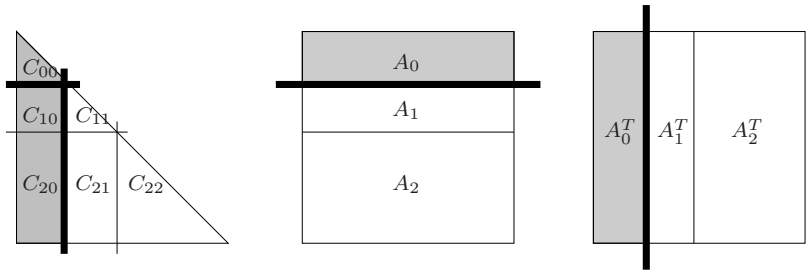
$$\texttt{fl}_{\texttt{inner\_syrk}} = \frac{n}{b}\texttt{fl}_{\texttt{diag}} = nk(b+1) \approx nkb \text{ flops.} \tag{3.2}$$

Therefore, the remaining operations are exclusively executed as general matrix-matrix operations, being:

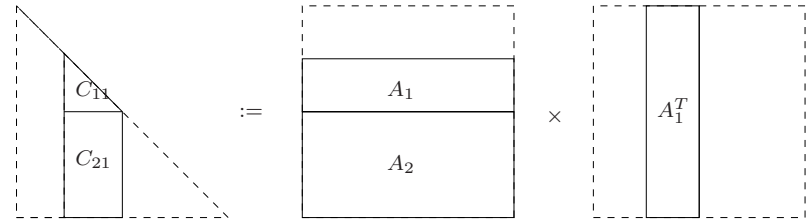
$$\texttt{fl}_{\texttt{gemm}} = kn(n+1) - nk(b+1) = kn^2 - nkb \text{ flops.} \tag{3.3}$$

Observe that, while the number of flops cast in terms of SYRK increases linearly with n (provided $b \ll n$), the number of flops cast in terms of GEMM increases quadratically with this dimension. This relationship, or GEMM-fraction of the algorithm (in a similar way as the *Level-3 fraction* in [70]), will be critical for the final performance attained by the algorithm. From the results obtained from the evaluation of GEMM in the NVIDIA CUBLAS implementation, the exploitation of this amount of GEMM-based flops is expected to have a considerable impact in the final performance results of the new GEMM-based SYRK implementation.

Step 1: Partitioning before iteration.



Step 2: Computation in iteration.



Step 3: Progress of partitioning in preparation for the next iteration.

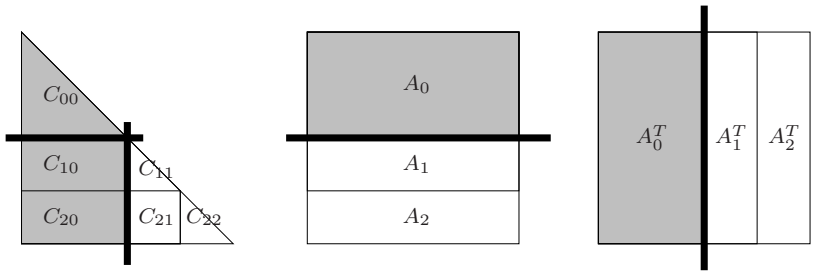


Figure 3.9: A visualization of the algorithm for matrix-panel variant of SYRK.

Name	Letter	Dimensions
Matrix	M	Both dimensions are large.
Panel	P	One of the dimensions is small.
Block	B	Both dimensions are small.

Table 3.4: Different names given to the partitioned sub-matrices according to their shapes.

3.3.2. Systematic development and evaluation of algorithmic variants

The GEMM-based approach exploits the usual near-optimal performance of the GEMM routine in many BLAS implementations. In addition to the performance of the specific kernels used in each routine, a second key factor in the final performance achieved for each implementation is the *shape* of the blocks involved in those inner kernels. For general-purpose architectures, those shapes have a direct impact on important factors such as cache memory exploitation, number of TLB misses or register usage [72], to name only a few.

For a given linear algebra operation, there usually exists a set of loop-based algorithms. The number of flops involved is exactly the same for all algorithmic variant, operating on the same data. The difference between them is basically the order in which operations are performed and, in blocked algorithms, the performance of the specific underlying BLAS operations.

In our case, the performance of the routines in NVIDIA CUBLAS in general, and that of the GEMM implementation in particular, are considerably influenced by the shape of its operands, as shown in the performance evaluation of Section 3.2.1, Figure 3.4. Thus, it is desirable to have multiple *algorithmic variants* at the disposal of the library user so that the best algorithm can be chosen for each situation. Even though the operations ultimately performed by each algorithmic variant are the same, there are many factors difficult to predict or control that can be exposed and solved by the different algorithmic variants.

Let us illustrate this with an example: the derivation of different algorithms for the GEMM operation. Given an $m \times n$ matrix X, consider two different partitionings of X into blocks of rows and blocks of columns:

$$X = \left(\begin{array}{c|c|c|c} \hat{X}_0 & \hat{X}_1 & \cdots & \hat{X}_{n-1} \end{array} \right) = \left(\begin{array}{c} \bar{X}_0 \\ \hline \bar{X}_1 \\ \hline \vdots \\ \hline \bar{X}_{m-1} \end{array} \right), \tag{3.4}$$

where $\hat{X}_j$ has n_b columns and $\bar{X}_i$ has m_b rows. For simplicity, we will assume hereafter that the number of rows/columns of X is an integer multiple of both m_b and n_b.

Depending on the dimensions of the sub-matrices (blocks) obtained in the partitioning process, we denote them as *matrices*, *panels* or *blocks*, as detailed in Table 3.4. The implementation of the matrix-matrix multiplication can be decomposed into multiplications with those sub-matrices. Thus, by performing different row and column partitioning schemes for the matrices A and B, it is possible to derive three different algorithmic variants, depending on the specific shapes of the sub-matrices involved. The name received by each variant depends on the shapes of the operands in them:

- **Matrix-panel variant** (GEMM_MP): Figure 3.7 represents an algorithm in which, for each iteration, the computation involves a *matrix* (A) and a *panel* of columns (B_1), to obtain

a new *panel* of columns of C. The algorithmic variant that proceeds this way is shown in Figure 3.8.

- **Panel-matrix variant** (GEMM_PM): In each iteration, a panel of rows of matrix A is multiplied by the whole matrix B, to obtain a new panel of rows of the result matrix C.
- **Panel-panel variant** (GEMM_PP): In the each iteration, a panel of columns of matrix A and a panel of rows of matrix B are multiplied to update the whole matrix C.

The algorithms in FLAME notation for the last two variants are shown in Figure 3.10.

Algorithm: GEMM_PM(A, B, C)

Partition $A \rightarrow \left(\frac{A_T}{A_B}\right)$, $C \rightarrow \left(\frac{C_T}{C_B}\right)$

where A_T has 0 rows, C_T has 0 rows

while $m(A_T) < m(A)$ **do**

Determine block size b

Repartition

$$\left(\frac{A_T}{A_B}\right) \rightarrow \left(\frac{A_0}{\frac{A_1}{A_2}}\right), \left(\frac{C_T}{C_B}\right) \rightarrow \left(\frac{C_0}{\frac{C_1}{C_2}}\right)$$

where A_1 has b rows, C_1 has b rows

$C_1 := C_1 + A_1 B$

Continue with

$$\left(\frac{A_T}{A_B}\right) \leftarrow \left(\frac{A_0}{\frac{A_1}{A_2}}\right), \left(\frac{C_T}{C_B}\right) \leftarrow \left(\frac{C_0}{\frac{C_1}{C_2}}\right)$$

endwhile

Algorithm: GEMM_PP(A, B, C)

Partition $A \rightarrow \left(A_L \middle| A_R\right)$, $B \rightarrow \left(\frac{B_T}{B_B}\right)$

where A_L has 0 columns, B_T has 0 rows

while $n(A_L) < n(A)$ **do**

Determine block size b

Repartition

$$\left(A_L \middle| A_R\right) \rightarrow \left(A_0 \middle| A_1 \middle| A_2\right),$$
$$\left(\frac{B_T}{B_B}\right) \rightarrow \left(\frac{B_0}{\frac{B_1}{B_2}}\right)$$

where A_1 has b columns, B_1 has b rows

$C := C + A_1 B_1$

Continue with

$$\left(A_L \middle| A_R\right) \leftarrow \left(A_0 \middle| A_1 \middle| A_2\right),$$
$$\left(\frac{B_T}{B_B}\right) \leftarrow \left(\frac{B_0}{\frac{B_1}{B_2}}\right)$$

endwhile

Figure 3.10: Algorithms for computing the matrix-matrix product. Panel-matrix variant (left), and panel-panel variant (right).

Several algorithmic variants with *matrix-panel*, *panel-matrix* and *panel-panel* partitionings are also possible for the rest of BLAS-3. To show this, we propose three different algorithmic variants for the SYRK operation $C := C - AA^T$ (with only the lower triangular part of C being updated), presented in FLAME notation in Figure 3.11. For each iteration of the loop, the algorithms in that figure also include the BLAS-3 kernel that has to be invoked in order to perform the corresponding operation.

The name of each algorithm describes the shape of the sub-matrices of A and A^T that appear as input operands in the operations performed in each iteration of the blocked algorithm. Three different algorithmic variants of the SYRK routine are proposed:

- **Matrix-panel variant** (SYRK_MP): For each iteration, the computation involves a *panel* of A^T (A_1^T) and a *sub-matrix* of A, to obtain a *panel* of columns of C.
- **Panel-matrix variant** (SYRK_PM): For each iteration of the *panel-matrix* variant, a panel of rows of matrix A (A_1) is multiplied by a sub-matrix of A^T, to obtain a new panel of rows of C.
- **Panel-panel variant** (SYRK_PP): In the *panel-panel* variant a panel of columns of matrix A and a panel of rows of matrix A^T are multiplied to update the whole symmetric matrix C at each iteration.

The different performances attained by each algorithmic variant of the SYRK routine exposed in Figure 3.11 are dictated by the specific routines invoked in each iteration and, more precisely, by the specific of exploitation of the underlying hardware derived from the different shapes of the operands involved. As an example, consider the differences between the *matrix-panel* and the *panel-matrix* variants:

- In each iteration of the *matrix-panel* variant, two invocations to BLAS-3 routines are performed to update the current block of columns of the result matrix C. The first operation, $C_{11} := C_{11} - A_1 A_1^T$, updates the diagonal block C_{11}, and thus is performed by a SYRK routine invocation. The second operation, $C_{21} := C_{21} - A_2 A_1^T$, is cast in terms of GEMM.
- For the *panel-matrix* algorithm, the SYRK invocation is executed on the same block, but the GEMM updates the panel of rows C_{01}.

Although the same routines are invoked in the two variants, the difference resides in the shapes of the operands of routine GEMM, a critical factor as we have shown. While in the *matrix-panel* variant a matrix (A_2) is multiplied by a panel of columns of matrix A^T (A_1^T) to update a panel of columns of matrix C (C_{21}), in the *panel-matrix* variant, a panel of rows (A_1) is multiplied by a matrix (A_0^T) to update a panel of rows of the matrix C (block C_{10} in the FLAME algorithm). Table 3.5 summarizes the operations and shapes involved in each algorithmic variant of the blocked SYRK implementation.

It is also possible to derive different algorithmic variants for the rest of the BLAS-3 routines. Specifically, three different variants, analogous to the *matrix-panel*, *panel-matrix* and *panel-panel* variants shown for the GEMM and the SYRK routines, have also been developed for routines SYMM, SYR2K, TRMM, and TRSM; see Figures A.1 to A.4.

3.4. Experimental results

By using the FLAME methodology, an implementation of one case[1] for each kernel of the Level-3 BLAS has been developed and evaluated. The new codes follow a high-level approach, invoking existing NVIDIA CUBLAS routines much in the line of higher-level libraries such as LAPACK. Therefore, no low-level programming is performed to tune the final codes. As an advantage of

[1] We consider a *case* as a given combination of the available parameters in each BLAS routine.

Algorithm: SYRK_MP_PM(C, A)

Partition

$$C \rightarrow \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right), A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)$$

where C_{TL} is 0×0, A_T has 0 rows

while $m(C_{TL}) < m(C)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right),$$

$$\left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right)$$

where C_{11} is $b \times b$, A_1 has b rows

SYRK_MP

$C_{11} := C_{11} - A_1 A_1^T$ (SYRK)

$C_{21} := C_{21} - A_2 A_1^T$ (GEMM)

SYRK_PM

$C_{10} := C_{10} - A_1 A_0^T$ (GEMM)

$C_{11} := C_{11} - A_1 A_1^T$ (SYRK)

Continue with

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right),$$

$$\left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right)$$

endwhile

Algorithm: SYRK_PP(C, A)

Partition $A \rightarrow \left(\begin{array}{c|c} A_L & A_R \end{array}\right)$

where A_L has 0 columns

while $n(A_L) < n(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_L & A_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_0 & A_1 & A_2 \end{array}\right)$$

where A_1 has b columns

$C := C - A_1 A_1^T$ (SYRK)

Continue with

$$\left(\begin{array}{c|c} A_L & A_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_0 & A_1 & A_2 \end{array}\right)$$

endwhile

Figure 3.11: Algorithms for computing SYRK. *Matrix-panel* and *panel-matrix* variants (left), and *panel-panel* variant (right).

this approach, the development process to attain high performance codes is made easier, while simultaneously, a full family of algorithms is at the disposal of the developer to choose the optimal one for each specific routine, and each specific case for that routine. The conclusions extracted from the algorithmic perspective (basically optimal algorithmic variants and best block sizes) can be directly applied to lower level codings, such as the native CUDA implementations in which the

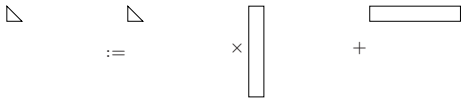
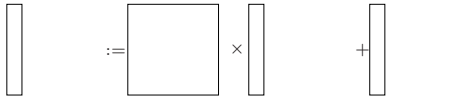
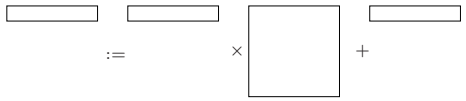
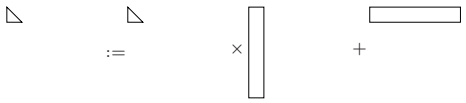
Variant	Operations in loop body	Routine	Shapes of the operands
SYRK_MP	$C_{11} := C_{11} - A_1 A_1^T$	SYRK	
	$C_{21} := C_{21} - A_2 A_1^T$	GEMM	
SYRK_PM	$C_{10} := C_{10} - A_1 A_0^T$	GEMM	
	$C_{11} := C_{11} - A_1 A_1^T$	SYRK	
SYRK_PP	$C := C - A_1 A_1^T$	SYRK	

Table 3.5: Operations and shapes of the operands involved in the blocked SYRK algorithmic variants. The partitionings assumed for each variant are those described in Figure 3.11.

NVIDIA CUBLAS routines are based. As an additional advantage, any internal improvement in the performance of the underlying implementation has an immediate impact on the final performance of the developed BLAS routines. Thus, direct performance improvements in the new implementations are expected as future tuned implementations of the GEMM routine are released by NVIDIA or by the scientific community.

The FLAME algorithms derived for each BLAS-3 routine expose two different lines of study to be considered to achieve high performance. For example, see the algorithms for the SYRK routine shown in Figure 3.11. First, the block size b must be chosen before the blocked algorithm begins. Second, different algorithmic variants must be considered in order to achieve the most suitable ones.

3.4.1. Impact of the block size

In addition to the particular algorithmic variant chosen, the block size is a critical parameter in the final performance of blocked algorithms in general, and particularly important for the blocked variants of the BLAS-3. The influence of the matrix dimensions on the performance the GEMM routine in NVIDIA CUBLAS (see in Figure 3.4) is a clear indicator of its capital importance in the final performance of the new routines. Many internal particularities of the hardware are directly related to this differences in performance, and they have a visible impact with the modification of the block size (internal memory organization, cache and TLB misses, memory alignment issues, etc.). From the programmer perspective, the ability to easily and mechanically evaluate the effect of the block size from a high-level point of view implies the possibility of extracting conclusions before the low-level programming effort begins. The insights extracted for the optimal block size

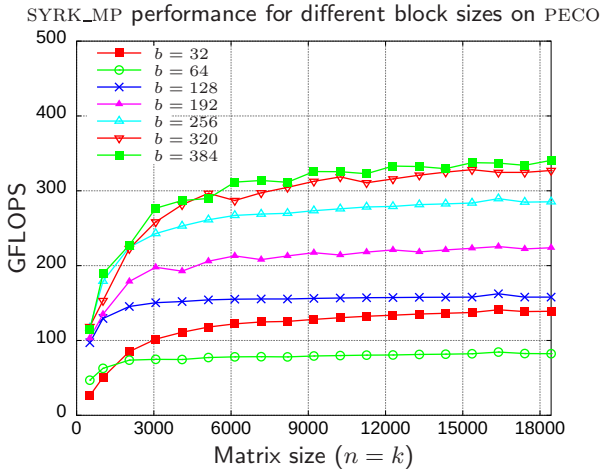


Figure 3.12: Performance of the new, tuned SYRK_MP implementation for multiple block sizes.

from this high-level approach are valid and can be applied (if desired) to low-level programming, driving to a faster code optimization.

As an illustrative example, Figure 3.12 shows the differences in performance for the *panel-matrix* variant of the SYRK routine using different block sizes. There are significant differences in the performance of the implementation depending on the block size. In general, the GFLOPS rates increase with the size of the block. This improvement is more important when using small blocks and is reduced as the tested block size is close to the optimal one. Figure 3.12 only reports performances for block sizes up to the optimal one; for larger block sizes, the performance rates are usually sustained and the performance does not improve. Observe in the figure the important performance difference depending on the selected block size. While 80 GFLOPS are attained for a block size $b = 32$, increasing this value to $b = 384$ boosts efficiency up to 320 GFLOPS. This speedup ($4\times$), obtained just by varying the block size, gives an idea of the importance of a thorough evaluation of this parameter when operating with blocked algorithms.

A detailed study of the impact of the block size has been carried out for each implementation. The performance results shown for every routine include the performance for the optimal block size from a whole set of block sizes.

3.4.2. Performance results for different algorithmic variants

The way in which calculations are organized in a given variant also plays a fundamental role in the final performance of the BLAS implementation. Once again, from the programmer perspective, having a family of algorithmic variants, and systematically evaluating these variants for each specific routine usually guides to a highly tuned implementation. Moreover, this methodology offers a way to guide the low-level programming effort to the appropriate parameters by performing a high-level previous evaluation of them.

Note that each implementation of the algorithmic variants for the BLAS routines is based on an initial partitioning and successive repartitioning processes. These partitionings basically translate into movements of pointers in the final codes, so no significant overhead is expected from them. For each iteration, the NVIDIA CUBLAS routines indicated in the FLAME algorithms are invoked, depending on the corresponding routine and algorithmic variants; these routines operate with the sub-blocks of the input and output matrices determined by the former partitioning processes.

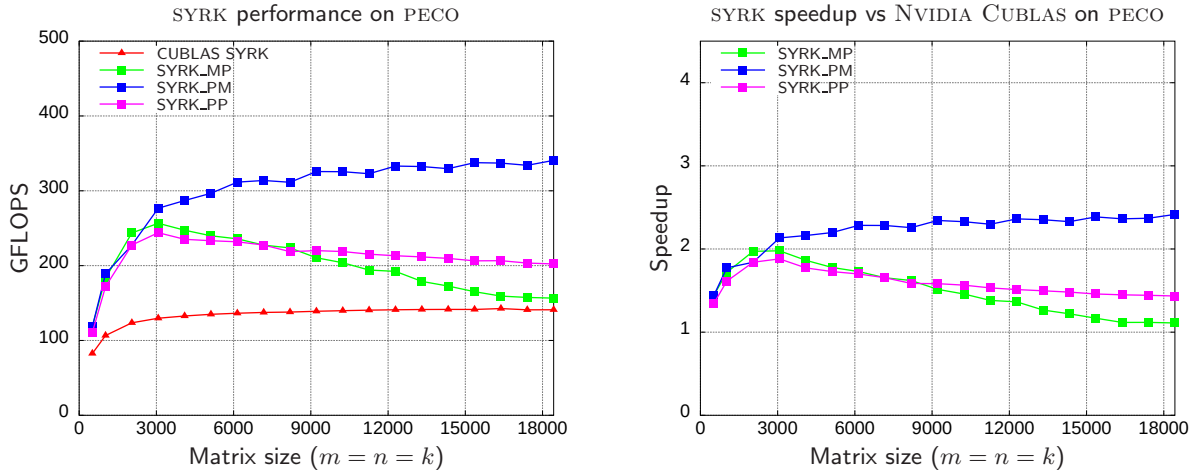


Figure 3.13: Performance (left) and speedup (right) of the three variants for the SYRK implementation.

For each evaluated routine, we compare the corresponding NVIDIA CUBLAS implementation and each new algorithmic variant. The speedup of the new codes is also shown, taking the NVIDIA CUBLAS performance of the routine as the reference for the comparison.

Figures 3.13 to 3.16 report the performance results for the evaluated BLAS routines and cases, and compare the performance of our new implementations with those from NVIDIA CUBLAS. The plots in the left of those figures show raw performances in terms of GFLOPS. The plots in the right illustrate the speedups attained by the new implementation of the routines taking the NVIDIA CUBLAS performance as the base for the calculation. As the block size is a critical factor in this type of blocked algorithms, a full evaluation of the performance of the routines has been carried out. Performance results only illustrate the performance attained with the optimal block size.

The performance behavior of the new implementations for the BLAS SYRK routine shown in Figure 3.13 is representative of the benefits that are expected from the new implementations. The advantages of our new GEMM-based implementations are demonstrated by observing the performances of the *matrix-panel* and the *panel-matrix* variants (especially the latter) that delivers a speedup near 2.5× for the largest matrices tested. This speedup is similar for smaller matrices. The performance of the *matrix-panel* variant is optimal for smaller matrices, but yields poorer performances for the largest matrices tested. This decreasing performance agrees with the behavior of the matrix-matrix multiplication shown in the GEMM evaluation (see Figure 3.3 for the case $C := C + AB^T$, in which this variant of the SYRK operation is based). Similar conclusions and performance results for the SYR2K and the TRMM routines are shown in Figure 3.15. The cases analyzed for each routine are the same as those used for the evaluation of NVIDIA CUBLAS in Section 3.2.1.

In general, for each new implementation there is at least one algorithmic variant that improves significantly the performance of the original NVIDIA CUBLAS implementations. The systematic evaluation of several algorithmic variants, and a detailed study of other critical parameters, such as block sizes, offer a set of conclusions related to the optimal algorithm and parameters for each specific routine, and drive to remarkable speedups for all BLAS routines compared with NVIDIA CUBLAS. As an example, the maximum speedup is 4× for SYMM, and the performance of the new implementations is higher than 300 GFLOPS for all implemented routines.

Further optimizations are possible from the basic algorithmic variants. For example, the performance of the SYMM implementations suffers from the poor performance of the NVIDIA CUBLAS

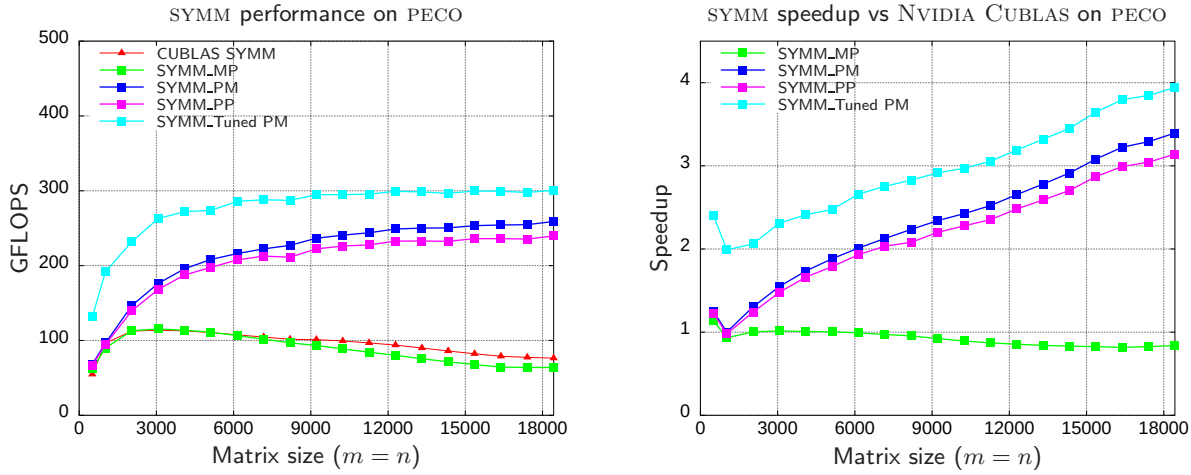


Figure 3.14: Performance (left) and speedup (right) of the three variants for the SYMM implementation.

SYMM implementation in all its algorithmic variants (see the performances of the basic algorithmic variants in Figure 3.14). Our basic SYMM implementation attains near 250 GFLOPS for the best variant, while the speed of the rest of our tuned implementations varies between 300 and 350 GFLOPS. Taking as reference the *panel-matrix* variant shown in Figure A.1, it is possible to observe that the operation involving a diagonal block of the input matrix A (block A_{11} in the algorithm) is the only one cast in terms of NVIDIA CUBLAS SYMM, which becomes the key bottleneck for the performance of the implementation.

A possible approach to this problem is to use an additional workspace to store the diagonal block of A in its full form, instead of storing only the lower triangular part. To do so, the algorithm has to explicitly copy the contents from the lower triangular part of the block into the upper triangular part, creating a symmetric matrix. Proceeding in this way, the operation $C_1 := C_1 + A_{11}B_1$, with A_{11} a block considered symmetric and stored in compact scheme, is symmetrized into the operation $C_1 := C_1 + \hat{A}_{11}B_1$, being $\hat{A}_{11}$ exactly the same matrix as A_{11}, but storing explicitly all the values in GPU memory. Thus, this operation can be cast exclusively in terms of a GEMM instead of using SYMM to perform it, potentially boosting the performance. The performance attained by this optimization based con the *panel-matrix* variant is also shown in Figure 3.14; in essence, proceeding this way a blocked general matrix-matrix multiplication is finally performed, introducing a small penalty related to the allocation of the workspace and the transformation of the compact symmetric matrix into canonical storage. This explains the difference in performance of this implementation compared with the performance of the NVIDIA CUBLAS GEMM implementation shown in Figure 3.3. Similar techniques can be applied to the rest of the routines involving symmetric matrices (SYRK and SYR2K).

One of the benefits of the GEMM-based approach is the direct impact of the inner-kernel (in this case, the GEMM routine) in the final performance of the rest of the BLAS implementations. For example, consider the performance of the NVIDIA CUBLAS implementation for the GEMM operation when matrix B is transposed; see Figure 3.16. As can be observed, the NVIDIA CUBLAS implementation performs particularly poor for large matrix sizes. We have developed blocked algorithms for this operation following the algorithms outlined in Figure 3.8, identifying a variant (the *panel-matrix* one) that performs particularly well, attaining a speedup 2.5× compared to the basic NVIDIA CUBLAS implementation for matrices of large dimension.

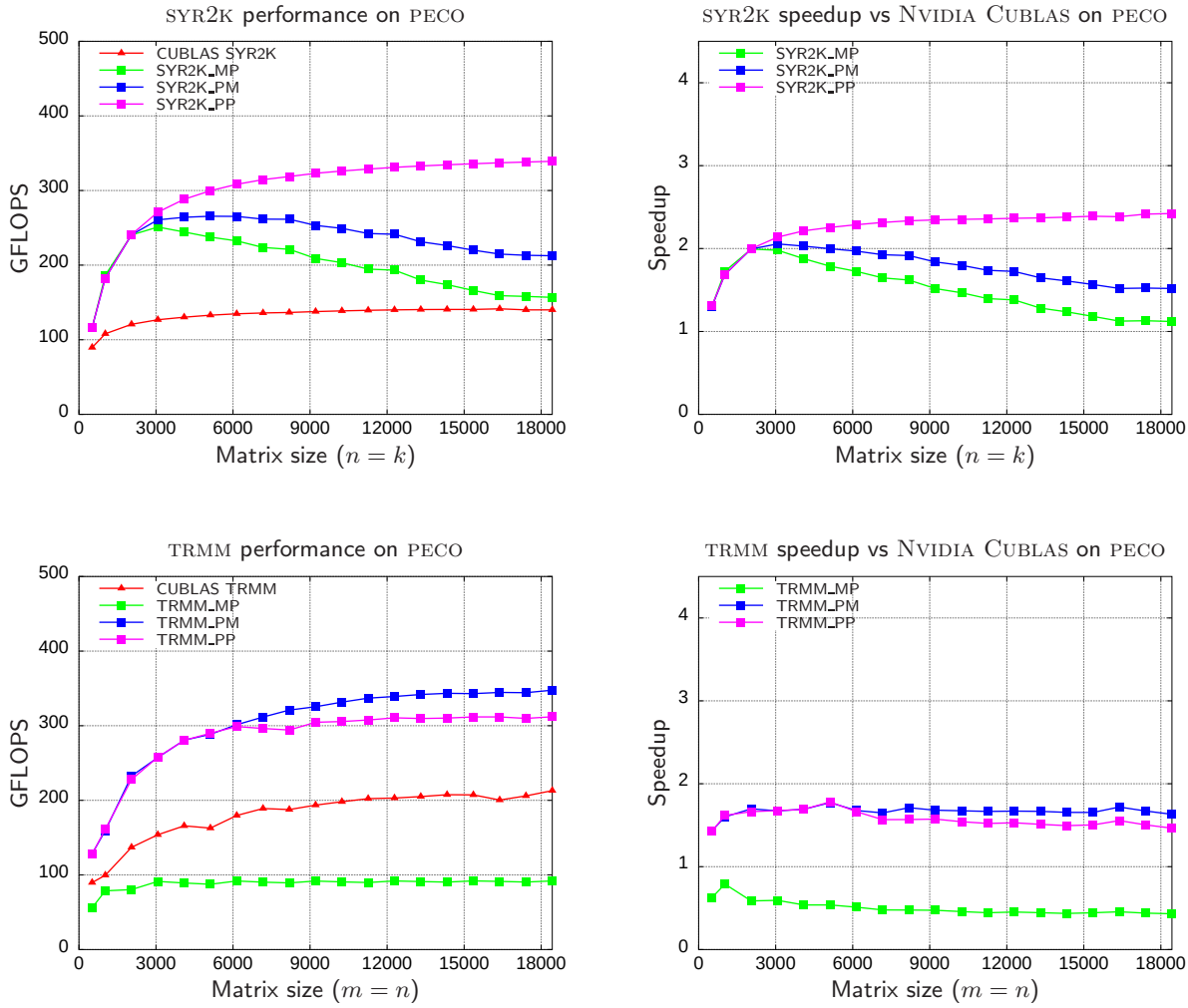


Figure 3.15: Top: performance (left) and speedup (right) of the three variants for the SYR2K implementation. Bottom: performance (left) and speedup (right) of the three variants for the TRMM implementation.

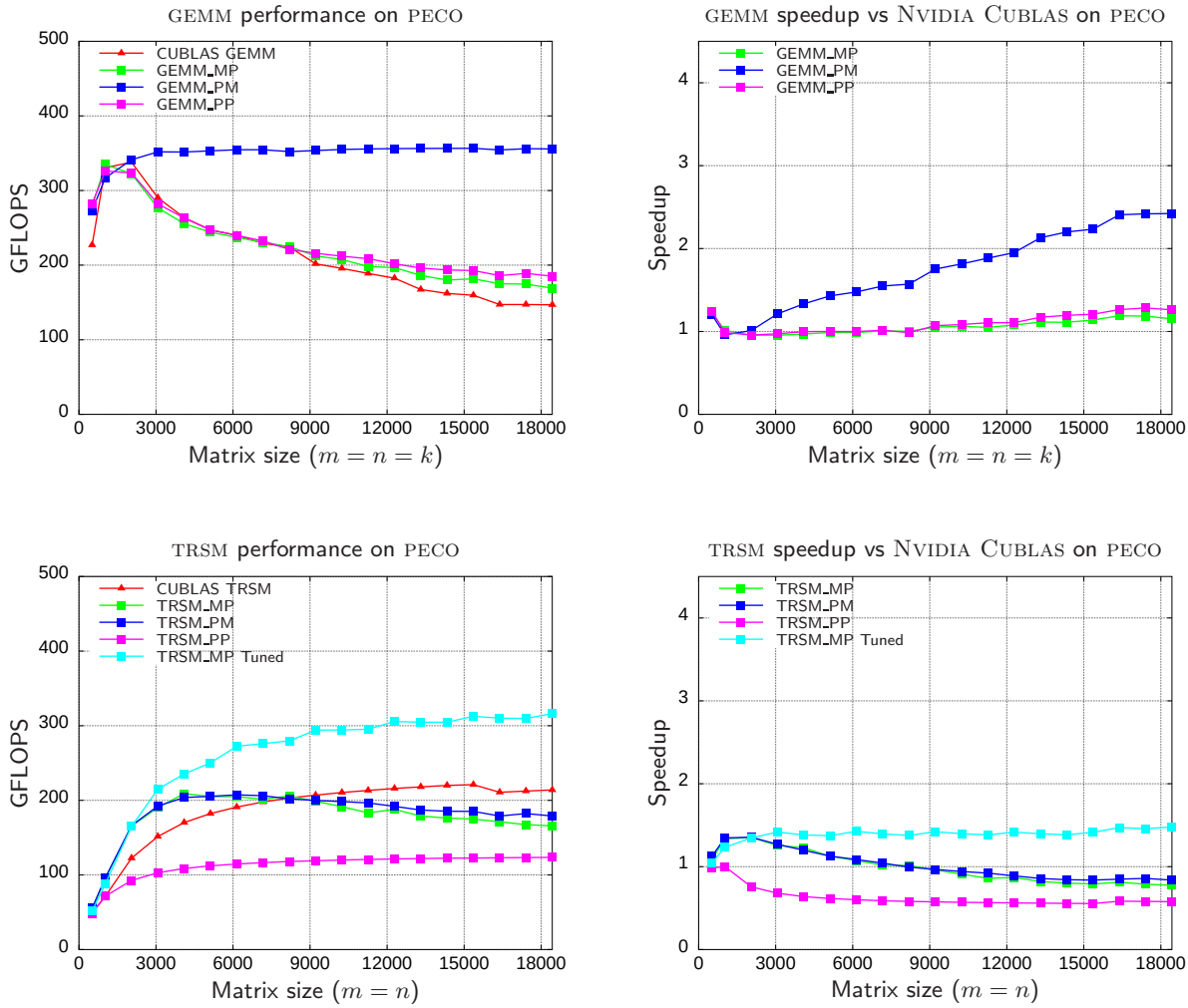


Figure 3.16: Top: Performance (left) and speedup (right) of the three variants for the GEMM implementation. Bottom: Performance (left) and speedup (right) of the three variants for the TRSM implementation.

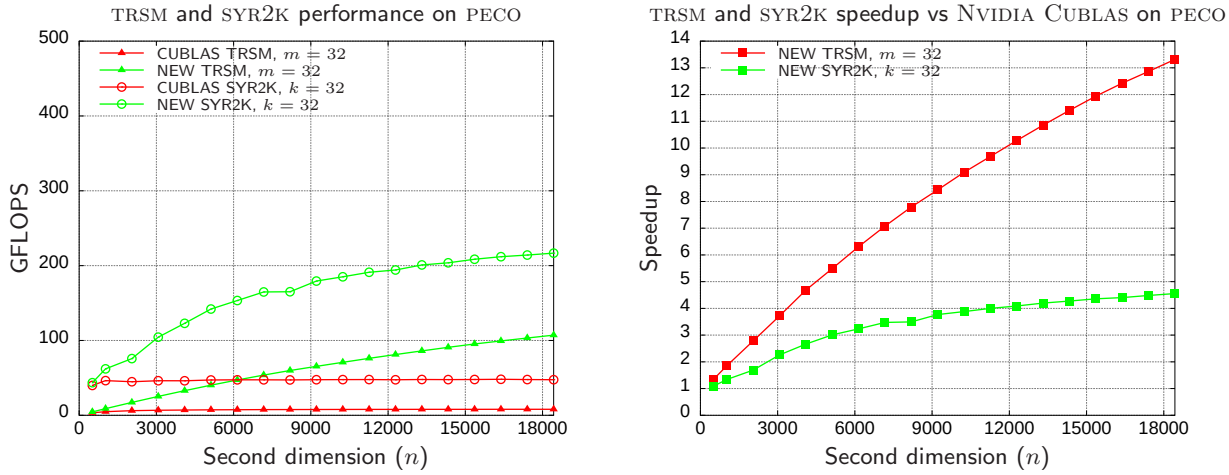


Figure 3.17: Performance of the tuned TRSM and SYR2K implementation on rectangular matrices (left) and speedup compared with NVIDIA CUBLAS (right).

This tuned case of the GEMM implementation has a direct application to routines such as the TRSM. The performance of the tuned TRSM implementations for the *matrix-panel* and *panel-matrix* variants performs particularly poor for large matrices, exactly as happened with the GEMM case exposed above. In fact, the behavior of TRSM when solving linear systems of the form $X = XA^T$ is dictated by the similar performance behavior of the GEMM when B is transposed, as this operation is the base of the blocked algorithms; see Figure A.4. Thus, the improvement of the performance of the general matrix-matrix multiplication shown in Figure 3.16 can be directly applied to the TRSM algorithms.

The performance and speedups attained by the blocked TRSM based on the optimized GEMM are shown in Figure 3.16. Note how, while the three basic algorithmic variants for the new TRSM implementation attain relatively poor performance results (even worse than the implementation from NVIDIA CUBLAS for large matrices), the reutilization of the tuned implementation of the GEMM routine provides a way to boost the performance of the original implementation from NVIDIA CUBLAS, attaining a speedup 1.5× for the largest problem dimension.

3.4.3. Performance results for rectangular matrices

BLAS-3 operations involving rectangular matrices are common in blocked implementation of many linear-algebra routines [144]. Figure 3.17 reports the performance results for two different BLAS-3 routines (TRSM and SYR2K) considering that their corresponding general operands are rectangular. The specific case for both routines is the same as the one illustrated throughout the rest of the evaluation. In the case of TRSM, the performance for the panel-matrix variant is shown; while in the case of SYR2K, the panel-panel variant is reported. These variants offered the highest performance results compared with the rest of the algorithmic variants. The tested routines operate with rectangular matrices with 32 columns (parameter n in TRSM, k in SYR2K); thus, the x axis in the figure indicates the number of rows of the result (matrix X for TRSM, C for SYR2K).

The performance and speedups attained for our new implementations compared with those in NVIDIA CUBLAS inherit the improvements observed in the evaluation of the routines for square matrices. For TRSM, the improvement is even higher that that for that case: our implementation attains 107 GFLOPS for the largest matrices (n = 18,432), while NVIDIA CUBLAS attains only 8 GFLOPS. The highest speedup in this case is almost 13× compared with the NVIDIA CUBLAS im-

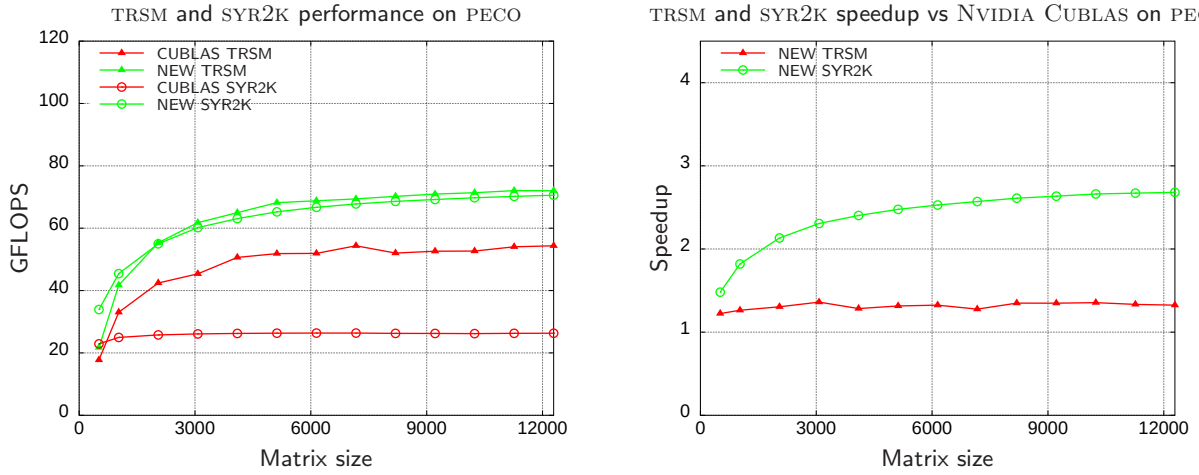


Figure 3.18: Performance of the tuned TRSM and SYR2K implementation on double precision matrices (left) and speedup compared with NVIDIA CUBLAS (right).

plementation. In the case of SYR2K, although the raw performance achieved by our implementation is higher (220 GFLOPS), the speedup obtained compared with the corresponding implementation in NVIDIA CUBLAS is 4.5×. In any case, these speedups are much higher than those attained for square matrices in previous experiments and demonstrate that the new implementations are also competitive compared with NVIDIA CUBLAS when the operands are rectangular.

3.4.4. Performance results for double precision data

The techniques presented to improve the performance employed in BLAS-3 routines in NVIDIA CUBLAS are independent from the arithmetic precision of the operations, provided the GEMM performance in NVIDIA CUBLAS delivers high performance in both simple and double precision. Similar qualitative results have been observed for all the operations, with our new implementations delivering remarkably better performances than those in NVIDIA CUBLAS.

Figure 3.18 compares the double-precision implementation of TRSM and SYR2K in NVIDIA CUBLAS and our new routines developed in this thesis. Only the optimal variants are presented, as the qualitative differences between different algorithmic variants are similar to those observed for single precision. The results correspond to square matrices.

As in the single-precision evaluations, the tuned implementations outperform those in NVIDIA CUBLAS for double precision for all the reported routines and cases. The accelerations are similar to those observed for single precision (see Figures 3.15 and 3.16), achieving a speedup 1.3× for TRSM and 2.7× for SYR2K.

Note, however, how the raw performance delivered by this particular GPU model when the operations are performed using double precision is not so competitive with those obtained for single precision. In fact, the 310 GFLOPS attained for the tuned single-precision TRSM implementation (see Figure 3.16) are reduced to only 73 GFLOPS for double-precision data.

The GEMM implementation was illustrative of the difficulties of the GPU to attain performances near its theoretical peak on single-precision arithmetic. In the results reported in Figure 3.2, an efficiency of only 40% was attained for this operation. As the number of double-precision units per GPU is quite reduced, the impact of the memory wall that was responsible of the poor efficiency in single-precision implementations is reduced, and thus the efficiency of the algorithms is dramatically

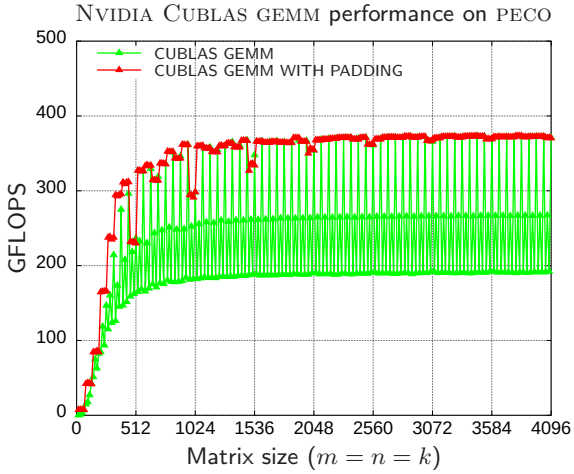


Figure 3.19: Detailed performance of the GEMM routine of NVIDIA CUBLAS.

increased. In the results reported in Figure 3.18, the attained performance represents a 93% of the peak performance of the architecture in double precision.

3.4.5. Padding

There is an additional factor with a relevant impact on the final performance of the NVIDIA CUBLAS implementation: the exact size of the operands. A fine-grained study of the performance of the GEMM implementation in NVIDIA CUBLAS reveals an irregular performance of the routine depending on the size of the input matrices. Figure 3.19 reports the performance of the GEMM implementation in NVIDIA CUBLAS for the operation $C := C + AB$ for small matrix dimensions, from $m = n = k = 8$ to $m = n = k = 4096$ in steps of 8. Comparing this evaluation with that performed in Figure 3.3, yields several rapid observations: First, better performance results are attained for matrix dimensions which are integer multiple of 32; optimal results are achieved for matrix dimensions that are an integer multiple of 64. Second, there exists a dramatic decrement in the performance of the GEMM routine for matrices that do not meet this restriction in their dimensions. This behavior is present in each implementation of the BLAS-3 routines in NVIDIA CUBLAS, not only for GEMM.

This irregular behavior is caused by two factors intimately related to the architectural details of the graphics processors with CUDA architecture. First, the *warp* size (or minimum scheduling unit) of modern GPUs (32 threads in the tested unit) is a critical factor. Depending on the size of the matrices to be multiplied, some threads for each *warp* may remain idle, and thus penalize the global performance of the implementation. Second, there exist some memory access restrictions (*data coalescing*) to attain the peak bandwidth between the graphics processor and GPU memory, reducing the number of memory transactions. This type of restrictions and optimizations are also related with the specific size of the data structures to be transferred to/from the graphics processor.

To avoid the sources of irregular behavior of the GEMM implementation (and equally to other BLAS-3 routines), NVIDIA suggests some common optimization practices [2]. However, as NVIDIA CUBLAS is a closed-source library, it is difficult or even impossible to apply those techniques to the existing NVIDIA CUBLAS codes. However, it is possible to avoid the irregular performance of the GEMM and other BLAS routines by applying a simple technique. Our proposal for GEMM is to *pad* with zeros the input matrices, transforming their dimensions into the next multiple of 64. With this transformation, we introduce a small overhead in the computation of the kernel, negligible for

large matrices, as the dimensions are increased in 63 columns/rows at most. The implementation creates and sets to zeros a padded matrix in GPU memory for each operand matrix, and then transfers the data from main memory to the correct position in GPU memory.

As a result of the application of this technique, the performance attained by the kernel with padding is uniform for all matrix sizes, hiding the irregular performance of original NVIDIA CUBLAS implementation. There is some overhead associated with the cost of the padding process and the non-contiguous store of the data in GPU memory during the transference of the matrices; however, its effect over the whole process is small, and the improvement when operating with dimensions that are not multiple of 64 greatly pays off, as can be observed in Figure 3.19.

3.5. Conclusions

Despite the amount of resources and efforts that NVIDIA has invested in the development of NVIDIA CUBLAS, this library offers an irregular performance depending on the specific routine being used. The performance of each routine implementation is very heterogeneous, depending not only in the specific operation that is being executed, but also on the shape of the operands and their size.

The main contributions of this chapter include a detailed evaluation of each BLAS routine in NVIDIA CUBLAS, and a collection of new highly tuned implementations. The optimization techniques used follow a high-level approach, with three main benefits:

- Better **programmability**, as no low-level optimizations are necessary, and the generation of new, high-performance codes is straightforward.
- Easy **code portability**, to reuse the developed algorithms and take benefit from further optimizations of the underlying building blocks in the same or a different architecture.
- Higher **performance**, by taking benefit of high-performance inner kernels and a efficient way of organizing the calculations.

The systematic derivation of several *algorithmic variants* and *blocked,* GEMM*-based* implementations allow the developer to gain insights that can be directly applied to lower-level codes (e.g. CUDA codes), with aspects that are critical to the performance of the optimized routines, such as the optimal block sizes, algorithmic variants or the transformation of the codes to achieve regular performances independently of the size of the operands (*padding*).

As a result, a full implementation of the basic routines in the Level-3 BLAS has been obtained, which delivers performances similar to those achieved by the highly tuned GEMM implementation, and speedups between 2.5× and 4× compared with their NVIDIA CUBLAS counterparts for square matrices, and $14x$ for rectangular matrices. The new codes have demonstrated their efficiency for both single and double precision, yielding similar speedups.

As the BLAS-3 are the basic building block for more complex algorithms, a detailed evaluation of their performance becomes necessary to understand the performance rates for those higher-level libraries; moreover, optimizations applied to the BLAS can be ported to higher-level codes to improve their performance.

CHAPTER 4

LAPACK-level routines on single-GPU architectures

The optimization of BLAS-3 routines on graphics processors naturally drives to a direct optimization of higher-level libraries based on them, such as LAPACK (*Linear Algebra PACKage*). However, given the complexity of the routines in this type of libraries, other strategies can be applied to further improve their performance.

In the case of GPU-based implementations, the optimizations applied to the BLAS routines in Chapter 3 can have a direct impact in LAPACK-level implementations, but we advocate for alternative strategies to gain insights and further improve the performance of those implementations using the GPU as an accelerating coprocessor.

In this chapter, we propose a set of improved GPU-based implementations for some representative and widely used LAPACK-level routines devoted to matrix decompositions. New implementations for the Cholesky and LU (with partial pivoting) decompositions, and the reduction to tridiagonal form are proposed and deeply evaluated.

In addition, a systematic evaluation of algorithmic variants similar to that presented in the previous chapter for the BLAS is performed for LAPACK-level routines, together with a set of techniques to boost performance. One of the most innovative technique introduced in this chapter is the view of the GPU as an accelerating co-processor, not only as an isolated functional unit as in the previous chapter. Thus, *hybrid*, collaborative approaches are proposed in which operations are performed in the most suitable architecture, depending on the particular characteristics of the task to be performed.

Single and double-precision results are presented for the new implementations and, as a novelty, a *mixed-precision iterative-refinement* approach for the solution of systems of linear equations is presented and validated. The goal of this technique is to exploit the higher performance delivered by modern graphics processors when operating in single-precision arithmetic, keeping at the same time full accuracy in the solution of the system.

As a result, a full family of implementations for widely used LAPACK-level routines is presented, which attains significant speedups compared with optimized, multi-threaded implementations on modern general-purpose multi-core processors.

The chapter is organized as follows. Section 4.1 surveys the nomenclature and the most important routines in the LAPACK library. Sections 4.2 and 4.3 introduce the theory underlying the Cholesky factorization, and the approaches and optimizations taken to implement it on the graphics

processor, respectively. Sections 4.4 and 4.5 overview the theory and algorithms necessary to compute the LU factorization with partial pivoting and its implementations on the GPU, respectively. Section 4.6 introduces the iterative refinement strategy for the solution of linear systems, reporting experimental results and validating it in the framework of GPU computing. Section 4.7 proposes a set of improvements to improve performance of the reduction to tridiagonal form routines in LAPACK using the SBR package. Finally, Section 4.8 reports the main contributions from the chapter.

All experiments presented through the chapter were carried out using a single GPU (Tesla C1060) on peco. The specific hardware and software details of the experimental setup were presented in Section 1.3.2.

4.1. LAPACK: Linear Algebra PACKage

The optimization of the routines in BLAS makes real sense if they are considered as the basic building block for libraries that perform more complex linear algebra operations. One of the most used libraries in this sense is LAPACK [108].

LAPACK offers routines to solve fundamental linear algebra problems, and contains the state-of-the-art in numerical methods. In the same direction as BLAS, LAPACK offers support for both dense and band matrices but, while BLAS is focused on the solution of basic linear algebra operations, LAPACK solves more complex problems, as, for example, linear equation systems, least square problems or eigenvalue and singular value problems.

LAPACK was created as a result of a project born at the end of the 80s. The main goal was to obtain a library with the same functionalities and improved performance than LINPACK [56] and EISPACK [107]. Those libraries, designed for vector processors, do not offer reasonable performance on current high performance processors, with segmented pipelines and complex memory hierarchies. This inefficiency is mainly due to the use of BLAS-1 routines, which does not exploit locality of reference, resulting in a sub-optimal usage of the memory subsystem. As a consequence, the offered routines devote most of the time to data movements from/to memory, with the subsequent penalty on the performance. For the same reasons that made necessary the development of higher levels of BLAS, the significant gap between the memory and processor speeds in current architectures required the redesign of existing libraries to optimize the number of memory accesses and even the access patterns [37].

The performance boost attained by the routines in LAPACK is due to two main reasons:

- The integration into the library of new algorithms that did not exist when older libraries were designed.
- The redesign of existing and new algorithms to make an efficient use of the BLAS library.

The legacy implementation of LAPACK is public domain [108], and includes drivers to test and time the routines. As with BLAS, some hardware manufacturers have implemented specific versions of LAPACK tuned for their architectures; however, the improvements introduced in these implementations are usually not as important as those introduced in BLAS. As the performance of many routine implementations relies on the performance of the underlying BLAS implementation, high-level modifications, such as the tuning of the block size, are the most common tweaks implemented in the proprietary versions.

For multi-core architectures, LAPACK extracts the parallelism by invoking a parallel version of BLAS. That is, the routines in LAPACK do not include any kind of explicit parallelism in their codes, but use a multi-threaded implementation of BLAS.

4.1.1. LAPACK and BLAS

The BLAS library offers a collection of efficient routines for basic linear algebra operations. For this reason those routines are used by LINPACK and LAPACK. The former internally uses BLAS-1, but as they operate on scalars and vectors, they do not allow a proper parallelization or the efficient use of the memory hierarchy. LAPACK uses routines from all BLAS levels, mostly from Level-3 BLAS, as:

- They yield high efficiencies on machines with hierarchical memory, not being limited by the slow data transfer rates between CPU and memory.
- They implement operations with more workload, being candidates for an efficient parallelization.

The possibility of using BLAS-3 routines led to a redesign of the algorithms implemented by LAPACK to work with blocks or sub-matrices [59, 55]. By using blocked-based algorithms, many operations can be cast in terms of the most efficient routines in BLAS, and the possibilities of parallelization are improved in two ways: the internal parallelization of individual block operations and the possibility of processing several blocks in parallel [51].

Although LAPACK contains BLAS-3 based routines, in some cases there exist also alternative implementations based on BLAS-1 and BLAS-2, that can be used in case non-blocked codes are needed. For example, the Cholesky factorization of a dense symmetric positive definite matrix is implemented as a blocked routine (named POTRF). At each iteration, this routine performs a Cholesky factorization of the current diagonal block, using an unblocked version of the Cholesky factorization routine (named POTF2).

The internal usage of BLAS offers other advantages:

- Portability and efficiency. There exist different BLAS implementations developed for several architectures, often with dramatically different characteristics. These specific implementations allow a migration of the applications without modifications in the code, attaining high performances at the same time. This efficiency is supposed to hold for future architectures, as hardware manufacturers are usually the first interested in delivering highly tuned BLAS implementations as soon as new products are released.
- Code legibility. The BLAS routines are well-known and their names clearly specify their functionality.
- Parallelism. Multi-threaded versions of BLAS allow a direct parallelization of the routines in LAPACK that internally invoke them.

In summary, the efficiency of LAPACK depends mainly on two factors: first, the efficiency of the underlying BLAS implementations; second, the amount of flops performed in terms of BLAS-3 routines.

4.1.2. Naming conventions

The name of each LAPACK routine describes its functionality. Similarly to BLAS, the names of LAPACK routines follow the format XYYZZZ, where:

- X: Indicates the data type involved in the routine: single-precision real data (S), double-precision real data (D), single-precision complex data (C), and double-precision complex data (Z).

- YY: Indicates the matrix type (or the type of the most significant matrix).
- ZZZ: Indicates the specific operation to be performed, e.g., QRF for the QR factorization.

4.1.3. Storage schemes and arguments

LAPACK is based on the internal usage of BLAS routines. Thus, the storage schemes and are the same as those defined for BLAS (see section 3.1.3), and the arguments defining matrices are very similar. For more information about these details, see [9].

4.1.4. LAPACK routines and organization

LAPACK offers three types of routines:

- *Driver routines*: Devoted to the solution of complex problems; e.g., linear systems, least square problems or eigenvalue problems.
- *Computational routines*: Perform basic operations, being invoked from the *drivers* to obtain partial results. Some examples include matrix factorizations or the solution of a linear system with an already factored matrix.
- *Utils*: Auxiliary routines for the *drivers* and the *computational routines*. These can be divided in:
  - Routines implementing low-level functionalities; e.g., scaling a matrix.
  - Routines implementing sub-tasks of the blocked algorithms; e.g., unblocked versions of the algorithms.
  - Support routines, as for example routines to query the optimal block size for the target machine.

4.1.5. Porting LAPACK-level routines to graphics processors

Due to the heavy use of LAPACK-level routines from many scientific and engineering applications, remarkable efforts have been performed in order to exploit the capabilities of these processors.

Preliminary works were performed using the former Cg/OpenGL frameworks for GPGPU with remarkable results [84, 87, 66]. This interest was renewed with the introduction of CUDA. In [142], optimized codes for common decompositions (Cholesky, LU and QR) are developed and presented as near-optimal for some given GPU models. Similar approaches following low-level coding efforts have been proposed in the last decade [132, 131, 140]. All of them pursue high-performance implementations on GPU, with minor contributions from the programmability perspective.

We advocate here for a high-level approach similar to that presented for the BLAS in Chapter 3. The benefits (ease of development, independence from the underlying architecture, code reuse,...) and key insights (evaluation of multiple algorithmic variants, thorough searches of optimal block sizes,...) are similar to those obtained for the BLAS. The main goal is to demonstrate how the FLAME high-level approach is perfectly valid to attain high performance on LAPACK-level operations using the GPU as an accelerating device.

4.2. Cholesky factorization

One of the most common strategies for the solution of linear equation systems commences with the factorization of the coefficient matrix, decomposing it as the product of two triangular matrices. These factors are used in a subsequent stage to obtain the solution of the original system by solving two triangular systems. The LU and the Cholesky factorization are decompositions of this type.

The LU factorization, combined with a partial row pivoting during the factorization process, is the common method to solve general linear systems of the type $Ax = b$. The Cholesky factorization plays the same role when the coefficient matrix A is symmetric and positive definite.

Definition A symmetric matrix $A \in \mathbb{R}^{n\times n}$ is positive definite if $x^T Ax > 0$ for all nonzero $x \in \mathbb{R}^n$.

The following definition establishes the necessary conditions for a matrix to be invertible, a necessary property for the calculation of both the Cholesky and the LU factorizations.

Definition A matrix $A \in \mathbb{R}^{n\times n}$ is invertible if all its columns are linearly independent and, thus, $Ax = 0$ if and only if $x = 0$ for all $x \in \mathbb{R}^n$.

The following theorem defines the Cholesky factorization of a symmetric definite positive matrix.

Theorem 4.2.1 *If $A \in \mathbb{R}^{n\times n}$ is symmetric positive definite, then there exists a unique lower triangular matrix $L \in \mathbb{R}^{n\times n}$ with positive diagonal entries such that $A = LL^T$.*

The proof for this theorem can be found in the literature [70]. The factorization $A = LL^T$ is known as the *Cholesky factorization*, and L is usually referred to as the *Cholesky factor* or the *Cholesky triangle* of A. Alternatively, A can be decomposed into an upper triangular matrix U, such that $A = UU^T$ with $U \in \mathbb{R}^{n\times n}$ upper triangular matrix.

Note how the Cholesky factorization is the first step towards the solution of a system of linear equations $Ax = b$ with A symmetric positive definite. The system can be solved by first computing the Cholesky factorization $A = LL^T$, then solving the lower triangular system $Ly = b$ and finally solving the upper triangular system $L^T x = y$. The factorization of the coefficient matrix involves the major part of the operations ($O(n^3)$ for the factorization compared with $O(n^2)$ for the system solutions), and thus its optimization can yield higher benefits in terms of performance gains.

The largest entries in a positive definite matrix A are on the diagonal (commonly referred to as a *weighty* diagonal). Thus, these matrices do not need a pivoting strategy in their factorization algorithms [70]. Systems with a positive definite coefficient matrix A constitute one of the most important cases of linear systems $Ax = b$.

The Cholesky factor of a symmetric positive definite matrix is unique. The Cholesky factorization is widely used in the solution of linear equation systems and least square problems. Although the Cholesky factorization can only be computed for symmetric positive definite matrices, it presents some appealing features, basically with respect to computational cost and storage requirements levels. These advantages make the Cholesky decomposition of special interest compared with other factorizations (e.g., the LU or QR factorizations).

4.2.1. Scalar algorithm for the Cholesky factorization

Consider the following partition of a symmetric definite positive coefficient matrix A, and its Cholesky factor L with lower triangular structure:

$$A = \left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right), \; L = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right),$$

with α_{11} and λ_{11} scalars, a_{21} and l_{21} vectors of $n-1$ elements, and A_{22} and L_{22} matrices of dimension $(n-1) \times (n-1)$ (the symbol $\star$ in the expression above denotes the symmetric part of A, not referenced through the derivation process). From $A = LL^T$, it follows that:

$$\left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right) \left(\begin{array}{c|c} \lambda_{11} & l_{21}^T \\ \hline 0 & L_{22}^T \end{array}\right) = \left(\begin{array}{c|c} \lambda_{11}^2 & \star \\ \hline \lambda_{11} l_{21} & l_{21} l_{21}^T + L_{22} L_{22}^T \end{array}\right),$$

so that

$$\begin{aligned} \lambda_{11} &= \sqrt{\alpha_{11}}, \\ l_{21} &= a_{21}/\alpha_{11}, \\ A_{22} - l_{21} l_{21}^T &= L_{22} L_{22}^T. \end{aligned}$$

Following these expressions, it is possible to obtain a recursive formulation of a scalar algorithm to calculate the Cholesky factorization, which rewrites the elements in the lower triangular part of A with those of the Cholesky factor L:

1. Partition $A = \left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right)$.
2. $\alpha_{11} := \lambda_{11} = \sqrt{\alpha_{11}}$.
3. $a_{21} := l_{21} = a_{21}/\lambda_{11}$.
4. $A_{22} := A_{22} - l_{21} l_{21}^T$.
5. Continue recursively with $A = A_{22}$ in step 1.

This algorithm is usually known as the *right-looking* algorithm for the Cholesky factorization because, once a new element of the diagonal of the Cholesky factor (λ_{11}) is calculated, the corresponding updates are applied exclusively to the elements below as well as to the right of the calculated element (that is, A_{21} and A_{22}).

With this algorithmic formulation, it is only necessary to store the entries of the lower triangular part of the symmetric matrix A. These elements are overwritten with the resulting factor L. Thus, in the operation $A_{22} := A_{22} - l_{21} l_{21}^T$, it is only necessary to update the lower triangular part of A_{22}, with a *symmetric rank-1 update*. Proceeding in this manner, the flops required by this algorithm is $n^3/3$.

The major part of the calculations in the algorithm are cast in terms of the symmetric rank-1, which performs $O(n^2)$ operations on $O(n^2)$ data. This ratio between memory accesses and effective calculations does not allow an efficient exploitation of the underlying memory hierarchy, and thus performance is dramatically limited by the memory access speed. In order to improve performance, this BLAS-2-based algorithm can be reformulated in terms of BLAS-3 operations (mainly matrix-matrix multiplications and rank-k updates). This is the goal of the blocked algorithm presented next.

4.2.2. Blocked algorithm for the Cholesky factorization

Deriving a blocked algorithm for the Cholesky factorization requires a different partitioning of matrices A and L. Specifically, let

$$A = \left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array}\right), \; L = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right),$$

where A_{11} and L_{11} are $b \times b$ matrices (with b, the block size, being usually much smaller than the dimension of A), and A_{22} and L_{22} matrices with $(n-b) \times (n-b)$ elements. From $A = LL^T$, we then have that:

$$\left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right) \left(\begin{array}{c|c} L_{11}^T & L_{21}^T \\ \hline 0 & L_{22}^T \end{array}\right) = \left(\begin{array}{c|c} L_{11}L_{11}^T & \star \\ \hline L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T \end{array}\right),$$

which yields the following expressions:

$$\begin{aligned} L_{11}L_{11}^T &= A_{11}, \\ L_{21} &= A_{21}L_{11}^{-T}, \\ A_{22} - L_{21}L_{21}^T &= L_{22}L_{22}^T. \end{aligned}$$

The blocked recursive algorithm for the calculation of the Cholesky factor of A, overwriting the corresponding elements of the lower triangular part of A, can be formulated as follows:

1. Partition $A = \left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array}\right)$, with A_{11} being a $b \times b$ sub-matrix.
2. $A_{11} := L_{11}$ such that $A_{11} = L_{11}L_{11}^T$ (that is, obtaining the Cholesky factor of A_{11}).
3. $A_{21} := L_{21} = A_{21}L_{11}^{-T}$.
4. $A_{22} := A_{22} - L_{21}L_{21}^{-T}$.
5. Continue recursively with $A = A_{22}$ with step 1.

This blocked algorithm is also a right-looking variant: after the factorization of the diagonal block A_{11}, only the blocks below and to its right are updated. Data only needs to be stored in the lower triangular part of A. Thus, the factorization of the diagonal block, $A_{11} := L_{11}$, and the update $A_{22} := A_{22} - L_{21}L_{21}^T$ only affect the lower triangular parts of the corresponding blocks. The cost of the blocked formulation of the Cholesky factorization is $n^3/3$ flops, provided $b \ll n$.

Blocked versions for the Cholesky factorization and other basic linear algebra routines are common in many software packages, such as LAPACK, and usually deliver high performance by invoking BLAS-3 kernels. In particular, in the blocked Cholesky factorization algorithm presented above, the calculation of the panel A_{21} and the update of the sub-matrix A_{22} can be cast exclusively in terms of BLAS-3 calls. In fact, the major part of the operations are carried out by the latter operation (a symmetric rank-b update); this operation requires $O(n^2b)$ operations on $O(n^2)$ data. Thus, in case b is significantly large, the blocked algorithm increases the ratio between operations and data accesses by a factor of b, and thus the data reutilization is improved. This is a critical factor to attain high performance on novel architectures, such as modern graphics processors, where blocked algorithms become absolutely necessary.

4.2.3. Algorithms in FLAME notation for the scalar and blocked Cholesky factorization

There exist different algorithmic variants for the Cholesky factorization. Figure 4.1 shows the three different algorithmic variants that can be derived for the Cholesky factorization of a symmetric

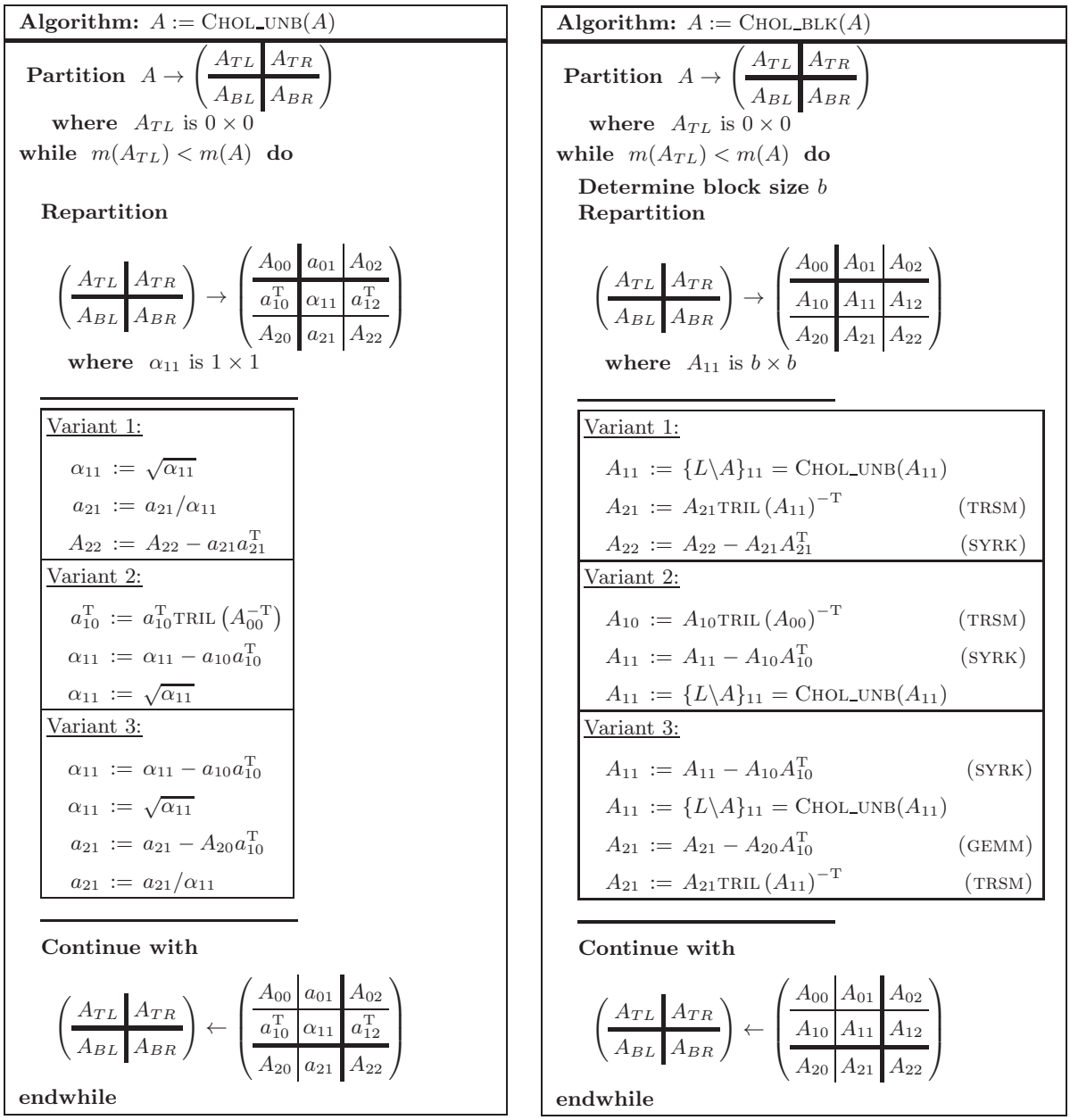


Figure 4.1: Algorithmic variants for the scalar (left) and blocked (right) algorithms for the Cholesky factorization.

positive definite matrix A; scalar variants are shown on the left of the figure, and blocked variants are shown on the right.

These variants have been obtained applying the FLAME methodology for the formal derivation of algorithms [138]. The lower triangular part of A is overwritten by the Cholesky factor L, without referencing the upper triangular part of A. In the blocked variants, the expression $A := \{L\backslash A\} = \text{CHOL\_UNB}(A)$ indicates that the lower triangle of block A is overwritten with its Cholesky factor L,

without referencing any entry of the upper part of A. In the blocked variants, the algorithm chosen to factorize the diagonal blocks can be the blocked algorithm itself (i.e., a recursive formulation) with a smaller block size, or any scalar algorithm. The latter option is the one illustrated in the figure.

The computational cost of all the algorithmic variants is the same ($n^3/3$ flops); the only difference are the specific BLAS kernels called from each algorithm, and the order in which they are invoked.

Consider for example the first variant of the blocked algorithm for the Cholesky factorization, and note how the algorithms illustrate, in an intuitive way, the operations that are carried out. Initially, A is partitioned into four different blocks, A_{TL}, A_{TR}, A_{BL} and A_{BR}, where the first three are empty (their sizes are 0×0, $0 \times n$ and $n \times 0$, respectively). Thus, at the beginning of the first iteration of the algorithm, A_{BR} contains the whole matrix A.

At each iteration of the algorithm, A_{BR} is repartitioned into four new blocks, namely A_{11}, A_{21}, A_{12} and A_{22}. The elements of the Cholesky factor L corresponding to the first two blocks (A_{11} and A_{21}) are calculated in the current iteration, and then A_{22} is updated. The operations that are carried out to compute the elements of the lower triangle of L_{11} and the whole contents of L_{21} with the corresponding elements of the Cholesky factor L constitute the main loop of the algorithm:

$$\begin{array}{lrcl} & A_{11} & := & \{L \backslash A\}_{11} = \textsc{Chol\_unb}(A_{11}) \\ (\textsc{trsm}) & A_{21} & := & A_{21} \textsc{tril}\,(A_{11})^{-T} \\ (\textsc{syrk}) & A_{22} & := & A_{22} - A_{21} A_{21}^T. \end{array}$$

To the right of the mathematical operations, we indicate the corresponding BLAS-3 routines that are invoked from the blocked algorithms in Figure 4.1. Function $\textsc{tril}\,(\cdot)$ returns the lower triangular part of the matrix passed as a parameter. Thus, the operation $A_{21} := A_{21} \textsc{tril}\,(A_{11})^{-T}$ stands for the solution of a triangular system (TRSM routine from BLAS).

Upon completion of the current iteration, the boundaries are shifted, restoring the initial 2×2 partitioning, so that A_{BR} is now composed by the elements that were part of A_{22} at the beginning of the iteration. Proceeding in this manner, in the following iteration the operations on the matrix are applied again on the new block A_{BR}.

The algorithm terminates when A_{BR} remains empty, that is, when the number of rows/columns of A_{TL} equals the number of rows/columns of the original matrix A. The main loop in the algorithm iterates while $m(A_{TL}) < m(A)$, where $m(\cdot)$ corresponds to the number of rows of a given matrix.

4.3. Computing the Cholesky factorization on the GPU

The FLAME algorithms presented in Figure 4.1 for the scalar and blocked Cholesky factorization capture the algorithmic essence of the operations to be implemented. This approach allows an easy translation of the algorithms into high-performance code written in traditional programming languages, like Fortran or C, by just manipulating the pointers to sub-matrices in the partitioning and repartitioning routines, and linking to high-performance parallel BLAS implementations. The FLAME project provides a specific API (FLAME/C) to facilitate the transition from FLAME algorithms to the corresponding C code. This approach hides the error-prone intricate indexing details and eliminates the major part of the debugging effort when programming linear algebra routines.

The main drawbacks of starting the development process of complex linear algebra libraries (such as LAPACK) on novel architectures from a low level approach are the same as for simpler libraries, such as BLAS. Many key parameters with a dramatic impact on the final performance

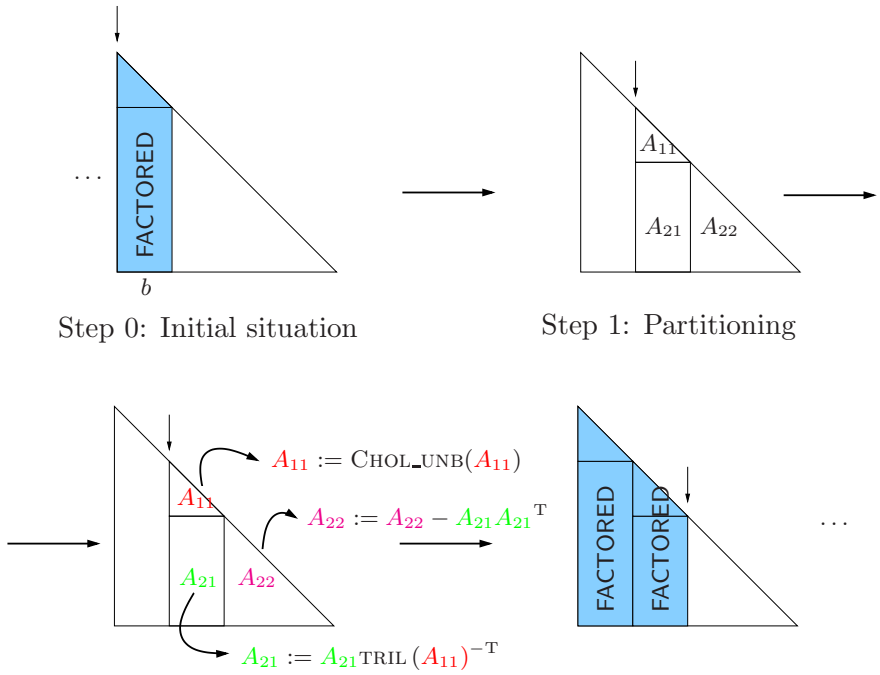


Figure 4.2: Diagram of the blocked Cholesky factorization algorithm, Variant 1.

(scalar and blocked implementations, block sizes on the latter, algorithmic variants, ...) can be evaluated more easily at a higher level of abstraction. These observations can be then applied to low level programming paradigms if desired, accelerating the development process of high performance native libraries.

In this section, a number of basic implementations for the scalar and blocked algorithmic variants for the Cholesky factorization presented in previous sections are exposed and their performance evaluated on modern multi-core and graphics processors. A systematic experimental analysis of the optimal algorithmic variants and block sizes is presented. From this basic implementations, successive optimizations, together with the benefits from the performance perspective are reported.

4.3.1. Basic implementations. Unblocked and blocked versions

The FLAME algorithms for the Cholesky factorization present three common stages at each iteration. First, an initial partitioning of the matrices. Second, invocations to the corresponding BLAS routines depending on the specific algorithmic variant. Third, a repartitioning process in preparation for the next iteration. Figure 4.2 illustrates the operations performed at each iteration of the blocked Cholesky algorithm, using Variant 1 from Figure 4.1.

Provided there exists an optimized BLAS implementation, parallel implementations of the Cholesky factorizations for general-purpose multi-core processors usually translate the first and third stages into movements of pointers to define the sub-matrices at each iteration, and the second stage into invocations to multi-threaded BLAS routines.

This approach can be also followed to use the GPU as the target architecture. In this case, the movements of pointers involve memory addresses in the GPU memory, and the invocations to BLAS routines use the NVIDIA CUBLAS implementation. As the NVIDIA CUBLAS routines operate exclusively with matrices previously allocated in the GPU memory, data has to be transferred

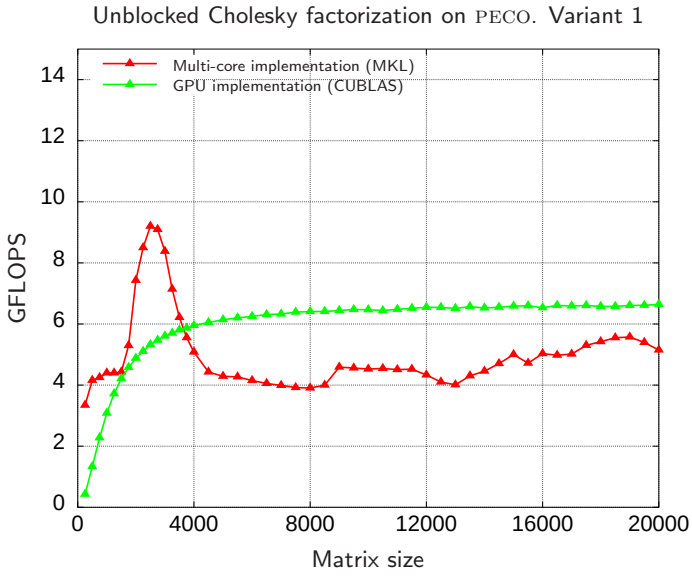


Figure 4.3: Performance of the single-precision scalar variant 1 of the Cholesky factorization on the multi-core processor and the graphics processor on PECO.

before the invocation to the CUBLAS kernels. In our implementations, the whole matrix to be decomposed is initially transferred to the GPU memory. The decomposition is then performed exclusively on the GPU; and the resulting factor is transferred back to main memory.

In order to provide initial implementations of the scalar and blocked algorithms for the Cholesky factorization, the equivalent codes for each algorithmic variant shown in Figure 4.1 have been developed on multi-core and graphics processors.

The scalar versions of the algorithms for the Cholesky algorithmic variants shown in Figure 4.1 imply the usage of BLAS-1 and BLAS-2 routines operating with vectors, and thus they usually deliver a poor performance on recent general-purpose architectures, with sophisticated cache hierarchies. In these type of operations, the access to memory becomes a real bottleneck in the final performance of the implementation. For the scalar algorithms, it is usual that the implementations on multi-cores attain better performance when the matrices to be factorized are small enough to fit in cache memory, reducing the performance as this size is increased and accesses to lower levels in the memory hierarchy are necessary.

To evaluate this impact on multi-core and graphics processors, the first experiment consists on the evaluation of the scalar algorithms presented for the Cholesky factorization. Figure 4.3 reports the performance of the Variant 1 of the scalar algorithm for the Cholesky factorization on a multi-core processor using a tuned multi-threaded BLAS implementation and its equivalent implementation on the GPU using NVIDIA CUBLAS. In the implementation using the graphics processor, only the square root of the diagonal element is performed on the CPU, transferring previously the diagonal element to main memory, and back to GPU memory when the update is done. Experimental evidences [23] revealed the benefits of using the CPU to perform this operation, even with the penalty introduced by the transfer of the scalar value between memory spaces. Performance results are given in terms of GFLOPS for increasing matrix sizes; hereafter, transfer times are not included in the performance results. We have chosen this algorithmic variant as the most illustrative, as the remaining variants offered worse performance results, but a similar qualitative behavior.

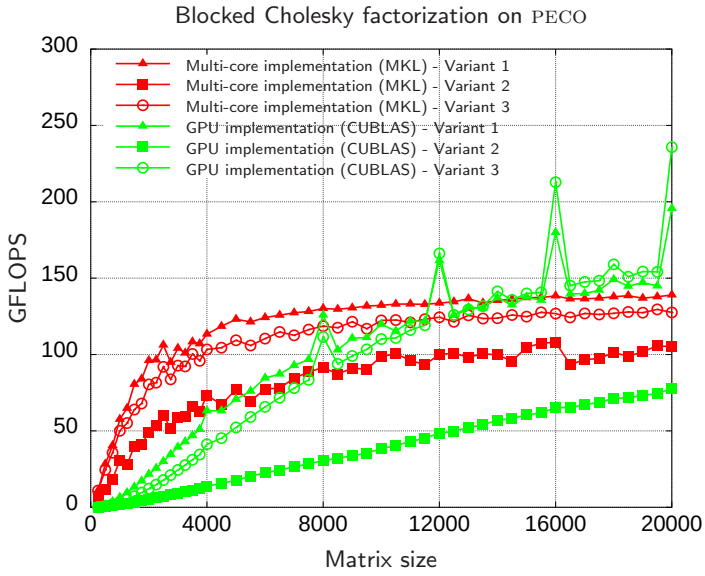


Figure 4.4: Performance of the single-precision blocked algorithmic variants of the Cholesky factorization on the multi-core processor and the graphics processor on PECO.

The performance rates attained by the scalar implementations, is far from the peak performance of the architecture for both the CPU and the GPU, and also very different from the performances attained for the BLAS-3 routines studied in Chapter 3. More specifically, the performances obtained in this case are 9.2 GFLOPS and 6.5 GFLOPS for the CPU and the GPU, respectively. These results represent a small percentage of the peak performance of the architectures. Considering the performance of the GEMM implementation as a realistic peak performance of both architectures (see Figure 3.1), the results obtained for the scalar Cholesky implementations are roughly a 7% of that maximum GFLOPS rate on the CPU and a 2% for the GPU.

As these kernels are part of the blocked variants (the factorization of the diagonal blocks is usually performed by invoking scalar variants of the factorization routines), it is important to analyze its performance. The cache memories in the general-purpose processors play a critical role for BLAS-1 and BLAS-2 operations that usually appear in the scalar variants of the codes. More specifically, they have a direct impact on the performance of the scalar routines for small matrix sizes (up to $n = 2{,}500$ in the unblocked Cholesky factorization, for example). For these matrix sizes, the use of the CPU offers important speedups compared with the equivalent implementation on the GPU. For larger matrices, however, the GPU (without a sophisticated memory hierarchy) attains higher performance ratios.

These performance results lead to the conclusion that a blocked version of the algorithms becomes necessary to attain high performance, not only on multi-core processors, but also on modern graphics processors, especially if large matrices are involved in the factorizations. For relatively small matrices, however, the usage of the CPU can be still an appealing option.

The performance of the blocked algorithms for the Cholesky factorization is shown in Figure 4.4. The experimental setup is the same as that used for the evaluation of the scalar algorithms. A full study of the block sizes has been carried out, reporting only the results corresponding to the optimal one in the figures. The figure reports the performance results for the three blocked algorithmic variants, both on the multi-core processor and the graphics processor. In the latter case, the diagonal block is factorized using the corresponding variant of the scalar algorithm.

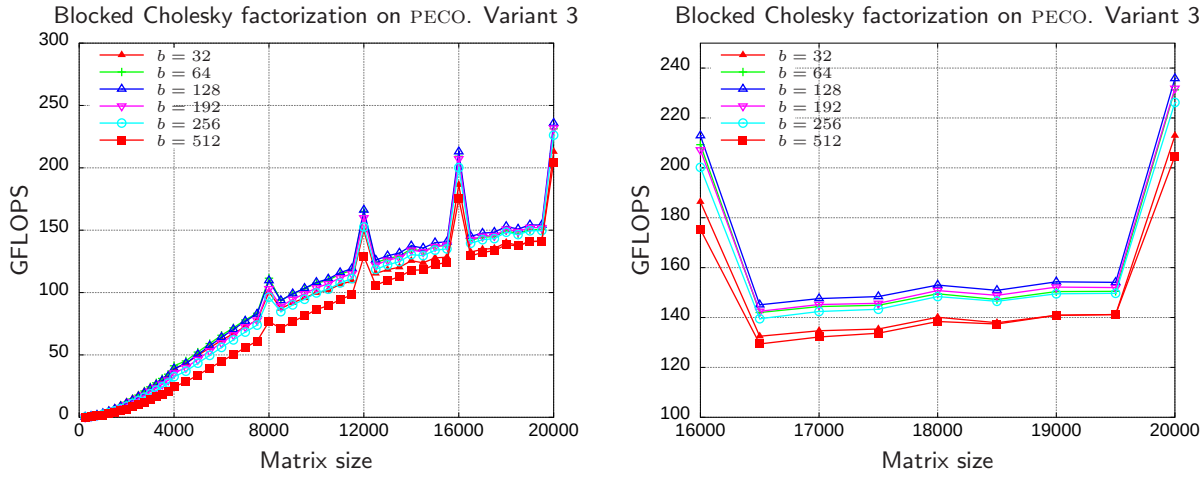


Figure 4.5: Impact of the block size on the single-precision Cholesky factorization using the graphics processor on PECO.

The attained performance for the blocked algorithms, independently from the target processor, is illustrative of the benefits expected for these type of implementations. These rates are closer to those attained for the BLAS-3 routines reported in Chapter 3. In this case, the peak performance obtained for the Cholesky factorization (using Variant 3) is 235 GFLOPS, which means a 65% of the performance achieved by the GEMM implementation in NVIDIA CUBLAS.

A deeper analysis of the performance rates involves a comparison between scalar and blocked implementations, a study of the impact of the block size and algorithmic variants on the final performance results on blocked algorithms and a comparison between single and double precision capabilities of the tested architectures. This analysis is presented progressively and motivates the improvements introduced in the following sections.

Impact of the block size

In the blocked Cholesky implementations, besides the specific algorithmic variant chosen, the block size (b) plays a fundamental role in the final performance of the implementation. In the GPU implementations, the block size ultimately determines the dimensions of the sub-matrices involved in the intermediate steps of the decomposition. As shown in Chapter 3, the implementations of the BLAS on GPUs deliver very different performance depending on the size of the operands. Thus, the operands form and size will ultimately determine the final performance of the Cholesky algorithms that make use of them.

Figure 4.5 shows GFLOPS rate of the first algorithmic variant for the blocked Cholesky implementation on the GPU. In the plots, we show the performance attained by the implementation with block size varying from $b = 32$ to $b = 512$. The right-hand side plot is a detailed view of the same experiment for a smaller set of matrix dimensions.

Taking as a driving example a coefficient matrix of dimension $n = 20{,}000$, performance grows as the block size is increased, reaching an optimal point for $b = 128$. From this value on, performance decreases, attaining the worst value for the largest block size tested ($b = 512$). The difference in performance between both extremes is significant: 204 GFLOPS for $b = 512$ and 237 GFLOPS for $b = 128$. This performance difference (an improvement of 16% between both values of b) gives an idea of the importance of a thorough and systematic evaluation of the block size to find the optimal one, and is a major contribution of our evaluation methodology. Similar behavior has been

observed in the rest of the algorithmic variants and matrix dimensions. All performance results shown in the rest of the evaluation only show the results for the optimal block size.

Scalar implementations vs. blocked implementations

As previously stated, blocked algorithms are the most convenient approach to the development of dense linear algebra routines such as the Cholesky factorization on modern multi- and many-core architectures.

In the target architectures, the scalar implementation of the algorithmic Variant 1 attains 6.5 GFLOPS on the graphics processor and 5.1 GFLOPS on the multi-core processor for the largest tested matrices: see Figure 4.3. In the case of the GPU, this is the best result from the whole set of tested dimensions. On the other hand, in the multi-core processor the role played by cache memory reveals for small matrices, attaining 9.2 GFLOPS for matrices of dimension $n = 2{,}500$. The performance of the scalar implementation on the CPU drops rapidly from this dimension on, attaining similar performance results than those attained on the GPU.

On the other hand, taking as a reference the best algorithmic variant for the blocked implementation on each processor, the differences with respect to the performance attained using the scalar implementations are remarkable, see Figure 4.4. The performance achieved on the GPU is 235 GFLOPS for the largest tested matrix, and 141 GFLOPS in the case of the CPU. The speedup obtained in this basic implementations (1.7× comparing the best blocked algorithmic variant on the CPU and on the GPU) is the base for further optimizations. In this case, the best performance results are attained for the largest tested matrices.

From the comparison between the scalar and the blocked implementations, the first conclusion to be extracted is the convenience of using blocked algorithms for the Cholesky factorization not only on multi-core processors, but also on modern graphics processors. In the latter architecture, the usage of BLAS-3 routines involves operations with higher arithmetic intensity, for which the GPU unleashes its true potential [82].

Performance of the different algorithmic variants

The results shown in Figure 4.4 show the capital importance of choosing the appropriate algorithmic variant in the computation of the Cholesky factorization. The specific underlying BLAS calls invoked in each variant (and shown in the algorithms of the Figures 4.1), as well as the specific shape of the involved sub-matrices depends on the algorithmic variant, determine the performance of the corresponding variant.

More specifically, the performance of the GPU implementations vary between 235 GFLOPS for Variant 3 to 77 GFLOPS for Variant 2. The performance attained for Variant 1 is similar to that attained for Variant 3. On the CPU, the attained performance rates are also greatly affected by the specific algorithmic variants, ranging from 105 GFLOPS to 141 GFLOPS for the worst and best variant, respectively.

Therefore, a systematic derivation of all the algorithmic variants for a given dense linear algebra operation (in this case, the Cholesky factorization) becomes absolutely necessary to find out the optimal one. The use of the high-level approach followed in our methodology eases this systematic derivation process. Moreover, the optimal variants on the GPU differ from those on the CPU, and are likely to be also different for other GPU models, BLAS implementation or accelerator type. This justifies the detailed development and evaluation of several algorithmic variants, together with its careful evaluation. The adopted high-level approach allows the library developer to extract critical

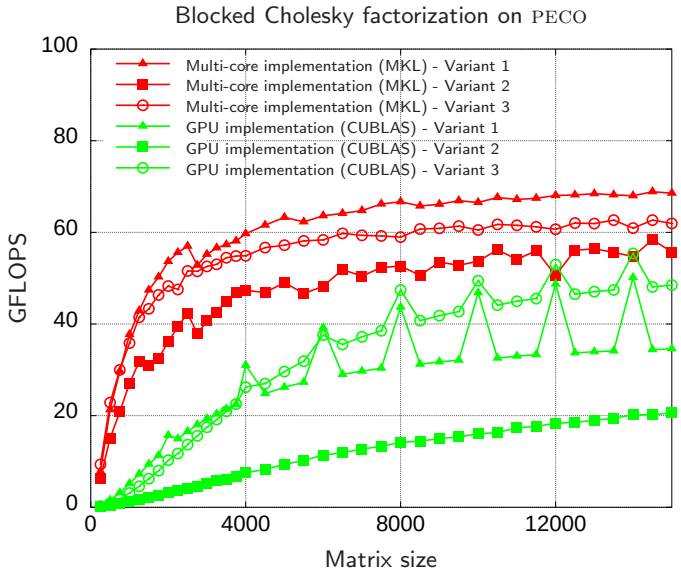


Figure 4.6: Performance of the double-precision blocked algorithmic variants of the Cholesky factorization on the multi-core processor and the graphics processor on PECO.

insights about the final performance of the implementations without having to develop low-level codes.

Single/double precision capabilities of the GPU

Figure 4.6 reports the performance rates for the three algorithmic variants of the Cholesky factorization on the multi-core processor and the GPU, using double precision. The experimental setup used for the test is identical to that used for single precision.

The comparison between the performances of the single and double precision implementations of each variant of the Cholesky factorization reports the capabilities of each type of processor depending on the arithmetic precision. Both processors experiment a reduction in the performance when operating with double precision data. However, this reduction is less important for the general-purpose processor: 50% of the performance attained for single precision arithmetic. The drop is dramatically higher in the case of the GPU: double-precision performs at 25% of the pace attained for single-precision arithmetic. Peak performances are 53 GFLOPS on the GPU, and 68 GFLOPS on the CPU. The qualitative behavior of the algorithmic variants is the same as that observed for the single-precision experiments.

A quick conclusion is that (pre-Fermi) GPUs as that evaluated in this chapter are not competitive in double precision for this type of operations. This observation motivates the iterative refinement approach presented in Section 4.6.

Impact of the matrix size in the final performance

The analysis of the performance results attained for the blocked implementations admits two different observations:

- From a coarse-grained perspective, the performance results attained with single/double-precision data (see Figures 4.4/4.6, respectively) show a different behavior depending on the target processor type. In general, the performance curves for the CPU implementations

accelerate earlier, attaining near optimal performance results for relatively small matrices. This is different for the GPU implementations, in which high performance results are only reached for large matrix dimensions. In single precision, for example, the CPU attains higher performance rates for matrices up to $n = 8{,}000$; from this size on, the GPU performance becomes competitive, attaining better throughput as the matrix size is increased.

- From a fine-grained perspective, a study of the implementations for different operand sizes reveals important differences in the performance depending on the specific size of the matrix. For example, the Cholesky factorization operating on single-precision data (Figure 4.4) shows an important performance improvement when the matrix dimension is an integer multiple of 64; observe the results for $n = 4{,}000, n = 8{,}000, n = 16{,}000, \ldots$. This behavior is inherited from that observed for the BLAS operations in NVIDIA CUBLAS, as was noted in Section 3.4.5, and is common for the Cholesky and, in general, other codes based on BLAS-3, using single and double precision.

To overcome the penalties observed in the basic blocked implementations, the following sections introduce refinements that can be applied simultaneously and/or progressively in order to improve both the performance of the factorization process and the accuracy of the solution of the linear system. These improvements include padding, a hybrid CPU-GPU implementation, and an iterative refinement procedure to improve the accuracy of the solution to the linear system.

4.3.2. Padding

Previous experiments have shown that Level 3 BLAS implementations of NVIDIA CUBLAS (specially the GEMM kernel) deliver much higher performance when operating on matrices with dimensions that are a multiple of 32 [20]. Similar conclusions have been extracted from the evaluation of Level-3 BLAS routines in Chapter 3. This situation is inherited by the Cholesky factorization implementations, as reported in previous sections.

Therefore, it is possible to improve the overall performance of the blocked Cholesky factorization process by applying the correct pad to the input matrix. Starting from a block size n_b that is multiple of 32, we pad the $n \times n$ matrix A to compute the factorization

$$\bar{A} = \begin{pmatrix} A & 0 \\ 0 & I_k \end{pmatrix} = \begin{pmatrix} L & 0 \\ 0 & I_k \end{pmatrix} \begin{pmatrix} L & 0 \\ 0 & I_k \end{pmatrix}^T,$$

where I_k denotes the identity matrix of order k, and k is the difference between the matrix size n and the nearest integer multiple of n_b larger than n ($k = 0$ if n is already a multiple of n_b). By doing this, all BLAS-3 calls operate on sub-matrices of dimensions that are a multiple of 32, and the overall performance is improved. Moreover, there is no communication overhead associated with padding as only the matrix A and the resulting factor L are transferred between main memory and video memory. Although we incur in a computation overhead due to useless arithmetic operations which depends on the relation between n and 32, for moderate to large n, this overhead is negligible.

Figure 4.7 shows the performance attained for the blocked algorithmic variants for the Cholesky factorization when input matrices are padded accordingly with the directives given above. The rest of the experimental setup is the same as that used for the evaluation without padding. It is important to realiza that the attained performance rates are more homogeneous, without peaks when matrix sizes are multiples of 32. This improvement offers a straightforward and cheap way of obtaining regular performance rates independently from the dimension of the problem.

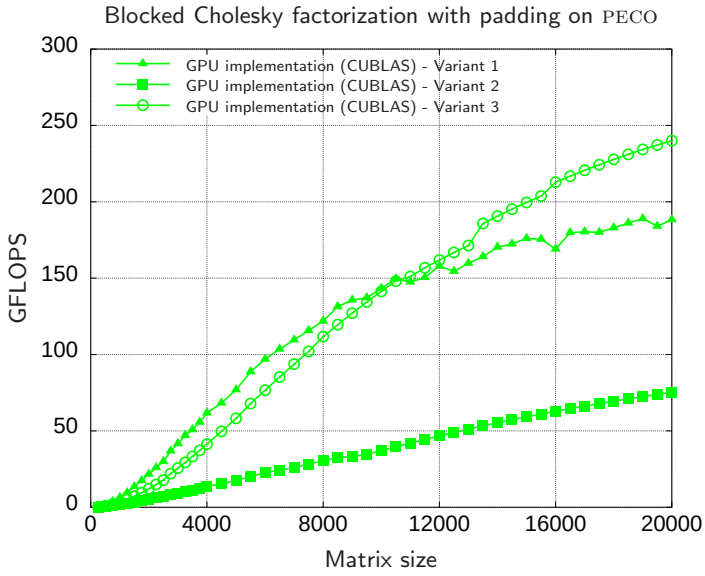


Figure 4.7: Performance of the single-precision blocked algorithmic variants of the Cholesky factorization on the graphics processor using padding on PECO.

4.3.3. Hybrid implementation

The results attained with the basic implementations show a remarkable difference in the performance of the factorizations for small matrices when they are carried out by the CPU or the GPU. In the first case, the performance is higher for small matrices, attaining close-to-peak performance already with the factorization of matrices of relatively small dimension. On the GPU side, the performance is low when the matrices to be decomposed are small, attaining higher performance results as the dimension is increased.

Hence, it is natural to propose a redesign of the basic blocked algorithm, in which each operation involved in the Cholesky factorization is performed in the most suitable processor. In this case, provided the optimal block size is relatively small (see Figure 4.5), the factorization of the diagonal blocks is a clear candidate to be offloaded to the CPU, as this architecture attains better efficiency in the factorization of small matrices.

Consider, e.g., any of the algorithmic variants for the Cholesky factorization shown in Figure 4.1. Each iteration of the algorithm implies the Cholesky factorization of the diagonal block A_{11}, usually of a small dimension. In the basic implementation, this factorization is performed exclusively on the GPU using an unblocked version of the Cholesky factorization. The main overhead due to the decomposition of the diagonal block on the GPU is directly related to its small dimension.

To tackle this problem, a hybrid version of the blocked algorithm for the Cholesky factorization has been developed, delegating some of the calculations previously performed on the GPU to the CPU. This approach aims at exploiting the different abilities of each processor to deal with specific operations. The advantage of the CPU is twofold: due to the stream-oriented nature of the GPU, this device pays a high overhead for small problem dimensions. On the other hand, the CPU is not affected by this and also delivers higher performance for some fine-grained arithmetic operations, specially the square root calculation, heavily used in the factorization of the diagonal block A_{11}.

In the Cholesky factorization, the hybrid algorithm moves the diagonal block from GPU memory to main memory, factorizes this block on the CPU using the multi-threaded BLAS implementation, and transfers back the results to the GPU memory before the computation on the GPU continues.

b	GPU → RAM	Fact. on CPU	RAM → GPU	Total CPU	Fact. on GPU
32	0.016	0.009	0.017	0.043	2.721
64	0.028	0.030	0.022	0.082	5.905
96	0.044	0.077	0.033	0.154	9.568
128	0.080	0.160	0.053	0.294	13.665
144	0.090	0.138	0.058	0.288	15.873
192	0.143	0.208	0.083	0.435	23.236
224	0.189	0.224	0.107	0.520	28.726
256	0.244	0.315	0.134	0.694	34.887
320	0.368	0.370	0.200	0.940	48.019
384	0.618	0.555	0.360	1.535	62.891
448	0.801	0.673	0.482	1.956	79.634
512	1.024	0.917	0.618	2.561	101.732
576	1.117	1.112	0.688	2.918	118.294
640	1.225	1.798	0.730	3.754	147.672
768	1.605	2.362	1.003	4.972	193.247

Table 4.1: Time breakdown for the factorization of one diagonal block for different block sizes on the CPU and on the GPU. Time in ms.

Whether this technique yields a performance gain depends on the overhead due to the data transfer between GPU and main memories as well as the performance of the Cholesky factorization of small matrices on the multi-core CPU.

To evaluate the impact of the intermediate data transfers, Table 4.1 analyzes the time devoted to intermediate data transfers to and from GPU memory necessary for the factorization of the diagonal block on the CPU. As shown, there is a major improvement in the factorization time for the tested block sizes on the CPU, even taking into account the necessary transfer times between memories (compare the columns "Total CPU", that includes data transfer times, with the column "Fact. on GPU").

Figure 4.8 reports the performance of the hybrid version of the blocked Cholesky algorithmic variants. Diagonal blocks are factorized using the multi-threaded MKL implementation, using the 8 cores of the architecture. The rest of the experimental setup is identical to that used in previous experiments.

The benefits of the hybrid approach are clear from the observation of the figure. With independence of the chosen variant, performance is improved compared with the rates attained for the padded variants (compare Figures 4.7 and 4.8). For example, Variant 3 of the blocked Cholesky factorization attains 241 GFLOPS in the version executed exclusively on the GPU and 301 GFLOPS in the hybrid version.

4.4. LU factorization

The LU factorization, combined with a pivoting strategy (usually partial row pivoting) is the most common method to solve linear systems when the coefficient matrix A does not present any particular structure. As for the Cholesky factorization, a necessary property of the coefficient matrix A is to be invertible.

The LU factorization of a matrix $A \in \mathbb{R}^{n \times n}$ involves the application of a sequence of $(n-1)$ Gauss transformations [144], that ultimately decompose the matrix into a product $A = LU$, where $L \in \mathbb{R}^{n \times n}$ is unit lower triangular and $U \in \mathbb{R}^{n \times n}$ is upper triangular.

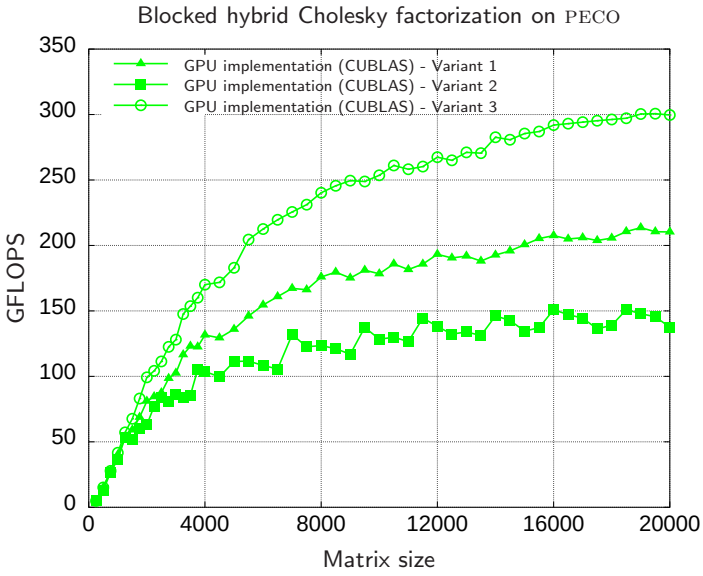


Figure 4.8: Performance of the single-precision blocked algorithmic variants of the Cholesky factorization on the graphics processor using padding and a hybrid execution approach on PECO.

Definition A Gauss transformation is a matrix of the form

$$L_k = \left(\begin{array}{c|c|c} I_k & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & l_{21} & I_{n-i-1} \end{array}\right), 0 \leq k < n. \tag{4.1}$$

An interesting property of the Gauss transformations is how they can be easily inverted and applied. The inverse of the Gauss transformation in (4.1) is given by

$$L_k^{-1} = \left(\begin{array}{c|c|c} I_k & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & -l_{21} & I_{n-i-1} \end{array}\right), 0 \leq k < n, \tag{4.2}$$

while, for example, applying the inverse of L_0 to a matrix A boils down to

$$L_0^{-1}A = \left(\begin{array}{c|c} 1 & 0 \\ \hline -l_{21} & I_{n-1} \end{array}\right)\left(\begin{array}{c|c} \alpha_{11} & a_{12}^{\mathrm{T}} \\ \hline a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^{\mathrm{T}} \\ \hline a_{21} - \alpha_{11}l_{21} & A_{22} - l_{21}a_{12}^{\mathrm{T}} \end{array}\right). \tag{4.3}$$

The main property of the Gauss transformations is the ability to annihilate the sub-diagonal elements of a matrix column. For example, taking $l_{21} := a_{21}/\alpha_{11}$ in (4.3), we have $a_{21} - \alpha_{11}l_{21} = 0$, and zeros are introduced in the sub-diagonal elements of the first column of A.

Another interesting property of the Gauss transformations is that they can be easily accumulated, as follows:

$$\left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & 1 & 0 \\ \hline L_{20} & 0 & I_{n-i-1} \end{array}\right) L_k = \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & 1 & 0 \\ \hline L_{20} & l_{21} & I_{n-i-1} \end{array}\right), \tag{4.4}$$

where $L_{00} \in \mathbb{R}^{k \times k}$.

Considering these properties, it is easy to demonstrate that the successive application of Gauss transformations corresponds in fact to an algorithm that computes the LU factorization of a matrix. Assume that after k steps of the algorithm, the initial matrix A has been overwritten with $L_{k-1}^{-1} \dots L_1^{-1} L_0^{-1} A$, where the Gauss transformations $L_j, 0 \leq j < k$ ensure that the sub-diagonal elements of column j of $L_{k-1}^{-1} \dots L_1^{-1} L_0^{-1} A$ are annihilated. Then

$$L_{k-1}^{-1} \dots L_1^{-1} L_0^{-1} A = \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array} \right),$$

where $U_{TL} \in \mathbb{R}^{k \times k}$ is upper triangular. Repartitioning

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & A_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} U_{00} & u_{12} & U_{02} \\ \hline 0 & \alpha_{11} & a_{12}^{\mathrm{T}} \\ \hline 0 & a_{21} & A_{22} \end{array} \right) \quad \text{and taking} \quad L_k = \left(\begin{array}{c|c|c} I_k & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & l_{21} & I_{n-k-1} \end{array} \right),$$

where $l_{21} := a_{21}/\alpha_{11}$, we have that

$$\begin{aligned} L_k^{-1}(L_{k-1}^{-1} \cdots L_1^{-1} L_0^{-1} A) &= \left(\begin{array}{c|c|c} I_k & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & -l_{21} & I_{n-k-1} \end{array} \right) \left(\begin{array}{c|c|c} U_{00} & u_{12} & U_{02} \\ \hline 0 & \alpha_{11} & a_{12}^{\mathrm{T}} \\ \hline 0 & a_{21} & A_{22} \end{array} \right) \\ &= \left(\begin{array}{c|c|c} U_{00} & u_{12} & U_{02} \\ \hline 0 & \mu_{11} & u_{12}^{\mathrm{T}} \\ \hline 0 & 0 & A_{22} - l_{21} u_{12}^{\mathrm{T}} \end{array} \right). \end{aligned}$$

Proceeding in this way, it is possible to reduce a matrix to an upper triangular form by successively applying Gauss transformations that annihilate the sub-diagonal entries, which effectively computes an LU factorization. Note that if $L_{n-1}^{-1} \dots L_1^{-1} L_0^{-1} A = U$, then $A = LU$, where $L = L_0 L_1 \dots L_{n-1}$ is built from the columns of the corresponding Gauss transformations.

Once the coefficient matrix is decomposed into the corresponding triangular factors, it is possible to obtain the solution of the linear system $Ax = b$ by solving two triangular systems: first, the solution of the system $Ly = b$ is obtained using progressive elimination; and then the system $Ux = y$ is solved by regressive substitution. As for the Cholesky factorization, these two stages involve lower computational cost, and thus the optimization effort is usually focused on the factorization stage.

To summarize, the LU factorization decomposes a matrix A into a unit lower triangular matrix L and an upper triangular matrix U such that $A = LU$. The following theorem dictates the conditions for the existence of the LU factorization.

Definition Given a partitioning of a matrix A as $A = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right)$, where $A_{TL} \in \mathbb{R}^{k \times k}$, matrices A_{TL}, $1 \leq k \leq n$, are called the leading principle sub-matrices of A.

Theorem 4.4.1 *A matrix $A \in \mathbb{R}^{n \times n}$ has an LU factorization if its $n-1$ leading principle sub-matrices of A are invertible. If the LU factorization exists and A is invertible, then the factorization is unique.*

The proof for this theorem can be found in the literature; see [70].

4.4.1. Scalar algorithm for the LU factorization

This section describes, using a formal derivation process, a scalar algorithm to obtain the LU factorization of a matrix $A \in \mathbb{R}^{n \times n}$. During this process, the elements below the diagonal of L overwrite the strictly lower triangular part of A, while the elements of U overwrite its upper triangular part. The diagonal of L is not explicitly stored as its elements are all 1.

Consider the following partitionings for the coefficient matrix A and its factors L and U:

$$A = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right), L = \left(\begin{array}{c|c} 1 & 0 \\ \hline l_{21} & L_{22} \end{array}\right), U = \left(\begin{array}{c|c} \mu_{11} & u_{12}^T \\ \hline 0 & U_{22} \end{array}\right),$$

where α_{11} and μ_{11} are scalars, a_{21}, l_{21}, and u_{12} vectors of $n-1$ elements, and A_{22}, L_{22} and U_{22} matrices of size $(n-1) \times (n-1)$. From $A = LU$,

$$\left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} 1 & 0 \\ \hline l_{21} & L_{22} \end{array}\right) \left(\begin{array}{c|c} \mu_{11} & u_{12}^T \\ \hline 0 & U_{22} \end{array}\right) = \left(\begin{array}{c|c} \lambda_{11}^2 & u_{12}^T \\ \hline \mu_{11} l_{21} & l_{21} u_{12}^T + L_{22} U_{22} \end{array}\right),$$

and the following expressions are derived:

$$\begin{array}{rcl} \mu_{11} & = & \alpha_{11}, \\ u_{21}^T & = & a_{12}^T, \\ l_{21} & = & a_{21}/\mu_{11}, \\ A_{22} - l_{21} u_{12}^T & = & L_{22} U_{22}. \end{array}$$

Proceeding in this manner, a recursive formulation of a scalar algorithm for the LU factorization is obtained; in this formulation, the elements of A are overwritten with those from the factors L and U as follows:

1. Partition $A = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right)$.
2. $\alpha_{11} := \mu_{11} = \alpha_{11}$.
3. $a_{12}^T := u_{12}^T = a_{12}^T$.
4. $a_{21} := l_{21} = a_{21}/\mu_{21}$.
5. $A_{22} := A_{22} - l_{21} u_{12}^T$.
6. Continue recursively with $A = A_{22}$ in step 1.

The scalar algorithm to calculate the LU factorization presents a computational cost of $2n^3/3$ flops. As in the Cholesky factorization, the major part of the calculation is due to the rank-1 update, that involves $O(n^2)$ arithmetic operations on $O(n^2)$ data. As usual, it is convinient to reformulate the algorithm in terms of operations with matrices (or rank-k updates), to improve the ratio between arithmetic operations and memory accesses, allowing a proper data reutilization that exploits the sophisticated memory hierarchy of modern processors. The blocked algorithm presented next pursues this goal.

4.4.2. Blocked algorithm for the LU factorization

Consider a partitioning of matrices A, L, and U:

$$A = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right), L = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right), U = \left(\begin{array}{c|c} U_{11} & U_{12} \\ \hline 0 & U_{22} \end{array}\right),$$

where A_{11}, L_{11} and U_{11} are sub-matrices of size $b \times b$ (with b much smaller than the dimension of A); and A_{22}, L_{22} and U_{22} are matrices with $(n-b) \times (n-b)$ elements. From $A = LU$:

$$A = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right) \left(\begin{array}{c|c} U_{11} & U_{12} \\ \hline 0 & U_{22} \end{array}\right) = \left(\begin{array}{c|c} L_{11}U_{11} & L_{11}U_{12} \\ \hline L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{array}\right),$$

from which the following expressions are obtained:

$$\begin{aligned} L_{11}U_{11} &= A_{11}, \\ U_{21} &= L_{11}^{-1}A_{12}, \\ L_{21} &= A_{21}U_{11}^{-1}, \\ A_{22} - L_{21}U_{12} &= L_{22}U_{22}. \end{aligned}$$

Thus, the recursive blocked algorithm for the calculation of the LU factorization of A, can be formulated as follows:

1. Partition $A = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right)$, with A_{11} of size $b \times b$.
2. $A_{11} := \{L\backslash U\}_{11}$ such that $A_{11} = L_{11}U_{11}$ (that is, A_{11} is overwritten with its LU factorization).
3. $A_{12}^T := U_{12}^T = L_{11}^{-T}A_{12}$.
4. $A_{21} := L_{21} = A_{21}/U_{21}$.
5. $A_{22} := A_{22} - L_{21}U_{12}^T$.
6. Continue recursively with $A = A_{22}$ in step 1.

The cost of this blocked formulation is $2n^3/3$ flops. In this algorithm, the calculation of the panels A_{12} and A_{21}, and the update of the sub-matrix A_{22} are cast in terms of Level-3 BLAS routines. The latter operation involves the major part of the calculations ($O(n^2b)$ operations on $O(n^2)$ data). As typically $b \gg 1$, the ratio between arithmetic operations and data accesses is increased, and thus data reutilization is improved.

4.4.3. LU factorization with partial pivoting

As noted before, the solution of a linear system of the form $Ax = b$ exists and is unique if the coefficient matrix A is invertible. However, this is not the only requisite for the existence of the LU factorization. For example, if A is invertible, but any of the elements in the diagonal is zero, the factorization is not feasible, as a division by zero appears during the algorithm (see step 4 of the scalar algorithm presented above).

Row pivoting is a strategy that can be applied to solve this issue: if the row with the zero diagonal element is swapped with a different row below that in the matrix (without a zero element in the corresponding entry), the factorization can continue normally. Note that these swappings must be applied as well to vector b before solving the linear system.

Another problem that can appear during the LU factorization is caused by diagonal elements with a small magnitude compared with the rest of the elements in its column. This situation involves a growth in the magnitude of the elements of U, with the consequent rounding errors in the presence of limited precision architectures, and a catastrophic impact on the accuracy of the results [144].

A row pivoting strategy that swaps the diagonal element with that of largest magnitude from those in and below the diagonal in the current column, guarantees that every element in L is equal or smaller than 1 in magnitude, and handles properly the growth of the elements in U, limiting the impact of the round-off errors. This technique is known as *partial row pivoting*, and will be implemented in our GPU codes.

Scalar algorithm for the LU factorization with partial pivoting

The introduction of the partial pivoting in the recursive scalar algorithm asks for two new steps that follow the initial partitioning of the coefficient matrix. First, the element with the largest magnitude in the first column of the current sub-matrix to be factorized has to be found. Second, the first row of the sub-matrix and the row of the selected element must be swapped.

The resulting algorithm is formulated as follows:

1. Partition $A = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right)$.
2. $\pi :=$ index of the element with the largest magnitude in $\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$.
3. Swap the row $\left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \end{array}\right)$ with the corresponding elements of the π-th row.
4. $\alpha_{11} := \mu_{11} = \alpha_{11}$.
5. $a_{12}^T := u_{12}^T = a_{12}^T$.
6. $a_{21} := l_{21} = a_{21}/\mu_{21}$.
7. $A_{22} := A_{22} - l_{21}u_{12}^T$.
8. Continue recursively with $A = A_{22}$ in step 1.

The same sequence of swappings (or *permutations*) must be applied to vector b in the solution of the linear system. This application is usually performed after the LU factorization of the coefficient matrix A is computed; thus, it is necessary to store the information that allows the application of this sequence. In practice, a vector of n elements is enough to store the permutation information and recover it afterwards.

To formally include row swapping in the LU factorization, we next introduce *permutation matrices*. The effect of a permutation matrix is to rearrange the elements of vectors and entire rows or columns of matrices.

Definition $P \in \mathbb{R}^{n \times n}$ is a permutation matrix if, when applied to the vector $x = (\chi_0, \chi_1, \ldots, \chi_{n-1})^T$, rearranges the order of the elements in that vector. Such a permutation can be represented by a vector of integers $p = (\pi_0, \pi_1, \ldots, \pi_{n-1})^T$, where $\{\pi_0, \pi_1, \ldots, \pi_{n-1}\}$ is a permutation of $\{0,1,\ldots,n-1\}$, and the scalars π_i indicate that the permuted vector is given by $Px = (\chi_{\pi_0}, \chi_{\pi_1}, \ldots, \chi_{\pi_{n-1}})$.

A permutation matrix is equivalent to the identity matrix with permuted rows.

Definition If P is the identity matrix with rows i and j swapped, then applying P to a matrix A as PA swaps rows i and j of A.

The algorithm shown above is implemented in LINPACK [56]. In this algorithm, the application of the Gauss transformations and permutations on b must be interleaved, exactly as has been shown for A to obtain U. Thus, if $P_k \in \mathbb{R}^{n \times n}$ and $L_k \in \mathbb{R}^{n \times n}$ are, respectively, the permutation matrix and the Gauss transformation applied in the k-th stage of the algorithm, it follows that $A = (P_1 L_0 P_1 L_1 \ldots P_{n-1} L_{n-1})U$ and, thus, the solution of the linear system $Ax = b$ can be obtained from $Ux = (L_{n-1}^{-1} P_{n-1} \ldots L_1^{-1} P_1 L_0^{-1} P_0)b$. A side effect of the interleaving of permutations and Gauss transformations is the impossibility of the derivation of an analogous blocked algorithm, as each Gauss transformation is applied in terms of a rank-1 update (GER routine in BLAS-2).

An alternative approach for row pivoting is implemented in LAPACK [9]. In the implementation proposed in this library, each interchange affects all elements in the involved rows. Thus, in the step 3 of the scalar algorithm, the elements to the left of the diagonal entry α_{11}, together with $\left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \end{array} \right)$, are permuted with *all* the elements in the row of index π. Pivoting with complete rows is equivalent to grouping the permutations, and hence it is not necessary to apply them in an interleaved manner with the Gauss transformations. More precisely, if matrix $P = P_{n-1} \ldots P_1 P_0$ contains the permutations applied during the factorization, then the solution of the linear system $Ax = b$ is obtained by solving the systems $Ly = Pb$ and $Ux = y$.

Adapting the recursive formulation of the algorithm for the LU factorization to this new pivoting scheme is not straightforward, mainly because the pivoting affects the sub-matrix on which the calculations are performed at each iteration (A_{22} in the previous iteration), as well as to the elements that have already been factored in previous iterations. In this case it is more convenient the formulation of the algorithm using the FLAME notation for both the scalar and the blocked algorithms.

Algorithms in FLAME notation for the scalar and blocked LU factorization with partial pivoting

The formal derivation process proposed by the FLAME project drives to five different algorithmic variants for the LU factorization. From these variants, only three can be combined with partial pivoting [138]. The scalar and blocked algorithms for these three variants are shown in Figure 4.9 (left and right, respectively). In the FLAME algorithms, $n(\cdot)$ represents the number of columns of a matrix, and $P(\pi_1)$ denotes the permutation matrix that, applied to a second matrix, swaps the first row of it with row π_1. If $p_1 = (\pi_1, \pi_2, \ldots, \pi_b)$, then $P(p_1)$ is the permutation matrix that, applied to a second matrix, swaps its first row with row π_1, the second row with row π_2, and so on, until the last swapping of row b with row π_b. This sequence of permutations is gathered by vector p. The procedure *PivIndex* returns the index of the element of largest magnitude of the vector passed as a parameter.

TRILU$(\cdot)$ returns the strictly lower triangular matrix stored in the matrix passed as a parameter, transforming it to a unit lower triangular matrix. Its usage implies that the operation $A_{12} :=$ TRILU$(A_{11})^{-T} A_{12}$ is the solution of a triangular system (routine TRSM in BLAS). The rest of the operations in the algorithm are cast in terms of GEMM-like operations.

Algorithm: $[A, p] := \text{LU\_PIV\_UNB}(A)$

Partition

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), \; p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$$

where A_{TL} is 0×0 and p_T has 0 elements

while $n(A_{TL}) < n(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \; \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline \pi_1 \\ \hline p_2 \end{array}\right)$$

where α_{11} and π_1 are 1×1

Variant 1:

$$\pi_1 := \text{PivIndex}\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$$

$$\left(\begin{array}{c|c|c} a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) := P(\pi_1) \left(\begin{array}{c|c|c} a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

$$a_{21} := a_{21}/\alpha_{11}$$

$$A_{22} := A_{22} - a_{21} a_{12}^T$$

Variant 2:

$$\left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array}\right) := P(p_0) \left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array}\right)$$

$$a_{01} := \text{TRILU}(A_{00})^{-1} a_{01}$$

$$\alpha_{11} := \alpha_{11} - a_{10}^T a_{01}$$

$$a_{21} := a_{21} - A_{20} a_{01}$$

$$\pi_1 := \text{PivIndex}\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$$

$$\left(\begin{array}{c|c} a_{10}^T & \alpha_{11} \\ \hline A_{20} & a_{21} \end{array}\right) := P(\pi_1) \left(\begin{array}{c|c} a_{10}^T & \alpha_{11} \\ \hline A_{20} & a_{21} \end{array}\right)$$

$$a_{21} := a_{21}/\alpha_{11}$$

Variant 3:

$$\alpha_{11} := \alpha_{11} - a_{10}^T a_{01}$$

$$a_{21} := a_{21} - A_{20} a_{01}$$

$$a_{12}^T := a_{12}^T - a_{10}^T A_{02}$$

$$\pi_1 := \text{PivIndex}\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$$

$$\left(\begin{array}{c|c|c} a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) := P(\pi_1) \left(\begin{array}{c|c|c} a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

$$a_{21} := a_{21}/\alpha_{11}$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \; \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline \pi_1 \\ \hline p_2 \end{array}\right)$$

endwhile

Algorithm: $[A, p] := \text{LU\_PIV\_BLK}(A)$

Partition

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), \; p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$$

where A_{TL} is 0×0 and p_T has 0 elements

while $n(A_{TL}) < n(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \; \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$$

where A_{11} is $b \times b$ and p_1 has b elements

Variant 1:

$$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), p_1\right] := \text{LU\_PIV\_UNB}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$$

$$\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right) := P(p_1) \left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right)$$

$$A_{12} := \text{TRILU}(A_{11})^{-1} A_{12}$$

$$A_{22} := A_{22} - A_{21} A_{12}$$

Variant 2:

$$\left(\begin{array}{c} A_{01} \\ \hline A_{11} \\ \hline A_{21} \end{array}\right) := P(p_0) \left(\begin{array}{c} A_{01} \\ \hline A_{11} \\ \hline A_{21} \end{array}\right)$$

$$A_{01} := \text{TRILU}(A_{00})^{-1} A_{01}$$

$$A_{11} := A_{11} - A_{10} A_{01}$$

$$A_{21} := A_{21} - A_{20} A_{01}$$

$$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), p_1\right] := \text{LU\_PIV\_UNB}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$$

$$\left(\begin{array}{c} A_{10} \\ \hline A_{20} \end{array}\right) := P(p_1) \left(\begin{array}{c} A_{10} \\ \hline A_{20} \end{array}\right)$$

Variant 3:

$$A_{11} := A_{11} - A_{10} A_{01}$$

$$A_{21} := A_{21} - A_{20} A_{01}$$

$$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), p_1\right] := \text{LU\_PIV\_UNB}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$$

$$\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right) := P(p_1) \left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right)$$

$$A_{12} := A_{12} - A_{10} A_{02}$$

$$A_{12} := \text{TRILU}(A_{11})^{-1} A_{12}$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \; \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$$

endwhile

Figure 4.9: Algorithmic variants for the scalar (left) and blocked (right) algorithms for the LU factorization with partial pivoting.

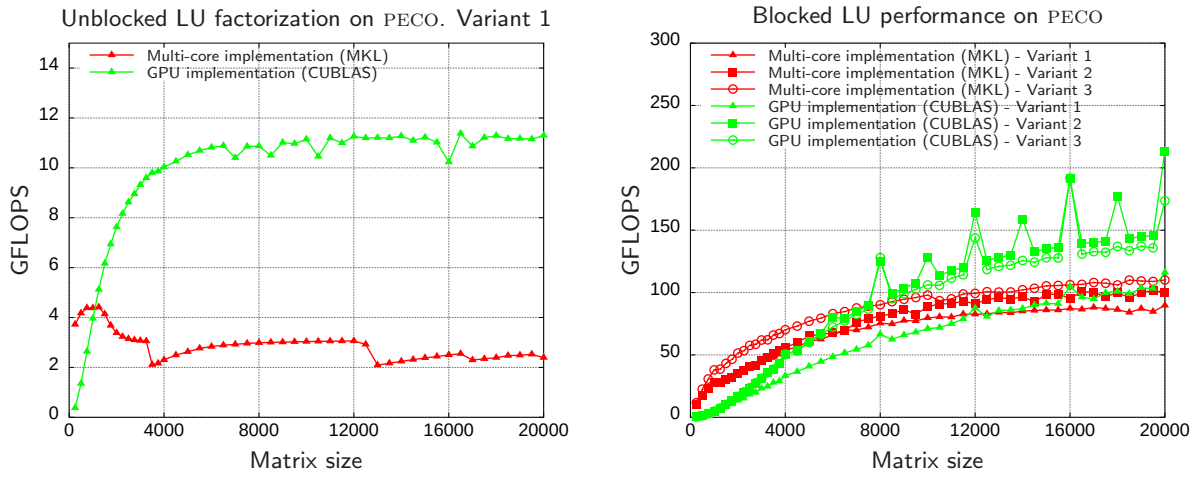


Figure 4.10: Performance of the single-precision unblocked (left) and blocked (right) LU factorization on the multi-core processor and the graphics processor on PECO.

4.5. Computing the LU factorization with partial pivoting on the GPU

4.5.1. Basic implementations. Unblocked and blocked versions

Given the unblocked and blocked formulations of the LU factorization with partial pivoting in Figure 4.9, the procedure followed to implement and evaluate the multi-core and GPU implementation is equivalent to that used for the Cholesky factorization.

Basic scalar and blocked implementations corresponding to the different algorithmic variants in Figure 4.9 have been implemented and evaluated on the multi-core CPU and the graphics processor. In these implementations, the information related to pivoting is kept in an integer vector IPIV stored in main memory (also for the GPU implementations). This vector, of dimension n for an $(n \times n)$ coefficient matrix, maintains the pivot indices used in the factorization process: for each position i of the vector, row i of the matrix was interchanged with row IPIV(i).

On the GPU, the row interchanges due to the pivoting scheme are performed using the BLAS-1 CUBLASSSWAP routine, that swaps two rows of a matrix stored on GPU memory. Given the high bandwidth delivered by the GPU memory, a lower penalty can be expected from these interchanges than is expected if the interchange occurs in main memory.

The plots in Figure 4.10 report the performance attained for the LU with partial pivoting implementations using the scalar (left-hand plot) and blocked (right-hand plot) algorithmic variants from Figure 4.9. Data transfers are not included in the time measurements, and performance is shown for the optimal block size in the blocked results.

A number of conclusions for the LU implementations which are similar to those extracted for the Cholesky factorization:

- The scalar implementations are only convenient on the CPU for relatively small matrix dimensions (up to $n = 1{,}024$ in the evaluated algorithmic variant). For these matrix dimensions, the cache system in the multi-core processor plays a critical role on the final attained performance. As in the Cholesky factorization, this motivates the derivation and development of blocked versions of the variants that make use of the advanced memory hierarchy of the multi-cores and the stream-oriented nature of the GPU.

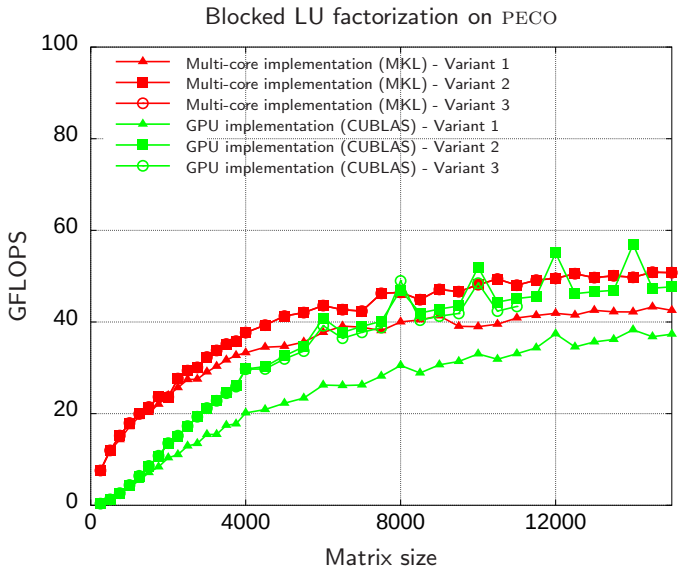


Figure 4.11: Performance of the double-precision blocked LU factorization on the multi-core and the graphics processor on PECO.

- The performance rates attained by the blocked implementations are closer to the peak performances of the architectures (107 GFLOPS for the multi-core and 211 GFLOPS for the GPU). These rates illustrate the appeal of blocked algorithms on both types of processors.

  As in the Cholesky factorization, the LU factorization on the GPU presents an irregular behavior, with performance peaks for selected matrix sizes, which can be solved by applying the appropriate padding.

  In addition, multi-core variants deliver higher performance for small to moderately-sized coefficient matrices. For the tested dimensions, GPU attains higher performance only for matrices bigger than $n = 8{,}000$. This also motivates the use of a hybrid approach, mixing the capabilities of each processor for each operation inside the factorization.

- In the blocked variants, the derivation and systematic evaluation of all available algorithmic variants (together with the appropriate selection of the block size for each one) makes the difference between an algorithmic variant that attains lower performance than that of the CPU (see, for example, Variant 1 in Figure 4.10) and another one attaining remarkable speedups (see, for example, Variants 2 and 3 in the figure).

- The weakness of the graphics processor when operating with double precision is also revealed in the LU factorization with partial pivoting. Figure 4.11 shows the performance results of the three algorithmic variants developed for the LU factorization, using double precision, on the multi-core platform and the graphics processors. In fact, the GPU implementations barely attain better results than the equivalent multi-core implementations for selected large matrix dimensions, and this improvement is not remarkable. For small/medium matrix sizes (up to $n = 8{,}000$), the multi-core architecture outperforms the GPU. This observation is similar to that for the Cholesky factorization, and motivates the usage of mixed-precision techniques, such as the iterative refinement presented in Section 4.6, to overcome this limitation of the GPUs while preserving the accuracy of the solution.

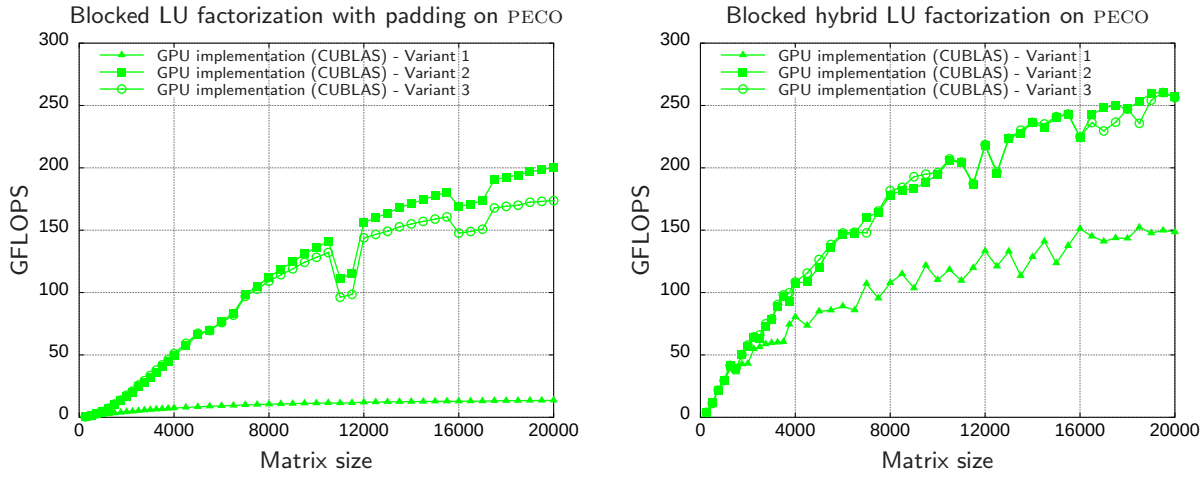


Figure 4.12: Performance of the single-precision blocked LU factorization with padding (left) and hybrid approach (right) on the graphics processor on PECO.

4.5.2. Padding and hybrid algorithm

Regarding matrix dimensions, two main conclusions can be obtained from the evaluation of the algorithmic variants derived for the LU factorization with partial pivoting. First, the irregular behavior of the routines depending on the specific matrix size. Second, the different skills of multi-cores and GPUs when factorizing small and large matrices. While the former architectures are especially efficient with matrices of small dimension, GPUs demonstrate their strength in the factorization of larger matrices.

As in the Cholesky factorization, padding and hybrid algorithms are the solutions proposed to solve these two issues, respectively.

Figure 4.12 (left) shows the performance of the three algorithmic variants developed for the LU factorization with partial pivoting on the GPU, applying the appropriate padding to the coefficient matrix. The experimental setup is identical to that used for the basic blocked implementation. Although the peak performance attained for the padded version of the algorithms is similar to that without padding, performance results are more homogeneous, without the presence of peaks in the performance curves shown in the plots.

A similar hybrid approach to that applied for the Cholesky factorization has been applied in the LU factorization with partial pivoting. In this case, at each iteration of the factorization process, the current column panel $\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$ is transferred to main memory; it is factored there using a scalar or blocked multi-threaded implementation; and then it is transferred back to the GPU, where the factorization continues.

Figure 4.12 (right) shows the performance of the three algorithmic variants using the hybrid approach. All hybrid variants improve their performance with respect to the corresponding GPU-only implementation. As an example, the peak performance of Variant 2 is increased from 211 GFLOPS to 259 GFLOPS by the hybrid approach.

4.6. Iterative refinement for the solution of linear systems

Graphics processors typically deliver single-precision arithmetic much faster than that of their double-precision counterparts. In previous chapters, significant differences between single- and

double-precision performance have been observed for the BLAS-3 routines; in previous sections of this chapter, similar conclusions have been derived from the Cholesky and LU factorizations. The demands that graphical applications set on data precision ultimately determine this behavior, as no double-precision support is usually needed for these applications. In addition, these needs will not likely change in future hardware and software generations. This gap between single- and double-precision performance does not only appear in graphical architectures. The Cell B.E. processor announces a theoretical peak performance of 204.8 GFLOPS in single-precision, reducing it to 20 GFLOPS when operating on double-precision data. For general-purpose processors, such as the Intel x87, the use of SSE units can boost performance by doing 4 single-precision flops per cycle, while this rate is reduced to 2 double-precision flops per cycle provided the SSE (or equivalent) extensions are supported by the processor.

However, double precision often the norm in scientific computing in general, and in linear algebra in particular. To solve the performance gap between single and double precision in modern graphics processors, a mixed-precision iterative refinement approach is proposed and evaluated in this section.

In the context of the solution of linear equations, iterative refinement techniques have been successfully applied for many years [104, 93]. Our mixed-precision iterative refinement process combines the benefits of using the GPU to boost single-precision routines with the accuracy of a full double-precision approach.

As a driving example, consider the solution of the linear system $Ax = b$ using Gaussian elimination with partial pivoting, where the coefficient matrix is factored as $PA = LU$, with L lower triangular, U upper triangular, and P the permutation matrix that results from the factorization. The idea that underlies the iterative process for the accurate solution of a dense linear equation system involves an iterative algorithm with four main steps:

$$\begin{aligned} &\text{Compute } r = b - Ax \\ &\text{Solve } \ Ly = Pr \\ &\text{Solve } \ Uz = y \\ &\text{Update } x' = x + z \end{aligned} \tag{4.5}$$

In a mixed precision iterative refinement process, the factorization and triangular solves are carried out using single precision, and only the calculation of the residual and the update of the solution is performed using double precision. Thus, most of the flops involved in the process are carried out in single precision. Naturally, these procedures (especially the factorization, that entails the major part of the calculations) are clear candidates to be executed on the GPU.

This approach has been successfully applied to modern hardware accelerators, such as the Cell B.E. [91]. As a novelty, we propose and evaluate a similar implementation on modern graphics processors that exploits the single precision capabilities of the GPU.

The impact of this approach is that the major part of the operations (factorization of the coefficient matrix and forward/backward substitutions) are performed in single precision, while only the residual calculation and update of the solution are performed in double precision. The striking advantadge is that the final accuracy of the solution will remain the same as that of a full double-precision approach, while the performance of the implementation will benefit from the high performance of the GPU in single-precision computations. The only drawback is the memory needs of the algorithm, as a (single-precision) copy of A must be explicitly stored to calculate the residual at each iteration.

In our solution, the factorization of matrix A is first computed on the GPU (in single precision arithmetic) using any of the algorithms proposed in the previous sections. Depending on the numerical properties and structure of the coefficient matrix, the Cholesky or the LU factorization with partial pivoting can be used. A first solution is then computed and iteratively refined on the CPU to double precision accuracy.

Algorithm 1 lists the steps necessary to perform the factorization, solution and iterative refinement to solve a dense linear system $Ax = b$. In this algorithm, the ‘‘(32)’’ subscript indicates single precision storage, while the absence of subscript implies double precision format. Thus, only the matrix-vector product Ax is performed in double precision (BLAS routine GEMV), at a cost of $O(n^2)$ flops (floating-point arithmetic operations), while the rest of the non-negligible arithmetic operations involve only single precision operands. The Cholesky factorization is computed on the GPU. A similar strategy can be applied to general systems using the LU factorization.

Our implementation of the iterative refinement algorithm iterates until the solution, $x^{(i+1)}$, satisfies the following condition:

$$\frac{\|r^{(i)}\|}{\|x^{(i+1)}\|} < \sqrt{\varepsilon},$$

where $r^{(i)}$ is the current residual and $x^{(i)}$ is the current solution, both with a double precision representation, and ε corresponds to the machine (double) precision of the platform. When this condition is satisfied, the algorithm iterates two more steps, and the solution is then considered to be accurate enough [38].

Algorithm 1 Iterative refinement using the Cholesky factorization

$A_{(32)}, b_{(32)} \leftarrow A, b$
$L_{(32)} \leftarrow$ GPU_CHOL_BLK$(A_{(32)})$
$x^{(1)}_{(32)} \leftarrow L^{-T}_{(32)}(L^{-1}_{(32)} b_{(32)})$
$x^{(1)} \leftarrow x^{(1)}_{(32)}$
$i \leftarrow 0$
repeat
  $i \leftarrow i + 1$
  $r^{(i)} \leftarrow b - A \cdot x^{(i)}$
  $r^{(i)}_{(32)} \leftarrow r^{(i)}$
  $z^{(i)}_{(32)} \leftarrow L^{-T}_{(32)}(L^{-1}_{(32)} r^{(i)}_{(32)})$
  $z^{(i)} \leftarrow z^{(i)}_{(32)}$
  $x^{(i+1)} \leftarrow x^{(i)} + z^{(i)}$
until $\frac{\|r^{(i)}\|}{\|x^{(i+1)}\|} < \sqrt{\varepsilon}$

Figure 4.13 summarizes the performance results attained for the solution of a linear system $Ax = b$ using our iterative refinement approach. The x axis of the plots represents the dimension of the coefficient matrix; the y axis represents the computation time, in seconds. The times on the left-hand side plot correspond to the case when A is symmetric definite positive, and thus the Cholesky factorization is used to decompose it. On the right-hand side plot, the LU factorization with partial pivoting is used to decompose the coefficient matrix. In both cases, the dark red area corresponds to the factorization time using single precision on the GPU, while the light red area corresponds to the iterative refinement time. The lines in the plots represent the computation time for a full double-precision approach, performing the factorization on the GPU (red line) or on the

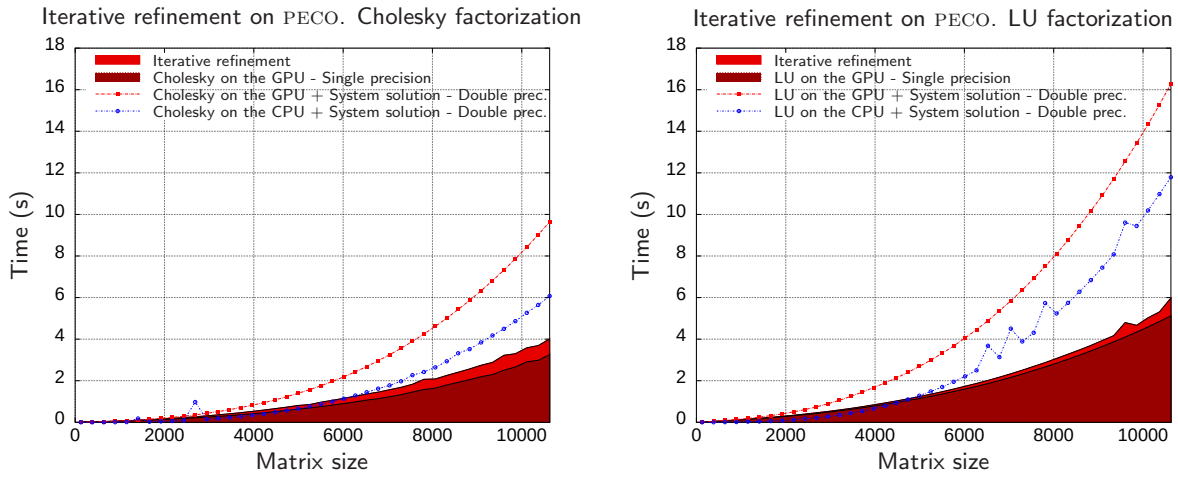


Figure 4.13: Time devoted to factorization and solution of a linear system using double precision on the CPU and the GPU, single precision on the GPU and an iterative refinement strategy on PECO. The Cholesky factorization (left) and the LU factorization (right) are employed to decompose the system matrix.

	Cholesky			LU		
n	Factorization	It. refinement	# iters.	Factorization	It. refinement	# iters.
1024	0.066	0.006	6	0.103	0.003	5
2048	0.153	0.034	5	0.261	0.018	5
4096	0.411	0.110	5	0.778	0.061	4
8192	1.640	0.435	6	2.843	0.229	5
10240	3.255	0.753	5	5.118	0.858	5

Table 4.2: Time devoted to factorization and iterative refinement using Cholesky and LU factorizations, and number of iterations necessary to regain full accuracy

CPU (blue line). In this experiment, the iterative refinement is performed exclusively on the CPU; thus, only the factorization of A is performed in the GPU using single precision.

As a first conclusion, the mixed precision approach with iterative refinement presents a remarkable performance gain compared with the double precision factorization and solution on the CPU or the GPU. Our approach permits using the GPU as the platform for the factorization, improving performance while maintaining the accuracy of the solution. If the GPU is only used for the factorization (see red line in the plots), it is not competitive compared with the CPU. With independence of the factorization routine, the time devoted to the iterative refinement process is smaller than that required for the factorization using single precision. Table 4.2 shows a breakdown of the time devoted to the factorization using the GPU and single precision, and the iterative refinement process. In addition, we include the number of iterations required for convergence. As shown, this number is typically small (between 4 and 6 iterations, depending on the matrix size and the factorization routine). For a theoretical study of the upper bound in the number of iterations necessary for convergence, see [123].

The iterative refinement of the solution of a linear equation system represents a trade-off between the high performance of current GPUs operating in single precision and the accuracy delivered by double-precision arithmetic.

4.7. Reduction to tridiagonal form on the graphics processor

We consider the solution of the *symmetric eigenvalue problem* $AX = X\Lambda$, where $A \in \mathbb{R}^{n\times n}$ is a *dense* symmetric matrix, $\Lambda = \text{diag}(\lambda_1,\lambda_2,\ldots,\lambda_n) \in \mathbb{R}^{n\times n}$ is a diagonal matrix containing the eigenvalues of A, and the j-th column of the orthogonal matrix $X \in \mathbb{R}^{n\times n}$ is an eigenvector associated with λ_j [70]. Given the matrix A, the objective is to compute its eigenvalues or a subset thereof and, if requested, the associated eigenvectors as well. Many scientific and engineering applications lead to large eigenvalue problems. Examples come from computational quantum chemistry, finite element modeling, multivariate statistics, and density functional theory. Especially in the latter, problems become particularly challenging because a significant fraction of the eigenvalues and eigenvectors needs to be computed [102].

Reduction to tridiagonal form is one of the most time-consuming parts of the algorithms that are employed to solve the symmetric eigenvalue problem. In this section, the LAPACK routines for the reduction to tridiagonal form are presented and evaluated on modern multi-core processors. Additionally, a different two-stage approach used by the SBR toolbox is also presented and evaluated. As a novelty, the most time-consuming parts of this algorithm are off-loaded to the GPU. We show the benefits of this hybrid approach and combine it with the tuned BLAS-3 routines presented in Chapter 3.

4.7.1. The symmetric eigenvalue problem

Efficient algorithms for the solution of symmetric eigenvalue problems usually consist of three stages:

1. Matrix A is first reduced to a (symmetric) tridiagonal form $T \in \mathbb{R}^{n\times n}$, by a sequence of orthogonal similarity transforms: $Q^T AQ = T$, where $Q \in \mathbb{R}^{n\times n}$ is the matrix representing the accumulation of these orthogonal transforms.

2. A tridiagonal eigensolver as, e.g., the $\texttt{MR}^3$ algorithm [52, 27] or the (parallel) $\texttt{PMR}^3$ [27] algorithm is then applied to matrix T to accurately compute its eigenvalues and, optionally, the associated eigenvectors.

3. Finally, when the eigenvectors of A are desired, a back-transform has to be applied to the eigenvectors of T. In particular, if $TX_T = X_T\Lambda$, with $X_T \in \mathbb{R}^{n\times n}$ representing the eigenvectors of T, then $X = QX_T$.

Both the first and last stage cost $O(n^3)$ floating-point arithmetic operations (flops) while the second stage based on the $\texttt{MR}^3$ algorithm only requires $O(n^2)$ flops. (Other algorithms for solving tridiagonal eigenvalue problems, such as the QR algorithm, the Divide & Conquer method, etc. [70] require $O(n^3)$ flops in the worst case.)

We re-evaluate the performance of the codes in LAPACK [9] and the *Successive Band Reduction* (SBR) toolbox [32] for the reduction of a symmetric matrix A to tridiagonal form. LAPACK routine SYTRD employs a simple algorithm based on Householder reflectors [70], enhanced with WY representations [53], to reduce A directly to tridiagonal form. Only half of its operations can be performed in terms of calls to Level-3 BLAS kernels, resulting in a poor use of the memory hierarchy. To overcome this drawback, the SBR toolbox first reduces A to an intermediate banded matrix B, and subsequently transforms B to tridiagonal form. The advantage of this two-step procedure is that the first step can be carried out using BLAS-3 kernels, while the cost of the second step is negligible provided a moderate band width is chosen for B.

A similar study was performed by B. Lang in [92]. The conclusions from that work were that the SBR toolbox could significantly accelerate the computations of the reduction to tridiagonal form compared to the approach in LAPACK. However, if the orthogonal transforms have to be accumulated, then the SBR routines were not competitive. Our interest in this analysis is motivated by the increase in the number of cores in general-purpose processors in the last years and the recent advances in more specific hardware accelerators like graphics processors. In particular, we aim at evaluating how the use of multiple cores in these architectures affects the performance of the codes in LAPACK and SBR for tridiagonal reduction and back-transform. Note that, because of the efficient formulation and practical implementation of the MR^3 algorithm, the reduction to tridiagonal form and the back-transform are currently the most time-consuming stages in the solution of large symmetric eigenvalue problems.

The main contribution of this section is a practical demonstration that the use of GPUs turns SBR to being a competitive approach for both the reduction to tridiagonal form and the accumulation of transforms. This changes the main message that was presented in [92].

4.7.2. Reduction to tridiagonal form. The LAPACK approach

Let $A \in \mathbb{R}^{n\times n}$ be a *dense* symmetric matrix; then it is possible to find an orthogonal matrix Q such that

$$Q^T AQ = T \tag{4.6}$$

is tridiagonal. This process is usually referred to as *t*ridiagonal decomposition or reduction to tridiagonal form, and constitutes the first step towards the solution of the symmetric eigenvalue problem. We illustrate how the routines in LAPACK compute (4.6) via Householder matrices.

LAPACK routine SYTRD is based on the classical approach of reducing A to a tridiagonal matrix $T \in \mathbb{R}^{n\times n}$ by a series of Householder reflectors H_1, H_2, ..., H_{n-2} [70]. Each Householder reflector is an orthogonal matrix of the form $H_j = I - \beta_j u_j u_j^T$, where $\beta_j \in \mathbb{R}$, $u_j \in \mathbb{R}^n$ with the first j entries zero, and I denotes hereafter the square identity matrix of the appropriate order. The purpose of each reflector H_j is to annihilate the entries below the sub-diagonal in the j-th column of $A_{j-1} = H_{j-1}^T \cdots H_2^T H_1^T A H_1 H_2 \cdots H_{j-1}$.

Routine SYTRD proceeds as follows. Let b denote the algorithmic block size and assume that we have already computed the first $j-1$ columns/rows of T. Consider

$$H_{j-1}^T \cdots H_2^T H_1^T A H_1 H_2 \cdots H_{j-1} = \left(\begin{array}{c|c|c} T_{00} & T_{10}^T & 0 \\ \hline T_{10} & A_{11} & A_{21}^T \\ \hline 0 & A_{21} & A_{22} \end{array}\right),$$

where $T_{00} \in \mathbb{R}^{j-1\times j-1}$ is in tridiagonal form and $A_{11} \in \mathbb{R}^{b\times b}$. With this partitioning, all entries of T_{10} are zero except for that one in its top right corner. Then, the following operations are computed during the current iteration of SYTRD:

1. The current panel $\left(\frac{A_{11}}{A_{21}}\right)$ is reduced to tridiagonal form by a sequence of b orthogonal transforms H_j,H_{j+1},...,H_{j+b-1}. Simultaneously, two matrices U,$W \in \mathbb{R}^{(n-j-b+1)\times b}$ are built

such that

$$H_{j+b-1}^T \cdots H_{j+1}^T H_j^T \left(\begin{array}{c|c|c} T_{00} & T_{10}^T & 0 \\ \hline T_{10} & A_{11} & A_{21}^T \\ \hline 0 & A_{21} & A_{22} \end{array}\right) H_j H_{j+1} \cdots H_{j+b-1}$$
$$= \left(\begin{array}{c|c|c} T_{00} & T_{10}^T & 0 \\ \hline T_{10} & T_{11} & T_{21}^T \\ \hline 0 & T_{21} & A_{22} - UW^T - WU^T \end{array}\right),$$

where T_{11} is tridiagonal and all entries of T_{21} are zero except for its top right corner.

2. The submatrix A_{22} is updated as $A_{22} := A_{22} - UW^T - WU^T$ where, in order to exploit the symmetry, only the lower (or the upper) half of this matrix is updated.

The simultaneous computation of U and W along with the reduction in Operation 1 is needed to determine the first column of the unreduced part, which defines the Householder reflector. While U simply contains the vectors $u_j, u_{j+1}, \ldots, u_{j+b-1}$ of the Householder reflectors $H_j, H_{j+1}, \ldots, H_{j+b-1}$, more work is needed to determine W. In fact, the bulk of the computation in Operation 1 lays in the formation of this matrix. For each reduced column in the panel, a new column of W is generated. This requires four panel-vector multiplications and one symmetric matrix-vector multiplication with the submatrix A_{22} as operand. The latter operation, computed with the BLAS-2 SYMV kernel, is the most expensive one, requiring roughly $2(n-j)^2 b$ flops. Operation 2 also requires $2(n-j)^2 b$ flops, but is entirely performed by the BLAS-3 kernel SYR2K for the symmetric rank-$2b$ update. The overall cost of performing the reduction $A \to T$ using routine SYTRD is therefore $4n^3/3$ flops provided that $b \ll n$.

Note that there is no need to construct the orthogonal factor $Q = H_1 H_2 \cdots H_{n-2}$ explicitly. Instead, the vectors u_j defining the Householder reflectors H_j are stored in the annihilated entries of A. Additional work-space is needed to store the scalars β_j, but this requires only $O(n)$ entries and is thus negligible. If the eigenvectors are requested, the back-transform QX_T is computed in $2n^3$ flops without ever forming Q. Using the compact WY representation [31], this operation can be performed almost entirely in terms of calls to BLAS-3 kernels. LAPACK routine ORMTR provides this functionality.

4.7.3. Reduction to tridiagonal form. The SBR approach

The SBR toolbox is a software package for symmetric band reduction via orthogonal transforms. SBR includes routines for the reduction of dense symmetric matrices to banded form (SYRDB), and the reduction of banded matrices to narrower banded (SBRDB) or tridiagonal form (SBRDT). Accumulation of the orthogonal transforms and repacking routines for storage rearrangement are also provided in the toolbox.

In this section we describe the routines SYRDB and SBRDT which, invoked in that order, produce the same effect as the reduction of a dense matrix to tridiagonal form via LAPACK routine SYTRD. For the SBR routine SYRDB, we also describe how to efficiently off-load the bulk of the computations to the GPU.

Reduction to banded form

Assume that the first $j-1$ columns of the matrix A have already been reduced to banded form with bandwidth w. Let b denote the algorithmic block size, and assume for simplicity that $j+w+b-1 \leq n$ and $b \leq w$; see Figure 4.14. Then, during the current iteration of routine SYRDB, the next b columns of the banded matrix are obtained as follows:

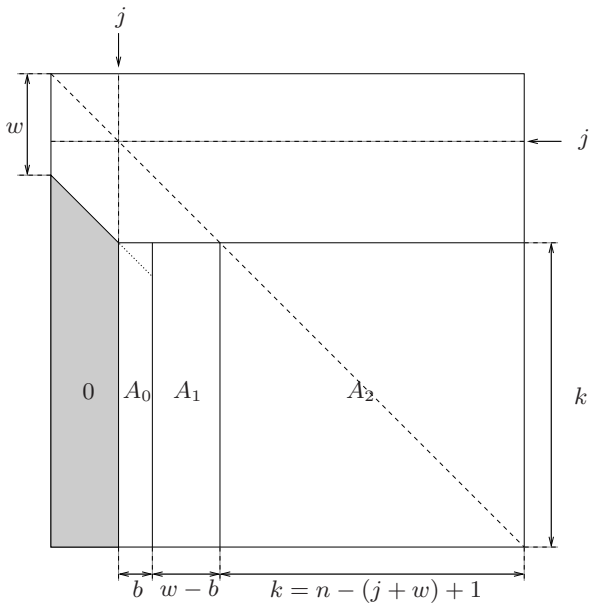


Figure 4.14: Partitioning of the matrix during one iteration of routine SYRDB for the reduction to banded form.

1. Compute the QR factorization of $A_0 \in \mathbb{R}^{k \times b}$, $k = n - (j + w) + 1$:

$$A_0 = Q_0 R_0, \tag{4.7}$$

where $R_0 \in \mathbb{R}^{b \times b}$ is upper triangular and the orthogonal factor Q_0 is implicitly stored as a sequence of b Householder vectors using the annihilated entries of A_0 plus b entries of a vector of length n. The cost of this first operation is $2b^2(k - b/3)$ flops.

2. Construct the factors of the compact WY representation of the orthogonal matrix $Q_0 = I + WSW^T$, with $W \in \mathbb{R}^{k \times b}$ and $S \in \mathbb{R}^{k \times k}$ upper triangular. The cost of this operation is about kb^2 flops.

3. Apply the orthogonal matrix to $A_1 \in \mathbb{R}^{k \times w-b}$ from the left:

$$A_1 := Q_0^T A_1 = (I + WSW^T)^T A_1 = A_1 + W(S^T(W^T A_1)). \tag{4.8}$$

By computing this operation in the order specified in the rightmost expression of (4.8), the cost becomes $4kb(w - b)$ flops. In case the bandwidth equals the block size ($w = b$), A_1 comprises no columns and, therefore, no computation is performed.

4. Apply the orthogonal matrix to $A_2 \in \mathbb{R}^{k \times k}$ from both the left and the right sides with $Y = WS^T$:

$$\begin{aligned} A_2 &:= Q_0^T A_2 Q_0 = (I + WY^T)^T A_2 (I + WY^T) && (4.9) \\ &= A_2 + YW^T A_2 + A_2 WY^T + YW^T A_2 WY^T. && (4.10) \end{aligned}$$

In particular, during this computation only the lower (or the upper) triangular part of A_2 is updated. In order to do so, (4.10) is computed as the following sequence of (BLAS) operations:

$$\text{(SYMM)} \quad X_1 := A_2 W, \tag{4.11}$$

$$\text{(GEMM)} \quad X_2 := \frac{1}{2} X_1^T W, \tag{4.12}$$

$$\text{(GEMM)} \quad X_3 := X_1 + Y X_2, \tag{4.13}$$

$$\text{(SYR2K)} \quad A_2 := A_2 + X_3 Y^T + Y X_3^T. \tag{4.14}$$

The major cost of Operation 4 is in the computation of the symmetric matrix product (4.11) and the symmetric rank-$2b$ update (4.14), each with a cost of $2k^2b$ flops. On the other hand, the matrix products (4.12) and (4.13) only require $2kb^2$ flops each. Therefore, the overall cost of this operation is approximately $4k^2b + 4kb^2$. This is higher than the cost of the preceding Operations 1, 2, and 3, which require $O(kb^2)$, $O(kb^2)$, and $O(\max(kb^2, kbw))$ flops, respectively. In summary, provided that b and w are both small compared to n, the global cost of the reduction of a full matrix to banded form is $4n^3/3$ flops. Furthermore, the bulk of the computation is performed in terms of BLAS-3 operations SYMM and SYR2K in (4.11) and (4.14), so that high performance can be expected in case a tuned BLAS is used.

The orthogonal matrix $Q_B \in \mathbb{R}^{n \times n}$ for the reduction $Q_B^T A Q_B = B$, where $B \in \mathbb{R}^{n \times n}$ is the (symmetric) banded matrix, can be explicitly constructed by accumulating the involved Householder reflectors at a cost of $4n^3/3$ flops. Once again, compact WY representations help in casting this computation almost entirely in terms of calls to BLAS-3. SBR implements this functionality in routine SYGTR.

Reduction to banded form on the GPU

Recent work on the implementation of BLAS and the major factorization routines for the solution of linear systems [20, 22, 142] has demonstrated the potential of GPUs to yield high performance on dense linear algebra operations which can be cast in terms of matrix-matrix products. In this subsection we describe how to exploit the GPU in the reduction of a matrix to banded form, orchestrating the computations carefully to reduce the number of data transfers between the host and the GPU.

During the reduction to banded form, Operation 4 is a natural candidate for being computed on the GPU, while, due to the kernels involved in Operations 1 and 2 (mainly narrow matrix-vector products), these computations are better suited for the CPU. Operation 3 can be performed either on the CPU or the GPU but, in general, $w - b$ will be small so that this computation is likely better suited for the CPU. Now, assume that the entire matrix resides in the GPU memory initially. We can then proceed to compute the reduced form by repeating the following three steps for each column block:

1. Transfer A_0 and A_1 back from GPU memory to main memory. Compute Operations 1, 2, and 3 on the CPU.
2. Transfer W and Y from main memory to the GPU.
3. Compute Operation 4 on the GPU.

Proceeding in this manner, upon completion of the algorithm, most of the banded matrix and the Householder reflectors are available in the main memory. Specifically, only the diagonal $b \times b$ blocks in A remain to be transferred to the main memory.

The construction of Q_B that produces the reduction to band form can also be easily done in the GPU, as this computation is basically composed of calls to BLAS-3 kernels.

Reduction to tridiagonal form

Routine SBRDT in the SBR toolbox is responsible for reducing the banded matrix B to tridiagonal form by means of Householder reflectors. Let Q_T denote the orthogonal transforms which yield this reduction, that is $Q_T^T B Q_T = T$. On exit, the routine returns the tridiagonal matrix T and, upon request, accumulates these transforms, forming the matrix $Q = Q_B Q_T \in \mathbb{R}^{n \times n}$ so that $Q^T A Q = Q_T^T (Q_B^T A Q_B) Q_T = Q_T^T B Q_T = T$.

The matrix T is constructed in routine SBRDT one column at the time: at each iteration, the elements below the first sub-diagonal of the current column are annihilated using a Householder reflector; the reflector is then applied to both sides of the matrix, resulting in a bulge which has to be chased down along the band. The computation is cast in terms of BLAS-2 operations at best (SYMV and SYR2 for two-sided updates, and GEMV and GER for one-sided updates) and the total cost is $6n^2w + 8nw^2$ flops.

Let Q denote the product of all the Householder reflectors needed to obtain T from B. These transforms are usually accumulated by applying the corresponding sequence of reflectors to the orthogonal matrix that results from the reduction of A to the band matrix B. These transformations can be applied in a blocked fashion by exploiting the WY representation [70]. The accumulation of Q is therefore entirely performed by the fast BLAS-3 GEMM kernel.

If the eigenvectors are desired, then the orthogonal matrix Q_B produced in the first step (reduction from full to banded form) needs to be updated with the orthogonal transforms computed during the reduction from banded to tridiagonal form building $Q_B Q_T$. This accumulation requires $O(n^3)$ flops and can be reformulated almost entirely in terms of calls to level-3 BLAS kernels, even though this reformulation is less trivial than for the first step [32]. However, the matrix $Q = Q_B Q_T$ still needs to be applied as part of the back-transform stage, to obtain $X := QX_T$, adding $2n^3$ flops to the cost of building the matrix containing the eigenvectors of A.

We do not propose to off-load the reduction of the banded matrix to tridiagonal form on the GPU as this is a fine-grained computation which does not lend itself to an easy implementation on this architecture. However, the accumulation of the orthogonal factor produced by this step and the back-transform merely require operations alike the matrix-matrix product, which can be efficiently computed on the GPU.

4.7.4. Experimental Results

Evaluating the performance of the routines for the reduction to tridiagonal form is an elaborate task, due to the large number of factors that have an influence on it. Among them, we will focus on block size, bandwidth for the SBR toolbox, and number of cores. In the following experiments we aim at determining the optimal configuration of these parameters, before proceeding to show a full comparison of the two approaches. For simplicity, we only report results for four problem sizes: 2,048, 6,144, 10,240 and 24,576, and the best bandwidths (w) and block sizes (b) detected in each case, though a complete experiment was carried out for many other values.

Building BLAS-2 and BLAS-3 kernels

We start by analyzing the performance of the low-level kernels involved in the reduction to condensed forms (tridiagonal and banded for the LAPACK and SBR approaches, respectively). For the LAPACK routine SYTRD, these are the symmetric matrix-vector product (SYMV) and the

symmetric rank-2k update (SYR2K). For the SBR routine SYRDB, the kernels are the symmetric matrix-matrix product (SYMM) and SYR2K. Table 4.3 reports results on 1, 4 and 8 cores of the Intel Xeon *Nehalem* processors in PECO. For the BLAS-3 kernels (SYMM and SYR2K), we also report the performance on the single GPU in this platform using the implementation in CUBLAS as well as our own implementations, which cast most computations in terms of the general matrix-matrix product [83] (column labeled as "Our BLAS"). The matrix dimensions of SYR2K and SYMM are chosen so that they match the shape of the blocks encountered during the reduction to condensed forms (n is the problem size, while k plays the role of the block size b for SYTRD and that of the bandwidth w for SYRDB). For reference, we also include the performance of the general matrix-vector product (GEMV) and matrix-matrix product (GEMM) kernels.

The performance of SYMV increases with the number of cores and is significantly higher than that of GEMV. When the problem size is $n = 2{,}048$, the matrix fits into the L3 caches of PECO (8 MB) which explains the much higher GFLOPS rate of SYMV. The same does not hold for GEMV as all the matrix needs to be stored and not just half of it (lower or upper triangle). We will consider these numbers as the basis for our investigation; in other words, justifying the figures yielded by the particular implementation of these kernels in the multi-threaded BLAS implementation is not the purpose of this dissertation.

The two BLAS-3 kernels involved in the computation increase their GFLOPS rate with the number of cores on PECO. This behavior plays a key role in the global performance of the whole reduction process on this architecture.

The results also illustrate the higher performance delivered by the GPU for most BLAS-3 kernels. Although our own implementations of the symmetric BLAS-3 kernels for the GPU deliver higher GFLOPS rates than those from CUBLAS, they are still quite below the performance of the matrix-matrix product kernel in CUBLAS. However, in all cases the performance attained by the GPU is higher than that delivered by the multi-core CPU. The improvement in the performance of the SYMM kernel is of special relevance for the reduction to tridiagonal form using the SBR routines.

The LAPACK approach

We next analyze the gains of a multi-core execution of the LAPACK routines SYTRD and, in case QX_T is required, ORMTR. Table 4.4 reports the execution time (in seconds) for different values of problem size, block size (b) and number of cores.

Let us comment first on the routine that performs the reduction from full to tridiagonal form, SYTRD. The block size does not play a role in the performance of it. Increasing the number of cores yields a reduction in the execution time on PECO but with moderate speed-up; for example, 3× and 4.2× speed-ups are attained for the largest problem sizes using, respectively, 4 and 8 cores.

The accumulation of the orthogonal transformations via routine ORMTR requires less time and is, in general, more efficient than the reduction stage. This is to be expected, as most of the computations in the accumulation are cast in terms of BLAS-3 kernels (GEMM), while only half of those in SYTRD are BLAS-3. Representative speed-ups of routine ORMTR in PECO are 3.7× and 6.6×, attained respectively using 4 and 8 cores for the largest problem size. Note also that this routine benefits from using block sizes (e.g., 192 and 256) much larger than those that resulted optimal for SYTRD.

The SBR approach

We next study the parallelism of the two-step SBR approach: reduction of a general matrix to band form (SYRDB) and subsequent reduction to tridiagonal form (SBRDT). Also, we include in the

	SYMV. $y := Ax + y$ $A \in \mathbb{R}^{n \times n}$ symmetric, x,$y \in \mathbb{R}^n$			GEMV. $y := Ax + y$ $A \in \mathbb{R}^{n \times n}$, x,$y \in \mathbb{R}^n$		
n	1 core	4 cores	8 cores	1 core	4 cores	8 cores
2048	8.26	34.5	60.0	5.15	8.47	8.31
6144	7.80	17.7	21.6	5.07	9.13	10.7
10240	6.69	18.3	22.1	4.70	9.26	11.2
24576	5.75	16.0	21.0	3.16	8.45	10.8

		SYR2K. $C := AB^T + BA^T + C$ A,$B \in \mathbb{R}^{n \times k}$, $C \in \mathbb{R}^{n \times n}$ symmetric					GEMM. $C := AB^T + C$ A,$B \in \mathbb{R}^{n \times k}$, $C \in \mathbb{R}^{n \times n}$			
n	k	1 core	4 cores	8 cores	CUBLAS	Our BLAS	1 core	4 cores	8 cores	CUBLAS
2048	32	14.0	55.4	91.0	53.2	53.2	15.0	58.6	116.4	157.2
	64	15.5	62.5	116.3	74.4	159.2	16.7	65.9	129.3	185.5
	96	16.5	65.3	122.2	78.0	162.9	17.3	68.4	135.9	192.2
6144	32	13.6	50.3	89.6	55.9	56.0	15.0	59.6	106.9	161.0
	64	15.7	59.8	112.3	78.4	124.2	16.7	66.6	123.1	185.0
	96	16.8	64.4	122.6	81.8	126.3	17.4	69.2	137.8	195.1
10240	32	13.8	51.2	83.9	56.4	56.4	15.0	59.6	116.5	159.3
	64	15.8	60.9	113.8	79.2	114.2	16.7	66.7	130.6	182.2
	96	16.9	65.3	123.7	78.1	116.7	17.5	69.4	137.7	187.3
24576	32	13.9	51.0	89.9	57.4	53.4	14.7	58.3	108.9	156.0
	64	16.0	60.9	113.6	79.1	116.9	16.6	64.6	129.9	186.2
	96	16.5	65.1	123.9	83.4	112.2	17.3	68.8	137.5	189.9

		SYMM. $C := AB + C$ $A \in \mathbb{R}^{n \times n}$ symmetric, B,$C \in \mathbb{R}^{n \times k}$					GEMM. $C := AB + C$ $A \in \mathbb{R}^{n \times n}$, B,$C \in \mathbb{R}^{n \times k}$			
n	k	1 core	4 cores	8 cores	CUBLAS	Our BLAS	1 core	4 cores	8 cores	CUBLAS
2048	32	12.2	29.7	46.9	89.7	106.5	15.0	59.9	99.2	177.5
	64	14.7	45.2	75.4	97.1	183.4	16.7	66.2	105.3	279.0
	96	15.7	49.3	80.4	97.9	189.4	17.4	68.7	133.5	290.1
6144	32	11.9	28.5	42.9	94.1	129.6	15.1	59.5	116.7	327.5
	64	14.5	43.9	71.8	99.1	188.4	16.8	66.5	132.2	339.3
	96	15.6	48.5	77.5	100.4	198.1	17.5	69.3	132.2	338.2
10240	32	11.1	25.3	39.4	76.0	113.5	15.0	59.5	116.7	321.9
	64	13.9	40.5	66.6	76.5	175.8	16.8	66.6	132.4	346.9
	96	15.2	45.3	73.4	77.5	180.0	17.4	69.4	135.8	348.0
20480	32	10.8	24.8	37.9	77.8	110.0	14.7	58.3	108.4	328.0
	64	13.5	39.3	63.3	66.8	176.7	16.6	66.0	131.3	344.9
	96	15.0	44.6	65.7	65.9	179.0	17.3	68.9	135.3	346.0

Table 4.3: Performance (in GFLOPS) of the BLAS kernels SYMV (top), SYR2K (middle) and SYMM (bottom) and the corresponding matrix-vector and matrix-matrix products (for reference) on PECO. Peak performance for 1, 4 and 8 cores of this platform are 18.2, 72.6 and 145.3 GFLOPS, respectively.

n	SYTRD: Full→Tridiagonal				ORMTR: Accum. QX_T			
	b	1 core	4 cores	8 cores	b	1 core	4 cores	8 cores
2048	32	1.11	0.34	0.23	128	1.22	0.41	0.27
	64	1.11	0.35	0.23	192	1.23	0.41	0.28
	96	1.11	0.36	0.25	256	1.27	0.41	0.27
6144	32	40.4	11.4	9.2	128	28.8	8.60	5.20
	64	39.9	11.3	8.4	192	28.2	8.32	5.09
	96	40.1	11.3	10.4	256	29.0	8.46	5.08
10240	32	156.3	52.4	40.6	128	128.5	36.6	21.1
	64	152.5	51.9	40.5	192	127.4	35.9	21.1
	96	152.6	52.1	40.9	256	126.4	35.3	21.9
24576	32	2522.1	812.5	590.3	128	1767.1	488.1	275.2
	64	2453.6	796.8	600.9	192	1732.0	471.5	272.0
	96	2444.4	795.0	582.4	256	1720.0	466.0	262.7

Table 4.4: Execution time (in seconds) for the LAPACK routine(s) on PECO.

analysis the routines that construct the orthogonal factor Q (SYGTR to build Q_B and SBRDT to accumulate $Q := Q_B Q_T$) and compute QX_T (GEMM) in the back-transform. Remember that while the computational cost of the first step is inversely proportional to the bandwidth, w, the cost of the second step is directly proportional to it. In other words, a larger bandwidth requires a smaller number of computations for the first step, transferring a larger part of the flops to the second step.

Table 4.5 displays results for these experiments. For the discussion, consider the following five cases:

1. Reduction of a dense matrix to banded form (Step 1). On the multi-core, the usage of a larger number of cores or the increase of the bandwidth results in a smaller execution time. The execution time on the GPU, on the other hand, is quite independent of w and outperforms the Intel-based architectures for all problem sizes except $n = 2{,}048$.

2. Reduction of banded matrix to tridiagonal form (Step 2). Using more than a single core yields no gain. As expected, a larger value of w translates into a longer execution time of this step.

3. Building the orthogonal factor resulting from Case 1 (Step 1). On the Intel cores, the execution time and parallelism of routine SYGTR is quite similar to those of SYRDB discussed in Case 1. Compared with the results obtained on 8 cores of PECO, the GPU in this platform accelerates the execution by a considerable factor, between 2.5×–3×.

4. Accumulating the orthogonal transforms corresponding to Case 2 (Step 2). By far, this is the most expensive operation of the five cases in the Intel cores, though it exhibits a certain degree of parallelism, which helps in reducing its weight on the overall process. The speed-up attained by the GPU for the larger problem dimensions is impressive.

5. Back-transform. The cost of this operation is comparable to that in Case 3. The best results is always attained with 8 cores and the GPU yields a notable acceleration.

Note that a study needs to take into account that the choice of bandwidth cannot be done independently from other factors. Therefore, we delay further comments on the data in the previous tables to the next subsection. There, we elaborate on the optimal combination of the factors that determine the overall performance of this approach.

		1st step (SYRDB): Full→Band				2nd step (SBRDT): Band→Tridiagonal		
n	w	1 core	4 cores	8 cores	GPU	1 core	4 cores	8 cores
2048	32	0.89	0.34	0.23	0.21	0.37	1.64	1.72
	64	0.81	0.28	0.19	0.20	0.45	1.08	1.03
	96	0.80	0.27	0.19	0.22	0.57	0.90	0.91
6144	32	23.6	8.3	5.2	2.78	3.48	14.93	17.1
	64	20.8	6.2	3.7	2.27	4.88	9.92	10.1
	96	19.9	5.9	3.6	2.29	5.42	8.23	8.91
10240	32	112.6	41.1	26.7	10.81	9.51	41.1	43.3
	64	95.9	29.6	18.7	9.72	11.7	27.5	26.3
	96	90.9	27.3	16.1	10.39	15.1	23.1	25.3
24576	32	1589.9	569.0	354.3	112.6	54.2	237.3	258.0
	64	1330.2	404.3	235.5	99.3	72.9	159.2	157.7
	96	1251.2	370.7	220.5	105.3	96.8	133.3	140.3

		1st step (SYGTR): Build Q_B				2nd step (SBRDT): Accum. $Q := Q_B Q_T$				Back-transform (GEMM): Comp. QX_T	
n	w	1 core	4 cores	8 cores	GPU	1 core	4 cores	8 cores	GPU	8 cores	GPU
2048	32	0.81	0.33	0.25	0.07	2.31	1.28	1.38	0.76		
	64	0.73	0.26	0.20	0.04	1.86	0.83	0.55	0.42	0.12	0.07
	96	0.70	0.25	0.19	0.03	1.61	0.54	0.36	0.26		
6144	32	21.2	7.35	7.04	1.68	65.4	33.0	35.7	6.24		
	64	19.0	5.83	3.86	1.77	51.8	22.1	14.5	3.09	3.36	1.50
	96	18.3	5.62	3.67	1.75	44.3	14.5	9.5	1.74		
10240	32	97.5	32.5	22.7	6.81	291.0	150.8	163.4	32.4		
	64	87.5	25.6	17.2	6.44	235.1	102.3	66.6	12.6	15.0	6.46
	96	84.1	24.7	16.5	5.61	203.1	67.2	43.9	6.27		
24576	32	1399.0	456.8	310.7	94.6	4149.5	2166.7	2403.4	101.8		
	64	1232.0	353.0	217.3	88.0	3390.2	1465.0	956.6	55.3	209.5	89.3
	96	1177.0	377.7	207.6	81.0	2898.4	969.9	638.8	30.9		

Table 4.5: Execution time (in seconds) for the SBR routines on PECO.

	Reduction to tridiagonal form		
	LAPACK	SBR	
n	PECO	PECO	PECO+GPU
2048	0.23	0.6	0.58
6144	8.4	8.58	6.26
10240	40.5	30.4	20.32
24576	582.4	308.4	166.8

	Reduction to tridiagonal form and back-transform		
	LAPACK	SBR	
n	PECO	PECO	PECO+GPU
2048	0.50	1.65	1.39
6144	13.5	25.6	14.6
10240	61.6	101.8	47.5
24576	845.1	1207.2	314.0

Table 4.6: Comparison of the execution time (in seconds) for the the LAPACK and SBR routines on PECO and SBR accelerated by the GPU on PECO.

Comparing the two approaches

Although the routines that tackle the symmetric eigenvalue problem are structured as a sequence of steps, these are not independent. Therefore, in general the tuning of parameters for each step cannot be done separately. For example, the bandwidth has to be kept constant through all the routines involved in the reduction. The block size, instead, can be adjusted for each routine. Additionally, on the multi-core processors, one may choose the degree of parallelism for each routine by fixing the number of threads employed for its execution. For example, consider the reduction to tridiagonal form of a problem of size $n = 10240$ when performed on the multi-core in PECO using the SBR routines. For bandwidths $w = 32$, 64 and 96, the best timings for the reduction to banded form using the corresponding SBR routine are 112.6, 29.6, and 16.1 seconds, using 1, 4 and 8 cores, respectively. The cost for the next stage, reduction from banded to tridiagonal form, is minimized when a single core is used, resulting in 9.51, 11.7 and 15.1 seconds for bandwidths 32, 64 and 96, respectively. Overall, the best combination, totaling 31.2 seconds, corresponds to bandwidth 64, using 8 cores for the first step and a single core for the second.

In Table 4.6, we collect results for an experimental comparison of the two approaches on both architectures: PECO, and the GPU in this platform for all steps except the reduction from banded to tridiagonal form using the SBR routines (labeled as "PECO+GPU"). For small and medium problem sizes LAPACK is the fastest approach. For the largest dimensions, the SBR approach greatly benefits from the acceleration enabled by the GPU, and outperforms LAPACK both in the reduction and back-transform stages.

In the reduction stage, the GPU delivers speedups of 1.5× and 1.8× for the two largest problem sizes compared with the best options (SBR or LAPACK) on any of the two Intel-based architectures. When the back-transform is also required, the speedups for these problem sizes become 1.3× and 2.7×.

4.8. Conclusions

In this chapter, we have demonstrated how the GPU can be a reliable approach to the acceleration of higher-level dense linear algebra routines. As driving examples, we have chosen representa-

tive and widely used LAPACK-level operations to illustrate a number of techniques that, following a high-level approach, improve the performance of the implementations.

In the first part of the chapter, we have addressed the Cholesky and LU with partial pivoting factorizations. The usage of a high-level approach allows us to systematically derive and implement a number of algorithmic variants. Among them, it is possible to choose the most convenient one for a given architecture or BLAS implementation.

The implementation of blocked and unblocked routines for both operations have yield a collection of conclusions that result from the study developed in the chapter. First, the usage of blocked implementations is a must on current graphics processors. The properties of modern GPUs transform them into a platform of special appeal for blocked computations. However, the capabilities of general-purpose multi-core processors when operating with small datasets is one of their main strengths. This divergence naturally drives to the design and development of hybrid algorithms, in which CPU and GPU collaborate in the solution of a problem. In our case, the usage of a hybrid approach has been successfully applied to both operations. Experimental results validate the advantadges of the solution.

Double precision is most often required in scientific codes. In our case, we have identified the poor performance of modern GPUs when operating on double-precision data. To solve this drawback in the context of the solution of systems of linear equations, we propose a mixed-precision iterative refinement approach, in which the major part of the computation is performed using single precision, but CPU and GPU collaborate to regain double-precision accuracy. Experimental results show that this approach can exploit the performance of modern GPUs when they operate using single-precision arithmetic while delivering the accuracy of double precision.

For the symmetric eigenvalue problem, we have evaluated the performance of existing codes for the reduction of a dense matrix to tridiagonal form and back-transform. Our experimental results confirm that the two-stage approach proposed in the SBR toolbox (reduction from full to banded form in the first stage followed by a reduction from banded to tridiagonal form in a second stage) delivers a higher parallel scalability than the LAPACK-based alternative on general-purpose multi-core architectures. However, when the orthogonal factors that define the back-transform have to be constructed and applied in the last stage, this results in a computation time considerably larger than that of LAPACK.

The use of a hardware accelerator like a GPU changes the message dramatically. By off-loading the level-3 BLAS operations in the SBR codes to the GPU, remarkable speed-ups are attained to the point that the SBR toolbox becomes a competitive alternative to the standard LAPACK-based algorithm. The reward did not come effortless, though. Specifically, the advantages came from two improvements: First, the application of the routines developed in Chapter 3 for the rank-2k and symmetric matrix-matrix product; second, a careful modification of the SBR routines to exploit the hardware elements of the hybrid CPU-GPU architecture and to minimize the number of data transfers between the host and the device memory spaces.

Part III

Matrix computations on multi-GPU systems

CHAPTER 5

Matrix computations on multi-GPU systems

Although graphics processors deliver a performance that only a few years ago was difficult to attain even using the most advanced distributed-memory machines, HPC necessities are in constant growth. In response to these requirements, computing systems with more than one hardware accelerator attached to them are the next evolutionary step in the GPU computing arena.

These new platforms offer appealing raw peak performance, but also pose a big challenge for the programmers and library developers. In this sense, the dilemma is to rewrite the existing libraries to adapt them to this new type of architectures, or reuse concepts successfully applied in shared- and distributed-memory programming for years and apply to these novel platforms.

Following with the approach proposed in previous chapters, our aim is to easily adapt existing libraries to new platforms and, more specifically, to machines with several GPUs or other type of hardware accelerators. However, the quest for programmability often implies a performance sacrifice. In this chapter, we pursue a straightforward adaptation of existing linear algebra codes to modern multi-GPU platforms while maintaining the performance.

To achieve this goal, we advocate for rescuing techniques and concepts applied to parallel architectures and programming for years, like task-parallelism, out-of-order execution, data-affinity, software-managed caches, algorithms-by-blocks, hierarchical block storage schemes or dynamic scheduling. Therefore, the goal is to renew and apply these techniques to multi-GPU systems, with a small influence in the final library codes, but a great impact on the final performance results.

In this chapter, we demonstrate how, starting from algorithms and codes designed for a sequential execution, it is possible to exploit the potential performance of modern multi-GPU systems. The ultimate motivation is to avoid the need of rebuilding existing dense linear algebra libraries from scratch to adapt them to this novel architecture.

Our solution consists in designing and implementing a run-time system capable of extracting parallelism and orchestrate the execution of parallel codes on multiple graphics processors. Performance results validate the solution for many well-known linear algebra implementations, from both performance and scalability perspectives. In addition, following the FLAME philosophy, the programmer is not aware of the architectural details of the underlying platforms, and thus existing codes are easily parallelized and adapted to the new architecture.

The chapter is structured as follows. Section 5.1 describes the programming model chosen for the multi-GPU parallelization of existing codes, and introduces the main differences with other well-known programming models for parallel environments. Section 5.2 introduces some common concepts regarding algorithm-by-blocks, out-of-order execution, and runtime support for parallel execution adopted by our solution. In Section 5.3 a detailed description of the run-time system is given, together with the necessary modifications introduced in order to boost performance. The improvements in performance are reported in Section 5.4 for the Cholesky factorization and in Section 5.5 for a set of BLAS operations. Finally, Section 5.6 reviews the main contributions of the chapter and summarizes the most remarkable insights extracted from it.

All experiments presented through the chapter were carried out using a multi-GPU system (Tesla S1070) on tesla2. The specific hardware and software details of the experimental setup were presented in Section 1.3.2.

5.1. Programming models for multi-GPU systems

5.1.1. Programming models for multi-core systems

In response to the increasing interest in multi-core architectures during the past decade, many approaches have been proposed to improve the programmer productivity and to adapt the existing codes to these novel architectures. In some of these programming models, the user is responsible of exposing parallelism, and a run-time systems exploits it, tailoring the execution to the specific underlying architecture.

Cilk [34] is a multi-threaded runtime system that allows the programmer the exposition of elements that can be safely executed in parallel. At run time, a scheduler is responsible of dividing the work into tasks and mapping them to the available computing resources. Based on ANSI-C with minor extensions, the modification of the serial code is minimal to adapt it to multi-core architectures. The scheduling in Cilk is based on work stealing [35] where tasks are assigned to threads, but idle threads can steal tasks from busy ones. Intel Thread Building Blocks (TBB) [122] follows a similar approach, also using work stealing. Both models defer the responsibility of managing data dependencies to the user, providing special constructs for identifying independent operations that can be executed in parallel.

OpenMP [47] is an extended and standard programming model that uses preprocessor directives (or `pragmas`) to denote parallel regions in the code. OpenMP 3.0 has introduced a work queuing mechanism [98] where tasks can be dynamically placed onto a task queue from which idle threads dequeue tasks to execute. Once more, the work queuing model relies on the user to identify data dependencies between tasks.

In a similar way, SMP Superscalar (SMPSs) [109] offers an adaptation of the StarSs programming model for multi-core architectures and a runtime system that also uses preprocessor directives to identify tasks. With a small set of OpenMP-like pragmas, the SMPSs compiler infrastructure allows users to annotate parts of the codes that will be considered as tasks, or minimum scheduling units. The pragmas provide information about the input and output data of each task in order to detect data dependencies at run time, and thus dynamically build a DAG and dispatch tasks to the available computing units. SMPSs inherits many of their features from CellSs [24, 110, 25], which demonstrates the portability of these type of programming models and run-time systems to other type of modern multi-threaded architectures.

The SuperMatrix runtime follows the philosophy of the FLAME project and performs the adaptation of `libflame` [149] to modern multi-core architectures. It shares many of the ideas implemented in the models previously described, focusing on the automatic parallelization of

dense [116, 117] and banded [119] linear algebra algorithms. The framework exploits the benefits of storing and indexing matrices by blocks and algorithms-by-blocks in general, applying techniques for dynamic scheduling and out-of-order execution (common features of superscalar processors) and a systematic management of data dependencies at run-time [41]. This type of techniques aim at reducing the impact of the parallelization on the programmer's code and reducing the effort of library developers.

Traditional approaches in the linear algebra domain aimed at extracting the parallelism at the BLAS level, as many operations are usually built on top of these routines. Thus, the efficient execution of serial codes on parallel architectures relied on the skills of the BLAS programmers, usually with a deep knowledge of the architectural details of the target platform.

More recently, the FLAME project has advocated and shown the benefits of a different approach: extracting the parallelism at a higher level, so that only a sequential, tuned implementation of the BLAS routines is necessary to be executed on each core. This is the approach from which we benefit to create a run-time system that efficiently adapts to an absolutely different and novel architecture, as explained in the following section.

5.1.2. Adaptation to multi-GPU systems

The idea underlying the adaptation of an existing programming model and run-time system to a multi-GPU platform inherits the concept introduced by previous models focused on multi-core architectures. For systems with multiple GPUs sharing common central resources, a possibility explored in [141] is to distribute the data among the GPU memories, and code in a message-passing style similar to that of the libraries ScaLAPACK [33] and PLAPACK [7]. It is possible to identify four main drawbacks in this approach:

- While the state-of-the-art numerical methods have not changed, following this approach will require a complete rewrite of dense linear algebra libraries (similar to the redesign of LAPACK for parallel distributed-memory architectures that was done in the ScaLAPACK and PLAPACK projects). Therefore, a considerable programming effort is necessary to cover a functionality like that of LAPACK. Note that coding at such low level can be quite complex and, therefore, programmability becomes the main burden towards the adaptation of the codes to novel architectures.

- The product that is obtained as a result of this style of programming will likely exhibit a parallelism similar to that of libraries based on multi-threaded implementations of the BLAS and far from that demonstrated by the dynamic scheduling techniques in the FLAME, PLASMA [36], Cilk, and SMPSs projects. While look-ahead techniques [127] can increase the scalability of this solution to a certain point, they do so at the cost of a much higher complexity.

- The adaptation of the new codes to future many-core architectures implies a total rewriting, and even minor variations in the existing architectures can imply deep modifications in the implementations.

- Message-passing is not an optimal programming paradigm for shared-memory architectures.

Our approach in this context is fundamentally different. In previous works [43, 45, 46, 116, 115, 117, 118], an overview of software tools and methods developed as part of the FLAME project is given; in these references, it is also shown how, when applied to a platform with multiple

cores/processors, the FLAME infrastructure provides an out-of-the-box solution that attains high performance almost effortlessly for the library developer. The key lies in maintaining a separation of concerns between the code and the target architecture by leaving the parallel execution of the operation in the hands of a runtime system.

The advantages of this approach are twofold:

- When a new platform appears, it is only the runtime system that needs to be adapted. The routines in the library, which reflect the numerical algorithms, do not need to be modified.
- The parallelism is identified and exploited by a runtime system which can be adapted to exploit different architectures.

We demonstrate how the FLAME framework provides the necessary tools and techniques to easily port existing sequential linear algebra codes to platforms with multiple hardware accelerators in general, and to architectures with multiple graphics processors in particular. In this sense, the object-based approach offered by `libflame`, together with the ease of identification of tasks in FLAME codes, the hierarchical block storage provided by the FLASH API and the existing infrastructure in SuperMatrix, transforms FLAME into the ideal infrastructure for programming alternative architectures.

One of the main contributions of the work in this chapter is the reformulation of multi-GPU systems, viewing them as as a multi-core system. Specifically, we consider this type of platforms a pool of executing units, considering each accelerator as only one "core", exactly in the same way as a multi-core processor is viewed by the programming models exposed above. Following with the analogies with programming models for multi-core architectures, the major need to exploit task-level parallelism is a high performance implementation of the kernels that execute the task code. Taking as an example linear algebra algorithms, NVIDIA CUBLAS (or our own improvements of NVIDIA CUBLAS introduced in previous chapters) can play the same role as the sequential BLAS implementations that run on each core do on multi-core architectures.

With this approach, certain parts of the infrastructure already developed can be taken from existing models (e.g., programming model, run-time structure, thread deployment,...) and many well-known techniques usually applied for multi-cores can be successfully applied to multi-GPU systems. Even more important, this view abstracts the programmer and the sequential codes from the underlying architecture, and exactly the same techniques can be applied to a wide variety of multi-accelerator platforms.

However, there are important differences between both classes of architectures that are directly translated into important modifications in the runtime system design and implementation in order to adapt it to modern multi-GPU architectures. The major one is directly derived from the existence of separated memory spaces in each accelerator and independent from main memory. Each GPU can only access data located in its own private memory, and communication between GPUs is only possible through main memory (see Figure 5.1). Thus, in a naive implementation of a run-time for this architecture, explicit communication instructions have to be added to the programming model to deal with data transfers and coherence between memory spaces. Naturally, this approach has two important limitations:

- *Programmability*: The programmer has to be aware of the existence of several independent memory spaces, and deal with implementation details such as integrity and data coherence. This dramatically affects the main philosophy of the FLAME project and the abstraction of linear algebra codes from the underlying architecture.

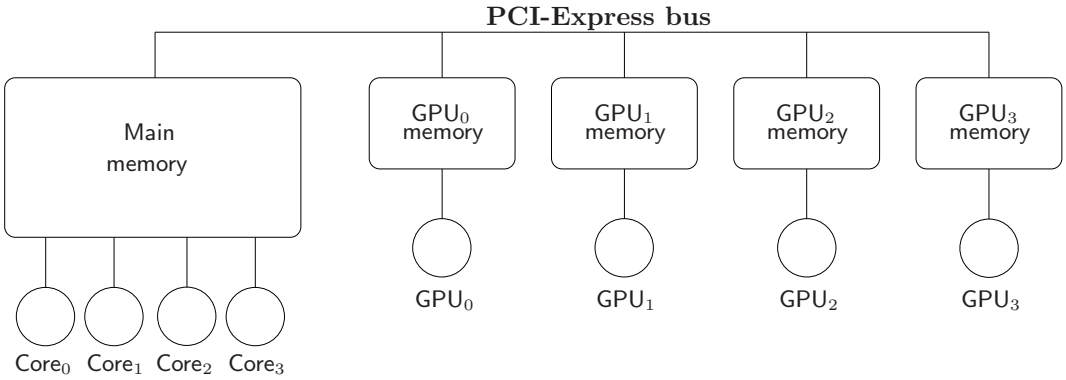


Figure 5.1: Schematic architecture of a multi-GPU system. Each GPU operates with data stored in its own private memory. Communication between GPUs is performed through main memory. Memory spaces are different for each GPU in the system, and for the main memory.

- *Performance*: Even with the programmer in charge of data transfers, some decisions to reduce the amount of transfers are hard to take at development/compilation time, and can only be optimized at runtime.

The developed runtime is in charge of managing data dependencies, task scheduling and the general parallelization of the sequential codes, and also of the management of data transfers. Proceeding in this manner, the runtime is responsible of maintaining data coherence among the different memory spaces, and applying effective techniques to reduce the impact of data transfers on the overall parallel performance. This approach completely abstracts the user from the underlying architecture, and thus the existence of separate memory spaces is fully hidden to the developer and transparent to the codes.

5.2. Linear algebra computation on multi-GPU systems

In this section, we introduce some key concepts that, when used in combination, drive to a natural and efficient adaptation of existing linear algebra codes to multi-GPU systems. These concepts include known techniques such as storage-by-blocks and algorithms-by-blocks, that naturally map algorithms to systems that exploit task-level parallelism. Dynamic scheduling and out-of-order execution are key techniques inherited from the superscalar processors that are now reintroduced in software as a way of automatically parallelizing sequential algorithms-by-blocks on novel architectures. Finally, the key concerns taken into account in the design and implementation of the run-time system, together with the key differences with existing systems are exposed and justified.

5.2.1. Storage-by-blocks and algorithms-by-blocks

Storage of matrices by blocks, where sub-matrices are physically stored and referenced in a hierarchical fashion, presents a few important advantages: a better exploitation of data locality [121], and thus better use of the memory hierarchy in modern architectures, and a more compact storage of matrices with a special structure [8].

From the algorithmic perspective, previous works have advocated for a new alternative to blocked algorithms and storage-by-columns proposed in LAPACK. This alternative, known as

```
FLA_Error FLASH_Chol_by_blocks_var1( FLA_Obj A )
{
  FLA_Obj ATL, ATR,      A00, A01, A02,
          ABL, ABR,      A10, A11, A12,
                         A20, A21, A22;

  FLA_Part_2x2( A,    &ATL, &ATR,
                      &ABL, &ABR,    0, 0, FLA_TL );

  while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) ) {
    FLA_Repart_2x2_to_3x3(
       ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
    /* ************* */   /* ******************** */
                            &A10, /**/ &A11, &A12,
       ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
       1, 1, FLA_BR );
    /*----------------------------------------------*/
    FLA_Chol_blk_var1( FLASH_MATRIX_AT( A11 ) );
    FLASH_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
                FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
                FLA_ONE, A11,
                         A21 );
    FLASH_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
                FLA_MINUS_ONE, A21,
                FLA_ONE,       A22 );
    /*----------------------------------------------*/
    FLA_Cont_with_3x3_to_2x2(
       &ATL, /**/ &ATR,        A00, A01, /**/ A02,
                               A10, A11, /**/ A12,
    /* *************** */   /* ****************** */
       &ABL, /**/ &ABR,        A20, A21, /**/ A22,
       FLA_TL );
  }
  return FLA_SUCCESS;
}
```

```
void FLASH_Trsm_rltn( FLA_Obj alpha, FLA_Obj L,
                                     FLA_Obj B )
/* Special case with mode parameters
     FLASH_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
                 FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
                 ...                              )
   Assumption: L consists of one block and
               B consists of a column of blocks     */
{
  FLA_Obj BT,      B0,
          BB,      B1,
                   B2;

  FLA_Part_2x1( B,    &BT,
                      &BB,    0, FLA_TOP );

  while ( FLA_Obj_length( BT ) < FLA_Obj_length( B ) ) {
    FLA_Repart_2x1_to_3x1( BT,       &B0,
                         /* ** */   /* ** */
                                    &B1,
                           BB,       &B2,    1, FLA_BOTTOM );
    /*----------------------------------------------*/
    FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
              FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
              alpha, FLASH_MATRIX_AT( L ),
                     FLASH_MATRIX_AT( B1 ) );
    /*----------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &BT,        B0,
                              B1,
                            /* ** */   /* ** */
                              &BB,        B2,    FLA_TOP );
  }
}
```

Figure 5.2: FLASH implementation of the algorithm-by-blocks for Variant 1 of the Cholesky factorization (left) and FLASH implementation of the function `FLASH_Trsm` involved in the algorithm (right).

algorithms-by-blocks [61], considers matrices as a hierarchical collection of sub-matrices, and thus those algorithms operate exclusively with these sub-matrices.

Although one of the main motivations for this class of algorithm-by-blocks is to increase performance [5, 77], this approach can also be used to design and implement runtime systems that exploit tasks parallelism on multi-core architectures and, in our case, on multi-GPU systems. The usage of storage-by-blocks and algorithms-by-blocks eases the task dependency analysis between tasks, and thus allows the development of efficient scheduling techniques to improve parallel performance on this type of architectures.

In response to the increasing interest in algorithms-by-blocks, the FLAME project introduced the FLASH API [95], facilitating the design, implementation and manipulation of matrices stored by blocks. As nearly all linear algebra operations can be expressed using the FLAME notation [29], the conversion of existing algorithms to algorithms-by-blocks is straightforward using the hierarchical matrices provided by the API.

To illustrate this point, Figure 5.2 (left) shows the implementation of the blocked algorithm for the Cholesky factorization using the FLASH API. This implementation is very similar to the corresponding one using the FLAME/C API. The main difference between both is in the repartitioning `FLA_Repart_2x2_to_3x3`: in FLAME/C, the size of `A11` is $b \times b$ (thus, it contains b^2 scalar elements), while in FLASH this block is of size 1×1, as it is considered as *one* element of a matrix of matrices.

In this type of implementations, routines `FLASH_Trsm` and `FLASH_Syrk` implement the algorithms-by-blocks in Figures 5.2 (right) and 5.3.

```
void FLASH_Syrk_ln( FLA_Obj alpha, FLA_Obj A,
                    FLA_Obj beta,  FLA_Obj C )
/* Special case with mode parameters
     FLASH_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
                 ...                                   )
   Assumption: A is a column of blocks (column panel) */
{
  FLA_Obj AT,      A0,      CTL,   CTR,      C00, C01, C02,
          AB,      A1,      CBL,   CBR,      C10, C11, C12,
                   A2,                       C20, C21, C22;

  FLA_Part_2x1( A,    &AT,
                      &AB,    0, FLA_TOP );
  FLA_Part_2x2( C,    &CTL, &CTR,
                      &CBL, &CBR,     0, 0, FLA_TL );

  while ( FLA_Obj_length( AL ) < FLA_Obj_length( A ) ) {
    FLA_Repart_2x1_to_3x1( AT,       &A0,
                         /* ** */   /* ** */
                                    &A1,
                           AB,       &A2,    1, FLA_BOTTOM );
    FLA_Repart_2x2_to_3x3(
       CTL, /**/ CTR,       &C00, /**/ &C01, &C02,

    /* ************ */   /* ****************** */
                          &C10, /**/ &C11, &C12,
       CBL, /**/ CBR,       &C20, /**/ &C21, &C22,
       1, 1, FLA_BR );
    /*--------------------------------------------*/
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
              alpha, FLASH_MATRIX_AT( A1 ),
              beta,  FLASH_MATRIX_AT( C11 ) );
    FLASH_Gepb( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
                alpha, A2,
                       A1,
                beta,  C21 );
    /*--------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &AT,        A0,
                                          A1,
                            /* ** */   /* ** */
                              &AB,        A2,    FLA_TOP );
    FLA_Cont_with_3x3_to_2x2(
       &CTL, /**/ &CTR,       C00, C01, /**/ C02,
    /* ************** */  /* ****************** */
       &CBL, /**/ &CBR,       C20, C21, /**/ C22,
       FLA_TL );
  }
}
```

```
void FLASH_Gepb_nt( FLA_Obj alpha, FLA_Obj A,
                                   FLA_Obj B,
                    FLA_Obj beta,  FLA_Obj C )
/* Special case with mode parameters
     FLASH_Gepb( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
                 ...                              )
   Assumption: B is a block and
               A, C are columns of blocks (column panels) */
{
  FLA_Obj AT,      A0,     CT,      C0,
          AB,      A1,     CB,      C1,
                   A2,              C2;

  FLA_Part_2x1( A,    &AT,
                      &AB,    0, FLA_TOP );
  FLA_Part_2x1( C,    &CT,
                      &CB,    0, FLA_TOP );

  while ( FLA_Obj_length( AT ) < FLA_Obj_length( A ) ) {
    FLA_Repart_2x1_to_3x1( AT,       &A0,
                         /* ** */   /* ** */
                                    &A1,
                           AB,       &A2,    1, FLA_BOTTOM );
    FLA_Repart_2x1_to_3x1( CT,       &C0,
                         /* ** */   /* ** */
                                    &C1,
                           CB,       &C2,    1, FLA_BOTTOM );
    /*--------------------------------------------*/
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
              alpha, FLASH_MATRIX_AT( A1 ),
                     FLASH_MATRIX_AT( B ),
              beta,  FLASH_MATRIX_AT( C1 ) );
    /*--------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &AT,        A0,
                                          A1,
                            /* ** */   /* ** */
                              &AB,        A2,    FLA_TOP );
    FLA_Cont_with_3x1_to_2x1( &CT,        C0,
                                          C1,
                            /* ** */   /* ** */
                              &CB,        C2,    FLA_TOP );
  }
}
```

Figure 5.3: FLASH implementation of the function `FLASH_Syrk` involved in the algorithm-by-blocks for Variant 1 of the Cholesky factorization.

The algorithm-by-blocks for Variant 1 of the Cholesky factorization is next described using an example for a matrix composed by 4×4 sub-matrices:

$$A = \begin{pmatrix} \bar{A}_{00} & \star & \star & \star \\ \bar{A}_{10} & \bar{A}_{11} & \star & \star \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \star \\ \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{pmatrix},$$

where we assume that each block $\bar{A}_{ij}$ is of dimension $b \times b$.

The loop in routine `FLASH_Chol_by_blocks_var1` will iterate four times and, at the beginning of each iteration, the partitioning will contain the following blocks:

$$\left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) = \begin{array}{cc} \text{First iteration} & \text{Second iteration} \\ \left(\begin{array}{c|c|c|c} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \hline \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \hline \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \hline \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{array}\right), & \left(\begin{array}{c|c|c|c} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \hline \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \hline \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \hline \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{array}\right), \\ \text{Third iteration} & \text{Fourth iteration} \\ \left(\begin{array}{cc|c|c} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \hline \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \hline \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{array}\right), & \left(\begin{array}{ccc|c} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \hline \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{array}\right). \end{array}$$

Thus, the operations performed on the blocks of the matrix during the first iteration of the loop will be:

$$A_{11} := \{L\backslash A\}_{11} = \text{CHOL}(A_{11}) \quad \equiv \quad \bar{A}_{00} := \{L\backslash \bar{A}\}_{00} = \text{CHOL}(\bar{A}_{00}), \tag{5.1}$$

$$A_{21} := A_{21}\text{TRIL}(A_{11})^{-\text{T}} \quad \equiv \quad \begin{pmatrix} \bar{A}_{10} \\ \bar{A}_{20} \\ \bar{A}_{30} \end{pmatrix} := \begin{pmatrix} \bar{A}_{10} \\ \bar{A}_{20} \\ \bar{A}_{30} \end{pmatrix} \text{TRIL}(\bar{A}_{00})^{-\text{T}}, \tag{5.2}$$

$$\begin{aligned} A_{22} := A_{22} - A_{21}A_{21}^{\text{T}} \quad \equiv \quad & \begin{pmatrix} \bar{A}_{11} & \star & \star \\ \bar{A}_{21} & \bar{A}_{22} & \star \\ \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{pmatrix} := \begin{pmatrix} \bar{A}_{11} & \star & \star \\ \bar{A}_{21} & \bar{A}_{22} & \star \\ \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{pmatrix} \\ & - \begin{pmatrix} \bar{A}_{10} \\ \bar{A}_{20} \\ \bar{A}_{30} \end{pmatrix} \begin{pmatrix} \bar{A}_{10} \\ \bar{A}_{20} \\ \bar{A}_{30} \end{pmatrix}^{\text{T}}. \end{aligned} \tag{5.3}$$

The expression in (5.2) operates on a panel of blocks; correspondingly, in routine `FLASH_Trsm` (Figure 5.2-right) the basic operations are also performed on blocks, and thus it is considered an algorithm-by-blocks. When this routine is used to solve the triangular system in (5.2) $B := BL^{-\text{T}}$, three iterations of the loop are performed:

$$\left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right) = \begin{array}{ccc} \text{First iteration} & \text{Second iteration} & \text{Third iteration} \\ \left(\begin{array}{c} \bar{A}_{10} \\ \hline \bar{A}_{20} \\ \bar{A}_{30} \end{array}\right), & \left(\begin{array}{c} \bar{A}_{10} \\ \bar{A}_{20} \\ \hline \bar{A}_{30} \end{array}\right), & \left(\begin{array}{c} \bar{A}_{10} \\ \bar{A}_{20} \\ \bar{A}_{30} \end{array}\right). \end{array}$$

Thus, the following subroutine calls and matrix operations are performed in those iterations:

$$B_1 := B_1 L^{-\mathrm{T}} \quad \equiv \quad \bar{A}_{10} := \bar{A}_{10}\mathrm{TRIL}\left(\bar{A}_{00}\right)^{-\mathrm{T}} \quad \text{First iteration,} \tag{5.4}$$

$$B_1 := B_1 L^{-\mathrm{T}} \quad \equiv \quad \bar{A}_{20} := \bar{A}_{20}\mathrm{TRIL}\left(\bar{A}_{00}\right)^{-\mathrm{T}} \quad \text{Second iteration,} \tag{5.5}$$

$$B_1 := B_1 L^{-\mathrm{T}} \quad \equiv \quad \bar{A}_{30} := \bar{A}_{30}\mathrm{TRIL}\left(\bar{A}_{00}\right)^{-\mathrm{T}} \quad \text{Third iteration.} \tag{5.6}$$

Analogously, as the rank-b update in (5.3) involves a blocked sub-matrix, the algorithms-by-blocks in Figure 5.3 provide the necessary functionality.

Similar transformations of the algorithms can be obtained for other variants of the Cholesky factorization, and also for other linear algebra operations. Note the differences between the algorithm-by-blocks for the Cholesky factorization in Figure 5.2 and the blocked algorithmic variants shown in Chapter 4. First, blocked algorithms assume a classical storage of data by columns, whereas algorithms-by-blocks deal with hierarchical matrices stored by blocks. In addition, while algorithms-by-blocks are decomposed into BLAS calls operating with blocks of the same dimension, blocked algorithms are transformed into BLAS calls with sub-matrices of arbitrary dimensions, yielding operations with different matrix shapes (see, for example, the *matrix-matrix*, *panel-matrix* or *panel-panel* blocked algorithms derived for our own implementation of the BLAS routines in Chapter 3).

5.2.2. Dynamic scheduling and out-of-order execution

The development of algorithms-by-blocks naturally facilitates the adoption of techniques akin to dynamic scheduling and out-of-order execution hardwired in superscalar processors. In this section we show how these techniques can be adopted to systematically expose task parallelism in algorithms-by-blocks, which is the base for our automatic parallelization of linear algebra codes on multi-GPU platforms.

Current superscalar processors can dynamically schedule scalar instructions to execution units as operands become available (keeping track of data dependencies). A similar approach can be adopted in software to automatically expose the necessary task parallelism, so that ready tasks are dynamically scheduled available tasks to the available execution resources. The only information that is needed is that necessary to track data dependencies between tasks. In this case, the specific blocks that act as operands of each task, and the *directionality* of those operands (*input* for read-only operands, *output* for write-only operands, or *input/output* for read-write operands).

Obtaining this information, it is possible to build a software system that acts like Tomasulo's algorithm [130] does on superscalar processors, dynamically scheduling operations to idle computing resources as operands become available, and tasks become ready for execution.

For example, consider the following partition of a symmetric definite positive matrix to be decomposed using the Cholesky factorization:

$$A = \begin{pmatrix} \bar{A}_{00} & \star & \star \\ \bar{A}_{10} & \bar{A}_{11} & \star \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} \end{pmatrix}.$$

Table 5.1 shows the operations (or *tasks*) that are necessary to factorize A using the algorithm in Figure 5.2. Initially, the original blocks that act as operands to the tasks are available for the corresponding tasks that make use of them. The availability of an operand is indicated in the table using a check tag (☑). Upon completion of a task, its output operand is checked in the subsequent entries in the table until that operand appears again as an output variable for another task.

Whenever all the operands of a given task are checked as available, the task can be considered as *ready*, and thus it can be scheduled for execution. Note how, depending on the order in which

	Task
1	$A_{00} := \text{CHOL}(A_{00})$
2	$A_{10} := A_{10} A_{00}^{-T}$
3	$A_{20} := A_{20} A_{00}^{-T}$
4	$A_{11} := A_{11} - A_{10} A_{10}^{T}$
5	$A_{21} := A_{21} - A_{20} A_{10}^{T}$
6	$A_{22} := A_{22} - A_{20} A_{20}^{T}$
7	$A_{11} := \text{CHOL}(A_{11})$
8	$A_{21} := A_{21} A_{11}^{-T}$
9	$A_{22} := A_{22} - A_{21} A_{21}^{T}$
10	$A_{22} := \text{CHOL}(A_{22})$

Original table

In		In/Out
		A_{00}☑
A_{00}		A_{10}☑
A_{00}		A_{20}☑
A_{10}		A_{11}☑
A_{10}	A_{20}	A_{21}☑
A_{20}		A_{22}☑
		A_{11}
A_{11}		A_{21}
A_{21}		A_{22}
		A_{22}

After execution of task 1

In		In/Out
A_{00}☑		A_{10}☑
A_{00}☑		A_{20}☑
A_{10}		A_{11}☑
A_{10}	A_{20}	A_{21}☑
A_{20}		A_{22}☑
		A_{11}
A_{11}		A_{21}
A_{21}		A_{22}
		A_{22}

After execution of task 2

In		In/Out
A_{00}☑		A_{20}☑
A_{10}☑		A_{11}☑
A_{10}☑	A_{20}	A_{21}☑
A_{20}		A_{22}☑
		A_{11}
A_{11}		A_{21}
A_{21}		A_{22}
		A_{22}

After execution of task 3

In		In/Out
A_{10}☑		A_{11}☑
A_{10}☑	A_{20}☑	A_{21}☑
A_{20}☑		A_{22}☑
		A_{11}
A_{11}		A_{21}
A_{21}		A_{22}
		A_{22}

After execution of task 4

In		In/Out
A_{10}☑	A_{20}☑	A_{21}☑
A_{20}☑		A_{22}☑
		A_{11}☑
A_{11}		A_{21}
A_{21}		A_{22}
		A_{22}

After execution of task 5

In		In/Out
A_{20}☑		A_{22}☑
		A_{11}☑
A_{11}		A_{21}☑
A_{21}		A_{22}
		A_{22}

After execution of task 6

In		In/Out
		A_{11}☑
A_{11}		A_{21}☑
A_{21}		A_{22}☑
		A_{22}

After execution of task 7

In		In/Out
A_{11}☑		A_{21}☑
A_{21}		A_{22}☑
		A_{22}

After execution of task 8

In		In/Out
A_{21}☑		A_{22}☑
		A_{22}

Table 5.1: An illustration of the availability of operands for each task involved in the Cholesky factorization of a 3×3 matrix of blocks using the algorithm-by-blocks shown in Figure 5.2. A ☑ indicates that the operand is available. Note how the execution of a task triggers the availability of new subsequent tasks.

ready tasks are scheduled for execution, the order in which future tasks are marked as ready and scheduled for execution can vary. Therefore, the actual order in which instructions are scheduled and executed does not have to match the sequential order in which tasks are found in the algorithm and added to the table. Consider, e.g., the first three tasks listed in Table 5.1. Initially, tasks 1, 2, and 3 have one of their operands available (A_{00}, A_{10} and A_{20}, respectively). Task 1 has all operands available, so it can proceed. Tasks 2 and 3, on the other hand, do not have all operands available. After factorizing the first diagonal block (task 1, that updates block A_{00}) the operand A_{00} marked as input for tasks 2 and 3 becomes available. At that moment, both tasks become *ready*, as all operands involved are available. The specific order in which those two tasks are scheduled for execution will not affect the correctness of the computation, but ultimately will influence the order in which future tasks are performed.

In addition to the out-of-order execution, this approach exposes parallelism at a task level (usually referred as *task-level parallelism* [67]). Following with the example above, after the execution of task 1, both tasks 2 and 3 become ready. As they are independent (that is, there is no data dependency between both tasks) given a system with a pool of execution resources, it is possible to schedule each task to a different execution unit such that both tasks are executed in parallel.

It is important to realize that the tasks to perform a given algorithm-by-blocks together with their associated input and output operand identification can be identified a priori, and a table similar to that shown in Figure 5.1 can be systematically built regardless the type of algorithm exposed.

Typically, the information gathered in the table can be represented as a directed acyclic graph (or *DAG*), in which nodes denote tasks, and edges denote data dependencies between them. For example, Figure 5.4 shows the DAG generated for the Cholesky factorization of a matrix of 4×4 blocks. The first two levels of the graph in Figure 5.4 offer a global view of two different key aspects of parallelism:

1. Consider the dependence relationship between tasks CHOL_0 and TRSM_1, where CHOL_0 performs the operation $A_{00} := \text{CHOL}(A_{00})$ and TRSM_1 performs the operation $A_{10} := A_{10}A_{00}^{-T}$. Note how block A_{00} is written by task CHOL_0 and read by subsequent task TRSM_1. Thus, task TRSM_1 cannot proceed until the execution of task CHOL_0 is completed, and thus there is a data dependence between them. This type of data dependence, usually referred as *flow-dependence* determines the order in which tasks can be issued to execution.

2. Consider tasks TRSM_1, TRSM_2, and TRSM_3. They are blocked as they are not ready for execution until the execution of task CHOL_0 is completed. When block A_{00} is updated by task CHOL_0, all these tasks become ready, and, as there is no data dependency between them, they can proceed concurrently. In this sense, it is possible to execute them in parallel provided there exist enough execution units available. Proceeding this way, a task is seen as the minimum execution unit, and parallelism is extracted at the task level. Thus, *task-level parallelism* is extracted from the structure of the DAG.

Following these two observations, parallelism can be extracted in algorithms-by-blocks at the *task level*, with an order of execution ultimately defined by the availability of data operands: this execution model is usually referred as a *data-flow execution*. These two concepts have been previously applied to multi-core systems [46, 109].

Our main contribution is to redesign this approach to automatically parallelize high-performance dense linear algebra codes on platforms based on multi-GPU architectures with no impact on the libraries already developed for single-GPU systems. This implementation introduces a significant difference from that for a multi-core target, as multi-GPU systems exhibit separate memory address

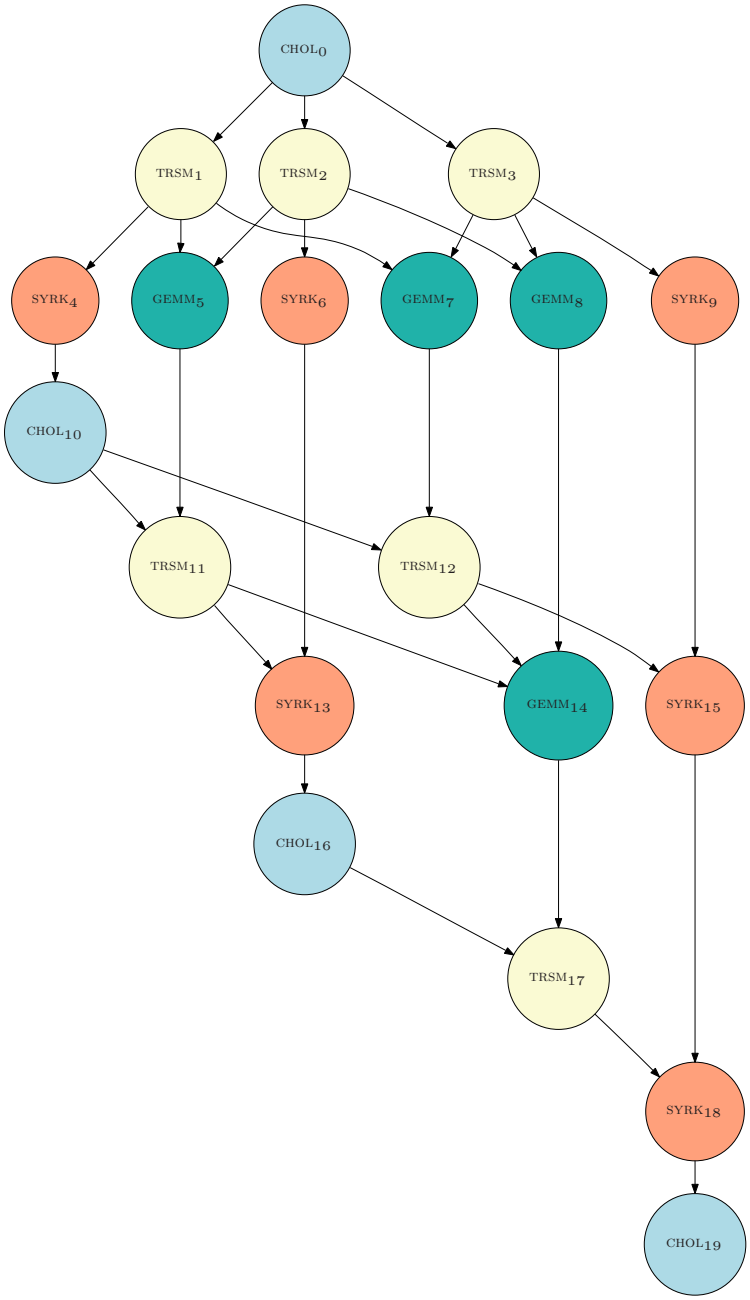


Figure 5.4: DAG for the Cholesky factorization of a 4×4 blocked matrix.

spaces for each execution unit. Given that this feature must remain transparent to the programmer, a run-time system must manage not only the *data-flow* execution and the *task-level* parallelism extraction, but also an appropriate scheme to deal with *data transfers*, reducing them as much as possible without the intervention of the programmer in order to minimize their impact on the final performance.

5.2.3. A runtime system for matrix computations on multi-GPU systems

Independently from the target architecture, there are two parts in a run-time system managing data-flow executions, each one related to a different code execution stage. The *analysis stage*, responsible of building the DAG associated with a specific algorithm; and the *dispatch stage*, responsible of handling data dependencies at runtime, data positioning issues and dispatching of tasks to execution units.

Data-flow execution entails the usage of a software system with two different jobs: tracking data-dependencies and issuing tasks to execution units at runtime. On multi-GPU systems, such a run-time system follows a hybrid execution model, in which the CPU is responsible for scheduling tasks to the accelerators while tracking dependencies at runtime, and the accelerators themselves are responsible for the actual computations and can be viewed as mere execution units.

The analysis stage

Our analysis stage follows many of the ideas implemented for multi-core architectures in the SuperMatrix runtime [120] from the FLAME project. During this stage, a single thread performs a symbolic execution of the code. This symbolic execution does not perform any actual operations as found in the code, but adds them to a queue of pending tasks as they are encountered. For the right-looking algorithm-by-blocks for the Cholesky factorization, this queuing is performed inside the calls to `FLA_Chol_unb_var1`, `FLA_Trsm`, `FLA_Syrk`, and `FLA_Gemm` as they are encountered in routines `FLASH_Chol_by_blocks_var1`, `FLASH_Trsm` and `FLASH_Syrk`, see Figures 5.2 and 5.3.

Proceeding this manner, the actual tasks involved in the algorithm-by-blocks, together with the existing data dependencies between them, are identified before the actual execution commences, and a DAG can be built to make use of it in during the *dispatch stage*.

The analysis stage is executed sequentially, and it is not overlapped with the dispatch stage. Other similar runtime systems for multi-core and many-core architectures, such as CellSs or SMPSs, interleave the analysis and dispatch stages, executing them concurrently. That is, the execution of tasks starts as soon as one task is stored in the pending tasks queue. Proceeding this way, the DAG is dynamically constructed, and tasks are issued from the pending queue following a producer-consumer paradigm. Under certain circumstances, this approach has remarkable benefits; for example, if the amount of memory necessary to store the information for the tasks becomes a real bottleneck (due to the limited amount of memory of the platform), the benefit of consuming tasks before the queue is completely built is evident.

However, given that the analysis stage does not perform actual computations, the cost of the symbolic execution stage of dense linear algebra operations is in general amortized during the dispatch stage, so we advocate for a separate execution of both stages. Also, the amount of memory consumed to explicitly store the full DAG can be considered negligible. Therefore, tasks are sequentially stored during the analyzer stage, and the dispatch stage is not invoked until the complete DAG is built.

The dispatch stage

Once the DAG is constructed, the execution of the dispatch stage commences. While the analysis stage in the case of multi-GPU systems remains basically unmodified from that of SuperMatrix, the dispatch stage is deeply changed to support the particularities of the novel architecture.

Once the dispatch stage begins, the runtime system does not refer anymore to the codes provided by the programmer. Instead, all the necessary information regarding tasks to be performed, data needed by each tasks, and data dependencies between them is stored in the pending tasks queue (and thus, the associated DAG) built during the analysis stage.

From a high level perspective, once the DAG has been constructed and the dispatch stage is invoked, one thread is spawned per accelerator in the system (or less, depending on the user's execution configuration). Each thread is bound to a specific accelerator, and is responsible for the management of that accelerator throughout the rest of the computation. Each thread monitors the queue of pending tasks, and as soon as a task becomes ready (that is, its data dependencies have been satisfied) and the corresponding accelerator is idle, task is executed in it. Upon completion, dependency information in the queue is properly updated.

There are important differences between a run-time system designed for a multi-core architecture and a run-time focused on multi-GPU architectures that are reported next.

5.3. Programming model and runtime. Performance considerations

In this section, we review the main design decisions and performance concerns and improvements made during the design and implementation of the run-time system for multi-GPU architectures. A few concepts have been directly inherited from the existing SuperMatrix out-of-order execution run-time for multi-core architectures. In essence, the analysis stage for multi-GPU systems does not differ to that used for multi-core architectures. In this section, we focus on the differences between both implementations and avoid details that have been already successfully applied and published for multi-core architectures [45, 46, 116].

5.3.1. Programming model

Figure 5.5 shows a sample code with the modifications necessary to transform a sequential code using the FLASH API to a code which can also run using the proposed runtime on a multi-GPU system. The code on the left side of the figure is analogous to the algorithm-by-blocks using the FLASH API shown in Figure 5.2. The code on the right part of the figure corresponds to a possible driver that initializes data structures and invokes the parallel Cholesky implementation.

Taking as a reference a driver that invokes a sequential version of the algorithm, the main differences with respect to a parallel version of the program are mainly two:

1. *Identification of a parallel region*: routines `FLASH_Queue_begin` and `FLASH_Queue_end` encapsulate a parallel region. The underlying runtime system is in charge of task identification and data dependency management inside this region.

2. *GPU activation*: routines `FLASH_enable_GPU` and `FLASH_disable_GPU` indicate the runtime that available GPUs must be used to execute each task inside a parallel region. Without these invocations, the runtime system would execute the tasks on the multi-core processors.

A rapid review of the codes in the figure reveals that differences between both codes are minimal, and are directly related to the structure of the underlying runtime system. In addition to the

runtime initialization and finalization procedures, there are two different parts in the code associated with the main stages of the runtime execution:

1. *Analysis stage*: This stage naturally corresponds to the FLAME-based part of the code in the figure. Each call to a routine starting with "FLASH_*" creates a new task and adds it to the DAG. As the directionality of the operands in linear algebra routines is known a priori, all the necessary information to track dependencies between tasks is known, and the whole DAG can be constructed at runtime. In fact, the calls to routines "FLASH_*" are just wrappers to functions that do not execute the real code, but just perform the DAG construction and handle dependency tracking (often referred as a *symbolic execution* of the code).

2. *Dispatch stage*: Once the first part of the execution is done, the DAG is completely built and data dependency information has been obtained. At this point, the actual parallel execution can proceed. The end of a parallel region activates the dispatch stage of the run-time system and thus the parallel execution can commence. All necessary information is now stored in the DAG, including task type, dependency tracking, and data flow information.

The code also illustrates how the user is isolated from the fact that the code is executed on a multi-GPU platform, as calls to the appropriate NVIDIA CUBLAS routines are embedded inside the task codes, and no data transfers are explicit in the codes as they are carried out by the the run-time system. This was one of the main goals in the design of the programming model, and here we demonstrate that the FLAME programming model is flexible enough to easily accommodate new architectures preserving the existing algorithms.

5.3.2. Temporal planification

In this section we offer an overview of the general mechanism that is used to schedule and dispatch tasks to the corresponding execution units in the multi-GPU architecture. Some of the ideas and techniques exposed have already been successfully implemented in the SuperMatrix runtime. Whenever possible, we only focus on the details that are necessary to adapt this runtime to the novel architecture.

Once the DAG has been built and all data dependency information is annotated, the dispatch stage commences with and invocation to the `FLASH_Exec_list_of_tasks`. Initially one thread is spawned per hardware accelerator. In our implementation, OpenMP is in charge of the creation, synchronization, and management of threads. From this point, each thread is responsible of the execution and data management of a given hardware accelerator.

Algorithm 2 presents the basic procedure that dictates the actions performed by each thread during the dispatch stage in order to schedule and execute tasks. In essence, the algorithm describes the general operations executed by the implemented runtime. We consider the DAG as a collection of tasks to be executed during the parallel execution stage. The tasks in this pool or *queue of pending tasks* are consumed by the threads following a consumer-producer scheme. When a thread becomes ready, it selects a ready task, that is, a task with all its data dependencies satisfied, and instructs the corresponding hardware accelerator to execute the task.

Note how in the algorithm, a ready state (from the data dependencies point of view) is not a sufficient condition to consider that a task is ready for the execution on a given accelerator. In addition, the procedure `is_ready` adds information about the identifier of the thread, which is necessary when data affinity policies are applied, as will be explained in the following sections. As a brief overview, we closely bind the accelerator in which a task can be executed to the specific data block that the task will update. In summary, each thread can dequeue a task from the pending

```
int FLA_Chol( FLA_Obj A ) {
  FLA_Obj  ATL, ATR,   A00, A01, A02,
           ABL, ABR,   A10, A11, A12,
                       A20, A21, A22;
  int      value;

  FLA_Part_2x2( A,  &ATL, /**/ &ATR,
                  /* ************** */
                    &ABL, /**/ &ABR,
            /* with */ 0, /* by */ 0, /* submatrix */ FLA_TL );

  while ( TRUE ){

    FLA_Repart_2x2_to_3x3(
                    ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
                    /* ************* */  /* ******************** */
                       /**/              &A10, /**/ &A11, &A12,
                    ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
           /* with */ 1, /* by */ 1, /* A11 split from */ FLA_BR );

    /* ********************************************************* */

    /* Chol_blk( A11 ). */
    value = FLASH_Chol( A11 );

    if ( value != FLA_SUCCESS ){
      value = k;
      break;
    }
    else
      k += b;

    /* A21 := A21 * inv( L11' ). */
    FLASH_Trsm( A11, A21 );

    /* tril( A22 ) := tril( A22 ) - tril( A21 * A21' ). */
    FLASH_Syrk( A21, A22 );

    /* ********************************************************* */

    FLA_Cont_with_3x3_to_2x2(
    &ATL, /**/ &ATR,      A00, A01, /**/ A02,
                          /**/            A10, A11, /**/ A12,
                    /* ************** */ /* ******************** */
                    &ABL, /**/ &ABR,      A20, A21, /**/ A22,
           /* with A11 added to submatrix */ FLA_TL );
  }

  return value;
}
```

```
int main( void ) {

  /* Linear and hierarchical matrix */
  FLA_Obj A, AH;

  /* Creation of linear matrix */
  FLA_Obj_create( FLA_FLOAT, n, n, 0, 0, & A );

  /* ... Initialization of matrix contents ...  */

  /* Transformation from linear to hierarchical */
  FLASH_Obj_create_hier_copy_of_flat( A, 1, & nb, & AH );

  /* Indicate usage of GPU */
  FLASH_Queue_enable_gpu();

  /* Begin parallel region */
  FLASH_Queue_begin();

  /* Cholesky factorization */
  FLA_Chol( AH );

  /* End parallel region */
  FLASH_Queue_end();

  /* Indicate end of usage of GPU */
  FLASH_Queue_disable_gpu();

  /* Transformation from hierarchical to linear */
  FLASH_Obj_flatten( AH, A );

  /* Free hierarchical matrix */
  FLASH_Obj_free( &AH );

  /* Free linear matrix */
  FLA_Obj_free( &A );
}
```

Figure 5.5: Code implementation of the Cholesky factorization using the FLASH API and our multi-GPU runtime system.

queue when it is ready and, in case data affinity policies are applied, only if the corresponding accelerator is responsible for the execution of that task.

Algorithm 2 General runtime algorithm. This procedure is executed by all threads in order to schedule and dispatch tasks to the corresponding GPUs.

```
for all task in pending_queue do
  if is_ready( task, th_id ) then
    add_to_ready_list( task, th_id )
  end if
end for
while ready_tasks( th_id ) do
  dequeue_from_ready()
  check_for_pending_transfers()
  load_operands( task, th_id )
  dispatch_task()
  manage_modified_operands( task, th_id )
  for all dependent_task do
    update_dependencies( dependent_task )
    if is_ready( dependent_task, th_id ) then
      add_to_ready_list( dependent_task, th_id )
    end if
  end for
end while
flush_cache( )
```

Once a thread dequeues a task from the pending queue, it adds it to a proprietary queue of ready tasks, that only contains tasks that are ready for execution on its accelerator. Thus, the design features an independent *ready queue* for each accelerator present in the system. Ready tasks are dequeued and scheduled for execution by the thread that owns the corresponding queue. Prior to the execution, data blocks bound to the task must be available in the corresponding GPU. At the end, the state of the GPU memory must be updated accordingly to the memory management policies that will be explained in the following section.

Once the task is executed, the corresponding thread is in charge of updating the data dependencies of tasks that were waiting for execution in the queue of pending tasks. Once this information is updated, the algorithm continues with the identification and execution of the new ready tasks.

Up to this point, the following aspects differentiate the run-time system tailored for multi-GPU architectures from that included in SuperMatrix for multi-core processors:

- There is a close binding between threads and execution units.
- The list of ready tasks is splitted, creating an independent queue bound to each execution unit.
- It is possible to include data affinity policies to exploit data locality and reduce data transfers between memory spaces.
- Data transfers are explicitly managed by the runtime, and not by the programmer.

Among these properties, data transfer management is the one that requires more challenging modifications in the run-time system, and, as will be demonstrated, the one that makes the strongest difference in performance.

5.3.3. Transfer management and spatial assignation

Modern multi-GPU systems present a (logically and physically) separate memory space owned by each GPU in the system, independent from the main memory space. This is a key feature of current multi-GPU architecture with a great impact on the design of of run-time systems, and that ultimately affects the final performance attained by the parallel codes. Hence, before a task can be executed on a given GPU, memory must be allocated on it, and data has to be transferred from main memory to GPU memory prior to the execution. Upon completion of the operation, the result must be transferred back to main memory. As an additional drawback, there is not direct communication between GPU memories, and thus every point-to-point communication between two GPUs must be carried out through main memory.

As demonstrated in Chapter 3, data transfers have a non-negligible impact on the final performance of the BLAS routines, especially if the operands are small. When dynamic scheduling is employed, moderate block sizes are selected, to increase task parallelism (concurrency) and, therefore, data transfers pose a severe limitation to attain high performance. In addition, the shared use of the PCI-Express bus in multi-GPU systems transforms it into a major bottleneck. In this sense, the reduction of data transfers will ultimately determine the actual performance attained in the parallel codes. In this section, we propose several improvements in the design of the runtime system to reduce the amount of data transfers between memories.

We propose incremental improvements in the data management system, starting from a basic implementation. For each version of the system, enumerated from 1 to 4, we explain the rationale, the modifications introduced in the procedures `load_operands` and `manage_modified_operands` in Algorithm 2, and sample traces that illustrate the benefits and drawbacks of each implementation.

Version 1: Basic implementation

The existence of separate memory spaces in multi-GPU systems naturally leads to a basic implementation of the run-time system, in which data transfers between main memory and GPU memory are closely bound to the amount and type of tasks to be executed on each accelerator.

In this basic implementation, scheduling a task to a given GPU involves three main stages. First, each block that is an input operand to the operation must be allocated and transferred to the corresponding GPU. Next, the operation is performed on the corresponding GPU. Finally, output operands are transferred back to main memory to keep memory consistency.

Algorithms 3 and 4 show the functionality of the procedures `load_operands` and `manage_modified_operands`, respectively. Note how all input operands are allocated and transferred to GPU memory before execution, and deallocated after it. Only output operands are transferred back to main memory after execution.

Table 5.2 shows an extract of the operations that are needed to compute the Cholesky factorization of a 4×4 matrix-of-blocks using a runtime that follows the guidelines of this basic implementation. For each task, the table includes the specific instruction executed on the accelerator on the second column. The third column specifies the GPU in which the task is executed. Note that in this basic implementation, tasks are dispatched to GPUs as they become ready, so the sequence of GPUs used can vary from one execution to another, with no restriction. In the fourth

Algorithm 3 Algorithm for loading operands to GPU. Version 1.

```
for all operand in operands( task ) do
  allocate_operand( operand )
  transfer_to_GPU( operand )
end for
```

Algorithm 4 Algorithm for updating operands in main memory. Version 1.

```
for all operand in operands( task ) do
  if output_operand( operand ) then
    transfer_to_main_memory( operand )
  end if
  deallocate_operand( operand )
end for
```

column, for each block involved in the computation of the task, a row is added to the table if a data transfer between main memory and the corresponding GPU memory is necessary.

Operands used by each task have to be allocated in the corresponding GPU before the task begins and transferred there prior to the execution of the task. After the completion of the task, output operands are retrieved back to main memory, and the corresponding blocks are erased from GPU memory.

The total amount of *writes* (transfers of data from main memory to GPU memory) and *reads* (transfers of data from GPU memories to main memory) are summarized in Table 5.2 for the extract of operations shown and for the whole computation of the Cholesky factorization of a 4×4 matrix of blocks. As only one operand is updated by each task, there are as many *reads* as operations are performed during the factorization. The number of *writes* is also dictated by the number of input operands for each operation, as no communication-reduction techniques are applied in this version.

This version of the runtime was developed to serve as a reference for the next ones, and to illustrate, by comparing performance results, the benefits yield by the following data transfer reduction techniques.

Version 2: Data affinity + write-through

Taking as an example the first two operations performed by GPU_0, it is possible to illustrate that some of the transfers performed by this basic implementation are actually unnecessary. In this case, the task labeled as number 1 in the table performs a "CHOL" task, $A_{00} := \text{CHOL}(A_{00})$, while the task labeled as number 3 in the table performs a "TRSM" task, $A_{20} := A_{20}A_{00}^{-T}$. The first CHOL operation needs a copy of block A_{00} in the memory of GPU_0, so it is transferred prior to the execution. Once this operation is completed, this block is removed from GPU_0, and is no longer available for future operations unless explicitly transferred there again. As in the example both operations are sequentially executed by the same GPU (in this case, GPU_0), the operation labeled with number 3 is executed immediately after operation number 1 on GPU_0. This second TRSM also needs block A_{00} as an input operand. Note that, although A_{00} has just been used by the same GPU, it has been erased at this point, and thus it must be transferred again to the memory of GPU_0.

In order to avoid this type of unnecessary transfers, and thus to improve data locality (therefore reducing the costly data transfers between the memory of the GPUs), we propose a workload distribution following a static mapping (or layout) of the data matrix to a logical grid of GPUs (for example, a grid of dimension 2×2 for four GPUs). Figure 5.6 shows an example of a bi-dimensional workload distribution of a 4×4 blocked matrix to a multi-GPU system with four GPUs (G_{00}, G_{01}, G_{10}, G_{11}). Bi-dimensional workload distribution in the context of shared-memory multiprocessors has been previously investigated in [98].

			Runtime Version 1		
#	Instruction		Block	Transfer	Comment
1	$A_{00} := \text{CHOL}(A_{00})$	GPU_0	A_{00}	CPU $\longrightarrow$ GPU_0	
			A_{00}	CPU $\longleftarrow$ GPU_0	A_{00} removed from GPU_0
2	$A_{10} := A_{10}A_{00}^{-T}$	GPU_1	A_{10}	CPU $\longrightarrow$ GPU_1	
			A_{00}	CPU $\longrightarrow$ GPU_1	
			A_{10}	CPU $\longleftarrow$ GPU_1	A_{10}, A_{00} removed from GPU_1
3	$A_{20} := A_{20}A_{00}^{-T}$	GPU_0	A_{20}	CPU $\longrightarrow$ GPU_0	
			A_{00}	CPU $\longrightarrow$ GPU_0	
			A_{20}	CPU $\longleftarrow$ GPU_0	A_{20}, A_{00} removed from GPU_0
4	$A_{30} := A_{30}A_{00}^{-T}$	GPU_1	A_{30}	CPU $\longrightarrow$ GPU_1	
			A_{00}	CPU $\longrightarrow$ GPU_1	
			A_{30}	CPU $\longleftarrow$ GPU_1	A_{30}, A_{00} removed from GPU_1
5	$A_{11} := A_{11} - A_{10}A_{10}^{T}$	GPU_3	A_{11}	CPU $\longrightarrow$ GPU_3	
			A_{10}	CPU $\longrightarrow$ GPU_3	
			A_{11}	CPU $\longleftarrow$ GPU_3	A_{11}, A_{10} removed from GPU_3
6	$A_{21} := A_{21} - A_{20}A_{10}^{T}$	GPU_2	A_{21}	CPU $\longrightarrow$ GPU_2	
			A_{20}	CPU $\longrightarrow$ GPU_2	
			A_{10}	CPU $\longrightarrow$ GPU_2	
			A_{21}	CPU $\longleftarrow$ GPU_2	A_{21}, A_{20}, A_{10} removed from GPU_2
7	$A_{31} := A_{31} - A_{30}A_{10}^{T}$	GPU_3	A_{31}	CPU $\longrightarrow$ GPU_3	
			A_{30}	CPU $\longrightarrow$ GPU_3	
			A_{10}	CPU $\longrightarrow$ GPU_3	
			A_{31}	CPU $\longleftarrow$ GPU_3	A_{31}, A_{30}, A_{10} removed from GPU_3
8	$A_{22} := A_{22} - A_{20}A_{20}^{T}$	GPU_0	A_{22}	CPU $\longrightarrow$ GPU_0	
			A_{20}	CPU $\longrightarrow$ GPU_0	
			A_{22}	CPU $\longleftarrow$ GPU_0	A_{22}, A_{20} removed from GPU_0
9	$A_{32} := A_{32} - A_{30}A_{20}^{T}$	GPU_1	A_{32}	CPU $\longrightarrow$ GPU_1	
			A_{30}	CPU $\longrightarrow$ GPU_1	
			A_{20}	CPU $\longrightarrow$ GPU_1	
			A_{32}	CPU $\longleftarrow$ GPU_1	A_{32}, A_{30}, A_{20} removed from GPU_1
10	$A_{33} := A_{33} - A_{30}A_{30}^{T}$	GPU_3	A_{33}	CPU $\longrightarrow$ GPU_3	
			A_{30}	CPU $\longrightarrow$ GPU_3	
			A_{33}	CPU $\longleftarrow$ GPU_3	A_{33}, A_{30} removed from GPU_3
11	$A_{11} := \text{CHOL}(A_{11})$	GPU_3	A_{11}	CPU $\longrightarrow$ GPU_3	
			A_{11}	CPU $\longleftarrow$ GPU_3	A_{11} removed from GPU_3
12	$A_{21} := A_{21}A_{11}^{-T}$	GPU_2	A_{21}	CPU $\longrightarrow$ GPU_2	
			A_{11}	CPU $\longrightarrow$ GPU_2	
			A_{21}	CPU $\longleftarrow$ GPU_2	A_{21}, A_{11} removed from GPU_2
			$\vdots$		

# writes in the example	25	**# writes in the complete execution**	36
# reads in the example	12	**# reads in the complete execution**	16

Table 5.2: Extract of the list of tasks executed by the runtime (Version 1) and data transfers needed to perform the Cholesky factorization of a blocked 4×4 matrix.

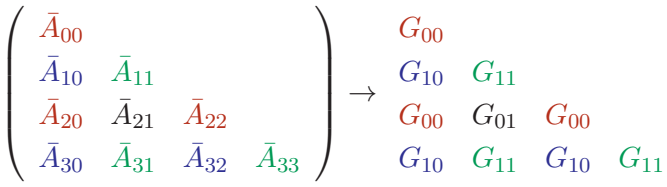


Figure 5.6: Cyclic 2-D mapping of the blocks in the lower triangular part of a 4×4 blocked matrix to four GPUs: G_{00}, G_{10}, G_{01}, and G_{11}.

In this scheme all operations that compute results which overwrite a given block are mapped to the same GPU, following an *owner-computes rule* [73]. Thus, e.g., in the Cholesky factorization the updates $\bar{A}_{21} := \bar{A}_{21} - \bar{A}_{20}\bar{A}_{10}^{\mathrm{T}}$ and $\bar{A}_{21} := \bar{A}_{21}\mathrm{TRIL}(\bar{A}_{11})^{-\mathrm{T}}$ are both performed in G_{01}.

Blocks are thus classified, from the viewpoint of a given GPU, into *proprietary* or *own* blocks (owned = written by it, following the "owner-computes" rule) and *non-proprietary* or *alien* blocks. This classification is known in advance (static), and does not change during the execution of the algorithm. Therefore, there is a tight data affinity that binds a given task to a given execution unit during the whole computation. For example, given the data distribution in Figure 5.6, block A_{00} is *owned* by GPU G_{00}, while block A_{10} is *alien* to it. As a consequence, during the dispatch stage any task that writes block A_{00} will be assigned to GPU G_{00}.

Initially all data blocks reside in main memory and the memory of all GPUs is empty. When a task is to be computed in a given GPU, block operands that are not already in the GPU memory are copied there. Proprietary blocks remain in that memory for the rest of the execution of the algorithm while alien blocks are discarded as soon as the operation is completed.

Algorithms 5 and 6 show the necessary steps to provide the functionality of the procedures `load_operands` and `manage_modified_operands`, respectively. Note that the `load_operands` algorithm includes an additional clause checking whether a proprietary block is already available in the memory of the corresponding GPU. After execution of the task, only *alien* blocks are deallocated from GPU memory.

A write-through policy is implemented in software to maintain the coherence between the proprietary blocks in the memory of the GPU and main memory, so that any update of a proprietary block is immediately propagated to the main memory. Proceeding this manner, there is no inconsistency between data in separate memory spaces. Also, there is no need to maintain the coherence between the GPU memory and main memory for non-proprietary blocks as these are read-only blocks.

Table 5.3 shows an extract of the operations necessary to perform a Cholesky factorization of a 4×4 matrix-of-blocks using a runtime modified to include the data-affinity strategy with the write-through coherence policy. For each executed task, the table includes the specific instruction executed on the accelerator on the second column. The third column specifies the GPU in which the task is executed. Remember that the GPU in which each task is executed in the basic implementation in Table 5.2 was the result of one possible execution but many variations were possible. On the other hand, the association between tasks and execution units in this implementation is tight, following in this case a 2D cyclic data distribution. Other workload distributions (block row-wise, block column-wise, cyclic variants or even a random distribution) are easily supported by the runtime system and, more important, are transparent to the developer.

Algorithm 5 Algorithm for loading operands to GPU. Version 2.

```
for all operand in operands( task ) do
  if own_operand (operand, th_id ) then
    if not_in_GPU (operand, th_id ) then
      allocate_operand( operand )
      transfer_to_GPU( operand )
    else
      {Own block is on GPU. Skip transfer.}
    end if
  else
    {Alien operand}
    allocate_operand( operand )
    transfer_to_GPU( operand )
  end if
end for
```

Algorithm 6 Algorithm for updating operands in main memory. Version 2.

```
for all operand in operands( task ) do
  if output_operand( operand ) then
    transfer_to_main_memory( operand )
  end if
  if alien_operand( operand, th_id ) then
    deallocate_operand( operand )
  end if
end for
```

In the example, when the task that computes the update $A_{21} := A_{21} - A_{20}A_{10}^{\mathrm{T}}$ is to be run at G_{01} (instruction #6 in the table), blocks A_{21}, A_{20}, and A_{10} are copied to the memory of this GPU; the update is computed, and the updated contents of A_{21} are propagated to main memory. Block A_{21} then remains in the GPU memory as G_{01} owns it, while the contents of A_{20} and A_{10} (that are alien blocks to that accelerator) are discarded. The benefits of this approach can be appreciated in future instructions: consider, e.g., the operation $A_{21} := A_{21}A_{11}^{-T}$ (instruction #12 in the table). In this case, only A_{11} is copied to the GPU memory as A_{21} is already there. Once this second update is carried out, following the write-through policy, the updated contents of A_{21} are sent back to main memory and A_{11} is discarded.

The benefits of this version of the runtime are clear from the summary of *writes* and *reads* shown at the bottom of the table. From the point of view of the *reads*, there is not any reduction in the number of transfers. On the other hand, the amount of *writes* is reduced by saving the transfers of *proprietary* blocks for all GPUs.

Version 3: Software cache + write-invalidate

Consider now the first two operations performed by GPU_1; it is possible to deduce that some of the transfers performed by these implementation are still unnecessary. In this case, the operation labeled as number 2 in the table performs a TRSM operation $A_{10} := A_{10}A_{00}^{-T}$, while the operation labeled as number 4 in the table performs a second TRSM operation $A_{30} := A_{30}A_{00}^{-T}$. The first TRSM operation needs a copy of block A_{00} in the memory of GPU_0, so it is transferred prior to the execution. Once the TRSM is completed, this block is removed from GPU_1, and is no longer available unless transferred back there. As both TRSM operations are sequentially executed by the same GPU (in this case, GPU_1), the operation labeled as number 4 is executed immediately after operation number 2 by GPU_1. This second TRSM also needs block A_{00} as an input operand. Thus, although A_{00} has just been used by the same GPU, it has been removed at this point, and thus it must be transferred again to the memory of GPU_1. Eliminating this unnecessary data transfers is the goal of the software cache introduced next.

			Runtime Version 2		
#	Instruction		Block	Transfer	Comment
1	$A_{00} := \text{CHOL}(A_{00})$	GPU_0	$\mathbf{A_{00}}$	CPU $\longrightarrow$ GPU_0	A_{00} kept in GPU_0
			$\mathbf{A_{00}}$	CPU $\longleftarrow$ GPU_0	
2	$A_{10} := A_{10}A_{00}^{-T}$	GPU_1	$\mathbf{A_{10}}$	CPU $\longrightarrow$ GPU_1	A_{10} kept in GPU_1
			A_{00}	CPU $\longrightarrow$ GPU_1	
			$\mathbf{A_{10}}$	CPU $\longleftarrow$ GPU_1	A_{00} removed from GPU_1
3	$A_{20} := A_{20}A_{00}^{-T}$	GPU_0	$\mathbf{A_{20}}$	CPU $\longrightarrow$ GPU_0	A_{20} kept in GPU_0
			$\mathbf{A_{00}}$	–	A_{00} already in GPU_0
			$\mathbf{A_{20}}$	CPU $\longleftarrow$ GPU_0	
4	$A_{30} := A_{30}A_{00}^{-T}$	GPU_1	$\mathbf{A_{30}}$	CPU $\longrightarrow$ GPU_1	A_{30} kept in GPU_1
			A_{00}	CPU $\longrightarrow$ GPU_1	
			$\mathbf{A_{30}}$	CPU $\longleftarrow$ GPU_1	A_{00} removed from GPU_1
5	$A_{11} := A_{11} - A_{10}A_{10}^{T}$	GPU_3	$\mathbf{A_{11}}$	CPU $\longrightarrow$ GPU_3	A_{11} kept in GPU_3
			A_{10}	CPU $\longrightarrow$ GPU_3	
			$\mathbf{A_{11}}$	CPU $\longleftarrow$ GPU_3	A_{10} removed from GPU_3
6	$A_{21} := A_{21} - A_{20}A_{10}^{T}$	GPU_2	$\mathbf{A_{21}}$	CPU $\longrightarrow$ GPU_2	A_{21} kept in GPU_2
			A_{20}	CPU $\longrightarrow$ GPU_2	
			A_{10}	CPU $\longrightarrow$ GPU_2	
			$\mathbf{A_{21}}$	CPU $\longleftarrow$ GPU_2	A_{20}, A_{10} removed from GPU_2
7	$A_{31} := A_{31} - A_{30}A_{10}^{T}$	GPU_3	$\mathbf{A_{31}}$	CPU $\longrightarrow$ GPU_3	A_{31} kept in GPU_3
			A_{30}	CPU $\longrightarrow$ GPU_3	
			A_{10}	CPU $\longrightarrow$ GPU_3	
			$\mathbf{A_{31}}$	CPU $\longleftarrow$ GPU_3	A_{30}, A_{10} removed from GPU_3
8	$A_{22} := A_{22} - A_{20}A_{20}^{T}$	GPU_0	$\mathbf{A_{22}}$	CPU $\longrightarrow$ GPU_0	A_{22} kept in GPU_0
			A_{20}	–	A_{20} already in GPU_0
			$\mathbf{A_{22}}$	CPU $\longleftarrow$ GPU_0	A_{20} removed from GPU_0
9	$A_{32} := A_{32} - A_{30}A_{20}^{T}$	GPU_1	$\mathbf{A_{32}}$	CPU $\longrightarrow$ GPU_1	A_{32} kept in GPU_1
			$\mathbf{A_{30}}$	–	A_{30} already in GPU_1
			A_{20}	CPU $\longrightarrow$ GPU_1	
			$\mathbf{A_{32}}$	CPU $\longleftarrow$ GPU_1	A_{20} removed from GPU_1
10	$A_{33} := A_{33} - A_{30}A_{30}^{T}$	GPU_3	$\mathbf{A_{33}}$	CPU $\longrightarrow$ GPU_3	A_{33} kept in GPU_3
			A_{30}	CPU $\longrightarrow$ GPU_3	
			$\mathbf{A_{33}}$	CPU $\longleftarrow$ GPU_3	A_{30} removed from GPU_3
11	$A_{11} := \text{CHOL}(A_{11})$	GPU_3	$\mathbf{A_{11}}$	–	A_{11} already in GPU_3
			$\mathbf{A_{11}}$	CPU $\longleftarrow$ GPU_3	
12	$A_{21} := A_{21}A_{11}^{-T}$	GPU_2	$\mathbf{A_{21}}$	–	A_{21} already in GPU_2
			A_{11}	CPU $\longrightarrow$ GPU_2	
			$\mathbf{A_{21}}$	CPU $\longleftarrow$ GPU_2	A_{11} removed from GPU_2
⋮					

# writes in the example	20	**# writes in the complete execution**	28
# reads in the example	12	**# reads in the complete execution**	16

Table 5.3: List of tasks executed by the runtime (Version 2) and data transfers needed to perform the Cholesky factorization of a blocked 4×4 matrix. Blocks in **bold** are owned by the corresponding GPU, according to a 2-D distribution.

The strategy illustrated in the previous section reduces the number of transfers from main memory to GPU memory of blocks that are modified by each GPU, but still produces a large amount of transfers of read-only blocks. These are blocks considered as *non-proprietary* or *alien* for a given processor. In this variant we implement a software cache of read-only blocks in each GPU memory to maintain recently used *non-proprietary* blocks. Table 5.4 shows an example of the Cholesky factorization on a multi-GPU system with four GPUs, with a software cache mechanism operating in place. As this technique is compatible with those explained above, both can be exploited simultaneously.

Consider, for example, the effect of this mechanism when GPU_1 solves the linear systems $A_{10} := A_{10}\text{TRIL}(A_{00})^{-\text{T}}$ and $A_{30} := A_{30}\text{TRIL}(A_{00})^{-\text{T}}$. A copy of A_{00} is transferred from main memory to the cache in the memory of GPU_1 before the first linear system is solved (task number 4 in the table, marked as HIT), and remains there for the solution of the second linear system, saving a second transfer.

As the software cache only stores *alien* blocks, they can be externally modified at any time of the execution. In this situation, incoherence can appear between copies in the software caches of any GPU that stores a given block and the proprietary GPU that has modified it. Thus, a cache coherence policy becomes necessary to guarantee a proper management of the potential incoherence between the state of the blocks in different memory spaces. In this version of the runtime, to complement the cache system, when a task which updates a given block is completed, the thread in the CPU in charge of its execution invalidates all read-only copies of that block in the memory of the other GPUs in the system, following a write-invalidate policy.

From a high-level perspective, we consider the memory of each GPU as a cache of the main memory, storing only those blocks that are *alien* to a given processor. The management of the cache is done exclusively in software by the runtime, and parameters such as the number cache lines, the associativity degree, or the replacement policy can be easily adapted by the programmer to different linear algebra algorithms or specific GPU configurations.

Algorithms 7 and 8 show the necessary steps to implement the functionality of the procedures `load_operands` and `manage_modified_operands`, respectively. Additional instructions to check if an input operand is either a *proprietary* or an *alien* block are added to the `load_operands` algorithm. Only if a block is proprietary and resides already in the memory of the corresponding GPU, the transfer is saved. The invalidate instruction is added to the `manage_modified_algorithms` to invalidate read only copies of the block in other GPU memories.

The replacement policy, currently LRU (Least-Recently Used first), and the number of blocks per cache can be easily modified in the runtime system.

The application of this improvement further reduces the number of *writes* to GPU memories compared with those obtained by Version 2 of the runtime; see Table 5.4. In this case, the reduction in the number of *writes* due to the lack of transfers of recently-used *own* blocks, and also to the reduction in the number of transfers of recently used *alien* blocks. As the completion of a task involves the automatic transfer of output operands (write-through policy), there is no reduction in the number of *reads* performed by the system.

Version 4: Software cache + write-back

The purpose of this version is to attain an additional reduction in the number of transfers from the memory of the GPUs to main memory that occur when *proprietary* blocks are updated by each GPU. In this case, the write-through policy is abandoned in favor of a write-back strategy which allows inconsistencies between proprietary blocks in the memory of the GPUs and main memory. Thus, blocks written by a GPU are updated in main memory only when a different GPU has to

			Runtime Version 3		
#	Instruction		Block	Transfer	Comment (Cache status)
1	$A_{00} := \text{CHOL}(A_{00})$	GPU_0	$\mathbf{A_{00}}$	CPU $\longrightarrow$ GPU_0	$G_0(-), G_2(-)$
			$\mathbf{A_{00}}$	CPU $\longleftarrow$ GPU_0	$G_1(-), G_3(-)$
2	$A_{10} := A_{10}A_{00}^{-T}$	GPU_1	$\mathbf{A_{10}}$	CPU $\longrightarrow$ GPU_1	$G_0(-), G_2(-)$
			A_{00}	CPU $\longrightarrow$ GPU_1	$G_1(A_{00}), G_3(-)$
			$\mathbf{A_{10}}$	CPU $\longleftarrow$ GPU_1	
3	$A_{20} := A_{20}A_{00}^{-T}$	GPU_0	$\mathbf{A_{20}}$	CPU $\longrightarrow$ GPU_0	$G_0(-), G_2(-)$
			$\mathbf{A_{00}}$	–	$G_1(A_{00}), G_3(-)$
			$\mathbf{A_{20}}$	CPU $\longleftarrow$ GPU_0	
4	$A_{30} := A_{30}A_{00}^{-T}$	GPU_1	$\mathbf{A_{30}}$	CPU $\longrightarrow$ GPU_1	$G_0(-), G_2(-)$
			A_{00}	HIT	$G_1(A_{00}), G_3(-)$
			$\mathbf{A_{30}}$	CPU $\longleftarrow$ GPU_1	
5	$A_{11} := A_{11} - A_{10}A_{10}^{T}$	GPU_3	$\mathbf{A_{11}}$	CPU $\longrightarrow$ GPU_3	$G_0(-), G_2(-)$
			A_{10}	CPU $\longrightarrow$ GPU_3	$G_1(A_{00}), G_3(A_{10})$
			$\mathbf{A_{11}}$	CPU $\longleftarrow$ GPU_3	
6	$A_{21} := A_{21} - A_{20}A_{10}^{T}$	GPU_2	$\mathbf{A_{21}}$	CPU $\longrightarrow$ GPU_2	$G_0(-), G_2(A_{20}, A_{10})$
			A_{20}	CPU $\longrightarrow$ GPU_2	$G_1(A_{00}), G_3(A_{10})$
			A_{10}	CPU $\longrightarrow$ GPU_2	
			$\mathbf{A_{21}}$	CPU $\longleftarrow$ GPU_2	
7	$A_{31} := A_{31} - A_{30}A_{10}^{T}$	GPU_3	$\mathbf{A_{31}}$	CPU $\longrightarrow$ GPU_3	$G_0(-), G_2(A_{20}, A_{10})$
			A_{30}	CPU $\longrightarrow$ GPU_3	$G_1(A_{00}), G_3(A_{10}, A_{30})$
			A_{10}	HIT	
			$\mathbf{A_{31}}$	CPU $\longleftarrow$ GPU_3	
8	$A_{22} := A_{22} - A_{20}A_{20}^{T}$	GPU_0	$\mathbf{A_{22}}$	CPU $\longrightarrow$ GPU_0	$G_0(-), G_2(A_{20}, A_{10})$
			A_{20}	–	$G_1(A_{00}), G_3(A_{10}, A_{30})$
			$\mathbf{A_{22}}$	CPU $\longleftarrow$ GPU_0	
9	$A_{32} := A_{32} - A_{30}A_{20}^{T}$	GPU_1	$\mathbf{A_{32}}$	CPU $\longrightarrow$ GPU_1	$G_0(-), G_2(A_{20}, A_{10})$
			$\mathbf{A_{30}}$	–	$G_1(A_{00}, A_{20}), G_3(A_{10}, A_{30})$
			A_{20}	CPU $\longrightarrow$ GPU_1	
			$\mathbf{A_{32}}$	CPU $\longleftarrow$ GPU_1	
10	$A_{33} := A_{33} - A_{30}A_{30}^{T}$	GPU_3	$\mathbf{A_{33}}$	CPU $\longrightarrow$ GPU_3	$G_0(-), G_2(A_{20}, A_{10})$
			A_{30}	HIT	$G_1(A_{00}, A_{20}), G_3(A_{10}, A_{30})$
			$\mathbf{A_{33}}$	CPU $\longleftarrow$ GPU_3	
11	$A_{11} := \text{CHOL}(A_{11})$	GPU_3	$\mathbf{A_{11}}$	–	$G_0(-), G_2(A_{20}, A_{10})$
			$\mathbf{A_{11}}$	CPU $\longleftarrow$ GPU_3	$G_1(A_{00}, A_{20}), G_3(A_{10}, A_{30})$
12	$A_{21} := A_{21}A_{11}^{-T}$	GPU_2	$\mathbf{A_{21}}$	–	$G_0(-), G_2(A_{20}, A_{10}, A_{11})$
			A_{11}	CPU $\longrightarrow$ GPU_2	$G_1(A_{00}, A_{20}), G_3(A_{10}, A_{30})$
			$\mathbf{A_{21}}$	CPU $\longleftarrow$ GPU_2	
			⋮		

# writes in the example	17	**# writes in the complete execution**	25
# reads in the example	12	**# reads in the complete execution**	16

Table 5.4: List of tasks executed by the runtime (Version 3) and data transfers needed to perform the Cholesky factorization of a blocked 4×4 matrix. Blocks in **bold** are owned by the corresponding GPU, following a 2-D distribution. Comments illustrate the status of the cache of each GPU during the execution.

Algorithm 7 Algorithm for loading operands to GPU memory. Version 3.

```
for all operand in operands( task ) do
  if own_operand ( operand, th_id ) then
    if not_in_GPU ( operand, th_id ) then
      allocate_operand( operand )
      transfer_to_GPU( operand )
    else
      {Own block is on GPU. Skip transfer.}
    end if
  else
    if not_in_cache ( operand ) then
      look_for_empty_cache_line( operand )
      transfer_to_GPU( operand )
    end if
  end if
end for
```

Algorithm 8 Algorithm for updating operands in main memory. Version 3.

```
for all operand in operands( task ) do
  if output_operand( operand ) then
    transfer_to_main_memory( operand )
    invalidate_in_other_caches( operand )
  end if
end for
```

compute a task that involves them. A software cache for read-only blocks and the write-invalidate policy are still in operation.

When the execution of the complete algorithm is completed, the data matrix in main memory must be updated with the contents of the blocks that have been updated in the memory of the proprietary GPU.

As an example of the benefits of this technique, consider operation #3 ($A_{20} := A_{20}A_{00}^{-T}$, performed by GPU$_0$) and operation #6 ($A_{21} := A_{21} - A_{20}A_{10}^{T}$, performed by GPU$_2$) shown in Table 5.5. The first TRSM writes block A_{20} but, instead of writing the result immediately to main memory, the transfer is delayed until another GPU needs it as an input operand. In this case, the GEMM operation performed by GPU$_2$ involves block A_{20}. At this point, the thread that controls this GPU asks for this block to the proprietary GPU and waits until the block is flushed back to main memory (see the comment "A_{20} flushed by GPU$_0$"). Only when this flush occurs, GPU$_2$ can proceed with the execution of the task.

Note that the benefits of this technique are given in terms of blocks written by a GPU that are no longer needed by other GPUs, as they will be only flushed to main memory at the end of the execution, not every time the block is updated by the proprietary GPU. Analyzing the summarized results from Table 5.5, the application of this strategy leads to a reduction in the number of *reads* or data transfers to main memory. The number of *writes* is the same as that of Version 3, as the behavior of the runtime for inherited from that version.

Algorithms 9 and 10 illustrate the functionality of the procedures `load_operands` and `manage_modified_operands` in Version 4 of the runtime, respectively. Upon completion of each task, the necessary steps are the same as those shown in Version 3 of the runtime, applying a write-invalidate for *proprietary* blocks. The modifications appear in the `load_operands` procedure and, more specifically, in the steps necessary to load an *alien* block that is invalid in cache. In this situation, the thread in charge of the GPU in which the task will be executed asks for the block to the GPU that owns the modified block. The execution of the task stalls until that request is serviced by the corresponding thread, and the block is flushed to main memory. At that point, the executing thread is able to continue with the execution.

			Runtime Version 4		
#	Instruction		Block	Transfer	Comment
1	$A_{00} := \text{CHOL}(A_{00})$	GPU_0	$\mathbf{A_{00}}$	CPU $\longrightarrow$ GPU_0	
			$\mathbf{A_{00}}$	–	
2	$A_{10} := A_{10}A_{00}^{-T}$	GPU_1	$\mathbf{A_{10}}$	CPU $\longrightarrow$ GPU_1	
			A_{00}	CPU $\longrightarrow$ GPU_1	A_{00} flushed by GPU_0
			$\mathbf{A_{10}}$	–	
3	$A_{20} := A_{20}A_{00}^{-T}$	GPU_0	$\mathbf{A_{20}}$	CPU $\longrightarrow$ GPU_0	
			$\mathbf{A_{00}}$	–	
			$\mathbf{A_{20}}$	–	
4	$A_{30} := A_{30}A_{00}^{-T}$	GPU_1	$\mathbf{A_{30}}$	CPU $\longrightarrow$ GPU_1	
			A_{00}	HIT	
			$\mathbf{A_{30}}$	–	
5	$A_{11} := A_{11} - A_{10}A_{10}^{T}$	GPU_3	$\mathbf{A_{11}}$	CPU $\longrightarrow$ GPU_3	
			A_{10}	CPU $\longrightarrow$ GPU_3	A_{10} flushed by GPU_1
			$\mathbf{A_{11}}$	–	
6	$A_{21} := A_{21} - A_{20}A_{10}^{T}$	GPU_2	$\mathbf{A_{21}}$	CPU $\longrightarrow$ GPU_2	
			A_{20}	CPU $\longrightarrow$ GPU_2	A_{20} flushed by GPU_0
			A_{10}	CPU $\longrightarrow$ GPU_2	
			$\mathbf{A_{21}}$	–	
7	$A_{31} := A_{31} - A_{30}A_{10}^{T}$	GPU_3	$\mathbf{A_{31}}$	CPU $\longrightarrow$ GPU_3	
			A_{30}	CPU $\longrightarrow$ GPU_3	A_{30} flushed by GPU_1
			A_{10}	HIT	
			$\mathbf{A_{31}}$	–	
8	$A_{22} := A_{22} - A_{20}A_{20}^{T}$	GPU_0	$\mathbf{A_{22}}$	CPU $\longrightarrow$ GPU_0	
			A_{20}	–	
			$\mathbf{A_{22}}$	–	
9	$A_{32} := A_{32} - A_{30}A_{20}^{T}$	GPU_1	$\mathbf{A_{32}}$	CPU $\longrightarrow$ GPU_1	
			$\mathbf{A_{30}}$	–	
			A_{20}	CPU $\longrightarrow$ GPU_1	
			$\mathbf{A_{32}}$	–	
10	$A_{33} := A_{33} - A_{30}A_{30}^{T}$	GPU_3	$\mathbf{A_{33}}$	CPU $\longrightarrow$ GPU_3	
			A_{30}	HIT	
			$\mathbf{A_{33}}$	–	
11	$A_{11} := \text{CHOL}(A_{11})$	GPU_3	$\mathbf{A_{11}}$	–	
			$\mathbf{A_{11}}$	–	
12	$A_{21} := A_{21}A_{11}^{-T}$	GPU_2	$\mathbf{A_{21}}$	–	
			A_{11}	CPU $\longrightarrow$ GPU_2	A_{11} flushed by GPU_3
			$\mathbf{A_{21}}$	–	
			⋮		

# writes in the example	17	**# writes in the complete execution**	25
# reads in the example	5	**# reads in the complete execution**	9

Table 5.5: List of tasks executed by the runtime (Version 4) and data transfers needed to perform the Cholesky factorization of a blocked 4×4 matrix. Blocks in **bold** are owned by the corresponding GPU, following a 2-D distribution.

Algorithm 9 Algorithm for loading operands to GPU memory. Version 4.

```
for all operand in operands( task ) do
  if own_operand (operand, th_id ) then
    if not_in_GPU (operand, th_id ) then
      allocate_operand( operand )
      transfer_to_GPU( operand )
    else
      {Own block is on GPU. Skip transfer.}
    end if
  else
    if not_in_cache( operand ) then
      ask_for_update( operand, th_owner )
      while not_updated_in_RAM do
        wait_for_update
      end while
      look_for_empty_cache_line( operand )
      transfer_to_GPU( operand )
    end if
  end if
end for
```

Algorithm 10 Algorithm for updating operands in main memory. Version 4.

```
for all operand in operands( task ) do
  if output_operand( operand ) then
    invalidate_in_other_caches( operand )
  end if
end for
```

The mechanism to conduct the requests of *alien* blocks is implemented by taking benefit from the private queue of ready tasks owned by each thread in the run-time system. As soon as a GPU needs an *alien* block to continue with its execution, a request is introduced in the queue of the thread that owns the valid copy of the block (that is always the proprietary of the block). This request can be viewed as a special task with no associated functionality other than signaling the thread to flush a given data block to main memory. On the other side of the communication, the proprietary thread polls for new ready tasks; if a request task is found, the flush of the corresponding block is carried out. In the general procedure executed by each thread (see Algorithm 2), this functionality is implemented in procedure `check_for_pending_transfers` each time a new task is ready for execution. This procedure just checks if the task just extracted corresponds to a request task; in that case, the effective data transfer is performed. To avoid the active wait shown in Algorithm 2, the waiting thread can re-enqueue the task and proceed with the execution of an alternative ready task while the necessary data transfer is satisfied. Only when all the operands of the task are effectively in GPU memory, the execution is performed.

Summary of changes

Each improvement introduced in the runtime aims at reducing the amount of data transfers necessary to execute a given parallel algorithm. Table 5.6 summarizes the techniques and benefits introduced by each version of the runtime. As can be observed, whenever possible the improvements are carried over successive versions, as the benefits introduced by each one are independent:

Version 1: Basic version. The execution of a task implies the transfer of all input operands from main memory to GPU memory, as well as the output operands from GPU memory to main memory.

	1	2	3	4
Techniques	-	Data-affinity	Data-affinity	Data-affinity
	-	Own/alien blocks	Own/alien blocks	Own/alien blocks
	-	Write-through	Write-through	Write-back
Software cache	-	-	*Alien* blocks cached	*Alien* blocks cached
	-	-	Write-invalidate	Write-invalidate
Benefits	-	Reduce number of writes to GPU (*own* blocks)	Reduce number of writes to GPU (*alien* blocks)	Reduce number of reads from GPU

Table 5.6: Summary of the techniques and benefits introduced by the successive improvements in the runtime.

Version 2: Static data layout combined with write-through. The number of *writes* of *own* blocks is reduced as *own* blocks already transferred to a GPU are not transferred there again by future instructions.

Version 3: Software-cache of *alien* blocks. The number of *writes* of *alien* blocks is reduced by keeping them in a cache managed in software.

Version 4: Software-cache of *alien* blocks with write-back. The number of *reads* of *own* blocks is reduced by only maintaining the coherence between main memory and GPU memories when it is strictly necessary.

As an additional advantage, the improvements introduced in the runtime and the design of the runtime itself are independent from the underlying architecture. Assume a system consisting of a machine with a central memory pool with multiple hardware accelerators connected to it, each one with an independent memory space. With this setup, the techniques exposed above and their benefits can implemented with minor modifications.

5.4. Experimental results

5.4.1. Impact of the block size

The block size is a critical factor to attain high performance in algorithms-by-blocks. Independently of the techniques used to reduce data transfers, to improve data affinity, etc. an incorrect election of this parameter lead to sub-optimal performance. The first set of experiments aims at determining the optimal value for the block size, as well as its effect on the final performance of the parallel implementation.

In our case, the optimal value for the block size is a trade-off between a number of factors:

- *Potential parallelism*: a smaller block size is translated into a finer granularity. A larger block dimension leads to less concurrency. Given the pool of execution units available in the system, the size of the block greatly influences the degree of parallel execution on the sub-problems, and reduces idle times.
- *Data transfer efficiency*: a small block size is translated into a larger number of data transfers of reduced dimension. The latency of the PCI-Express bus plays a critical role in this case, and data transfers of small blocks have a significant performance penalty. Conversely, a

larger block dimension translates into higher effective bandwidth, and thus benefits the final performance of the implementation.

- *Inner BLAS kernels performance*: as shown in Chapter 3, the BLAS implementations on the GPU deliver higher performances when operating with relatively large matrices, as those derived from the utilization of a big block size.

Taking into account the combination of these factors, the performance of parallel implementations will be affected by the chosen block size from two perspectives:

1. If the block size is too small, data transfers become a serious bottleneck in the computation, and BLAS executions are inefficient on the GPU.

2. If the block size is too large, the degree of concurrency decreases and resources are wasted. In addition, data transfers through the bus are efficient, and so are BLAS executions.

In summary, a thorough analysis of the performance of the runtime system for a given linear algebra algorithm-by-blocks should show an increasing performance as the block size is increased (and thus, BLAS executions and data transfers are more efficient), attaining a peak point as potential parallelism in the algorithm reaches the optimal point. From this dimension on, parallelism decreases, and thus parallel efficiency should drop.

Consider for example the performance results in Figure 5.7. The figure shows the performance of the right-looking variant of the Cholesky factorization in GFLOPS (Y axis) for the most tuned version of the runtime implementation for several matrix sizes (Z axis), with block size (X axis) increasing from a relatively small size ($b = 128$) to a larger one ($b = 1600$). In this example, the four GPUs available on TESLA2 were used, following a 2-D data layout. Similar results have been observed for the rest of the implementations of the runtime and data affinity options. The figure offers information on the optimal block size for each matrix size.

As predicted, the behavior of the performance curves for each matrix size is quite regular and predictable. In each case, performance is increased as the block size grows until it reaches an optimal value. From this point on, a drop in the performance rate occurs. These results demonstrate the assumptions made on the BLAS and data transfer performance, and potential parallelism. In fact, note how the optimal block size (shown in the figure as a green line) is also increased as matrix size grows. Thus, taking into account only the potential parallelism factor, the optimal block size partially depends on the total matrix dimension.

For the rest of the experimental results in this chapter, performance values for the optimal block size are shown. A complete analysis of the performance attained using a wide range of block sizes has been carried out for each operation and runtime version.

5.4.2. Number of data transfers

Once the optimal block size is found and fixed for a given matrix size, the benefits of the different implementations of the runtime are mostly dictated by the number of data transfers necessary to perform a given operation. In fact, the reduction in the number of data transfers is the main goal of the successive improvements in the runtime implementation.

To demonstrate the benefits of the incremental improvements introduced from a basic runtime version, the next experiment evaluates the performance of the implementation of the algorithm-by-blocks for the Cholesky factorization (right-looking variant) on the four GPUs of TESLA2, for a fixed block size ($b = 768$), using the four versions of the runtime, and a 2-D data layout when

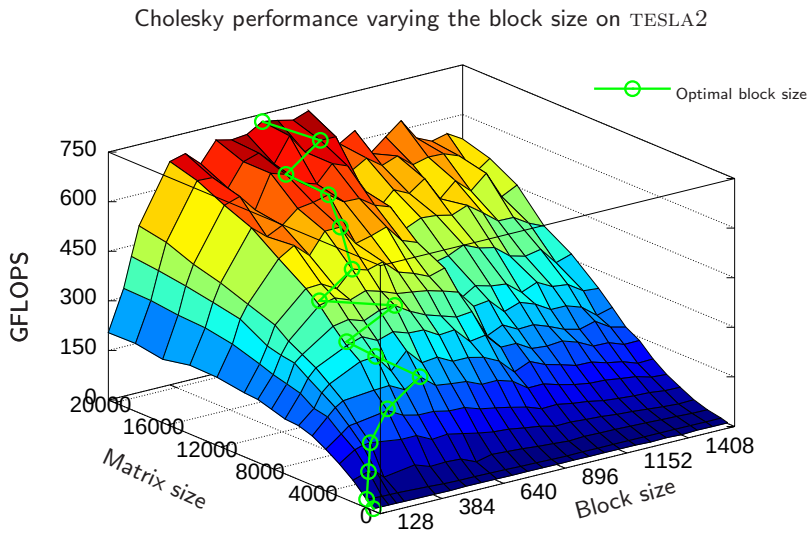


Figure 5.7: Performance of the Cholesky factorization using the 4 GPUs in TESLA2, varying the block size. Runtime version 4.

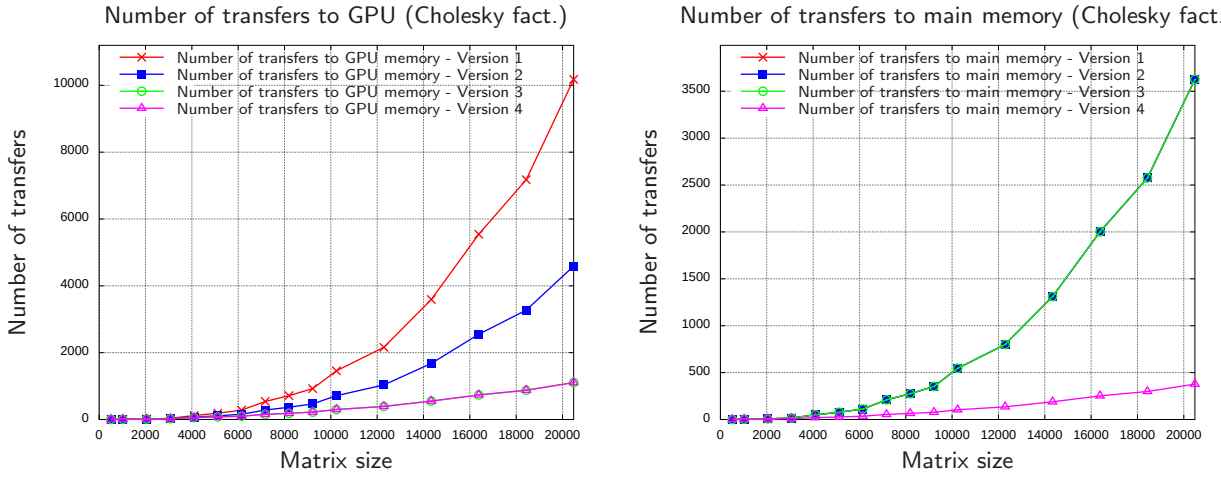


Figure 5.8: Number of data transfers for the Cholesky factorization using 4 GPUs on TESLA2. Block size is $b = 768$.

appropriate (versions 2 to 4). In this case, instead of raw performance attained by each one version, we analyze the number of data transfers to and from the GPU memory. We consider the number of *writes* (transfers to GPU memory) and *reads* (transfers to main memory) for each runtime version. Thus, the aim of the experiment is to evaluate the impact on the number of data transfers, and to demonstrate how each improvement in the runtime affects the writes or the reads of data (or even both).

Figure 5.8 reports the amount of data transfers for the experiment described above. The left-hand side plot shows the amount of *writes* for each implementation of the runtime; the plot on the right-hand side shows the corresponding number of *reads*.

Let us focus first on the left plot. The number of writes is dramatically reduced by the application of two different techniques:

- Data-affinity and the concept of *own* and *alien* blocks in version 2 of the runtime greatly reduces the amount of necessary data transfers from main memory to GPU memory for those blocks considered as property of the assigned GPU. As an example, consider the reduction in the number of data *writes* for a matrix size of $n = 20{,}480$. In this case, data transfers from main to GPU memory are reduced from 10,179 in the basic runtime implementation to 4,590 in the implementation introducing data-affinity, saving thus more than half of the total data transfers.

- The usage of a cache of *alien* blocks in the GPU further decreases the number of data *writes*, as non-proprietary blocks that are read once by a given GPU do not have to be transferred again if used in a future instruction (unless they are invalidated by the proprietary of the block). This improvement is shared by versions 3 and 4 of the runtime, as the way caches work in both versions is common from the point of view of the data *writes*. For a matrix size $n = 20{,}480$, the number of writes is reduced from 10,179 to 1,106.

To sum up, the introduction of data-affinity and a software cache can reduce the amount of transfers from main memory to GPU memory in a factor $9\times$ for the tested matrix sizes.

An inspection of data *reads* reveals that the benefits are similar. In this case, no transfer reduction is introduced in the three first versions of the runtime. However, the introduction of *write-back* as the coherency policy between main memory and GPU memory in version 4 reduces the amount of *reads* dramatically. For instance, for a matrix size $n = 20{,}480$, the amount of *reads* from main to GPU memory is reduced from 3,627 in versions 1-3 to 377 in version 4 of the runtime.

5.4.3. Performance and scalability

The progressive reduction of the amount of data transfers derived from the improvements introduced in the run-time system yields direct gains in the performance of the implemented linear algebra routines. To evaluate how this reduction impacts the final performance of the parallel executions, we show the rates attained for the Cholesky factorization using the four GPUs of TESLA2. For this experiment, all BLAS routines are mapped to NVIDIA CUBLAS calls. The factorization of the diagonal blocks is performed in the multi-core processor, following the hybrid approach introduced in Chapter 4, as the optimal block size is small and this particular operation is more suited for this type of processor.

We propose an evaluation of the performance and scalability of the three algorithmic variants for the Cholesky factorization analogous to those shown in Chapter 4 (see Figure 4.1). We evaluate how the successive improvements introduced in the transfers management affect the performance of each algorithmic variant.

Figure 5.9 shows the performance attained for the Cholesky factorization of a matrix of increasing size on the four versions of the runtime, with 2-D data distribution for versions 2-4. The plots show the performance of the algorithmic variants (Variant 1 in the top plot, Variant 2 in the middle plot and Variant 3 in the bottom plot).

A careful observation of the performance results offers an idea of the impact of the reduction of data transfers while improving data transfer management. As an starting point, consider in the following discussion the results attained for Variant 1 of the algorithm. Peak performance for the basic implementation (Version 1 of the runtime), in which data is transferred to GPU memory prior to the execution of the task and retrieved back after its conclusion, is 357 GFLOPS. The usage of data affinity and the storage of *own* blocks in the corresponding GPU (Version 2 of the runtime) boosts this performance up to 467 GFLOPS. With the introduction of the software cache mechanism with a *write-through* coherence policy between main memory an GPU memory (Version 3 of the runtime) performance is increased to 544 GFLOPS. Finally, the use of a combination of software cache and a *write-back* policy is translated into the best raw performance, attaining a peak of 705 GFLOPS. Note that this performance is not still fully stabilized, and could be further increased for larger matrices, provided there was enough memory space in the aggregated GPU memory space.

Similar qualitative results and improvements are shown for algorithmic variants 2 and 3 (middle and bottom plots, respectively). The speedup ratios are similar to those achieved for the first variant using different runtime versions. Considering raw performance, variant 3 attains a peak performance of 798 GFLOPS, whereas variants 1 and 2 achieve 735 GFLOPS and 751 GFLOPS, respectively. As was the case in the evaluation of single-GPU implementations, these results demonstrate the benefits of providing a whole family of algorithmic variants for multi-GPU systems in order to find the optimal one for a given operation.

To analyze the scalability of the proposed software solution, in Figure 5.10 we evaluate the scalability (left-hand side plots) and reports the speed-up (right-hand side plots) of the different algorithmic variants of the algorithm-by-blocks.

As an illustrative example, consider the top plot, which corresponds to the algorithmic variant 1. No bottlenecks are revealed in the scalability of this variant: the performance of the system steadily improves as the number of GPUs is increased, and larger problem sizes report higher execution rates. The speed-ups are calculated comparing the performance attained by the algorithms-by-blocks using 2–4 GPUs with that of executing same algorithm on a single graphics processor. For the largest problem dimension, the results show speed-ups of 1.9×, 2.5×, and 2.9× using respectively 2, 3, and 4 GPUs for variant 1; 1.7×, 2.2×, and 3× for variant 2; and 1.8×, 2.6×, and 3.2× for variant 3. Note that the PCI-Express bus becomes a bottleneck as the number of GPUs increases. This justifies the necessity of a proper data transfer policy as those implemented in our runtime.

5.4.4. Impact of data distribution

Data layout is an additional factor that can affect the performance attained for a given operation using any of the tuned variants of the developed runtime. The performance differences are mostly explained by a higher data locality when using runtime versions that make use of software caches.

Figure 5.11 shows the performance of the proposed algorithmic variants for the Cholesky factorization using the four GPUs of TESLA2, and version 4 of the runtime. The experimental conditions are the same as those set in previous experiments. In this experiment, we report the performance results attained for three different data distributions: cyclic 2-dimensional (top plot), column-wise distribution (middle plot) and row-wise distribution (bottom plot).

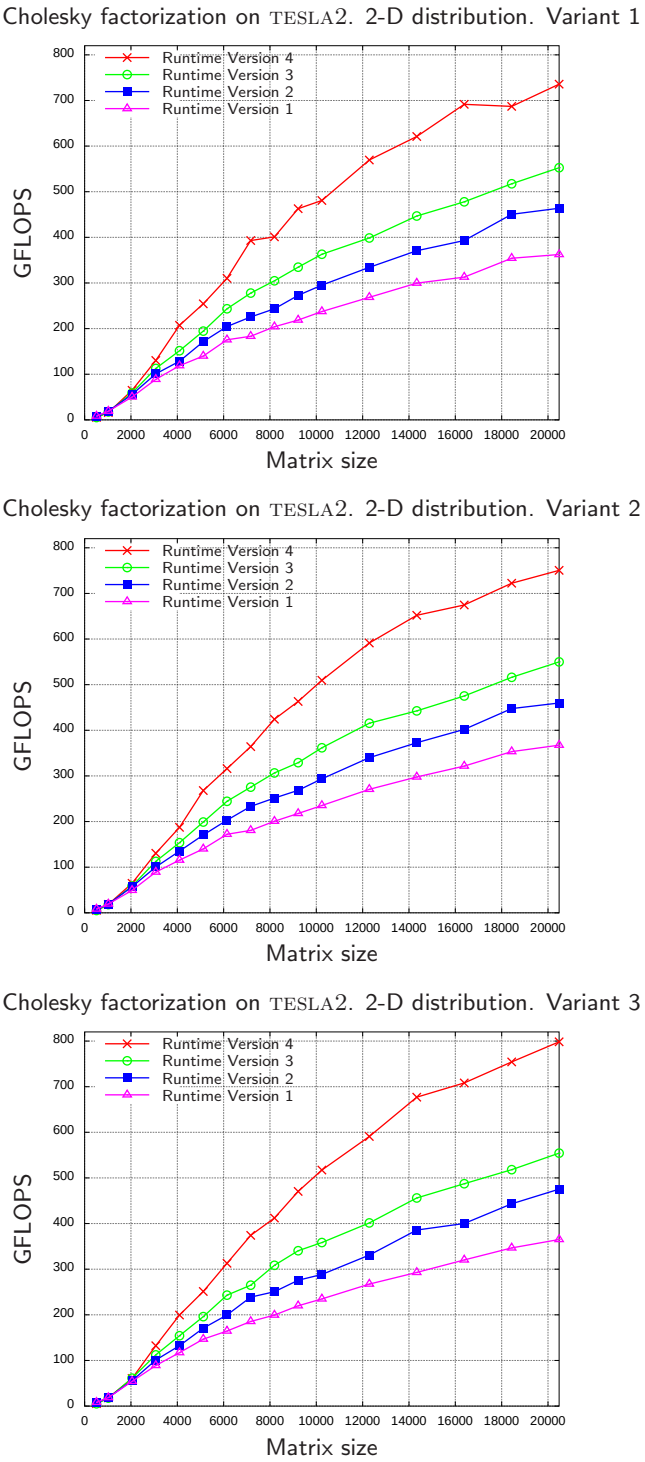


Figure 5.9: Performance comparison of different algorithmic variants of the Cholesky factorization using 4 GPUs on TESLA2, using a 2-D distribution. Top: algorithmic variant 1. Middle: algorithmic variant 2. Bottom: algorithmic variant 3.

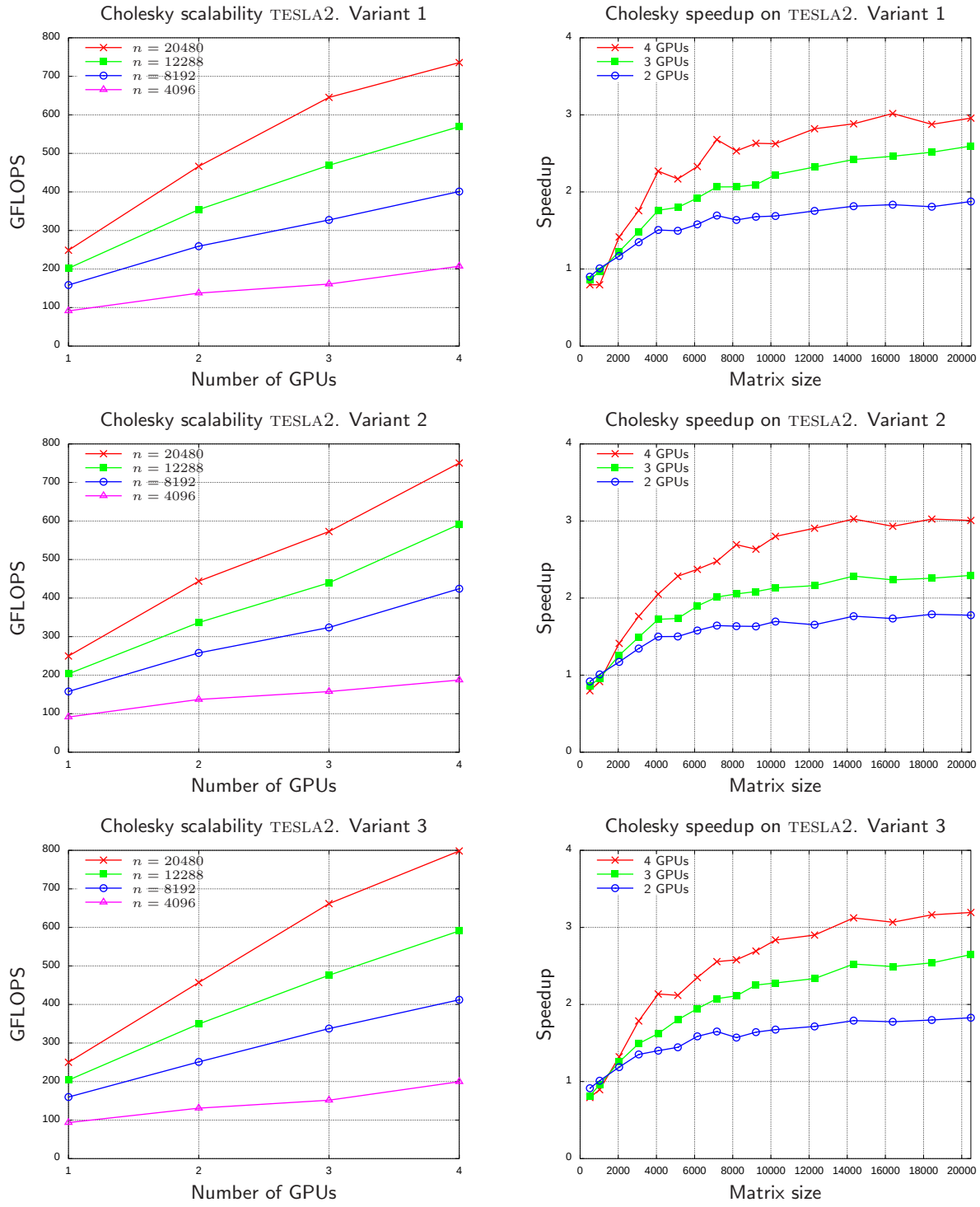


Figure 5.10: Scalability (left) and speedup (right) of the Cholesky factorization on TESLA2.

For the specific algorithmic variant implemented, the data distribution policy can play an important role on the final performance of the parallel implementations, especially for large matrices.

Variant 3 is the most efficient algorithmic variant independently from the selected data distribution, with negligible performance differences among them. On the other hand, consider the performance of Variant 3 for different data distributions and a large matrix dimension ($n = 20{,}480$). In this case, the performance attained by the runtime-based implementation varies from 741 GFLOPS for the 2-D cyclic distribution, to 753 GFLOPS for the column-wise layout and down to 625 GFLOPS for the row-wise distribution. Although smaller, there are also subtle differences in the performance of Variant 1 depending on the chosen data distribution.

The conclusion of this experiments is that spatial assignment of data (and therefore, tasks) to GPUs is also an important factor with relevant impact on the final performance, much like algorithmic variant, block size and data transfer policy were revealed in previous experiments. A careful study of this parameter becomes critical to achieve optimal results for every operation.

5.4.5. Performance comparison with other high-performance implementations

In order to gain a better understanding of previous results for the multi-GPU setup, Figure 5.12 compares the performances of the algorithm-by-blocks for the Cholesky factorization with those of optimized implementations of these operations on current high-performance platforms:

- `Algorithm-by-blocks` on four GPUs: Our algorithm-by-blocks for the Cholesky factorization, using the version 4 of the runtime system, and executed on TESLA2 using the four available GPUs.
- `Algorithm-by-blocks` on one GPU: Our algorithm-by-blocks for the Cholesky factorization, using the version 4 of the runtime system, and executed on TESLA2 using only one GPU.
- `Blocked-algorithm` on one GPU: Implementation of the hybrid Cholesky algorithm as presented at Chapter 4 on a single GPU of TESLA2. To be consistent with previous results, the time to transfer the data from main memory to GPU memory and retrieve the results is included.
- `Algorithm-by-blocks` on the multi-core processors of TESLA2: Our algorithms-by-blocks for the Cholesky factorization, using the SuperMatrix runtime system [45], and executed on TESLA2 using the its eight cores.
- MKL `spotrf` on the multi-core processors of TESLA2: A multi-threaded implementation of the corresponding LAPACK routine executed using the eight cores of TESLA2.

The benefits of using the runtime-based approach can be derived by comparing the performance results on 4 GPUs with those obtained for one GPU using the blocked algorithm. More precisely, the performance is increased from 267 GFLOPS for the mono-GPU blocked implementation to roughly 800 GFLOPS for the multi-GPU implementation.

In addition, the overhead introduced by the runtime is not important, as can be realized by comparing the results for the mono-GPU implementations using the blocked algorithm and the algorithm-by-blocks combined with the runtime (267 GFLOPS for the first implementation and 249 GFLOPS for the runtime-based implementation, for the largest tested matrices). The lack of code modifications to deal with multi-GPU systems further justifies the usage of the runtime-based approach and validates the solution also for mono-GPU architectures.

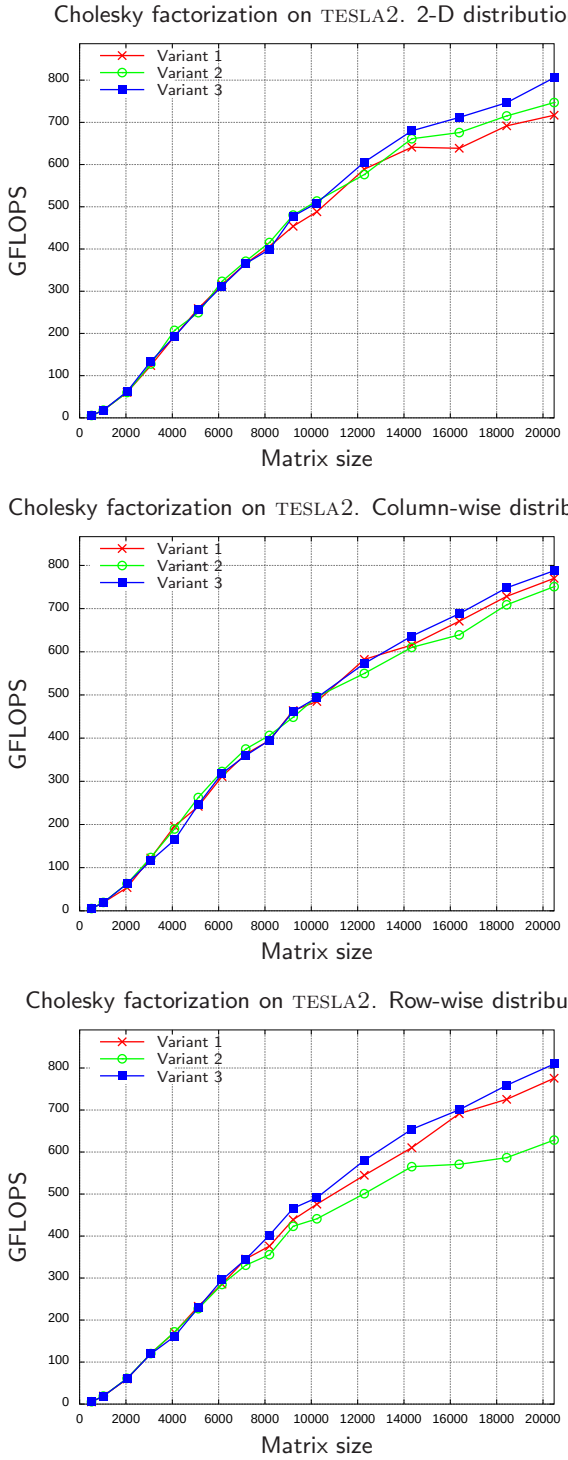


Figure 5.11: Performance comparison of different algorithmic variants of the Cholesky factorization using 4 GPUs on TESLA2. Top: 2-D distribution. Middle: column-wise distribution. Bottom: row-wise distribution.

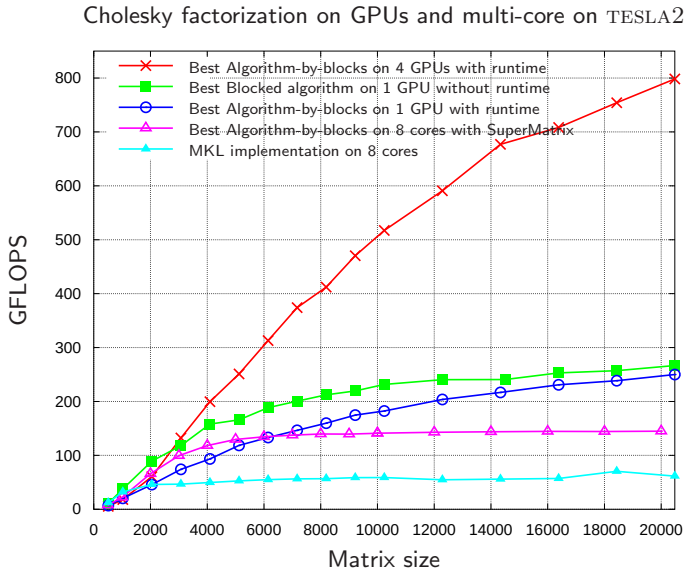


Figure 5.12: Performance of the Cholesky factorization using GPUs and multi-core on TESLA2.

The algorithm-by-blocks using the SuperMatrix runtime roughly attains 150 GFLOPS using the 8 cores in TESLA2. This figure shows how, with minor modifications in the FLAME codes, performance can be boosted from 150 GFLOPS to 800 GFLOPS by using the GPUs as the underlying architecture, implementing the same algorithm-by-blocks. Comparing the implementation of the algorithm-by-blocks with the optimized implementation in MKL gives an idea of the benefits of this type of algorithms, as the optimized library only attains 70 GFLOPS for the Cholesky factorization using the same experimental setup.

5.5. Multi-GPU implementations for the BLAS

The Cholesky factorization is an appropriate example to illustrate the possibilities of a runtime system like that developed for multi-GPU systems. The algorithm-by-blocks exhibits a number of characteristics that are appealing to exploit the benefits of the improvements introduced in the runtime (heterogeneity in the type of tasks, multiple data dependencies, use of the CPU or the GPU depending on the specific task type,...).

In this section, we report and analyze the performance results for three common BLAS routines, implemented using algorithms-by-blocks and on a multi-GPU system. These results illustrate how a common run-time system can be used as the base for several implementations, offering appealing performance results without affecting the programmability of the solution. In fact, the developed system can be applied to obtain a full BLAS implementation for multi-GPU systems, without any modification on existing algorithms-by-blocks for the routines.

Although minor new insights can be extracted from a deep analysis of the performance results for BLAS routines, the major part of the observations were already gained from the experimentation with the Cholesky factorization. Thus, we focus on reporting experimental results, comparing them with other high-performance implementations on systems with one GPU and/or multi-core architectures when possible. The main goal is to illustrate how the approach based on a runtime and minor modifications to the original FLAME code can be applied to a wide variety of linear algebra routines with appealing performance results.

5.5.1. Triangular system solve (TRSM)

The TRSM routine is a challenging implementation on multi-GPU platforms, since it presents a rich variety of data dependencies. In this case, we evaluate a particular case of the TRSM routine, even though similar results are expected for other cases.

In this case, the evaluation focuses on $XA^T = B$, where A is lower triangular and B is overwritten with the solution X. We show performance results obtained by performing three different experiments:

- *Evaluation of the performance of the runtime versions*: Figure 5.13 shows the performance of the TRSM implementation executed on 4 GPUs of TESLA2. The figure shows results for the four versions of the developed runtime. Like was the case for the Cholesky factorization, the reduction in the number of data transfers plays a significant role in the final performance of the parallel implementation. Performance results vary from 416 GFLOPS using Version 1 of the runtime up to 926 GFLOPS for version 4.

- *Comparison to single-GPU implementations*: Figure 5.14 shows a comparison of the performance of the multi-GPU implementation using the run-time system with two different mono-GPU implementations. The first one is the implementation offered by NVIDIA CUBLAS. The second one is the best implementation described in Chapter 3. For the largest matrices that a single GPU can hold, 206 GFLOPS are attained using NVIDIA CUBLAS on one GPU, 320 GFLOPS using our own implementation of the routine on one GPU, and 900 GFLOPS using the most tuned version of the runtime and four GPUs. Note that, for small matrices (up to $n = 3{,}072$ in the example), the single-GPU implementations are more efficient than the multi-GPU implementations, mainly due to the small amount of parallelism that can be extracted if a block size which is relatively large is chosen to attain high performance in the inner kernels.

- *Impact of the layout on the performance*: Unlike the conclusions extracted for the Cholesky factorization (see Figure 5.11), data distribution plays a fundamental role in the TRSM implementation on multi-GPU architectures. Figure 5.15 shows the performance of version 4 of the runtime using the 4 GPUs of TESLA2, for three different data distributions: 2-D cyclic, row-wise and column-wise. As reported in the plot, there are minor performance differences between the 2-D cyclic distribution and the row-wise distribution. However, the difference in performance between those two and the column-wise distribution is remarkable: 926 GFLOPS are attained for the 2-D cyclic distribution, but this rate is reduced to 731 GFLOPS for the column-wise distribution.

Multi-GPU implementations often deliver high gains for problems of large dimension. For instance, consider the TRSM performance in Figure 5.14. For a moderate matrix dimension ($n = 5{,}120$), single-GPU implementations achieve a performance rate that is close to the peak achieved for the largest tested matrix. However, the performance rate attained on 4 GPUs is still half of that attained for the largest matrix tested (425 GFLOPS vs. 926 GFLOPS). This is a frequent behavior of multi-GPU implementations and finally determines whether this type of architectures is appropriate to tackle a given problem depending on its size.

5.5.2. Symmetric rank-k update (SYRK)

The next study is focused on a representative case of this operation, $C := C - AA^T$, where C is symmetric and only the lower triangular part of this matrix is stored and computed. We carry

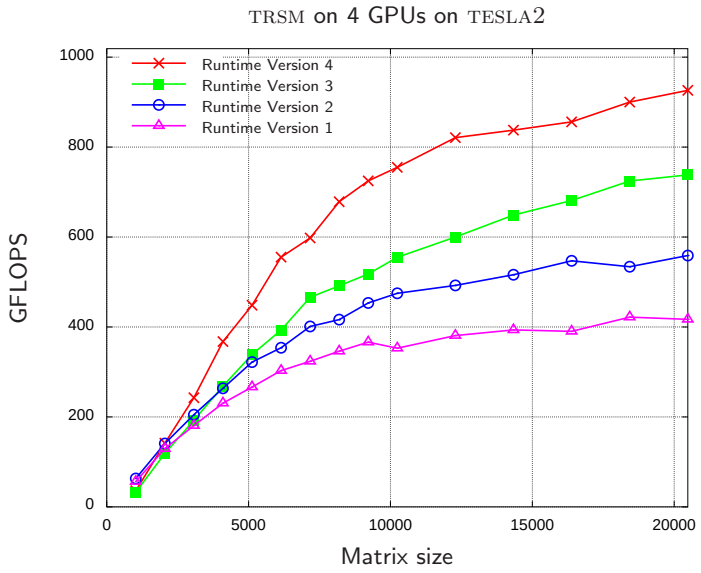


Figure 5.13: Performance of the TRSM implementation using 4 GPUs on TESLA2.

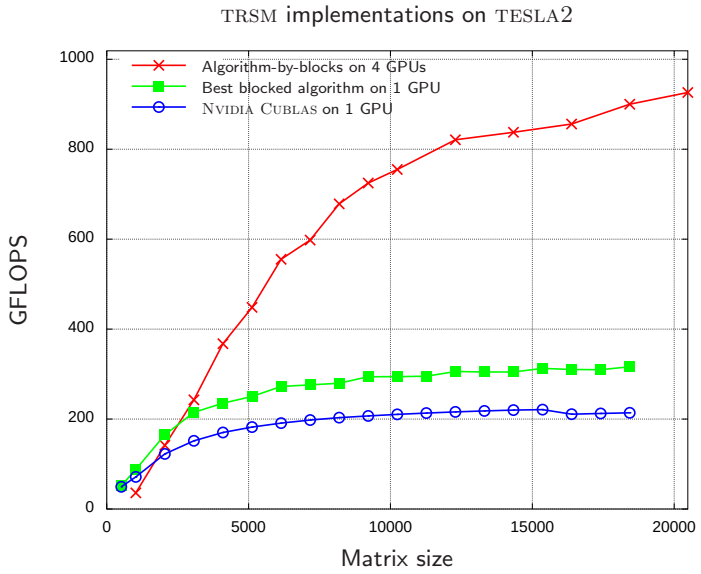


Figure 5.14: Performance of several TRSM implementations on TESLA2.

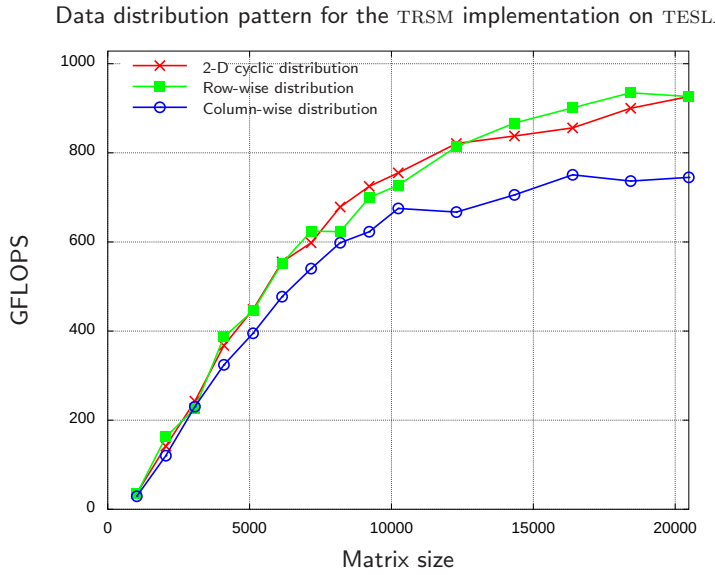


Figure 5.15: Impact of the data distribution pattern on the TRSM implementation on TESLA2.

out the analysis for operations with square matrices, although similar results are expected for other experimental configurations.

- *Evaluation of the performance of the runtime versions*: Figure 5.16 (left) reports the performance of the developed versions of the runtime for the SYRK routine on four GPUs of TESLA2. The performance results validate the successive refinements introduced in the runtime to reduce data transfers. Performance is increased from 402 GFLOPS in the basic version of the runtime to almost 1 TFLOP using the most sophisticated version.

- *Comparison to mono-GPU implementations*: Figure 5.16 (right) compares the best result attained using the run-time system on the multi-GPU setup (with four GPUs) with two different implementations of mono-GPU codes: the NVIDIA CUBLAS implementation and our own tuned implementation using blocked algorithms. The benefits of the run-time system are clear: for the largest tested matrices, performance is increased from 161 GFLOPS offered by NVIDIA CUBLAS and 321 GFLOPS of our tuned implementation to 965 GFLOPS of the multi-GPU configuration using the runtime on the four GPUs of TESLA2. Once more, the multi-GPU setup shows its potential for relatively large matrices, where distributing the computation among several GPUs is more appropriate.

5.5.3. Matrix-matrix multiplication (GEMM)

The matrix-matrix multiplication is usually evaluated on novel systems and its performance shown as illustrative of the performance that can be attained by the architecture. Following this trend, we report performance results for a specific case of matrix-matrix multiplication using our run-time system on TESLA2.

The study considers the particular case $C := C - AB^T$, where A, B, and C are square matrices. Similar results are expected for other experimental configurations. For this evaluation, we skip the detailed report of the performance offered by the different versions of the runtime, as the insights

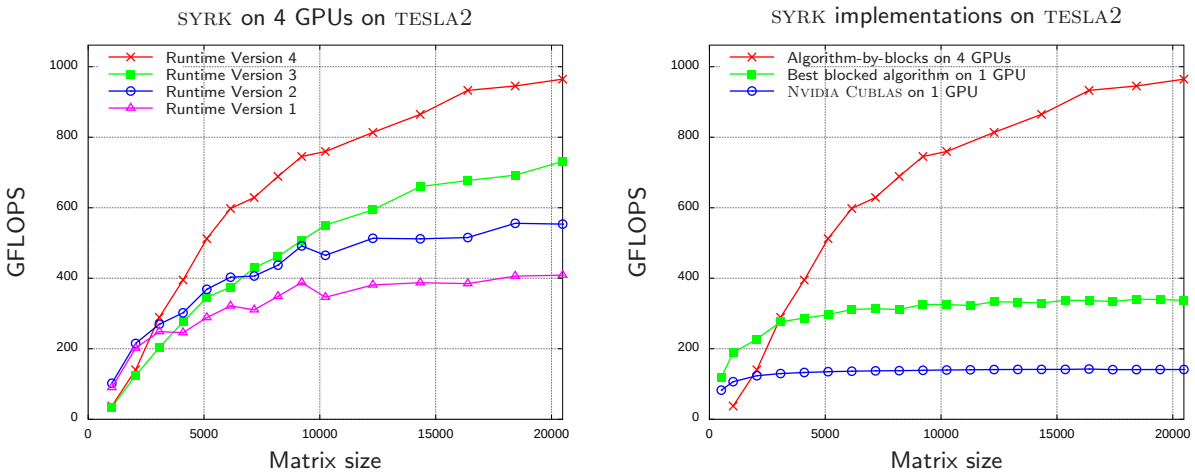


Figure 5.16: Performance (left) and comparison with mono-GPU implementations (right) of the SYRK implementation using 4 GPUs on TESLA2.

which could be gained from it are very similar to those already obtained for the Cholesky factorization and other BLAS routines. Instead, we perform a comparison of the algorithm-by-blocks on four GPUs of TESLA2, the same algorithm on only one GPU, our tuned blocked implementation on one GPU, and the NVIDIA CUBLAS implementation.

Figure 5.17 compares the best performance attained using the run-time system on the multi-GPU setup (with four GPUs) with the three mono-GPU implementations. Note that we attain more than 1 TFLOP of peak performance using four GPUs in the same system for the largest tested matrices. In particular, the performance rate for $n = 20{,}480$ is 1.1 TFLOP.

Compared with the single-GPU implementations, we attain 376 GFLOPS in our best blocked algorithm implementation (see Chapter 3 for more details), 295 GFLOPS in a mono-GPU implementation using the developed runtime, and 119 GFLOPS using the NVIDIA CUBLAS implementation.

5.6. Conclusions

The emergence of a new hardware architecture usually involves extensive efforts from the software point of view in order to exploit its full potential. Multi-GPU systems are not an exception, and several works have advocated for low-level ad-hoc implementations to fully exploit the huge performance available in this type of architectures.

Following the rationale of the rest of this thesis, our approach and main contribution is essentially different. We advocate for a high-level approach, which abstracts the library developer from the particularities of the underlying architecture, and still considers performance as the main goal of our implementations.

To accomplish this, our first contribution is a reformulation of multi-GPUs, viewing them as a multi-core architecture, and considering each GPU in the system as a single core. With this analogy, many well-known concepts and techniques successfully applied in the past for shared- and distributed-memory programming can be also applied to modern multi-GPU architectures.

However, there are specific characteristics of this kind of architectures that pose challenging difficulties for the implementation of efficient run-time systems; specifically, we refer to data transfers and separate memory spaces. In response to this problem, a second contribution of the chapter is

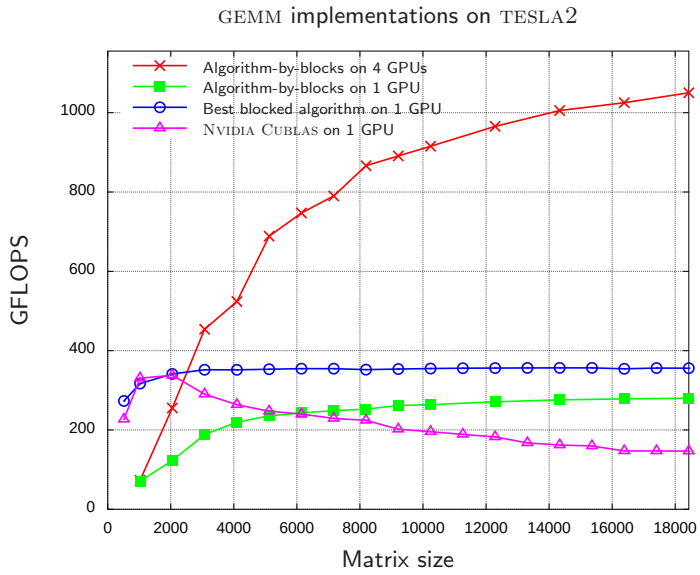


Figure 5.17: Performance comparison with mono-GPU implementations of the GEMM implementation using 4 GPUs on TESLA2.

a run-time system that is not only responsible of exploiting task parallelism, scheduling tasks to execution units or tracking data dependencies, but also transparently and efficiently handling data transfers between GPUs.

We have introduced techniques to reduce the amount of data transfers and thus increase data locality as well. We have also validated the efficiency of the runtime by evaluating many well-known dense linear algebra operations. The scalability and peak performance of the implementations is remarkable. Although the programmability of the solution is difficult to measure, the FLAME programming model allows a straightforward transition between existing sequential codes and parallel codes exploiting task parallelism.

Another remarkable contribution of the work is the fact that the major part of the concepts and techniques presented are not exclusive of a given runtime system or even a specific architecture. From this point of view, similar techniques have been applied by the author of the thesis to port the SMPSs runtime to platforms with multiple GPUs in a transparent way for the programmer [16]. This runtime (GPUSs) has been successfully tested with other type of hardware accelerators (ClearSpeed boards [50]) with similar performance results, is a clear demonstration of the portability of the proposed solution.

Part IV

Matrix computations on clusters of GPUs

CHAPTER 6

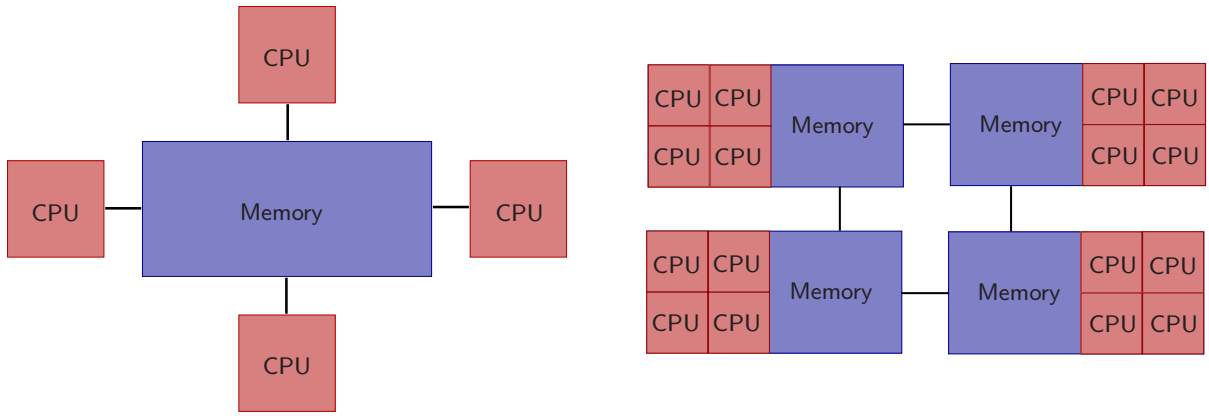
Matrix computations on clusters of GPUs

In the previous chapter, we have demonstrated how multi-GPU systems can be efficiently used to attain high performance without major changes from the programmability point of view. However, the scalability of this type of platforms is a problem without an easy solution in the near future. The main bottleneck remains in the PCIExpress bus. Systems with up to four GPUs attached to the same PCIExpress bus are relatively extended nowadays, but including a higher number of GPUs incurs in a serious bottleneck in data transfers with current technology.

To address this problem, clusters with a reduced number of hardware accelerators attached to each node seem an effective solution to the performance demands of large-scale HPC applications. As the performance of the interconnection networks (e.g. Infiniband) improves, the gap between them and the PCIExpress is reduced and they commence to be comparable in bandwidth and latency. Thus, the overhead introduced by the usage of distributed memory can be masked by the second penalty induced by the usage of the PCIExpress bus.

From the software point of view, as of today there are no dense linear algebra libraries adapted for the extension of the nodes with hardware accelerators (e.g., GPUs). In this chapter, we propose an extension of the well-known PLAPACK library to adapt it to clusters of GPUs. The selection of this library is based on its modular and layered design and, following the programmability goals stated during the rest of the dissertation, on its high-level approach from the developer point of view. We propose different techniques to improve performance and reduce data transfers, and show experimental results for some extended dense linear algebra operations from BLAS and LAPACK on a large GPU cluster.

The chapter is structured as follows. Sections 6.1 and 6.2 introduce the basic concepts behind distributed-memory architectures and message-passing programming, respectively, that will be useful in the rest of the chapter. In Section 6.3 we offer an review of the most extended libraries for dense linear algebra computation on distributed-memory architectures. This overview includes present and forthcoming libraries. In Section 6.4 we expose the layered structure of the PLAPACK library. Section 6.5 describes the process and design decisions taken to port PLAPACK to clusters with GPUs. Experimental results on a large GPU cluster are given in Section 6.6. Finally, Section 6.7 summarizes the main contributions of the chapter.



(a) UMA architecture **(b)** NUMA architecture

Figure 6.1: Shared-memory architectures. UMA (a) and NUMA (b) implementations

All experiments presented through the chapter were carried out using up to 32 nodes of a cluster of GPUs (LONGHORN) with a fast Infiniband interconnection. The specific hardware and software details of the experimental setup were presented in Section 1.3.2.

6.1. Parallel computing memory architectures

The development of clusters of GPUs (distributed-memory architectures with one or more GPUs attached to each node in the cluster) is the natural result of the evolution of HPC architectures through the years. The reason underlying this change is the continuous grow in the performance requirements of scientific and engineering applications. In this section we review the basic features of shared-memory, distributed-memory, hybrid shared-distributed memory and accelerated architectures.

6.1.1. Shared memory architectures

In general, *shared-memory architectures* are characterized by the ability of the processors in the system to access to a common global memory space. This fact implies that, although processors can work independently on the same shared resources, when a memory location is changed by a processor this modification is visible to all other processors.

Considering the relation between the specific core that performs a memory transaction and the access time imposed by the architecture to accomplish it, shared-memory architectures can be divided into two main groups. In UMA (*Unified Memory Access*) architectures, the access time to memory is constant independently from the processor that requests the memory access, provided there are no contentions. In these architectures, cache coherency is maintained in hardware. This architecture is widely represented by SMP processors. On the other hand, NUMA (*Non-Unified Memory Access*) architectures are usually built as sets of SMP processors communicated via a fast interconnect. In these systems a processor of an SMP can directly access memory physically located in other SMPs. This implies that not all processors experience the same access time to a given memory address. If cache coherence is maintained, the architecture is usually referred as cc-NUMA (*Cache Coherent NUMA*). Figure 6.1 shows an schematic diagram of the UMA and NUMA implementations of typical shared-memory architectures.

The main advantages of this type of architecture are their programmability and performance. First, the existence of a global address space provides a user-friendly programming view. Second, data sharing and communication between tasks is fast due to the proximity of memory to CPUs.

On the other hand, the disadvantages come from scalability and data coherency. Its primary drawback is the lack of scalability between memory and CPUs: the number of processing units that can be added to a shared-memory architecture without losing performance due to limited memory bandwidth is fixed and small. Moreover, the addition of more processors can rapidly increase traffic on the shared memory-CPU interconnection, and for cache coherent systems, this implies a fast increase in traffic associated with cache/memory management. In addition, it is the programmer's responsibility to build the synchronization constructs that ensure a correct access pattern to global shared memory.

As a final problem of the shared-memory architecture, the cost of designing and building shared-memory machines with increasing number of processors becomes a real problem as the limit of the memory bandwidth is reached by the processing power of the increasing number of computing units. Thus, distributed-memory machines or hybrid distributed-shared machines appear as the natural solution to the problem.

6.1.2. Distributed memory and hybrid architectures

Like shared-memory systems, *distributed-memory architectures* can present a number of different configurations and variations, even though all share some basic features. A distributed-memory architecture presents a set of processors, each one with a local memory, interconnected through a network. Memory addresses in one processor are private and do not map to other processors. There is no common global address space for all processors. In this sense, as all processors operate independently and the modifications in private data by one of them do not affect the rest, there is no cache coherence concept in distributed-memory architectures.

Explicit communications are required if a processor needs data that are private to other processor. The programmer is in charge of defining which data and when that data needs to be transferred. Analogously, process synchronization is responsibility of the programmer. Although fast interconnection networks are commonly used, there is no limitation in the type of network that can be used in these architectures.

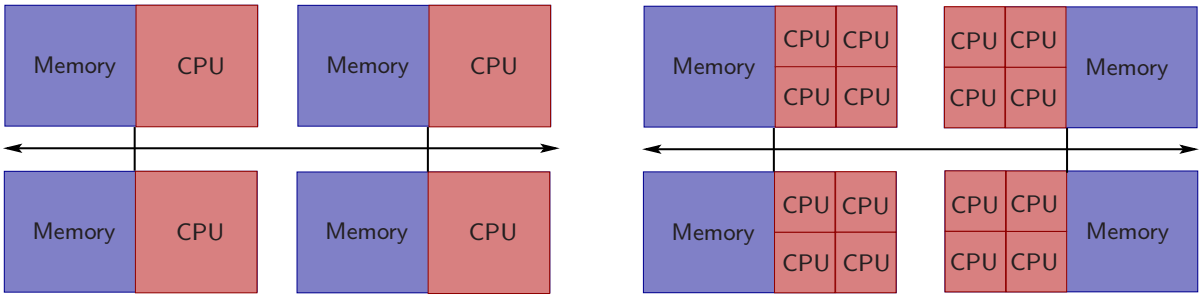
The reasons for the arise and success of distributed-memory architectures are directly related to the main advantages of this type of architectures, namely:

Memory: In shared-memory architectures, memory does not scale with the number of processors, and becomes a dramatic bottleneck as this number reaches a given limit. In distributed-memory architectures, memory is scalable with number of processors.

No cache coherence protocols: In distributed-memory architectures, each processor can efficiently access its own private memory without interferences and without the overhead associated to the use of cache-coherence protocols.

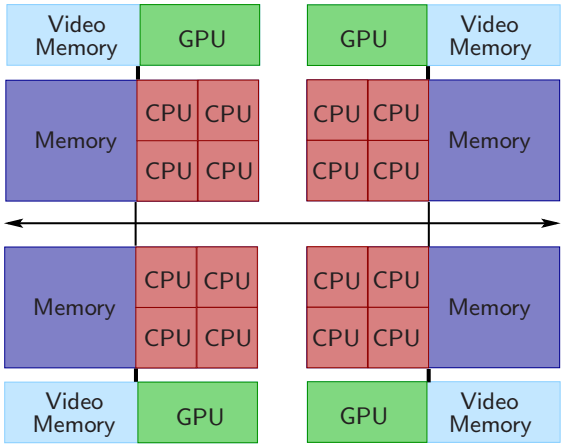
Cost: Distributed-memory architectures can use commodity technology for processors and networking. Shared-memory computers are usually an expensive solution for HPC.

On the other hand, distributed-memory architectures present a number of disadvantages. First, in this model the programmer is responsible for many of the aspects associated with data communication and synchronization between different processes. In addition, the adaptation of existing codes with complex data structures can become a problem from the programmability point of



(a) Distributed-memory architecture

(b) Hybrid architecture



(c) Accelerated hybrid architecture

Figure 6.2: Distributed-memory architectures. Classic (a), hybrid (b) and accelerated hybrid (c) implementations

view. A non-uniform memory access (NUMA) prevails as the cost of the communication between two processors is not uniform and must be taken into account on the software side.

As a variation of purely distributed-memory machines, in practice*hybrid distributed-shared memory architectures* are currently the most common alternative in the HPC arena. Each component in the architecture is typically a cache-coherent SMP architecture. As processors in each node can address own memory as local, other processors can address that machine's memory as global. An interconnection network is still needed for the communications between SMPs. Current trends indicate that this type of architecture is likely to prevail and even increase its numbers in the near future. The advantages and drawbacks of this type of architecture are directly inherited from those of purely shared and distributed-memory architectures. Figure 6.2 shows an schematic diagram of common distributed-memory and hybrid architectures.

6.1.3. Accelerated hybrid architectures

The complexity inherent to hybrid distributed-shared memory architectures has been increased with the introduction of new processing elements in each computing node. These *accelerated hybrid distributed-shared memory architectures* have been recently introduced in response to the higher necessity of computing power from modern HPC applications.

The appearance of accelerated hybrid distributed-memory architectures, with mixed CPU-GPU capabilities per node, is a natural step in response to:

- The poor scalability of multi-GPU systems, mainly due to the bottleneck of the PCIExpress bus as the number of accelerators in the system grows.
- The technological improvement of modern inter-node networks (e.g., 10 Gigabit Ethernet and Infiniband). As of today, inter-node networks performance is in the same order of magnitude as current PCIExpress specification. In this sense, the PCIExpress bottleneck becomes a secondary player in the computations, not the main one.

The introduction of new independent memory spaces per node (one or more, depending on the use of one or more GPUs per node) introduces a new burden for the programmer. Whether the separated memory spaces must be visible for the library developer or not is ultimately decided by the design of the chosen software infrastructure. In this chapter, we demonstrate how a careful design can drive to a transparent port of existing distributed-memory codes to accelerated codes without major performance penalty in the framework of dense-linear algebra implementations.

6.2. Parallel programming models. Message-passing and MPI

Computational models are useful to give a conceptual and high-level view of the type of operations that are available for a program. They are mostly independent from the underlying architecture, and do not expose any specific syntax or implementation detail. Parallel computational models focus on programming parallel architectures. In principle, any of the parallel models can fit on any of the architectures mentioned in Section 6.1; however, the success and effectiveness of the implementation will depend on the gap between the model definition and the target architecture.

A number of different parallel computational models have been proposed, namely:

Shared memory model: Each process accesses to a common shared address space using load and store operations. Synchronization mechanisms such as semaphores or locks must be used to coordinate access to memory locations manipulated concurrently by more than one process.

Threads model: The shared memory model can be applied to multi-threaded systems, in which a single process (or address space) has several program counters and execution stacks associated (*threads*). Threads are commonly associated with shared memory architectures. POSIX threads and OpenMP are the most common implementations of the threads computational model.

Message passing model: In this model, a set of processes use their own local memory during computation. Potentially, multiple processes can coexist in the same physical machine, or execute on different machines. In both cases, processes communicate by sending an receiving messages to transfer data in local memories. Data transfers usually require cooperative operations to be executed by each process involved in the communication. The MPI interface is the most extended implementation of the message passing model.

Hybrid models: In this model, two or more parallel programming models are combined. Currently, the most common example of a hybrid model combines the message passing model (MPI) with either the threads model (POSIX threads or OpenMP). The success of this hybrid model is directly related to the increasingly common hardware environment of clusters of SMP machines.

Distributed-memory architectures are the target of the work developed in this chapter. From the definition of the message passing model, it is the one that best fits to distributed-memory

architectures. The success of the message-passing model is based on a number of advantages compared to other parallel computational models [74]:

Universality: The message passing model is valid for any system with separate processors that can communicate through an interconnection network. The performance or nature of the processors or the network is not taken into account by the model. Thus, virtually any modern supercomputer or cluster of workstations can fit in this characterization.

Performance: The adaptation of message passing to distributed-memory architectures makes it the perfect model for high performance computing. In these architectures, memory-bound applications can attain even super-linear speedups, and message passing is the common used programming paradigm. In addition, the control that the message passing model offers to the programmers makes it suitable for high performance demanding implementations. On shared memory architectures, the use of message passing provides more control over data locality in the memory hierarchy than that offered by other models such as the shared memory one.

Expressivity: Message-passing model is a complete model useful to fully express the essence of parallel algorithms.

Ease of debugging: With only one process accessing data for reading/writing, the debugging process in message-passing model is usually easier than in other models like, e.g., shared-memory.

Historically, a number of message passing implementations have been available since the 1980s. In response to this heterogeneity in the message passing implementations, the MPI standard was introduced in 1994 [74] in its first version and in 1996 [125] in its second version. MPI has become the de facto standard for programming distributed-memory architectures. In addition, it is perfectly suitable for shared memory architectures where message passing is performed through memory copies.

The introduction of hybrid distributed-memory architectures with hardware accelerators (GPUs) per node, represents a new challenge from the software point of view. As an example, current dense linear algebra libraries for distributed-memory machines (ScaLAPACK, PLAPACK or Elemental, to name a few) are based on message-passing libraries like MPI or PVM. The introduction of new memory spaces bound to the new accelerators introduces the challenge of porting these libraries to novel architectures. If the goal is to abstract the programmer from the existence of hardware accelerators in the system, the problem is even harder to solve.

6.3. Dense linear algebra libraries for message-passing programming

In this section, we review three different alternatives for dense linear algebra computations on distributed-memory architectures. These libraries are illustrative of the past (and present) of the field (ScaLAPACK and PLAPACK), and the state-of-the-art in distributed-memory libraries (in the case of the Elemental framework).

6.3.1. ScaLAPACK

With the advent and popularization of distributed-memory machines, it seemed clear that LAPACK needed to be redesigned to adapt its contents. ScaLAPACK [49] was the response of the creators of LAPACK to port its functionality to distributed-memory machines. ScaLAPACK

is defined as a message-passing library for distributed-memory machines, offering in-core and out-of-core computations for dense linear algebra implementations. It implements a subset of the functionality of the LAPACK library.

Motivation, goals and design decisions

During the early stages of LAPACK, target architectures for this project were vector and shared-memory parallel computers. The ScaLAPACK library was initially thought as a continuation of LAPACK, focused on distributed-memory machines. To accomplish this target, ScaLAPACK delegates the communication layer to well-known message-passing communication libraries such as PVM [128] and MPI [125].

According to the ScaLAPACK User's Guide [33], six main goals underlied the initial idea of ScaLAPACK:

- Efficiency, to run message-passing codes as fast as possible.
- Scalability, to keep efficiency as the problem size as the number of processors grows.
- Reliability, including error bounds.
- Portability, across a wide variety of parallel machines.
- Flexibility, so users can construct new routines from well-designed parts.
- Ease of use, making the interfaces of LAPACK and ScaLAPACK as similar as possible.

Historically ScaLAPACK has been a library of success in part due to the accomplishment of many of these initial goals. For example, portability is guaranteed by the development and promotion of standards, especially for low-level computation and communication routines. The usage of BLAS and BLACS (Basic Linear Algebra Communication Subprograms) warrants the correct migration of the implementations through different architectures, provided there exist implementations of BLAS and BLACS for them. To ensure efficiency, the library relies on the performance of the underlying BLAS libraries, and scalability is accomplished with a optimal implementation of BLACS.

However, two of the initial goals of ScaLAPACK, *flexibility* and *ease of use* are arguable. The aim of the authors of ScaLAPACK was to create a library flexible enough to allow the scientific community to develop new routines or adapt existing ones to their necessities. As of today, 15 years after its introduction, and after becoming the most widely extended library for dense and banded linear algebra computations on distributed-memory machines, ScaLAPACK still lacks of many of the functionality present in LAPACK, mainly due to its difficult programming.

Software components and structure

Figure 6.3 describes the software hierarchy adopted in the design and implementation of ScaLAPACK. Elements labeled as *local* are invoked from a single process, and their arguments are stored only in the corresponding processor. Elements labeled as *global* are parallel and synchronous routines; their arguments (matrices and vectors) are distributed across more than one processor.

The BLAS and LAPACK components have already been described in Chapters 3 and 4, respectively. After observing the great benefits of BLAS on the performance, functionality and modularity of LAPACK, the ScaLAPACK designers aimed at building a parallel set of BLAS, called PBLAS

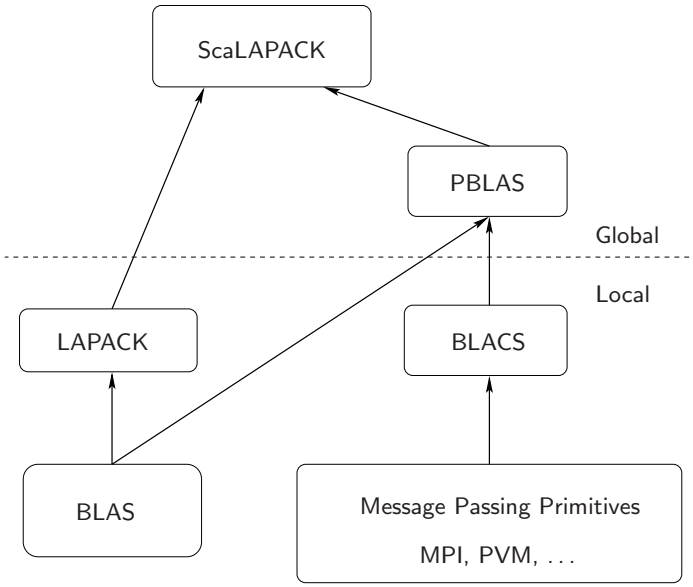


Figure 6.3: ScaLAPACK software hierarchy.

(Parallel BLAS) [48, 112]. PBLAS implements some of the functionality of BLAS and keeps an analogous interface (as much as possible), using a message passing paradigm.

The BLACS (Basic Linear Algebra Communication Subprograms) [10, 58] are a set of message-passing routines designed exclusively for linear algebra. Schematically, they implement a computational model in which processes, organized into a one- or two-dimensional grid, store pieces (blocks) of matrices and vectors. The main goal of BLACS was portability, and a specific design of their functions to address common linear algebra communication patterns. It implements synchronous send/receive routines to transfer matrices or sub-matrices between two processes, broadcast sub-matrices and perform reductions. It also implements the concept of *context*, that allows a given process to be a member of several disjoint process grids (alike MPI *communicators*).

Data distribution in ScaLAPACK

One design decision of ScaLAPACK is that all global data involved in the computations has to be conveniently distributed among processors before the invocation to the corresponding routine is performed. LAPACK uses block-oriented algorithms to maximize the ratio of floating-point operations per memory reference and increase data reuse. ScaLAPACK uses block-partitioned algorithms to reduce the frequency of data transfers between different processes, thus reducing the impact of network latency bound to each network transmission.

For in-core dense matrix computations, ScaLAPACK assumes that data is distributed following a *two-dimensional block-cyclic* [73] layout scheme. Although there exist a wide variety of data distribution schemes, ScaLAPACK chose the block-cyclic approach mainly because of its scalability [54], load balance and communication [78] properties.

Figure 6.4 illustrates a two-dimensional block-cyclic distribution layout. Here, we consider a parallel computer with P processes, arranged in a $P_r \times P_c = P$ rectangular array. Processes are indexed as (p_r, p_c), with $0 \leq p_r < P_r$, and $0 \leq p_c < P_c$. The distribution pattern of a matrix is defined by four parameters: the dimensions of the block (r, s), the number of processors in a column, P_c and the number of processors in a row, P_r. Note how, in the block-cyclic data distribution, blocks of consecutive data are distributed cyclically over the processes in the grid.

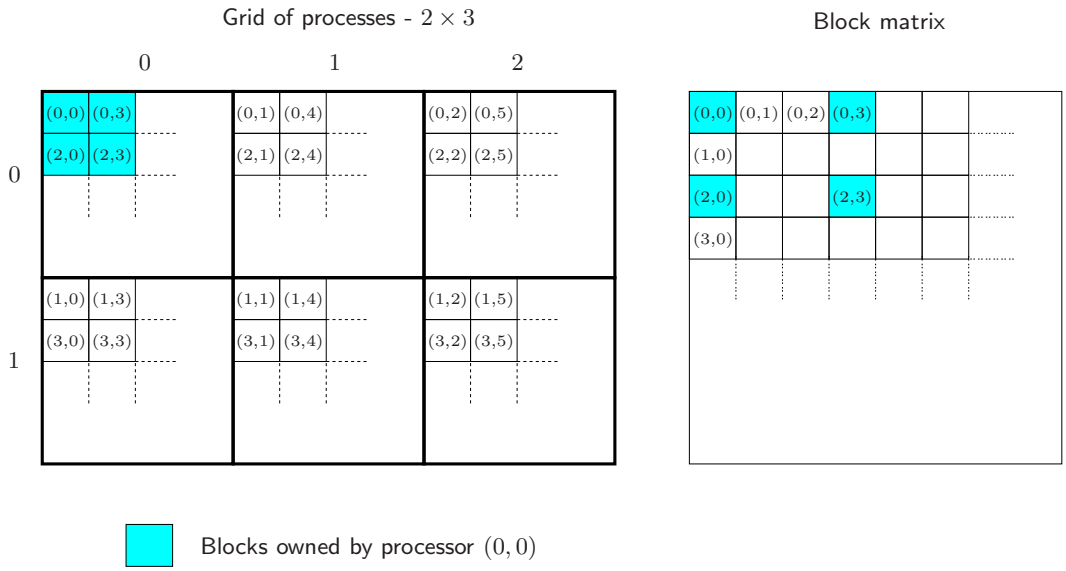


Figure 6.4: Block-cyclic data distribution of a matrix on a 2×3 grid of processes

Programmability

This is one of the weakest points of ScaLAPACK. To illustrate a practical example of the usage of ScaLAPACK and establish a base for a comparison with PLAPACK, we show the prototype of the `P_GEMM` function. `P_GEMM` is the PBLAS routine in charge of performing a distributed matrix-matrix multiplication. The interface of this routine is specified as:

```
1 P_GEMM( TRANSA, TRANSB, M, N, K, ALPHA, A, IA, JA, DESCA,
                                          B, IB, JB, DESCB,
3                                  BETA, C, IC, JC, DESCC );
```

The nomenclature and options of the invocations are directly inherited from BLAS and LAPACK. Besides the intricate indexing and dimension handling, parameters `DESCx`, with `x` being `A`, `B`, or `C` describe the two-dimensional block-cycling mapping of the corresponding matrix. This array descriptor is an integer array with 9 elements.

The first two entries of `DESCx` are the descriptor type and the BLACS context. The third and fourth ones denote the dimensions of the matrix (number of rows and columns, respectively). The fifth and sixth entries specify the row and column block sizes used to distribute the matrix. The seventh and eighth entries identify the coordinates of the process containing the first entry of the matrix. The last one is the leading dimension of the local array containing the matrix elements.

This syntax gives an idea of the complex usage of ScaLAPACK as a library. The programmer is not abstracted from data distributions. Many of the parameters managed internally by ScaLAPACK have to be explicitly handled by the programmer.

The usage of ScaLAPACK as an infrastructure to build new routines is also possible. In that case, complexity appears in the usage of BLACS and PBLAS as the underlying libraries to support communication and parallel execution. The description of those software elements is out of the scope of this dissertation. Further information can be found in the ScaLAPACK documentation.

6.3.2. PLAPACK

PLAPACK [137, 18, 7] is a library and an infrastructure for the development of dense and banded linear algebra codes. As a library, it provides the whole functionality of the three BLAS

levels, plus some of the functionality at the LAPACK level. As an infrastructure, it provides the necessary tools to allow the library developer to easily implement new distributed-memory routines with relatively small effort and high reliability. It has been widely used [150, 17, 146, 151] through the years, mainly due to its reliability and ease of programming.

Natural algorithmic description

PLAPACK introduced a methodology for algorithm description, development and coding that was the kernel of the FLAME methodology. Thus, programmability and natural port of algorithm descriptions directly to code lie the foundations of the infrastructure.

The main advantages of the methodology underlying PLAPACK can be easily illustrated by reviewing its implementation of the Cholesky factorization, an operation already introduced in previous chapters (a more detailed PLAPACK implementation of the Cholesky factorization will be given in Section 6.4.2). As in previous chapters, the goal is to illustrate how the algorithmic description of the operation can be translated directly into a code ready to execute on distributed-memory architectures.

The mathematical formulation of the Cholesky factorization of a, in this case, symmetric positive definite matrix A, given by

$$A = LL^T,$$

yields the following steps to obtain the Cholesky factorization, as detailed in previous chapters (assuming that the lower triangular part of A is overwritten by its Cholesky factor L):

1. $\alpha_{11} := \lambda_{11} = \sqrt{\alpha_{11}}$.
2. $a_{21} := l_{21} = a_{21}/\lambda_{11}$.
3. $A_{22} := A_{22} - l_{21}l_{21}^T$.
4. Compute the Cholesky factor of A_{22} recursively, to obtain L_{22}.

Three main observations naturally link to the programming style proposed by PLAPACK:

1. The blocks in the algorithms are just references to parts of the original matrices.
2. There is not an explicit indexing of the elements in the original matrix. Instead there is an indexing of parts of the *current* matrix, not the original one.
3. Recursion is inherently used in the algorithm.

Consider the following fragment of PLAPACK code which can be viewed as a direct translation of the algorithm for the Cholesky factorization:

```
PLA_Obj_view_all( a, &acur );
while( TRUE ){
  PLA_Obj_global_length( acur, &size );
  if( size == 0 ) break;
  PLA_Obj_split_4( acur, 1, 1, &a11, &a12,
                               &a21, &acur );
  Take_sqrt( a11 );
  PLA_Inv_scal( a11, a21 );
  PLA_Syr( PLA_LOW_TRIAN, min_one, a21, acur );
}
```

First, all the information that describes a distributed matrix (in this case, matrix A), including the information about the layout of its elements among processes is encapsulated in a data structure (or *object*) referenced in the code by `a`. From this data structure, multiple references, or *views* to the whole object (or parts of it) can be created. For example, function `PLA_Obj_view_all` creates a reference `acur` to the same data as `a`. Object properties are extracted to guide the execution of the algorithm. In the example, routine `PLA_Obj_global_length` extracts the current size of the matrix referenced by `acur`. In this case, when this value is zero, the recursion ends.

Matrix partitioning is performed in the code with the routine `PLA_Obj_split_4`, creating references to each one of the new four quadrants of the original matrix. From these sub-matrices, each one of the operations needed by the algorithm are performed on the corresponding data: `Take_sqrt` obtains the square root of α_{11}, a_{21} is scaled by $1/\alpha_{11}$ by invoking the routine `PLA_Inv_scal` and the symmetric rank-1 update of A_{22} is computed in `PLA_Syr`.

How parallelism is extracted from these operations will be illustrated by means of particular examples in the next sections. Both the internals of those operations and the higher-level implementation shown for the Cholesky factorization deliver a natural transition between the algorithmic essence of the operation and its implementation on distributed-memory architectures.

Data distribution in PLAPACK

PLAPACK was a revolution not only because of the novel programming paradigm that it introduced (that was, in fact, the kernel of the FLAME project), but also because of the approach taken to distribute data among processes. Typically, message-passing libraries devoted to matrix computations distribute data to be processed by initially decomposing the matrix. PLAPACK advocates for a different approach. Instead of that, PLAPACK considers how to decompose *the physical problem* to be solved.

In this sense, the key observation in PLAPACK is how the elements of the vectors involved in the computation are usually related with the significant data from the physical point of view. Thus, distribution of vectors instead of matrices naturally determines the layout of the problem among the processes. For example, consider the linear transform:

$$y = Ax$$

Here, matrix A is merely a relation between two vectors. Data in the matrix do not have a relevant significance themselves. The nature of specified by the vector contents.

This insight drives to the data distribution strategy followed by PLAPACK, usually referred to as PMBD or *Physically Based Matrix Distribution*. The aim is to distribute *the problem* among the processes, and thus, in the former expression, one starts by partitioning x and y and assigning these partitions to different processes. The matrix A can be then distributed consistently with the distribution chosen for the vectors.

We show here the process to distribute vectors and, therefore, derive matrix distributions alike PLAPACK. We start by defining the partitioning of vectors x and y:

$$P_x x = \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline \cdots \\ \hline x_{p-1} \end{array} \right)$$

and

$$P_y y = \left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline \dots \\ \hline y_{p-1} \end{array} \right),$$

where P_x and P_y are permutations that order the elements of vectors x and y, respectively, in some given order. In this case, if processes are labeled as $P_0, \dots, P_{p-1}$, then sub-vectors x_i and y_i are assigned to process P_i.

Then, it is possible to conformally partition matrix A:

$$P_y A P_x^T = \left(\begin{array}{c|c|c|c} A_{0,0} & A_{0,1} & \dots & A_{0,p-1} \\ \hline A_{1,0} & A_{1,1} & \dots & A_{0,p-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{p-1,0} & A_{p-1,1} & \dots & A_{p-1,p-1} \end{array} \right).$$

From there, we have that:

$$\left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline \dots \\ \hline y_{p-1} \end{array} \right) = P_y y = P_y A x = P_y A P_x^T P_x x = \left(\begin{array}{c|c|c|c} A_{0,0} & A_{0,1} & \dots & A_{0,p-1} \\ \hline A_{1,0} & A_{1,1} & \dots & A_{0,p-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{p-1,0} & A_{p-1,1} & \dots & A_{p-1,p-1} \end{array} \right) \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline \dots \\ \hline x_{p-1} \end{array} \right),$$

and thus:

$$\left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline \dots \\ \hline y_{p-1} \end{array} \right) = \left(\begin{array}{c} A_{0,0} \\ \hline A_{1,0} \\ \hline \vdots \\ \hline A_{p-1,0} \end{array} \right) x_0 + \left(\begin{array}{c} A_{0,1} \\ \hline A_{1,1} \\ \hline \vdots \\ \hline A_{p-1,1} \end{array} \right) x_1 + \ldots + \left(\begin{array}{c} A_{0,p-1} \\ \hline A_{1,p-1} \\ \hline \vdots \\ \hline A_{p-1,p-1} \end{array} \right) x_{p-1}.$$

Note how this expression shows the natural association between the sub-vectors of $P_x x$ and the corresponding blocks of columns of $P_y A P_x^T$.

Analogously,

$$y_i = \sum_{j=0}^{p-1} A_{i,j} x_j$$

means that there is also a natural association between sub-vectors of $P_y y$ and corresponding blocks of rows of $P_y A P_x^T$.

Previous studies [54, 78] concluded that scalability of the solutions is improved if matrices are assigned to processes following a two-dimensional matrix distribution. In this case, the p nodes are logically viewed as a $r \times c = p$ mesh. This forces a distribution of sub-vectors to the two-dimensional mesh. PLAPACK uses a column-major order for this distribution. Now, the distribution of matrix A requires that blocks of columns $A_{i,*}$ are assigned to the same column of nodes as sub-vector x_j, and blocks of rows $A_{i,*}$ to the same rows of nodes as sub-vector y_i, as seen in Figure 6.5.

The figure shows the mechanism used by PLAPACK to induce a matrix distribution from vector distributions. Consider a 3×4 mesh of processes, each one represented by a box in the diagram in the top of the figure. Each sub-vector of x and y is assigned to the corresponding node following a column-major ordering. Projecting the indices of y to the left, the distribution of the matrix row-blocks of A is determined. Projecting the indices of x to the top, the distribution of column-blocks

of A is determined. The diagrams in the bottom show the same information from different points of view. In the left-hand side, the resulting distribution of the sub-blocks of A is shown. In the right-hand diagram, the matrix is partitioned into sub-blocks, with the indices in the sub-blocks indicating the node to which the sub-block is mapped.

Data redistribution, duplication and reduction in PLAPACK

Besides computation and data distribution, communication is the third main building block in PLAPACK. The fundamentals of data communication in PLAPACK are based on the mechanisms used to redistribute, duplicate and reduce data. These are the basic operations required to support the complete functionality of the PLAPACK infrastructure.

We illustrate here one example of redistribution, duplication and reduction. Further information and examples can be found in the literature.

Redistribution of a vector to a matrix row. Consider a vector x distributed to the mesh of processes according to the inducing vector for a given matrix A; see Figure 6.6. Observe how blocks of columns of A are assigned to nodes by *projecting* the indices of the corresponding sub-vectors of the inducing vector distribution.

The transformation of x into a row of matrix A requires a *projection* onto that matrix row. The associated communication pattern is a *gather* of the sub-vectors of x within columns of nodes to the row that holds the desired row of A.

The inverse process requires a *scatter* within columns of A, as shown in Figure 6.6.

For the necessary communication patterns for the redistribution of a vector to a matrix column (and inverse) or a matrix row to a matrix column (and inverse), see [137].

Spread (duplicate) of a vector within rows (or columns) of nodes. Duplication (spread) of vectors, matrix rows or matrix columns within rows or columns of nodes is also a basic communication scheme used in PLAPACK. Upon completion, all rows of nodes or columns of nodes own a copy of the distributed row or column vector.

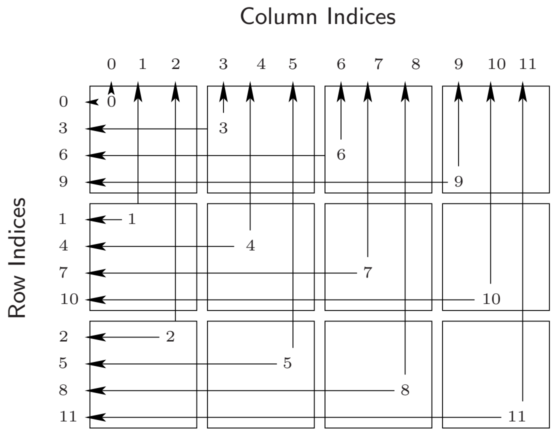
Consider the result of gathering a vector as illustrated in Figure 6.7. If the goal is to have a copy of this column (row) vector within each column (row) of nodes, it is necessary to *collect* the vector within rows (columns) of nodes, as shown in Figure 6.7.

For the necessary communication patterns for the duplication of a matrix row (or column) within rows (or columns) of nodes, see [137].

Reduction of a spread vector. Given a previously spread vector, matrix row or matrix column, multiple copies of the data coexist in different processors. Usually, as part of a distributed operation, multiple instances exist, each one holding a partial contribution to the global result. In that situation, local contributions must be *reduced* into the global result.

Often, the reduction is performed within one or all rows or columns of the process mesh. Figure 6.8 shows this situation. Consider that each column of processes holds a column vector that has to be summed element-wise into a single vector. In this situation, a *distributed reduction* of data that leaves a sub-vector of the result on each node has to be performed.

Note how the observations for data consolidation, duplication and reduction yield a systematic approach to all possible data movement situations in PLAPACK.



(a)

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9	0,10	0,11
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9	3,10	3,11
6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7	6,8	6,9	6,10	6,11
9,0	9,1	9,2	9,3	9,4	9,5	9,6	9,7	9,8	9,9	9,10	9,11
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	9,10	1,11
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8	4,9	4,10	4,11
7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7	7,8	7,9	7,10	7,11
10,0	10,1	10,2	10,3	10,4	10,5	10,6	10,7	10,8	10,9	10,10	10,11
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8	2,9	2,10	2,11
5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8	5,9	5,10	5,11
8,0	8,1	8,2	8,3	8,4	8,5	8,6	8,7	8,8	8,9	8,10	8,11
1,10	1,11	11,2	11,3	11,4	11,5	11,6	11,7	11,8	11,9	11,10	11,11

(b)

	0			1			2			3		
0	0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3	0,3	0,3
1	1,0	1,0	1,0	1,1	1,1	1,1	1,2	1,2	1,2	1,3	1,3	1,3
2	2,0	2,0	2,0	2,1	2,1	2,1	2,2	2,2	2,2	2,3	2,3	2,3
0	0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3	0,3	0,3
1	1,0	1,0	1,0	1,1	1,1	1,1	1,2	1,2	1,2	1,3	1,3	1,3
2	2,0	2,0	2,0	2,1	2,1	2,1	2,2	2,2	2,2	2,3	2,3	2,3
0	0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3	0,3	0,3
1	1,0	1,0	1,0	1,1	1,1	1,1	1,2	1,2	1,2	1,3	1,3	1,3
2	2,0	2,0	2,0	2,1	2,1	2,1	2,2	2,2	2,2	2,3	2,3	2,3
0	0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3	0,3	0,3
1	1,0	1,0	1,0	1,1	1,1	1,1	1,2	1,2	1,2	1,3	1,3	1,3
2	2,0	2,0	2,0	2,1	2,1	2,1	2,2	2,2	2,2	2,3	2,3	2,3

(c)

Figure 6.5: Induction of a matrix distribution from vector distributions. (a) Sub-vectors of x and y distributed to a 3×4 mesh of processes in column-major order. The numbers represent the index of the sub-vector assigned to the corresponding node. (b) Resulting distribution of the sub-blocks of A. Indices in the figure represent the indices of the sub-blocks of A. (c) The same information, from the matrix point of view. Indices represent the node to which the sub-blocks are mapped.

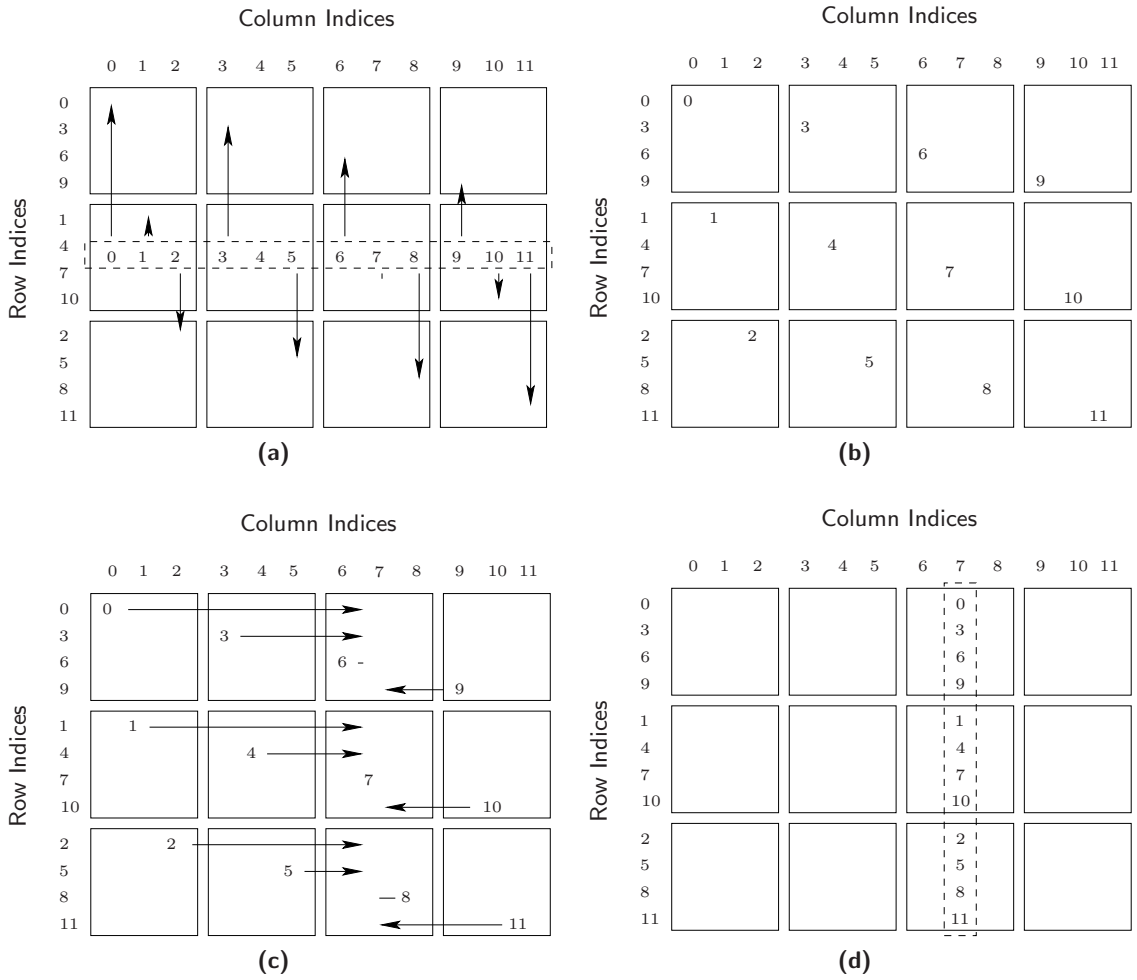


Figure 6.6: (a) and (b): Redistribution of a matrix row (or, equivalently, a vector projected onto a row of processes in the mesh) to a vector distributed like the inducing vector. (c) and (d): Redistribution of a vector distributed like the inducing vector to a matrix column (or, equivalently, a vector projected onto a column of processes in the mesh). Observe how the whole process (a)-(d) performs the transpose of a matrix row to a matrix column.

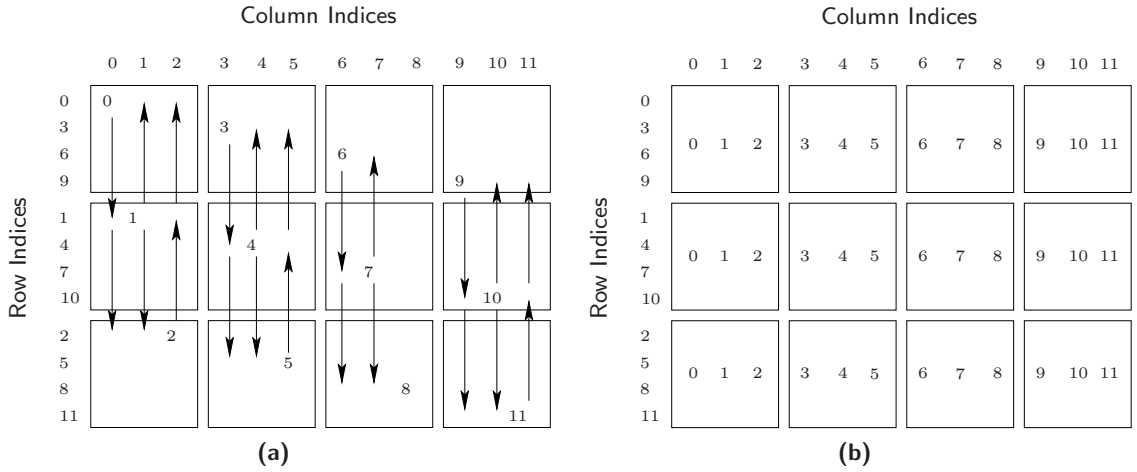


Figure 6.7: Spreading a vector within columns of processes.

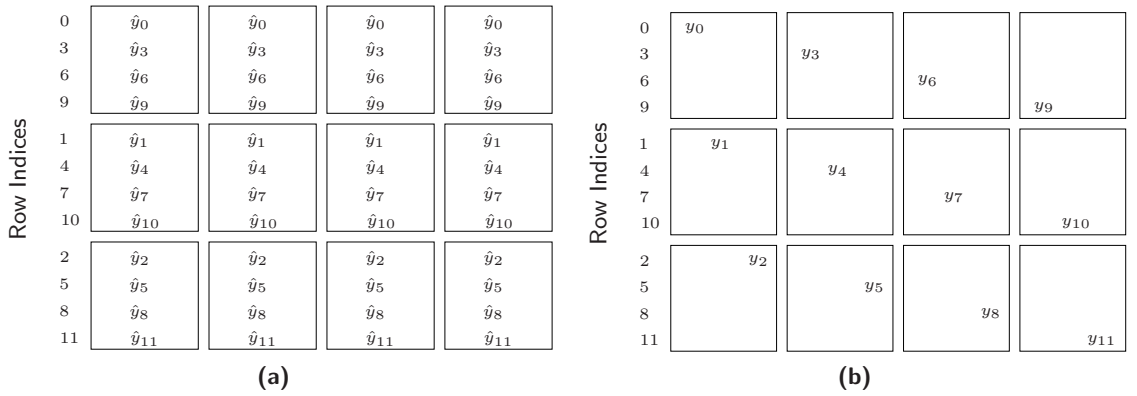


Figure 6.8: Reduction of a distributed vector among processes.

6.3.3. Elemental

Both ScaLAPACK and PLAPACK are relatively old software packages. With the advent of new many-core architectures, close to distributed-memory clusters inside a chip [81, 103], the interest in investigating and developing novel message-passing libraries is renewed, as they will have to be mapped to single-chip environments in a near future.

Elemental [113] is a new framework for the development of message-passing dense matrix computations that incorporates many of the insights introduced by the scientific community since the appearance of ScaLAPACK and PLAPACK one decade ago. Besides being a complete rewrite, with many of the functionality offered by other dense linear algebra libraries, for distributed-memory architectures, Elemental proposes many novelties from the lowest level of its development and philosophy. Many of these new contributions are beyond the scope of this thesis, but the layered approach of the framework makes it feasible to port it to accelerated hybrid architectures in a similar way as the port of the PLAPACK infrastructure has been carried out in the framework of this dissertation.

6.4. Description of the PLAPACK infrastructure

6.4.1. Layered approach of PLAPACK

Figure 6.9 offers an overview of the layered structure of the PLAPACK infrastructure. The components in the bottom line of the table are machine-dependent. The rest of the infrastructure was designed to be machine-independent. In fact, this is the key decision that makes it possible to perform a clean port to accelerated clusters without affecting independent modules of the library.

Machine-dependent layer: PLAPACK aims at using standardized components, namely MPI for communication, BLAS for local kernel execution, standard memory management systems (*malloc/calloc*), etc.

Machine-independent layer: PLAPACK offers a number of higher-level interfaces to the machine-dependent layer. These interfaces include wrappers for MPI communications, memory management routines, distribution primitives, and BLAS interfaces.

PLAPACK abstraction layer: It provides an abstraction that isolates users from details such as indexing, communication, and local computation.

- Linear algebra objects manipulation. All information related to the description of a linear algebra object (vectors or matrices) is stored in an opaque object; this is the same approach adopted years later by the FLAME methodology. This component of the abstraction layer allows the programmer to create, initialize, manipulate and destroy objects. It also provides mechanisms to perform indexing within matrices, sub-matrices and vectors.
- Copy/reduce: duplication and consolidation of data. PLAPACK does not provide explicit routines for the communication of data. Instead, the objects themselves describe how data has to be distributed or duplicated. Communication itself is reduced to copy or reduction from one object to another.
- PLAPACK local BLAS. Rather than explicitly requiring that the user extracts object data and invokes BLAS calls, PLAPACK provides a full set of routines to correctly

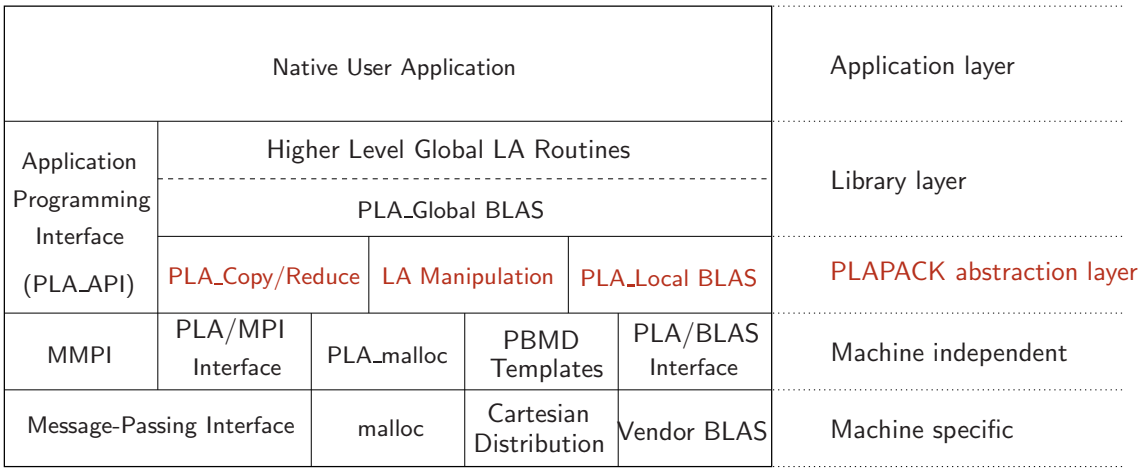


Figure 6.9: PLAPACK software hierarchy. The PLAPACK abstraction layer (in red) is the only part of the infrastructure that needs to be modified in order to adapt it to an accelerated cluster.

extract data and call the appropriate local BLAS routine to perform per-process local BLAS invocations.

Library layer: PLAPACK was created not only as a library but also as an infrastructure to create new application-level libraries on top of it.

- PLAPACK global BLAS. PLAPACK provides a full global BLAS implementation that allows the library developer to build higher-level libraries based on it without exposing communication routines.
- PLAPACK higher-level linear algebra routines. Although these routines can be built directly from the global BLAS provided by the infrastructure, higher performance is expected if the abstraction layer functionalities are directly used. PLAPACK offers implementation of a wide variety of LAPACK-level routines for message-passing architectures.

PLAPACK application interface: Even though the abstraction layer offers routines to manage linear algebra objects, the PLAPACK API allows the programmer to directly manipulate the internals of those objects directly from the application.

6.4.2. Usage of the PLAPACK infrastructure. Practical cases

We next propose three different implementations of dense linear algebra operations to illustrate the main functionality of the PLAPACK infrastructure. We show the necessary steps to implement the matrix-vector multiplication, matrix-matrix multiplication and the Cholesky factorization using PLAPACK. Each example aims at showing a given functionality of the infrastructure. In the matrix-vector multiplication, we focus on how the communication is performed in PLAPACK. In the matrix-matrix multiplication, we illustrate how PLAPACK deals with partitioning and repartitioning objects. In the Cholesky factorization, we show how LAPACK-level routines can be implemented using the distributed BLAS routines provided by PLAPACK, extracting parallelism inside those BLAS calls transparently for the programmer.

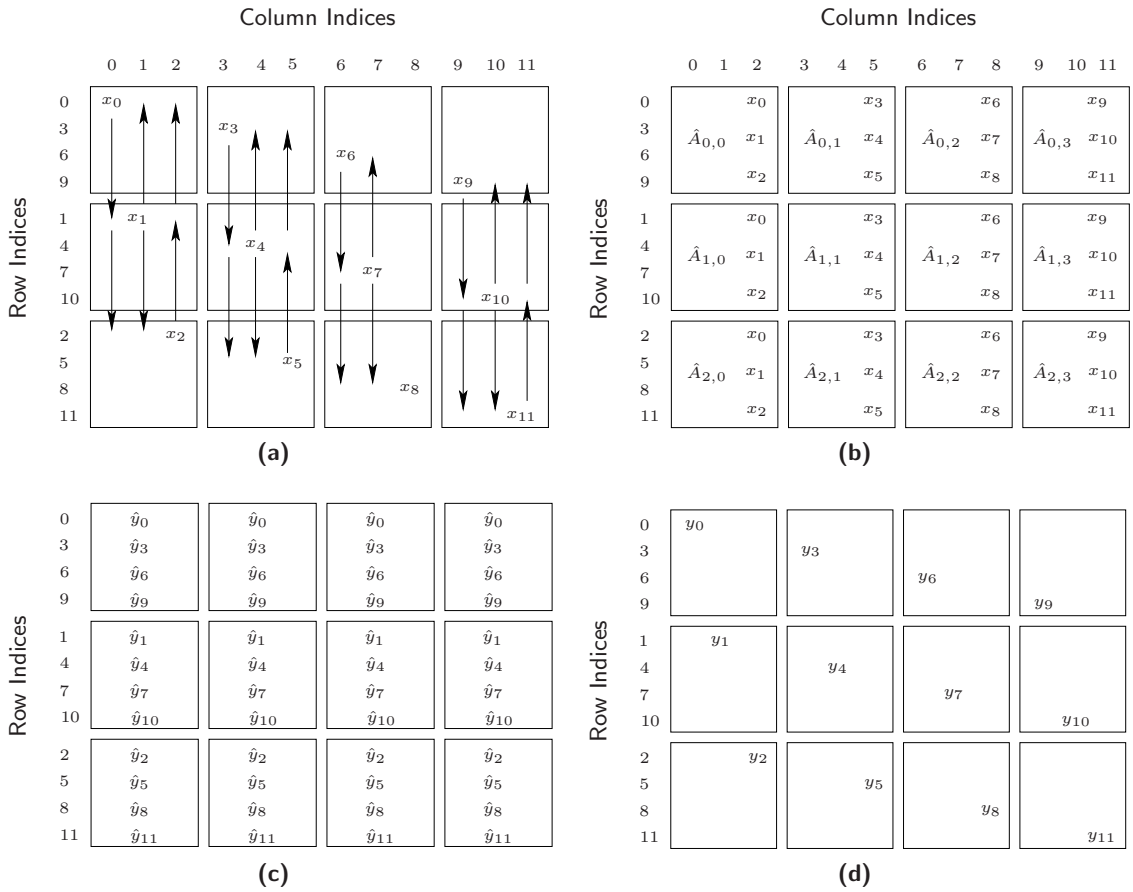


Figure 6.10: Parallel matrix-vector multiplication. (a) Sub-vectors of x are distributed among processes in column-major order. (b) x is spread within columns. After that, local matrix-vector products can commence. (c) Each process has a partial contribution to the result, which needs to be summed within rows of processes. (d) Sub-vectors of y end up being distributed between processes in column-major order.

Parallelizing matrix-vector multiplication

To illustrate the use of PLAPACK as an infrastructure for developing new dense linear algebra operations, we show the parallel implementation of the dense matrix-vector multiplication using PLAPACK. We consider the basic case

$$y = Ax$$

where A is an $m \times n$ matrix. Three different objects must be created in the driver program to handle vectors x and y and matrix A. We will refer to these objects as `x`, `y`, and `a`. In addition, a template is created with a $r \times c$ communicator and block size n_b (templates are the abstraction introduced by PLAPACK to describe internally the distribution of objects within processes). With this template, vectors x and y are aligned with the first element of the template vector, and A with the upper-left element of the template matrix.

Consider that x and y are distributed as vectors. In this case, we assume that x and y are identically distributed across the processes. Notice how, by *spreading* (collecting) vector x within columns, all the necessary elements of x are duplicated, so that local matrix vector multiplication

can start on each process. After this local computation is done, a *reduction* (a summation in this case) within rows of processes of the local partial results yields vector y.

Therefore, three different steps are necessary to perform the operation:

1. Spread elements of x within columns of processes.
2. Perform the local matrix-vector multiply.
3. Perform a *distributed reduce* of the local results within rows. The global result is left in vector y.

We next illustrate how PLAPACK perform these steps. PLAPACK uses a mechanism to communicate based on the description of the initial and final distribution as objects, performing then a *copy* or *reduction* between them. All communication patterns in PLAPACK follow this approach. Thus, the following code produces the necessary spread of the elements of x within columns of processes:

```
PLA_Obj_datatype( a, &datatype );
PLA_Pvector_create( datatype, PLA_PROJ_ONTO_ROW, PLA_ALL_ROWS,
                    n, template, PLA_ALIGN_FIRST, &xdup );
PLA_Copy( x, xdup );
```

The requirement of a row vector or column vector to exist within one or all row(s) or column(s) of processes is present in many parallel implementations. To create such an object, it is necessary to create a *projected vector* and perform the actual copy between the original and the projected vectors. Routine `PLA_Pvector_create` in the code above yields this object creation. All the communication details, including misalignment issues and the necessary MPI calls are encapsulated inside routine `PLA_Copy`.

After all processes have executed the previous code, all necessary information is locally available to perform the local matrix-vector multiplies. PLAPACK needs to create duplicated multi-scalars to hold constants 0 and 1 in all processes. In addition, a duplicated projected column vector has to be created to hold the result. PLAPACK provides routines to create those objects, and to perform the local matrix-vector multiply, as follows:

```
PLA_Mscalar_create( datatype, PLA_ALL_ROWS, PLA_ALL_COLS, 1, 1, template, &one );
PLA_Obj_set_to_one( one );

PLA_Mscalar_create( datatype, PLA_ALL_ROWS, PLA_ALL_COLS, 1, 1, template, &zero );
PLA_Obj_set_to_one( zero );

PLA_Pvector_create( datatype, PLA_PROJ_ONTO_COL, PLA_ALL_COLS,
                    m, template, PLA_ALIGN_FIRST, &ydup );

PLA_Local_gemv( PLA_NO_TRANSPOSE, one, a, xdup, zero, ydup );
```

The last step reduces the local results (in the different instances of the duplicated projected vector `ydup`) into a single vector y:

```
PLA_Obj_set_to_zero( y );
PLA_Reduce( ydup, MPI_SUM, y );
```

As in the `PLA_Copy` routine, all lower-level details underlying the reduction are encapsulated inside the routine `PLA_Reduce`.

Note how, beyond the implementation details explained and those that can be obtained in the literature, there is an important factor that will have capital importance for the GPU implementations. All communications are encapsulated in only two routines: `PLA_Copy` and `PLA_Reduce`. The

developer is not exposed to any other communication primitive or detail, neither at a lower nor at a higher level. The code pattern is common for all communications: objects are created according to the way data has to be duplicated or reduced, and the proper PLAPACK communication routine is invoked. The infrastructure is designed to deal with lower level details without intervention of the programmer.

Parallelizing matrix-matrix multiplication

We next describe a blocked algorithm to illustrate how to perform a scalable matrix-matrix multiplication on message-passing architectures using PLAPACK. This approach was introduced in 1995 by van de Geijn and Watts [135], and is known as SUMMA (Scalable Universal Matrix Multiplication Algorithm). We only illustrate here the formation of the matrix product $C := \alpha AB + \beta C$, where A, B, and C are matrices, and α, β are scalars.

For simplicity, consider that all three matrices A, B, and C are $n \times n$. We use the following partitionings:

$$X = \left(\begin{array}{c|c|c|c} x_0 & x_1 & \ldots & x_{n-1} \end{array} \right),$$

with $X \in A, B, C$ and $x \in a, b, c$, where x_j is the j-th column of matrix X. Similarly

$$X = \left(\begin{array}{c} \hat{x}_0^T \\ \hline \hat{x}_1^T \\ \hline \ldots \\ \hline \hat{x}_{n-1}^T \end{array} \right),$$

where $\hat{x}_i^T$ represents the i-th row of matrix X.

Observe that

$$C = AB = a_0 \hat{b}_0^T + a_1 \hat{b}_1^T + \ldots + a_{n-1} \hat{b}_{n-1}^T$$

and thus the concurrent execution of the matrix-matrix multiplication can be performed as a set of rank-1 updates, with vectors y and x being the appropriate column and row of A and B, respectively.

Higher performance can be obtained if one replaces the matrix-vector multiplication with multiplication of a matrix by a panel of vectors. In this case, rank-1 updates are substituted by rank-k updates. Let A and B be partitioned as:

$$A_{cur} = \left(\begin{array}{c|c} A_1 & A_2 \end{array} \right)$$

and

$$B_{cur} = \left(\begin{array}{c} B_1 \\ \hline B_2 \end{array} \right),$$

where A_1 and B_1 are $m \times s$ and $s \times n$ matrices, respectively. In that case

$$C \leftarrow \beta C$$
$$C \leftarrow \alpha [A_1 B_1 + A_2 B_2],$$

which leads to the algorithm:

1. Scale $C \leftarrow \beta C$.
2. Let $A_{cur} = A$ and $B_{cur} = B$.

3. Repeat until done

   - Determine width of the split. If A_{cur} and B_{cur} are empty, break the loop.
   - Partition

     $$A_{cur} = \left(\begin{array}{c|c} A_1 & A_2 \end{array} \right)$$

     and

     $$B_{cur} = \left(\frac{B_1}{B_2} \right)$$

     where A_1 and B_1 are $m \times s$ and $s \times n$ matrices, respectively.
   - Update $C \leftarrow C + \alpha A_1 B_1$.
   - Let $A_{cur} = A_2$ and $B_{cur} = B_2$.

In this case, the symbol $\leftarrow$ indicates assignment, while the symbol $=$ indicates a reference or a partitioning. Note that this approach is usually referred as the *panel-panel variant*, as the basic operation uses panels of each matrix to perform the rank-k update. The code that follows gives the PLAPACK implementation for the operation. Note how PLAPACK provides routines to perform the partitioning and repartitioning of the matrices necessary in this algorithm. The codes also introduces the concept of *projected multi-vector*. A multi-vector can be considered as a group of vectors that are treated as a whole and are operated (and communicated) simultaneously. All vectors in a multi-vector are of equal length and present identical alignment. Intuitively, a multi-vector can be seen as a matrix with a few columns, where all columns are equally distributed, like vectors.

```
int pgemm_panpan( int nb_alg, PLA_Obj alpha, PLA_Obj a,
                                             PLA_Obj b,
                              PLA_Obj beta,  PLA_Obj c )
{
  PLA_Obj acur = NULL, a1 = NULL, a1dup = NULL, a1dup_cur = NULL,
          bcur = NULL, b1 = NULL, b1dup = NULL, b1dup_cur = NULL,
          one  = NULL;

  int size, width_a1, length_b1;

  /* Scale c by beta */
  PLA_Scal( beta, c );

  /* Create mscalar 1 */

  /* Take a view of all of both a and b */
  PLA_Obj_view_all( a, &acur );
  PLA_Obj_view_all( b, &bcur );

  /* Create duplidated pmvectors for spreading panels of a and b */
  PLA_Pmvector_create_conf_to( c, PLA_PROJ_ONTO_COL,
                                  PLA_ALL_COLS, nb_alg, &a1dup );
  PLA_Pmvector_create_conf_to( c, PLA_PROJ_ONTO_ROW,
                                  PLA_ALL_ROWS, nb_alg, &b1dup );

  /* Loop until no more of a and b */
  while( TRUE ) {
    /* Determine width of next update */
    PLA_Obj_width ( acur, &width_a1 );
    PLA_Obj_length( bcur, &length_b1 );
```

```
32      if( ( size = min( width_a1, length_b1, nb_alg ) ) == 0 ) break;

34      /* Split off first col of acur */
        PLA_Obj_vert_split_2( acur, size,     &a1, &acur );
36      /* Split off first row of bcur */
        PLA_Obj_horz_split_2( bcur, size,     &b1,
38                                            &bcur );

40      /* Size the workspaces */
        PLA_Obj_vert_split_2( a1dup, size,    &a1dup_cur, PLA_DUMMY );
42      PLA_Obj_vert_split_2( b1dup, size,    &b1dup_cur,
                                              PLA_DUMMY );
44
        /* Annotate the views */
46      PLA_Obj_set_orientation( a1, PLA_PROJ_ONTO_COL );
        PLA_Obj_set_orientation( b1, PLA_PROJ_ONTO_ROW );
48
        /* Spread a1 and b1 */
50      PLA_Copy( a1, a1dup_cur );
        PLA_Copy( b1, b1dup_cur );
52
        /* Perform local rank-size update */
54      PLA_Local_Gemm( PLA_NOTRANS, PLA_NOTRANS, alpha, a1dup_cur,
                                                         b1dup_cur,
56                                                 one,   c );
    }
58
    /* Free views */
60
}
```

PLAPACK provides routines to handle data partitioning and repartitioning. Also, see how communication details are hidden for the programmer. The communication pattern is identical to that shown for the matrix-vector product: objects are created and data distribution is specified; then, single invocations to `PLA_Copy` between objects hide the intricate communication details from the programmer.

Implementing the Cholesky factorization

The implementation of the Cholesky factorization using PLAPACK illustrates the methodology that can be used to make use of the PLAPACK infrastructure and exploit the distributed versions of the BLAS routines offered by the library.

The following code illustrates a distributed PLAPACK implementation for the blocked right-looking variant of the Cholesky factorization.

```
1   int chol_right_blas3( PLA_Obj a )
    {
3     PLA_Obj      acur = NULL, a11 = NULL, a21 = NULL,
                   one = NULL, min_one = NULL;
5     PLA_Template template;
      MPI_Datatype datatype;
7     int          nb_alg, size;

9     /* Extract suggested block size for the bulk of the computation */
      PLA_Environ_nb_alg( PLA_OP_SYM_PAN_PAN, &nb_alg );
11
```

```
  /* Create multiscalars one and min_one ... */

  /* Take a view (reference) of A */
  PLA_Obj_view_all( a, acur );

  while( TRUE ) {
    /* Check dimension of Acur */
    PLA_Obj_global_length acur, &size );

    /* If current size is 0, Acur is empty */
    if( ( size = min( size, nb_alg ) ) == 0 ) break;

    /* Split Acur = / A_11  *    \ */
    /*              \ A_21  Acur / */
    PLA_Obj_split_4( acur, size, size, &a11, PLA_DUMMY,
                                       &a21, &acur );

    /* Compute the Cholesky factorization of A_11 */
    chol_right_blas2( a11 );

    /* Update A_21 <- A_21 A_11^(-T) */
    PLA_Trsm( PLA_SIDE_RIGHT, PLA_LOW_TRIAN, PLA_TRANS, PLA_NONUNIT_DIAG,
                                             one, a11, a21 );

    /* Update Acur <- Acur - A_21 A_21^T */
    PLA_Syrk( PLA_LOW_TRIAN, PLA_NO_TRANS, min_one, a21, acur );
  }

  PLA_Obj_free( &a11 ); PLA_Obj_free( &a21 ); PLA_Obj_free( &acur );
  PLA_Obj_free( &one ); PLA_Obj_free( &min_one );

  PLA_Temp_set_comm_dir( template, PLA_DIR_TEMP_ROW, old_dir_row );
  PLA_Temp_set_comm_dir( template, PLA_DIR_TEMP_COL, old_dir_col );
}
```

In the PLAPACK code shown above for the BLAS-3 based implementation of the right-looking variant of the Cholesky factorization, the block size b must be initially chosen. Typically, this block size equals the width of L_{21} which makes the symmetric rank-k update most efficient. To accomplish that, the PLAPACK infrastructure offers querying functions to get the optimal value for the block size for a given BLAS call (in the example, routine `PLA_Environ_nb_alg( ...)`. Note how, in this case, the communication patterns and actual communication calls are encapsulated into the (parallel) BLAS calls `PLA_Trsm` and `PLA_Syrk`, which gives an overview of how the PLAPACK infrastructure can be used as a library to build higher-level parallel implementations such as the Cholesky factorization, without explicitly exposing communication details.

6.5. Porting PLAPACK to clusters of GPUs

There are three main reasons underlying the election of PLAPACK as the target infrastructure to port to clusters of GPUs. First, its layered approach yields an easy identification and adaptation of exactly the necessary parts of the library without impact in others. Second, PLAPACK was the origin of the FLAME methodology, which turns the port consistent with the rest of the methodologies and APIs used throughout the rest of the thesis. Third, the object-based nature of PLAPACK allows us to cleanly separate concerns and encapsulate inside the linear algebra object the existence of different memory spaces and architectures whithin a node. In addition, communication patterns

```
int PLA_Chol( int b, PLA_Obj A )
{
  PLA_Obj  ABR = NULL,
           A11 = NULL,      A21 = NULL;
  /* ... */

  /* View ABR = A */
  PLA_Obj_view_all( A, &ABR );

  while ( TRUE ) {
    /* Partition  ABR = / A11  ||   *  \
     *                  | =====  ===== |
     *                  \ A21  || ABR  /
     * where A11 is b x b              */
    PLA_Obj_split_4( ABR, b, b,
                     &A11, PLA_DUMMY,
                     &A21, &ABR );

    /* A11 := L11 = Cholesky Factor( A11 ) */
    PLA_Local_chol( PLA_LOWER_TRIANGULAR, A11 );

    /* Update A21 := L21 = A21 * inv( L11' ) */
    PLA_Trsm( PLA_SIDE_RIGHT, PLA_LOWER_TRIANGULAR,
              PLA_TRANSPOSE,  PLA_NONUNIT_DIAG,
              one, A11,
                   A21 );

    /* Update A22 := A22 - L21 * L21' */
    PLA_Syrk( PLA_LOWER_TRIANGULAR, PLA_NO_TRANS,
              minus_one, A21,
              one,       ABR );
  }
  /* ... */
}
```

```
int CUPLA_Chol( int b, PLA_Obj A )
{
  PLA_Obj  ABR = NULL,
           A11 = NULL,      A21 = NULL;
  /* ... */

  /* View ABR = A */
  PLA_Obj_view_all( A, &ABR );

  while ( TRUE ) {
    /* Partition  ABR = / A11  ||   *  \
     *                  | =====  ===== |
     *                  \ A21  || ABR  /
     * where A11 is b x b              */
    PLA_Obj_split_4( ABR, b, b,
                     &A11, PLA_DUMMY,
                     &A21, &ABR );

    /* A11 := L11 = Cholesky Factor( A11 ) */
    CUPLA_Local_chol( PLA_LOWER_TRIANGULAR, A11 );

    /* Update A21 := L21 = A21 * inv( L11' ) */
    CUPLA_Trsm( PLA_SIDE_RIGHT, PLA_LOWER_TRIANGULAR,
                PLA_TRANSPOSE,  PLA_NONUNIT_DIAG,
                one, A11,
                     A21 );

    /* Update A22 := A22 - L21 * L21' */
    CUPLA_Syrk( PLA_LOWER_TRIANGULAR, PLA_NO_TRANS,
                minus_one, A21,
                one,       ABR );
  }
  /* ... */
}
```

Figure 6.11: Original PLAPACK code for the right-looking variant of the Cholesky factorization (left). Equivalent accelerated code using GPUs (right).

are limited and can be systematically used without exception. As the number of communication routines is restricted to copy and reduction, the use of accelerators does not imply major changes to neither the user's code nor the internals of PLAPACK, as explained in this section.

Programmability is the main goal of our port of the PLAPACK infrastructure to clusters of GPUs. Besides the easy methodology proposed by PLAPLACK for the development of new dense linear algebra routines, the existence of a new memory space bound to each GPU in the cluster must remain transparent for the user. Neither explicit memory transfers nor different linear algebra routines have to be added to existing codes in order to unleash GPU acceleration. Adding the complexity of managing a new set of memory spaces would imply a new burden to the intricate task of message-passing programming. Only by satisfying this condition can the existing implementations be easily ported to the novel architecture. Looking ahead slightly, the major modifications that will be introduced into the original PLAPACK code will merely consist in changes in the names of the routines, as can be observed by comparing both codes shown in Figure 6.11.

A homogeneous distributed architecture features one independent memory space per node in the system. If accelerators are attached to each node, new independent memory spaces appear bound to each one of them.

The appearance of new memory spaces introduces two different possibilities to port the existing implementation. We propose a scheme in which one process is created per accelerator in the system. With this approach, local data bound to each accelerator can be physically stored in either the node main memory (which we call a *host-centric storage scheme*) or the accelerator memory (which we refer to as a *device-centric storage scheme*) for the major part of the computation time. The moment in which data is transferred from main memory to the accelerator memory (or vice-versa) is what ultimately differentiates both implementations.

This section describes both implementations, detailing the differences between them from the viewpoint of the necessary changes on the user's code and the internals of the PLAPACK infrastructure.

6.5.1. Host-centric storage scheme

The host-centric storage scheme is a naive port of the original PLAPACK infrastructure to hybrid distributed-memory architectures. In essence, the idea underlying this option is similar to that used in the first version of the multi-GPU runtime introduced in Chapter 5.

Following a close approach, data reside in main memory for the major part of the computation. Only when a local BLAS invocation has to be performed, data are temporarily allocated and transferred to the memory of the corresponding GPU, the BLAS execution is performed, and data are retrieved back into its original address in main memory. After data have been used, the associated buffer in GPU memory is destroyed and data are kept exclusively in main memory for the rest of the computation.

The changes required in the infrastructure are limited to the local BLAS module. Using this strategy, the BLAS calls `PLA_*` are in fact wrappers with the following minor modifications:

1. Initially, local buffers are allocated in GPU memory.
2. Local buffers in main memory bound to an object are transferred to GPU memory.
3. Local BLAS calls are replaced by GPU-specific BLAS calls (in this case, NVIDIA CUBLAS).
4. The contents of local buffers in GPU memory are transferred back to main memory upon completion of the BLAS invocation.
5. Local buffers in GPU memory are destroyed.

No further changes are necessary from the programmer perspective. There are two main characteristics in this approach. First, data reside in main memory of the nodes during most of the computation time. Second, data transfers between main memory and GPU memories are bound exclusively to *computations*.

The original PLAPACK code would remain unmodified in the accelerated version, provided the corresponding wrappers keep the same name than the original ones. As an example, the code in the left listing of Figure 6.11 can be accelerated by just linking the appropriate BLAS wrappers.

6.5.2. Device-centric storage scheme

A drastically different approach to port PLAPACK to accelerated cluster is possible. In the device-centric storage scheme, data reside in GPU memories for the major part of the computations. Only when a node transfer is required, data is temporarily transferred to main memory. On the other side of the communication, data is received in main memory and immediately transferred to GPU memory, where the computation continues. Thus, in this case, data transfers between GPU memories and main memories are bound exclusively to *communications*.

We describe in this section our approach to retarget PLAPACK to a cluster equipped with hardware accelerators following the device-centric storage scheme. Given that current NVIDIA accelerator boards include a RAM of 4 GBytes, we do not expect that the size of the device memory becomes a limiting factor for most applications. Otherwise, one could still handle the

device memory as a cache of the host memory, by implementing a software coherence protocol similar to that introduced in Chapter 5.

In order to describe the changes required by the device-centric approach, we will again consider the right-looking variant for the Cholesky factorization (shown in Section 6.4.2).

Communicating data in the Cholesky factorization

The primary vehicles for communication in PLAPACK are the copy and reduce operations. The approach in this library is to describe the distribution for the input and output using linear algebra objects, and then to *copy* or to *reduce* from one to the other. Thus, a prototypical communication is given by the call `PLA_Copy( A, B )` where included in the descriptors `A` and `B` is the information for the respective distributions. The `PLA_Copy` routine determines how data must be packed, which collective communication routine must be called to redistribute them, and how the contents must be unpacked after the communication. What this means is that, with the addition of memory local to an accelerator, the necessary data movement with the host processors that perform the communication needs to be added. This can be accomplished by hiding the details completely within the `PLA_Copy` routine. The `PLA_Reduce` routine allows contributions from different processors to be consolidated, and similar implementation details can be encapsulated inside it.

Data Movement in the Cholesky Factorization

Let us focus on the call `PLA_Trsm( ..., A11, A21 )`. In the particular implementation of the Cholesky factorization given in Section 6.4.2, `A11` exists within one process and `A21` within one column of processors, with all elements of a given row of `A21` assigned to the same processor. Inside this routine, `A11` is broadcast so that all processors that own part of `A21` receive a copy, after which local triangular solves complete the desired computation. The call to `PLA_Syrk` performs similar data movements and local computations. Although PLAPACK includes more complex implementations that require more intricate data movements, we purposely focus on a simple implementation since it captures the fundamental issues.

Necessary changes to PLAPACK

We describe how porting to an exotic architecture, like accelerators with local memories, is facilitated by a well-layered, object-based library like PLAPACK. We do so by exposing a small sampling of the low-level code in PLAPACK and discussing how this code has to be changed. We will employ the copy from a PLAPACK object of matrix type to a second object, of the same type, in order to illustrate the modular structure of PLAPACK and the series of changes that had to be made to accommodate the use of accelerators in PLAPACK. This copy is implemented in routine `PLA_Copy` from matrix to matrix, which we transformed into `CUPLA_Copy` from matrix to matrix. Several cases are treated within the copy routine. For example, when both matrices are aligned to the same template (i.e., the contents are distributed among the nodes following the same pattern), local copies from the buffer containing the elements of the source matrix to the one with those of the target matrix suffice, as shown by the following excerpt of PLAPACK code:

```
/* PLAPACK PLA_Copy between aligned matrices */
if ( align_col_from == align_col_to ) {
  if ( align_row_from == align_row_to ){
    PLA_Local_copy( Obj_from, Obj_to );
    done = TRUE;
  }
}
```

In PLAPACK, the local copy inside `PLA_Local_copy` is performed using (several invocations to) the BLAS-1 routine `scopy`; in our port to clusters of GPUs we invoke the analogous copy counterpart from the CUDA runtime to move data between different positions of the device memory. A more elaborate case occurs when the two matrices feature different alignments. For simplicity, consider that the matrices share the same column alignment but differ in the row alignment. Thus, each column process has to send the corresponding block to each one of the processes in the same column. For efficiency, before these blocks are sent, they are packed in an auxiliary buffer to hide part of the communication latency by increasing the granularity of messages. This is done in PLAPACK as follows:

```
/* PLAPACK PLA_Copy between column aligned matrices */
while ( TRUE ){
  PLA_Obj_split_size( from_cur, PLA_SIDE_TOP,
                      &size_from, &owner_from );
  PLA_Obj_split_size( to_cur,   PLA_SIDE_TOP,
                      &size_to,   &owner_to);

  if ( 0 == ( size = min( size_from, size_to ) ) )
    break;

  PLA_Obj_horz_split_2( from_cur, size,
                        &from_1,  &from_cur );
  PLA_Obj_horz_split_2( to_cur,   size,
                        &to_1,    &to_cur );

  if ( myrow == owner_from && owner_from == owner_to ){
    PLA_Local_copy( from_1, to_1 );
  }
  else{
    if ( myrow == owner_from ) {
      PLA_Obj_get_local_contents( from_1, PLA_NO_TRANS,
          &dummy, &dummy, buffer_temp, size, 1 );

      MPI_Send( BF( buffer_temp ),
          size * local_width, datatype,
          owner_to, 0, comm_col );
    }
    if ( myrow == owner_to ) {
      MPI_Recv( BF( buffer_temp ),
          size * local_width, datatype,
          owner_from, MPI_ANY_TAG, comm_col, &status );

      PLA_Obj_set_local_contents(
          PLA_NO_TRANS, size, local_width,
          buffer_temp, size, 1, to_1 );
    }
  }
}
PLA_free( buffer_temp );
```

In PLAPACK, routine `PLA_Obj_get_local_contents` copies the local contents of the object into a buffer:

```
/* PLAPACK PLA_Obj_get_local_contents */

int PLA_Obj_get_local_contents   (
  PLA_Obj    obj,          int    trans,
  int        *rows_in_buf, int    *cols_in_buf,
```

```
    void        *buf,          int     leading_dim_buf,
    int         stride_buf)

    /* ... */
    for ( j=0; j<n; j++ ){
      tempp_local = buf_local + j*leading_dim_buf*typesize;
      tempp_obj   = buf_obj   + j*ldim*typesize;
      memcpy( tempp_local, tempp_obj, m*typesize );
    }
  /* ... */
```

An equivalent effect is achieved in our approach with a simple invocation to CUBLAS routine `cublasGetMatrix`, which packs the data into a contiguous buffer while simultaneously retrieving them from the device memory:

```
  /* Accelerated PLA_Obj_get_local_contents */

  int CUPLA_Obj_get_local_contents   (
    PLA_Obj     obj,           int     trans,
    int         *rows_in_buf,  int     *cols_in_buf,
    void        *buf,          int     leading_dim_buf,
    int         stride_buf)

    /* ... */
    cublasGetMatrix( m, n, typesize,
                     buf_obj,   ldim,
                     buf_local, leading_dim_buf );
    /* ... */
```

Analogous changes are needed in the remaining cases of `PLA_Copy`, as well as `PLA_Reduce`, the second key routine devoted to data duplication and consolidation in PLAPACK, which together embed all data communication (transfer) in PLAPACK. It is important to note that those changes are mainly mechanical, and similar communication schemes are repeated for all cases inside the copy and reduce routines.

Thus, only three of the modules inside the PLAPACK layered infrastructure must be modified to obtain an accelerated version of the whole setup. In fact, those three modules are the whole *abstraction layer* of the infrastructure, which validates the ability of PLAPACK to accommodate novel architectures (abstracting the rest of the part of the infrastructure), and justifies the decision of precisely porting this library to the accelerated system:

Local BLAS. Local BLAS calls are executed on the GPU, so the corresponding BLAS implementation must be invoked for each PLAPACK local BLAS call. We provide a set of equivalent BLAS routines with the prefix `CUPLA_` to implement this functionality.

Linear Algebra Manipulation routines. Objects are created with an associated buffer in GPU memories. Thus, object creation and manipulation routines must deal with this difference. We implement a set of alternative implementations with the prefix `CUPLA_*` in order to maintain the functionality of the original calls, but operate with data stored on the GPU.

PLA_Copy/Reduce. Communication routines must be modified according to the methodology explained.

With these changes, user's codes remain basically unmodified. Only the appropriate `CUPLA_*` routines have to be used instead of the existing `PLA_*` invocations when needed. No explicit memory allocations or data transfers between memory spaces are added to user's codes. As an example,

consider the necessary changes in the Cholesky factorization routine shown in Figure 6.11. An hypothetical driver would include the appropriate adapted codes to allocate, manage and free objects bound to the GPU memory space.

6.6. Experimental results

The goal of the experiments in this section is twofold. First, to report the raw performance that can be attained with the accelerated version of the PLAPACK library for two common linear algebra operations: the matrix-matrix multiplication and the Cholesky factorization. Second, to illustrate the scalability of the proposed solution by executing the tests on a moderate number of GPUs.

LONGHORN is a hybrid CPU/GPU cluster designed for remote visualization and data analysis. The system consists of 256 dual-socket nodes, with a total of 2,048 compute cores (Intel Xeon Nehalem QuadCore), 512 GPUs (128 NVIDIA Quadro Plex S4s, each containing 4 NVIDIA FX5800), 13.5 TBytes of distributed memory and a 210 TBytes global file system. The detailed specifications of the cluster were illustrated in Table 1.2.

In our experiments we only employ 16 compute nodes from LONGHORN. The results for 1, 2, 4, 8 and 16 GPUs make use of one of the GPUs in each node, with one MPI process per computing node. The results on 32 GPUs (there are two GPUs per node) were obtained using two MPI processes per node. This setup also illustrates the ability of the accelerated version of PLAPACK to deal with systems equipped with more than one accelerator per node (e.g., in a configuration with nodes connected to NVIDIA Tesla S1070 servers as the TESLA2 machine evaluated in Chapter 5). Here, only those results that are obtained for optimal values of the distribution and algorithmic block sizes are shown.

The matrix-matrix multiplication is frequently abused as a showcase of the highest attainable performance of a given target architecture. Following this trend, we have developed an implementation of this operation based on the PLAPACK `PLA_Gemm` routine. The performance results for the matrix-matrix multiplication are shown in Figure 6.12. The highest performance achieved with the adapted version of the matrix multiplication routine is slightly over 8 TFLOPS when using 32 GPUs for matrices of dimension 61,440 $\times$ 61,440.

A similar experiment has been carried out for the Cholesky factorization. In our implementation of the PLAPACK routine `PLA_Chol`, the factorization of the diagonal blocks is computed in the CPU using the general-purpose cores while all remaining operations are performed on the GPUs. This hybrid strategy has been successfully applied in previous chapters and studies [22, 23, 142]. Figure 6.13 reports the performance of the Cholesky routine on LONGHORN, which delivers 4.4 TFLOPS for matrices of dimension 102,400 $\times$ 102,400.

A quick comparison between the top performance of the matrix-matrix multiplication using the accelerated version of PLAPACK (8 TFLOPS) and the (theoretical) peak performance of the CPUs of the entire system (41.40 TFLOPS in single-precision arithmetic) reveals the advantages of exploiting the capabilities of the GPUs: using only 6% of the graphics processors available in the cluster (32 out of 512 GPUs) it is possible to attain 20% of the peak performance of the machine considering exclusively the available CPUs.

Figure 6.14 illustrates the scalability of the device-centric implementation matrix-matrix multiplication routine. Compared with the performance of the "serial" tuned implementation of the matrix-matrix product routine in NVIDIA CUBLAS, our routine achieves a 22$\times$ speedup on 32 GPUs, which demonstrates the scalability of the solution. Two main reasons account for the performance

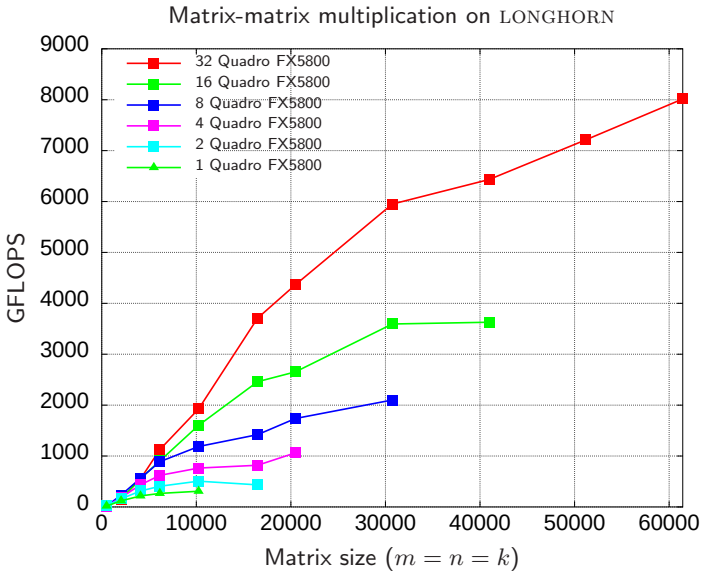


Figure 6.12: Performance of the device-centric implementation of GEMM on 32 GPUs of LONGHORN.

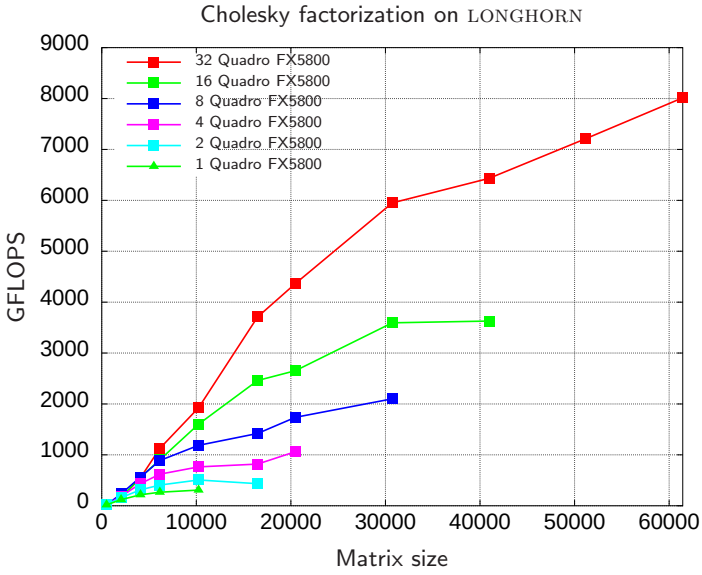


Figure 6.13: Performance of the device-centric implementation of the Cholesky factorization on 32 GPUs of LONGHORN.

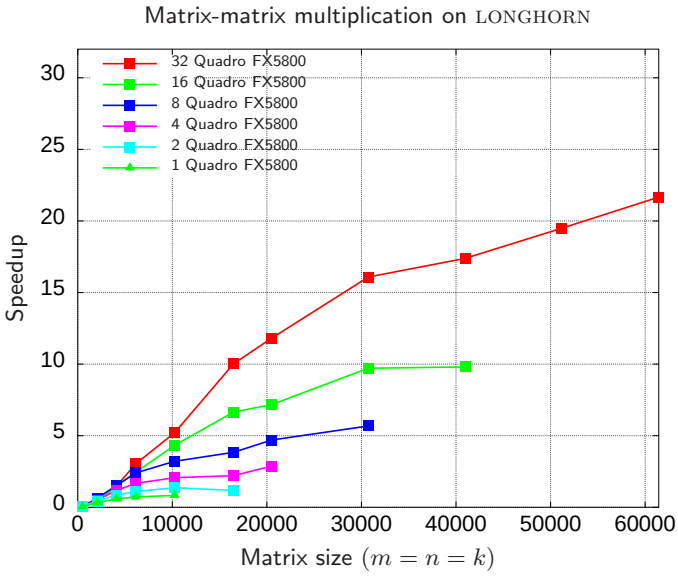


Figure 6.14: Speed-up of the device-centric GEMM implementation on LONGHORN.

penalty: the Infiniband network and the PCIExpress bus (mainly when 32 GPUs/16 nodes are employed as the bus is shared by two GPUs per node).

The plots in Figure 6.15 evaluate the performance of the original PLAPACK implementation and the GPU-accelerated version of the library for the matrix-matrix multiplication (left-side plot) and the Cholesky implementation (right-side plot). Results are shown only on 16 nodes of LONGHORN. Thus, the plots report the performance of the PLAPACK implementation using 128 Intel Nehalem cores (8 cores per node) versus those of the accelerated library using 16 and 32 GPUs (that is, one or two GPUs per each node). For the matrix-matrix multiplication, the highest performance for PLAPACK is 2.3 TFLOPS, while the accelerated version of the library attains 3.6 TFLOPS for 16 GPUs and 6.4 TFLOPS for 32 GPUs. The speedups obtained by the accelerated routines are 1.6× and 2.8×, respectively. For the Cholesky implementation, the PLAPACK routine attains a peak performance of 1.6 TFLOPS, compared with the 2.6 and 4.5 TFLOPS achieved by the accelerated versions of the routines on 16 and 32 GPUs, respectively. In this case, the corresponding speedups are 1.7× and 2.8×, respectively.

Note how the performance of the CPU-based version of PLAPACK is higher for matrices of small dimension. This fact is usual in GPGPU algorithms, and has already been observed in other works [22, 142] and in previous chapters. In response to this, hybrid algorithms combining CPU and GPU execution are proposed in those works. In the PLAPACK library, the implementation of hybrid algorithms would require a deep modification of internals of the library. Other approaches for multi-GPU computing computations can be integrated in the library to optimally exploit the heterogeneous resources in each node of the cluster. Independent studies, such as StarPU [13] or studies from the author of this thesis, such as the runtime proposed in Chapter 5 or GPUSs [16] can be integrated into the library to automatically deal with the heterogeneous nature of each node (multiple GPUs and general-purpose multi-core processors).

The benefits of using an approach in which data are kept in the GPU memory during the whole computation are captured in Figure 6.16. The results report the performance of the host-centric approach, in which data is stored in the main memory of each node and transferred to the GPU when it is strictly necessary, and the device-centric approach, in which data is stored (most of the time) in the GPU memory. The advantages of the second approach, in terms of higher performance,

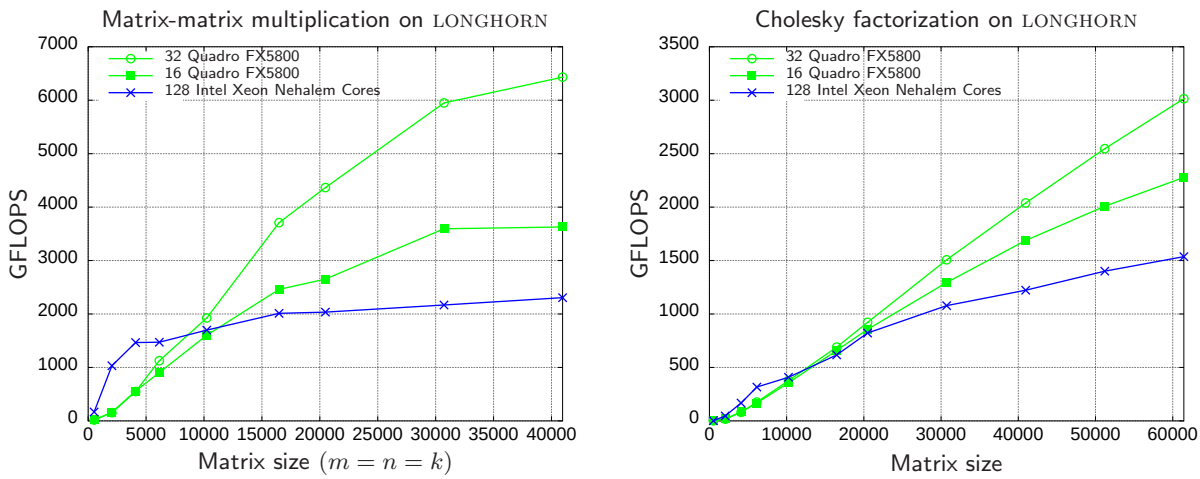


Figure 6.15: Performance of the device-centric implementation of GEMM (left) and the Cholesky factorization (right) compared with that of PLAPACK on 128 cores of LONGHORN.

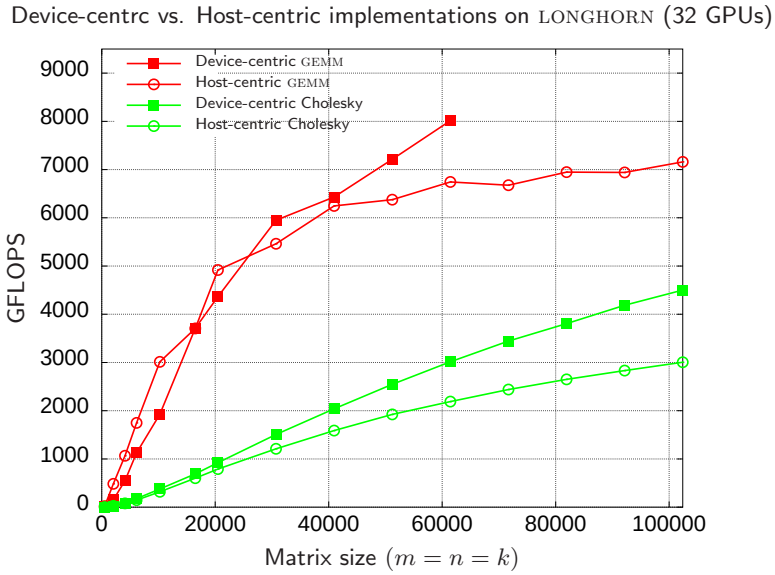


Figure 6.16: Cholesky factorization and GEMM on 16 nodes of LONGHORN, using the host-centric and device-centric storage approach for the accelerated implementation.

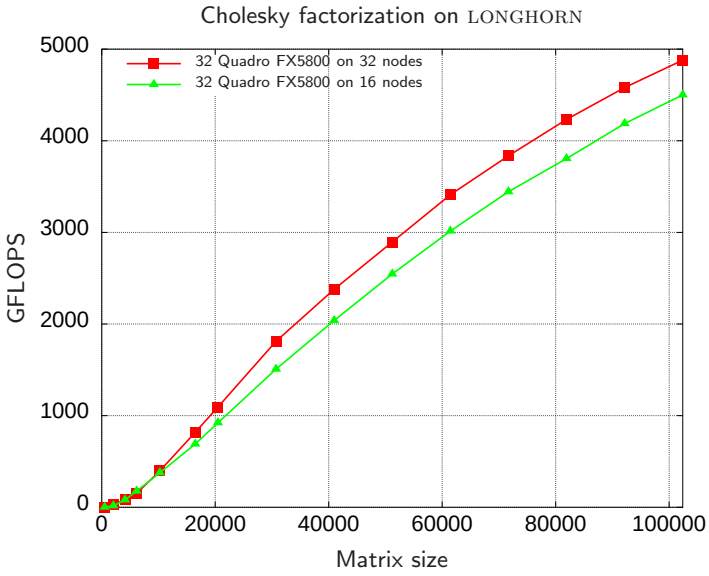


Figure 6.17: Performance of the device-centric implementation of GEMM on 16 nodes of LONGHORN, using 1 or 2 GPUs per node.

are clear, especially for large matrices. On the other hand, in the host-centric approach, the size of the problem that can be solved is restricted by the amount of main memory in the system, which is usually larger than the device memory (see the tested sizes for the matrix-matrix multiplication in Figure 6.16). In principle, this can be solved transparently to the programmer in the device-centric approach, by handling the device memory as a cache of the host memory as proposed in Chapter 5.

Figure 6.17 shows the performance of the accelerated version of the Cholesky factorization in PLAPACK executed on 32 GPUs of LONGHORN. The results illustrate the difference in performance between a configuration in which one GPU per node is used (using a total of 32 nodes) and that of a configuration where two GPUs per node are used for the calculations (for a total of 16 nodes). In the latter case, the penalty introduced by the common usage of the PCIExpress bus in each node is less than 8% for the largest matrices. Although the difference in performance is non-negligible, the multi-GPU configuration delivers interesting performance results, and thus the trade-off between acquisition costs and raw performance must be also taken into account.

6.7. Conclusions

In previous chapters, we have demonstrated how multi-GPU systems can be an appealing solution to implement high-performance dense linear algebra routines. However, given the bottleneck introduced by the PCIExpress bus as the number of GPU sharing it increases, clusters of GPUs, with a reduced number of GPUs per node, are the natural evolution towards high performance GPU-based large-scale systems.

Porting existing distributed-memory codes to hybrid GPU-CPU clusters may be a challenging task. We have presented an approach to mechanically port the routines of the dense linear algebra message-passing library PLAPACK to a hybrid cluster consisting of nodes equipped with hardware accelerators. By initially placing all data in the memory of the accelerators, the number of PCIExpress transfers between the memories of host and device is reduced and performance is improved. All data transfers are embedded inside PLAPACK communication (copy) and consolidation (re-

duce) routines so that the retarget of the library routines is mostly automatic and transparent to the user.

The experimental results have demonstrated that the integration of GPUs in the nodes of a cluster is an efficient, cheap and scalable solution for the acceleration of large dense linear algebra problems. Furthermore, PLAPACK has also demonstrated its portability to novel architectures. From the perspective of the user, the development of GPU-accelerated codes becomes a transparent task with the adaptation of the library to clusters of GPUs.

CHAPTER 7

Conclusions

7.1. Conclusions and main contributions

The main goal of the dissertation is the *design, development and evaluation of programming strategies to improve the performance of existing dense linear algebra libraries on systems based on graphics processors*. This primary goal has been pursued keeping some important premises in the framework of dense linear algebra computations:

Performance: All linear algebra routines, independently from the target architecture, have attained important speedups compared with their counterpart tuned versions for modern CPUs. When possible, a comparison with tuned GPU implementations of the routines (such as NVIDIA CUBLAS in the case of single-GPU BLAS developments) has demonstrated that the proposed implementations also attain better performance results for all routines. These results validate the application of the FLAME technology, and the high-level approach in the development of new codes, not only from the programmability point of view, but also from the performance perspective.

Programmability: As a second contribution of the thesis, we have applied the FLAME methodology, and a high-level developing approach, to different GPU-based architectures. Remarkably, no low-level CUDA codes have been written in the framework of the routines developed for single-GPU architectures, systems based on multiple GPUs or clusters of GPUs, yet high performances were obtained. This gives an idea of the benefits of the application of FLAME also for novel architectures, and its benefits on programmability, with independence from the target architecture in which those routines are executed.

Versatility: As a third contribution, the separation of concerns introduced by FLAME (and, similarly, by PLAPACK) enables the application of strategies similar to those developed in this dissertation to virtually any accelerator-based architecture similar to those used in our evaluations. Our purpose was to detach the programmer's view from the underlying architecture. Proceeding in this manner, programming models and user's codes remain basically unmodified for accelerated or non-accelerated platforms. In addition, the concepts approached at the architectural level are common for other types of accelerated platforms, not only GPU-based systems.

In addition to the proposed high-level methodologies and optimization techniques, as a result of the developed work we have obtained a full family of implementations for BLAS-level and LAPACK-level routines on three different GPU-based platforms: single-GPU, multi-GPU, and clusters of GPUs. Furthermore, many of the techniques described in this dissertation are general enough to be applied to other linear algebra implementations if a full, optimized implementation of BLAS-level and LAPACK-level implementations is desired on present and future accelerator-based architectures.

In general, the developed work has proved that a high-level approach, as that pursued in the framework of the FLAME project, is perfectly valid to port existing linear algebra libraries to novel architectures and to attain high performance at the same time. In addition, this type of methodologies presents two main advantages. First, ease of code development: FLAME provides APIs and tools to easily perform the transition from algorithms to code [26, 126]. Second, flexibility and portability of the developed codes: similar simultaneous and independent studies have addressed low-level coding strategies for linear algebra implementations on the same target architectures as ours. Although performance results for those works are remarkable, further coding efforts will be necessary if those codes need to be adapted to future accelerator-based architectures. With our methodologies, the programming low-level effort is focused on a reduced number of basic routines, on top of which the rest of the codes are developed.

Whether GPUs will remain as a reference in accelerator-based platforms is not clear at this time. Manufacturers are working together to apply unified programming models, such as OpenCL. From the software perspective, these efforts should derive in common software developments for all type of accelerator-based platforms. However, the programming efforts required to tune these codes to each specific platform, will remain nonnegligible. In addition, the extension of the developed routines to more sophisticated platforms, such as systems with multiple accelerators or clusters of GPUs, will remain a non-trivial task. These problems will be even more severe if heterogeneous systems, with accelerators of different nature sharing common resources, appear in response to the heterogeneity of some applications.

We have validated the proposed methodologies on three different architectures: systems with one GPU, systems with multiple GPUs, and clusters of GPUs. The general conclusions extracted from the programmability and performance points of view are common for the three architectures. However, several specific contributions have been developed for each platform. The following sections detail those specific contributions.

7.1.1. Contributions for systems with one GPU

A full set of implementations for BLAS-level routines and for some representative LAPACK-level routines has been developed and evaluated on systems with one GPU.

A complete implementation and evaluation of the main Level-3 BLAS routines has been performed. Experimental results reveal how, taking advantage of the high-performance of the GEMM implementation in NVIDIA CUBLAS, all derived BLAS routines outperform the corresponding implementations in the library from NVIDIA. The programming effort is reduced as no low-level CUDA code is necessary. In addition, as shown, a similar methodology is common for *every* routine in Level-3 BLAS; results have been validated for single and double precision, extracting similar qualitative improvements in performance. Following the FLAME philosophy, a systematic development and evaluation of several algorithmic variants for each routine, together with a thorough evaluation to find the optimal block size for each one, offers a complete family of tuned implementations for each BLAS routine. From those results results, the optimal combination of block size and algorithmic variant has been determined.

Performance results improve those of the proprietary library from NVIDIA (NVIDIA CUBLAS), attaining remarkable speedups. Despite the efforts devoted by the company towards the optimization of the library, the advantages of our high-level approach are obvious: speedups between 2x and 4x are attained for all the evaluated routines in Level-3 BLAS, achieving peak speedups of about 14x in specific cases (for example, rectangular matrices).

Several contributions are derived from the implementation of LAPACK-level routines on systems with one GPU. As for the BLAS implementation, a systematic derivation and evaluation of several algorithmic variants has been performed for the Cholesky factorization and the LU factorization with partial pivoting. The detailed study of the optimal block size for each variant yield remarkable speedups compared with tuned multi-threaded implementations on modern multi-core processors.

For the first time, a GPU-accelerated system is viewed as a hybrid architecture; in response to this, we propose a set of *hybrid* implementations, in which each sub-operation in the factorizations is executed in the most suitable computing resource. Performance results for the hybrid approach improve the basic implementations in all cases.

Regarding precision, a previous evaluation of the BLAS routines reveals a dramatic performance drop in modern graphics processors depending on the floating-point arithmetic precision employed. To deal with this limitation, we apply a *mixed precision iterative refinement* approach which combines the performance potential of modern GPUs with the accuracy in results of double precision in the solution of linear systems using the Cholesky factorization or the LU factorization.

The reduction to tridiagonal form has been implemented on systems with one GPU in order to illustrate the capabilities of the architecture for other type of LAPACK-level implementations. In this case, we have performed a comparison between the implementations in LAPACK and our approach based on the SBR toolkit, following hybrid approach. Performance results have shown that the latter option, using the GPU to accelerate the critical parts of the algorithm, outperforms the LAPACK implementation.

7.1.2. Contributions for multi-GPU systems

On systems with multiple GPUs sharing common resources, the programming effort is multiplied due to the existence of several independent memory spaces, one per GPU attached to the system, plus that of main memory. Given that no direct GPU-GPU communication is possible, and that the PCIExpress bus becomes the main bottleneck as the number of GPUs is increased, we propose a *runtime* that transparently and efficiently manages data allocation and communication between GPUs. In addition, the runtime orchestrates the execution of the routines by exploiting *task-level* parallelism in a transparent manner from the programmer point of view.

In this sense, a second major contribution in our work is the consideration of each GPU in the system as a single processing unit, delegating further parallelization levels to ad-hoc GPU implementations, such as NVIDIA CUBLAS.

Performance results for a set of representative BLAS routines and for the Cholesky factorization reveal the benefits of our approach. We propose a set of improvements to boost performance, namely the consideration of each GPU memory as a *software-managed cache* of recently-used blocks, or the application of well-known techniques in computer architecture such as a *write-back* policy that allows inconsistencies of blocks of data between main memory and GPU memory, or a *write-invalidate* policy to keep coherence of data blocks among different memory spaces. The ultimate goal of these policies is the reduction of the number of data movements between accelerators and main memory. These techniques have demonstrated their benefits for all the evaluated operations.

The proposed approach is fundamentally different from that of the MAGMA project. First, the integration of our methodology into `libflame` provides a straightforward, *full* port of all the

algorithms-by-blocks already developed in the library to multi-GPU systems. This huge functionality is not supported in the current release of MAGMA (version 1.0, December 2010). Second, our approach is essentially transparent to the programmer, without delegating decisions such as data distribution, data transfers or memory management to the programmer level. Third, our optimizations at runtime level are applicable to all routines supported by the library; no specific optimizations are required depending on the specific implemented routine.

7.1.3. Contributions for clusters of GPUs

Our contributions on distributed-memory architectures equipped with GPUs are focused on the adaptation of the PLAPACK infrastructure to this type of platforms. We demonstrate how, by adopting a modular design in the development of the library, the modifications required are not dramatic. Moreover, this requirements towards the acceleration of the library are transparent for the programmer. We have shown how, using our approach, accelerated and non-accelerated codes present minimal differences. The existence of new memory spaces and the associated data transfers are transparent for the library developer.

We have proposed two different approaches to modify the PLAPACK library. In the *host-centric approach*, data is stored in main memory most of the time, and data transfers to and from the GPUs are bound exclusively to computations. In the *device-centric approach* data is kept in GPU memories, and data transfers are bound exclusively to communications.

Experimental results on a large GPU cluster reveal remarkable performance results and speedups compared with CPU-based implementations. Although our experimental results are restricted to GEMM and the Cholesky factorization, similar improvements are expected for other implementations. As of today, no similar ports of distributed-memory linear algebra routines have been made to address accelerated clusters, so a comparison of performance results is not possible.

7.2. Related publications

The scientific contributions developed for this thesis has been validated with several peer-reviewed publications in national and international conferences, and international journals. Each one of the topics explained in this document is supported by, at least, one international publication.

The following sections list the main publications derived from the thesis. We divide them into papers directly related to the thesis topics, papers indirectly related to the thesis topics but with some degree of relationship with dense linear algebra computations on GPU-based platforms, and papers unrelated to the thesis topics, but with relation with GPU computing. For the first group of publications, we provide a brief abstract of the main contents of the paper. Only international conferences and journals are listed.

7.2.1. Publications directly related with the thesis topics

Chapter 3. BLAS on single-GPU systems

The first step towards the optimization of the BLAS on graphics processors was introduced in [82]. The paper identifies the BLAS-3 level as the most suitable candidate to attain high performance on current graphics architectures. The first advances and results towards the optimization of BLAS-3 level were introduced in [20]. The programmability issue was solved by introducing APIs inside the FLAME framework to deal with dense linear algebra implementations on single-GPU systems (FLAME@lab in [21] as a Matlab/Octave interface, and FLAG/C [148] as a C API).

Finally, the improvement techniques introduced as the main contribution of Chapter 3 were first introduced in [83].

The following is a detailed list of the main publications related to this topic:

Igual, F. D., Mayo, R., and Quintana-Ortí, E. S. Attaining high performance in general-purpose computations on current graphics processors. *High Performance Computing for Computational Science - VECPAR 2008: 8th International Conference, Toulouse, France, June 24-27, 2008. Revised Selected Papers* (2008), 406–419. Conference Proceedings

> The increase in performance of the last generations of graphics processors (GPUs) has made this class of hardware a co-processing platform of remarkable success in certain types of operations. In this paper we evaluate the performance of linear algebra and image processing routines, both on classical and unified GPU architectures and traditional processors (CPUs). From this study, we gain insights on the properties that make an algorithm more likely to deliver high performance on a GPU.

Barrachina, S., Castillo, M., Igual, F. D., Mayo, R., and Quintana-Ortí, E. S. Evaluation and tuning of the level 3 CUBLAS for graphics processors. In *Proceedings of the 10th IEEE Workshop on Parallel and Distributed Scientific and Engineering Computing, PDSEC 2008* (2008), pp. CD–ROM. Conference Proceedings

> The increase in performance of the last generations of graphics processors (GPUs) has made this class of platform a co-processing tool with remarkable success in certain types of operations. In this paper we evaluate the performance of the Level 3 operations in Nvidia Cublas, the implementation of BLAS for Nvidia GPUs with unified architecture. From this study, we gain insights on the quality of the kernels in the library and we propose several alternative implementations that are competitive with those in Nvidia Cublas. Experimental results on a GeForce 8800 Ultra compare the performance of Nvidia Cublas and the new variants.

Zafont, M. J., Martin, A., Igual, F., and Quintana-Orti, E. S. Fast development of dense linear algebra codes on graphics processors. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing. Workshop on High-Level Parallel Programming Models & Supportive Environments*, IEEE Computer Society, pp. 1–8. Conference Proceedings

> We present an application programming interface (API) for the C programming language that facilitates the development of dense linear algebra algorithms on graphics processors applying the FLAME methodology. The interface, built on top of the Nvidia Cublas library, implements all the computational functionality of the FLAME/C interface. In addition, the API includes data transference routines to explicitly handle communication between the CPU and GPU memory spaces. The flexibility and simplicity-of-use of this tool are illustrated using a complex operation of dense linear algebra: the Cholesky factorization. For this operation, we implement and evaluate all existing variants on an Nvidia G80 processor, attaining speedups 7x compared with the CPU implementations.

Barrachina, S., Castillo, M., Igual, F. D., Mayo, R., and Quintana-Ortí, E. S. FLAG@lab: An M-script API for linear algebra operations on graphics processors. In *9th International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA 2008)*. (To appear as Lecture Notes in Computer Science). Conference Proceedings

> We propose two high-level application programming interfaces (APIs) to use a graphics processing unit (GPU) as a co-processor for dense linear algebra operations. Combined with an extension of the FLAME API and an implementation on top of NVIDIA CUBLAS, the result is an efficient and user-friendly tool to design, implement, and execute dense linear algebra operations on the current generation of NVIDIA graphics processors, of wide-appeal to scientists and engineers. As an application of the developed APIs, we implement and evaluate the performance of three different variants of the Cholesky factorization.

CONFERENCE PROCEEDINGS

IGUAL, F. D., QUINTANA-ORTÍ, G., AND VAN DE GEIJN, R. Level-3 BLAS on a GPU: Picking the low hanging fruit. In *ICCMSE 2010: Proceedings of the Eighth International Conference of Computational Methods in Sciences and Engineering* (2011), AIP Conference Proceedings. (To appear. Also published as FLAME Working Note 37).

> The arrival of hardware accelerators has created a new gold rush to be the first to deliver their promise of high performance for numerical applications. Since they are relatively hard to program, with limited language and compiler support, it is generally accepted that one needs to roll up one's sleeves and tough it out, not unlike the early days of distributed memory parallel computing (or any other period after the introduction of a drastically different architecture). In this paper we remind the community that while this is a noble endeavor, there is a lot of low hanging fruit that can be harvested easily. Picking this low hanging fruit benefits the scientific computing community immediately and prototypes the approach that the further optimizations may wish to follow. We demonstrate this by focusing on a widely used set of operations, the level-3 BLAS, targeting the NVIDIA family of GPUs.

Chapter 4. LAPACK-level routines on single-GPU systems

In [22], we introduced the first results to date using NVIDIA CUBLAS as the underlying building block to develop LAPACK-level routines. In addition, we evaluated several algorithmic routines for the Cholesky and LU factorization with partial pivoting. For the first time, we applied the mixed-precision iterative refinement approach to the graphics processors, exploiting their high performance in single-precision arithmetic. In [23], we extended this evaluation to double-precision arithmetic. In [30] we proposed a method to accelerate the reduction to condensed forms using the GPU as the underlying platform.

The following is a detailed list of the main publications related to this topic:

CONFERENCE PROCEEDINGS

BARRACHINA, S., CASTILLO, M., IGUAL, F. D., MAYO, R., AND QUINTANA-ORTÍ, E. S. Solving dense linear systems on graphics processors. In *Proceedings of the 14th International Euro-Par Conference* (2008), E. Luque, T. Margalef, and D. Benítez, Eds., Lecture Notes in Computer Science, 5168, Springer, pp. 739–748.

> We present several algorithms to compute the solution of a linear system of equations on a GPU, as well as general techniques to improve their performance, such as padding and hybrid GPU-CPU computation. We also show how iterative refinement with mixed-precision can be used to regain full accuracy in the solution of linear systems. Experimental results on a G80 using NVIDIA CUBLAS 1.0, the implementation of BLAS for NVIDIA GPUs with unified architecture, illustrate the performance of the different algorithms and techniques proposed.

Barrachina, S., Castillo, M., Igual, F. D., Mayo, R., Quintana-Ortí, E. S., and Quintana-Ortí, G. Exploiting the capabilities of modern GPUs for dense matrix computations. *Concurrency and Computation: Practice and Experience 21*, 18 (2009), 2457–2477. Journal

> We present several algorithms to compute the solution of a linear system of equations on a graphics processor (GPU), as well as general techniques to improve their performance, such as padding and hybrid GPU-CPU computation. We compare single and double precision performance of a modern GPU with unified architecture, and show how iterative refinement with mixed precision can be used to regain full accuracy in the solution of linear systems, exploiting the potential of the processor for single precision arithmetic. Experimental results on a GTX280 using Nvidia Cublas 2.0, the implementation of BLAS for Nvidia GPUs with unified architecture, illustrate the performance of the different algorithms and techniques proposed.

Bientinesi, P., Igual, F. D., Kressner, D., and Quintana-Ortí, E. S. Reduction to condensed forms for symmetric eigenvalue problems on multi-core architectures. In *PPAM (1)* (2009), pp. 387–395. Conference Proceedings

> We investigate the performance of the routines in LAPACK and the Successive Band Reduction (SBR) toolbox for the reduction of a dense matrix to tridiagonal form, a crucial preprocessing stage in the solution of the symmetric eigenvalue problem, on general-purpose multi-core processors. In response to the advances of hardware accelerators, we also modify the code in SBR to accelerate the computation by off-loading a significant part of the operations to a graphics processor (GPU). Performance results illustrate the parallelism and scalability of these algorithms on current high-performance multi-core architectures.

Bientinesi, P., Igual, F. D., Kressner, D., Petschow, M., and Quintana-Ortí, E. S. Condensed forms for the symmetric eigenvalue problem on multi-threaded architectures. *Concurrency and Computation: Practice and Experience 23*, 7 (2011), 694–707. Journal

> We investigate the performance of the routines in LAPACK and the Successive Band Reduction (SBR) toolbox for the reduction of a dense matrix to tridiagonal form, a crucial preprocessing stage in the solution of the symmetric eigenvalue problem, on general-purpose multi-core processors. In response to the advances of hardware accelerators, we also modify the code in the SBR toolbox to accelerate the computation by off-loading a significant part of the operations to a graphics processor (GPU). The performance results illustrate the parallelism and scalability of these algorithms on current high-performance multi-core and many-core architectures.

Chapter 5. Matrix computations on multi-GPU systems

The work in [114] was in fact the first contribution, as far as we know, that utilized a run-time system to exploit task parallellism on dense linear algebra operations using multiple GPUs. A similar approach was taken to implement GPUSs, as detailed in [16]. Facing the programmability problem, the work on the extension of the StarSs programming model and its adaptation to future heterogeneous multi-core and many-core architectures derived in the extension proposals of the OpenMP standard presented in [15] and [14].

The following is a detailed list of the main publications related to that topic:

CONFERENCE PROCEEDINGS

QUINTANA-ORTÍ, G., IGUAL, F. D., QUINTANA-ORTÍ, E. S., AND VAN DE GEIJN, R. A. Solving dense linear systems on platforms with multiple hardware accelerators. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2009), ACM, pp. 121–130.

> The FLAME methodology, combined with the SuperMatrix runtime system, yields a simple yet powerful solution for programming dense linear algebra operations on multicore platforms. In this paper we provide evidence that this approach solves the programmability problem for this domain by targeting a more complex architecture, composed of a multicore processor and multiple hardware accelerators (GPUs, Cell B.E., etc.), each with its own local memory, resulting in a platform more reminiscent of a heterogeneous distributed-memory system. In particular, we show that the FLAME programming model accommodates this new situation effortlessly so that no significant change needs to be made to the codebase. All complexity is hidden inside the SuperMatrix runtime scheduling mechanism, which incorporates software implementations of standard cache/memory coherence techniques in computer architecture to improve the performance. Our experimental evaluation on a Intel Xeon 8-core host linked to an NVIDIA Tesla S870 platform with four GPUs delivers peak performances around 550 and 450 (single-precision) GFLOPS for the matrix-matrix product and the Cholesky factorization, respectively, which we believe to be the best performance numbers posted on this new architecture for such operations.

CONFERENCE PROCEEDINGS

AYGUADÉ, E., BADIA, R. M., IGUAL, F. D., LABARTA, J., MAYO, R., AND QUINTANA-ORTÍ, E. S. An extension of the StarSs programming model for platforms with multiple GPUs. In *Euro-Par* (2009), pp. 851–862.

> While general-purpose homogeneous multi-core architectures are becoming ubiquitous, there are clear indications that, for a number of important applications, a better performance/power ratio can be attained using specialized hardware accelerators. These accelerators require specific SDK or programming languages which are not always easy to program. Thus, the impact of the new programming paradigms on the programmer's productivity will determine their success in the high-performance computing arena. In this paper we present GPU Superscalar (GPUSs), an extension of the Star Superscalar programming model that targets the parallelization of applications on platforms consisting of a general-purpose processor connected with multiple graphics processors. GPUSs deals with architecture heterogeneity and separate memory address spaces, while preserving simplicity and portability. Preliminary experimental results for a well-known operation in numerical linear algebra illustrate the correct adaptation of the runtime to a multi-GPU system, attaining notable performance results.

CONFERENCE PROCEEDINGS

AYGUADE, E., BADIA, R. M., CABRERA, D., DURAN, A., GONZALEZ, M., IGUAL, F., JIMENEZ, D., LABARTA, J., MARTORELL, X., MAYO, R., PEREZ, J. M., AND QUINTANA-ORTÍ, E. S. A proposal to extend the OpenMP tasking model for heterogeneous architectures. In *IWOMP '09: Proceedings of the 5th International Workshop on OpenMP* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 154–167.

> OpenMP has recently evolved towards expressing unstructured parallelism, targeting the parallelization of a broader range of applications in the current multicore era. Homogeneous multicore architectures from major vendors have become mainstream, but

with clear indications that a better performance/power ratio can be achieved using more specialized hardware (accelerators), such as SSE-based units or GPUs, clearly deviating from the easy-to-understand shared-memory homogeneous architectures. This paper investigates if OpenMP could still survive in this new scenario and proposes a possible way to extend the current specification to reasonably integrate heterogeneity while preserving simplicity and portability. The paper leverages on a previous proposal that extended tasking with dependencies. The runtime is in charge of data movement, tasks scheduling based on these data dependencies, and the appropriate selection of the target accelerator depending on system configuration and resource availability.

Ayguadé, E., Badia, R., Bellens, P., Cabrera, D., Duran, A., Ferrer, R., Gonzalez, M., Igual, F., Jimenez-GonzÃ¡lez, D., Labarta, J., Martinell, L., Martorell, X., Mayo, R., Perez, J., Planas, J., and Quintana-Ortí, E. Extending OpenMP to survive the heterogeneous multi-core era. *International Journal of Parallel Programming 38* (2010), 440–459. Journal

This paper advances the state-of-the-art in programming models for exploiting task-level parallelism on heterogeneous many-core systems, presenting a number of extensions to the OpenMP language inspired in the StarSs programming model. The proposed extensions allow the programmer to write portable code easily for a number of different platforms, relieving him/her from developing the specific code to off-load tasks to the accelerators and the synchronization of tasks. Our results obtained from the StarSs instantiations for SMPs, the Cell, and GPUs report reasonable parallel performance. However, the real impact of our approach in is the productivity gains it yields for the programmer.

Chapter 6. Matrix computations on clusters of GPUs

In [65], we introduce the porting of the PLAPACK infrastructure to clusters of GPUs:

Fogue, M., Igual, F. D., Quintana-Ortí, E. S., and van de Geijn, R. A. Retargeting plapack to clusters with hardware accelerators. In *International Conference on High Performance Computing and Simulation (HPCS 2010)* (2010), pp. 444 –451. Conference Proceedings

Hardware accelerators are becoming a highly appealing approach to boost the raw performance as well as the price-performance and power-performance ratios of current clusters. In this paper we present a strategy to retarget PLAPACK, a library initially designed for clusters of nodes equipped with general-purpose processors and a single address space per node, to clusters equipped with graphics processors (GPUs). In our approach data are kept in the device memory and only retrieved to main memory when they have to be communicated to a different node. Here we benefit from the object-based orientation of PLAPACK which allows all communication between host and device to be embedded within a pair of routines, providing a clean abstraction that enables an efficient and direct port of all the contents of the library. Our experiments in a cluster consisting of 16 nodes with two Nvidia Quadro FX5800 GPUs each show the performance of our approach.

7.2.2. Publications indirectly related with the thesis topics

Related to dense linear algebra implementations on systems with one or multiple GPUs, a parallel research has been performed regarding out-of-core computations using hardware accelerators.

In these publications, we explore the possibility of solving large dense linear systems stored on disk, accelerating in-core calculations by using the graphics processors. Those local routines are based on the BLAS implementations proposed in Chapter 3. The work in [40] presents a MATLAB/OCTAVE interface to accelerate out-of-core computations using hardware accelerators in the framework of linear algebra. In [60] we propose a novel strategy to efficiently virtualize graphics processors on high performance clusters:

JOURNAL QUINTANA-ORTÍ, G., IGUAL, F., MARQUÉS, M., QUINTANA-ORTÍ, E. S., AND VAN DE GEIJN, R. Programming OOC matrix algorithms-by-tiles on multithreaded architectures. *ACM Trans. Math. Softw.* (Submitted).

JOURNAL CASTILLO, M., IGUAL, F. D., MARQUÉS, M., MAYO, R., QUINTANA-ORTÍ, E. S., QUINTANA-ORTÍ, G., RUBIO, R., AND VAN DE GEIJN, R. A. Out-of-core solution of linear systems on graphics processors. *International Journal of Parallel, Emergent and Distributed Systems 24*, 6 (2009), 521–538.

CONFERENCE PROCEEDINGS DUATO, J., IGUAL, F. D., MAYO, R., PEÑA, A. J., QUINTANA-ORTÍ, E. S., AND SILLA, F. An efficient implementation of GPU virtualization in high performance clusters. In *Euro-Par Workshops* (2009), pp. 385–394.

7.2.3. Other publications

Image processing is a discipline in which GPUs have historically delivered near-optimal performances. As an orthogonal research line, several publications in this field have been obtained during the development of this thesis. These publications are focused on a lower level approach, presenting fine-grained optimizations and ad-hoc improvements for current GPUs on biomedical image processing. We list some of the most important publications in this area:

JOURNAL IGUAL, F., MAYO, R., HARTLEY, T., CATALYUREK, U., RUIZ, A., AND UJALDON, M. Color and texture analysis on emerging parallel architectures. *Journal of High Performance Computing Applications* (2010). (Published online).

CONFERENCE PROCEEDINGS HARTLEY, T. D., CATALYUREK, U., RUIZ, A., IGUAL, F., MAYO, R., AND UJALDON, M. Biomedical image analysis on a cooperative cluster of gpus and multicores. In *Proceedings of the 22nd annual international conference on Supercomputing* (New York, NY, USA, 2008), ICS '08, ACM, pp. 15–25.

CONFERENCE PROCEEDINGS IGUAL, F., MAYO, R., HARTLEY, T., CATALYUREK, U., RUIZ, A., AND UJALDON, M. Exploring the gpu for enhancing parallelism on color and texture analysis. In *From Multicores and GPUs to Petascale. 14th International Conference on Parallel Computing (ParCo 2009)* (2010), vol. 19 of *Advances in Parallel Computing*, IOS Press, pp. 299–306.

CONFERENCE PROCEEDINGS IGUAL, F., MAYO, R., HARTLEY, T., CATALYUREK, U., RUIZ, A., AND UJALDON, M. Optimizing co-occurrence matrices on graphics processors using sparse representations. In *9th International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA 2008)*. (To appear as Lecture Notes in Computer Science).

7.3. Software efforts and technological transfer

The insights and efforts in the framework of this thesis have been translated into software products and collaborations with companies. The software efforts include the release of `libflame` [149] as an open source library at the disposal of the scientific community. As of its release date, the

Operation	LAPACK name	libflame name	FLAME/C	FLASH	GPU support	type support
Level-3 BLAS						
General matrix-matrix multiply	?gemm	Gemm	√	√	√	sdcz
Hermitian matrix-matrix multiply	?hemm	Hemm	√	√	√	sdcz
Hermitian rank-k update	?herk	Herk	√	√	√	sdcz
Hermitian rank-2k update	?her2k	Her2k	√	√	√	sdcz
Symmetric matrix-matrix multiply	?symm	Symm	√	√	√	sdcz
Symmetric rank-k update	?syrk	Syrk	√	√	√	sdcz
Symmetric rank-2k update	?syr2k	Syr2k	√	√	√	sdcz
Triangular matrix multiply	?trmm	Trmm	√	√	√	sdcz
Triangular solve with multiple right-hand sides	?trsm	Trsm	√	√	√	sdcz
LAPACK-level						
Cholesky factorization	?potrf	Chol	√	√	√	sdcz
LU factorization without pivoting	N/A	LU_nopiv	√	√	√	sdcz
LU factorization with partial pivoting	?getrf	LU_piv	√	√	√	sdcz
LU factorization with incremental pivoting	N/A	LU_incpiv		√		sdcz
QR factorization (via UT Householder transforms)	?geqrf	QR_UT	√			sdcz
QR factorization (via incremental UT Householder trans.)	N/A	QR_UT_inc		√		sdcz
LQ factorization (via UT Householder transforms)	?gelqf	LQ_UT	√			sdcz
Up-and-Downdate Cholesky/QR factor	N/A	UDdate_UT	√			sdcz
Up-and-Downdate Cholesky/QR factor (via incremental UT Householder-like transforms)	N/A	UDdate_UT_inc		√		sdcz
Triangular matrix inversion	?trtri	Trinv	√	√	√	sdcz
Triangular transpose matrix multiply	?lauum	Ttmm	√	√	√	sdcz
Symmetric/Hermitian positive definite inversion	?potri	SPDinv	√	√	√	sdcz
Triangular Sylvester equation solve	?trsyl	Sylv	√	√	√	sdcz
Reduction from a symmetric/Hermitian definite generalized eigenproblem to standard form	[sc]sygst [cz]hegst	Eig_gest	√	√	√	sdcz
Reduction to upper Hessenberg form	?gehrd	Hess_UT	√			sdcz
Reduction to tridiagonal form	[sd]sytrd [cz]hetrd	Tridiag_UT	√			sdcz
Reduction to bidiagonal form	?gebrd	Bidiag_UT	√			sdcz

Table 7.1: Dense linear algebra operations supported by `libflame`, which has full support for all four of the floating point data types: single and double precision real, single and double precision complex; `?` expands to one of {`sdcz`}.

library was the only dense linear algebra software product with multi-GPU capabilities and a wide functionality. It implements a large subset of the LAPACK functionality and a major part of the techinques illustrated in Chapter 5. A detailed description of the functionalities of the `libflame` library can be found in Table 7.1.

The interest on this type of run-time systems has been translated into collaborations and awards from well-known companies:

Microsoft Research showed their interest in the three main research lines presented in this thesis (developments for single-GPU, multi-GPU and clusters of GPUs). As part of the agreement with the company, codes for BLAS on one and multiple GPUs, LAPACK-level approaches on one and multiple GPUs, and similar routines for distributed memory approaches will be integrated in `libflame` with the support of Microsoft. A commercial license of the library has been acquired by Microsoft as part of the agreement.

Nvidia granted the HPC&A group with the Nvidia *Professor Partnership Award 2008* for its work on multi-GPU systems. At the same time, many of the graphics hardware used for performance evaluation in this document have been generously donated by the company in the framework of this collaboration.

PetaPath, manufacturers of ClearSpeed boards [50] signed an agreement with the HPC&A group to demonstrate the adaptability of the multi-GPU system developed to heterogeneous systems with other type of hardware accelerators. A prototype of the GPUSs runtime system was also developed and tested on this type of platforms.

7.4. Open research lines

GPU Computing is a relatively novel discipline, and thus many research lines remain open after the conclusion of this thesis. Some of them can be adapted from existing ideas from other arenas; others are new; it is likely that the last group of ideas will evolve with graphic architectures and programming paradigms.

The following list details some of the open research lines related to this thesis:

- The NVIDIA CUBLAS versions used for the evaluation and development of the ideas in this thesis do not support the overlapping of calculations on the GPU and data transfers. With the advent of newer versions that support this feature, the introduction of overlapping techniques on both single-GPU, multi-GPU and clusters of GPUs will open a new research line in order to hide the bus latency. More specifically, the runtime-based approach for systems with multiple GPUs will require a full redesign in order to deal and exploit this overlapping capabilities.

- Although the port of PLAPACK to clusters of GPUs combines a better programmability for message-passing architectures and remarkable performance, will soon be replaced by Elemental [113]. Fortunately, many of the design decisions in the Elemental framework are similar to those adopted in the early development of PLAPACK. A port of the Elemental framework is also in mind to adapt it to clusters of GPUs.

- In the adaptation of message-passing libraries, inter-node parallelism and data transfer reduction between processes is accomplished by an appropriate algorithm choose. In the case described in this thesis, one process per GPU is spawn. However, when more than one GPU per node is available, data transfers between memory spaces can be redundant using this approach. An alternative approach would be to employ one process per node, relying the management of multiple GPUs inside the node to a run-time system as that described in Chapter 5.

- The improvements described in Chapter 5 pursue the goal of data transfer reduction, without taking into account the scheduling policies to accomplish it. An alternative, but compatible approach is based in the modification of scheduling policies in order to assign tasks to the most affine computing resource, using a technique usually referred as *cache affinity* [44]. This techniques have already been implemented in the public release of `libflame`, but further research is still in the roadmap.

- While GPUs offer a near-optimal GFLOPS/price ratio, the main disadvantage of this hardware is power consumption. Energy-aware GPU computing is a field to be explored in the near future. Run-time systems provide a powerful tool to monitor and manage the configuration of the different computing units (in this case GPUs) according to their execution status, or the ability to redesign the scheduling policies in order to take into account the power consumption issue.

- Other improvements and research lines will be ultimately dictated by the technological evolution of graphics hardware. To name three possible improvement scenarios related to each one of the parts of this thesis:

    - Simultaneous execution of kernels can boost performance of block-oriented BLAS-3 algorithms by executing, if possible, the operations on the blocks simultaneously on a single-GPU system. This feature is already included in the latest NVIDIA GPUs.

- An scenario where direct GPU-GPU communication on multi-GPU systems is possible in the near future. In this case, provided the PCIExpress bus would disappear as the main bottleneck in the system, other strategies can be considered to improve performance. More intelligent scheduling policies, in which tasks are mapped to the most *affine* accelerator (considering which data are necessary for the execution of the task and where those data are located) have already been investigated in the framework of the `libflame` development [42].
- Future technological improvements include direct communication between GPU memories via interconnection networks (namely Infiniband). Adapting those new technologies to our developments would yield higher performance at no cost from the programmability level.
- Current hardware trends include the integration of the GPU as an on-chip co-processor to the general-purpose unit. NVIDIA has recently revealed the integration of ARM processors and graphics processors, and AMD has developed similar products in the framework of the FUSION project. If this novel architectures are sucessful, many of the techniques and methodologies proposed in this thesis are likely to need further adaptation to them. However, we believe that many of the ideas and techniques investigated would have a relevant impact on the performance of dense linear algebra implementations on these novel architectures without dramatic conceptual modifications.

APPENDIX A

FLAME algorithms for the BLAS-3 routines

The following algorithms correspond to the algorithmic variants of the routines SYMM (Figure A.1), SYR2K (Figure A.2), TRMM (Figure A.3), and TRSM (Figure A.4) developed and evaluated in Chapter 3. Algorithms are described using the FLAME notation.

Algorithm: SYMM_PP_PM(C, A, B)

Partition $C \rightarrow \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right)$, $B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right)$,
$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
where C_T, B_T have 0 rows, A_{TL} is 0×0
while $m(C_T) < m(C)$ **do**
Determine block size b
Repartition

$$\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline C_2 \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right),$$

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where C_1, B_1 have b rows, A_{11} is $b \times b$

Symm_pp	
$C_0 := C_0 + A_{10}^T B_1$	(GEMM)
$C_1 := C_1 + A_{11} B_1$	(SYMM)
$C_2 := C_2 + A_{21} B_1$	(GEMM)
Symm_pm	
$C_1 := C_1 + A_{10} B_0$	(GEMM)
$C_1 := C_1 + A_{11} B_1$	(SYMM)
$C_1 := C_1 + A_{21}^T B_2$	(GEMM)

Continue with

$$\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) \leftarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline C_2 \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right),$$

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

Algorithm: SYMM_MP(C, A, B)

Partition $C \rightarrow \left(\begin{array}{c|c} C_L & C_R \end{array}\right)$,
$B \rightarrow \left(\begin{array}{c|c} B_L & B_R \end{array}\right)$
where C_L has 0 columns,
B_L has 0 columns
while $n(C_L) < n(C)$ **do**
Determine block size b
Repartition

$$\left(\begin{array}{c|c} C_L & C_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array}\right),$$
$$\left(\begin{array}{c|c} B_L & B_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array}\right)$$

where C_1 has b columns,
B_1 has b columns

$C_1 := C_1 + AB_1$ (SYMM)

Continue with

$$\left(\begin{array}{c|c} C_L & C_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_0 & C_1 & C_2 \end{array}\right),$$
$$\left(\begin{array}{c|c} B_L & B_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array}\right)$$

endwhile

Figure A.1: Algorithms for SYMM.

Algorithm: SYR2K_MP(A, B, C)

Partition $A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)$, $B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right)$,

$C \rightarrow \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right)$

where A_T has 0 rows, B_T has 0 rows, C_{TL} is 0×0

while $m(A_T) < m(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right),$$

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right)$$

where A_1 has b rows, B_1 has b rows, C_{11} is $b \times b$

Syr2k_mp

$C_{01} := C_{01} + A_0 B_1^T$	(GEMM)
$C_{01} := C_{01} + B_0 A_1^T$	(GEMM)
$C_{11} := C_{11} + A_1 B_1^T + B_1 A_1^T$	(SYR2K)

Syr2k_pm

$C_{12} := C_{12} + A_1 B_2^T$	(GEMM)
$C_{12} := C_{12} + B_1 A_2^T$	(GEMM)
$C_{11} := C_{11} + A_1 B_1^T + B_1 A_1^T$	(SYR2K)

Continue with

$$\left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right),$$

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right)$$

endwhile

Algorithm: SYR2K_PP(A, B)

Partition $A \rightarrow \left(\begin{array}{c|c} A_L & A_R \end{array}\right)$,

$B \rightarrow \left(\begin{array}{c|c} B_L & B_R \end{array}\right)$

where A_L has 0 columns, B_L has 0 columns

while $n(A_L) < n(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_L & A_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_0 & A_1 & A_2 \end{array}\right),$$
$$\left(\begin{array}{c|c} B_L & B_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array}\right)$$

where A_1 has b columns, B_1 has b columns

$C := C + A_1 B_1^T + B_1 A_1^T$ (SYR2K)

Continue with

$$\left(\begin{array}{c|c} A_L & A_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_0 & A_1 & A_2 \end{array}\right),$$
$$\left(\begin{array}{c|c} B_L & B_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array}\right)$$

endwhile

Figure A.2: Algorithms for SYR2K.

Algorithm: TRMM_PP_PM(A, B)

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right)$

where A_{TL} is 0×0, B_T has 0 rows

while $m(A_{TL}) < m(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right),$$

$$\left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right)$$

where A_{11} is $b \times b$, B_1 has b rows

Trmm_pp

$B_0 := B_0 + A_{01}B_1$ (GEMM)

$B_1 := A_{11}B_1$ (TRMM)

Trmm_pm

$B_1 := A_{11}B_1$ (TRMM)

$B_1 := B_1 + A_{12}B_2$ (GEMM)

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right),$$

$$\left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right)$$

endwhile

Algorithm: TRMM_MP(A, B)

Partition $B \rightarrow \left(\begin{array}{c|c} B_L & B_R \end{array}\right)$

where B_L has 0 columns

while $n(B_L) < n(B)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} B_L & B_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array}\right)$$

where B_1 has b columns

$B_1 := AB_1$ (TRMM)

Continue with

$$\left(\begin{array}{c|c} B_L & B_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array}\right)$$

endwhile

Figure A.3: Algorithms for TRMM.

Algorithm: TRSM_PP_PM(A, B)

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$,
$B \rightarrow \left(\begin{array}{c|c} B_L & B_R \end{array}\right)$
where A_{TL} is 0×0,
B_L has 0 columns
while $m(A_{TL}) < m(A)$ **do**
Determine block size b
Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right),$$
$$\left(\begin{array}{c|c} B_L & B_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array}\right)$$
where A_{11} is $b \times b$,
B_1 has b columns

Trsm_pp

$B_1 := B_1 - B_0 A_{10}^T$	(GEMM)
$B_1 := B_1 A_{11}^{-T}$	(TRSM)

Trsm_pm

$B_1 := B_1 A_{11}^{-T}$	(TRSM)
$B_2 := B_2 - B_1 A_{21}^T$	(GEMM)

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right),$$
$$\left(\begin{array}{c|c} B_L & B_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array}\right)$$
endwhile

Algorithm: TRSM_MP(A, B)

Partition $B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right)$
where B_T has 0 rows
while $m(B_T) < m(B)$ **do**
Determine block size b
Repartition

$$\left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right)$$
where B_1 has b rows

$B_1 := B_1 A^{-T}$ (TRSM)

Continue with

$$\left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right)$$
endwhile

Figure A.4: Algorithms for TRSM.

Bibliography

[1] *NVIDIA CUDA Architecture. Introduction and Overview.* NVIDIA Corporation, 2009.

[2] *NVIDIA CUDA C Programming Best Practices Guide.* NVIDIA Corporation, 2009.

[3] *NVIDIA's Next Generation CUDA Compute Architecture: Fermi Whitepaper.* NVIDIA Corporation, 2010.

[4] Advanced Micro Devices Inc. http://www.amd.com/acml.

[5] Agarwal, R. C., and Gustavson, F. G. Vector and parallel algorithms for cholesky factorization on ibm 3090. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 1989), ACM Press, pp. 225–233.

[6] Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., and Tomov, S. Numerical linear algebra on emerging architectures: The plasma and magma projects. *Journal of Physics: Conference Series Vol. 180*.

[7] Alpatov, P., Baker, G., Edwards, C., Gunnels, J., Morrow, G., Overfelt, J., van de Geijn, R., and Wu, Y.-J. J. PLAPACK: Parallel linear algebra package – design overview. In *Proceedings of SC97* (1997).

[8] Andersen, B. S., Gustavson, F. G., and Wasniewski, J. A recursive formalation of Cholesky factorization of a matrix in packed storage. LAPACK Working Note 146 CS-00-441, University of Tennessee, Knoxville, May 2000.

[9] Anderson, E., Bai, Z., Demmel, J., Dongarra, J. E., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A. E., Ostrouchov, S., and Sorensen, D. *LAPACK Users' Guide.* SIAM, Philadelphia, 1992.

[10] Anderson, E., Benzoni, A., Dongarra, J., Moulton, S., Ostrouchov, S., Tourancheau, B., and van de Geijn, R. Basic linear algebra communication subprograms. In *Sixth Distributed Memory Computing Conference Proceedings* (1991), IEEE Computer Society Press, pp. 287–290.

[11] Anderson, E., Benzoni, A., Dongarra, J., Moulton, S., Ostrouchov, S., Tourancheau, B., and van de Geijn, R. Lapack for distributed memory architectures:

Progress report. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing* (Philadelphia, 1992), SIAM, pp. 625–630.

[12] Anderson, E., Dongarra, J., and Ostrouchov, S. Lapack working note 41: Installation guide for lapack. Tech. rep., Knoxville, TN, USA, 1992.

[13] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International Euro-Par Conference* (Delft, The Netherlands, Aug. 2009), vol. 5704 of *Lecture Notes in Computer Science*, Springer, pp. 863–874.

[14] Ayguade, E., Badia, R., Bellens, P., Cabrera, D., Duran, A., Ferrer, R., Gonzalez, M., Igual, F., Jimenez-Gonzalez, D., Labarta, J., Martinell, L., Martorell, X., Mayo, R., Perez, J., Planas, J., and Quintana-Orti, E. Extending OpenMP to survive the heterogeneous multi-core era. *International Journal of Parallel Programming 38* (2010), 440–459. 10.1007/s10766-010-0135-4.

[15] Ayguade, E., Badia, R. M., Cabrera, D., Duran, A., Gonzalez, M., Igual, F., Jimenez, D., Labarta, J., Martorell, X., Mayo, R., Perez, J. M., and Quintana-Ortí, E. S. A proposal to extend the OpenMP tasking model for heterogeneous architectures. In *IWOMP '09: Proceedings of the 5th International Workshop on OpenMP* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 154–167.

[16] Ayguadé, E., Badia, R. M., Igual, F. D., Labarta, J., Mayo, R., and Quintana-Ortí, E. S. An extension of the StarSs programming model for platforms with multiple GPUs. In *Euro-Par* (2009), pp. 851–862.

[17] Badía, J. M., Movilla, J. L., Climente, J. I., Castillo, M., Marqués, M., Mayo, R., Quintana-Ortí, E. S., and Planelles, J. Large-scale linear system solver using secondary storage: Self-energy in hybrid nanostructures. *Computer Physics Communications 182*, 2 (2011), 533–539.

[18] Baker, G., Gunnels, J., Morrow, G., Riviere, B., and van de Geijn, R. PLAPACK: High performance through high level abstraction. In *Proceedings of ICCP98* (1998).

[19] Bargen, B., and Donnelly, T. P. *Inside DirectX*. Microsoft Press, 1998.

[20] Barrachina, S., Castillo, M., Igual, F. D., Mayo, R., and Quintana-Ortí, E. S. Evaluation and tuning of the level 3 CUBLAS for graphics processors. In *Proceedings of the 10th IEEE Workshop on Parallel and Distributed Scientific and Engineering Computing, PDSEC 2008* (2008), pp. CD–ROM.

[21] Barrachina, S., Castillo, M., Igual, F. D., Mayo, R., and Quintana-Ortí, E. S. FLAG@lab: An M-script API for linear algebra operations on graphics processors. In *Proceedings of PARA 2008* (2008), LNCS, Springer-Verlag Berlin Heidelberg. To appear.

[22] Barrachina, S., Castillo, M., Igual, F. D., Mayo, R., and Quintana-Ortí, E. S. Solving dense linear systems on graphics processors. In *Proceedings of the 14th International Euro-Par Conference* (2008), E. Luque, T. Margalef, and D. Benítez, Eds., Lecture Notes in Computer Science, 5168, Springer, pp. 739–748.

[23] Barrachina, S., Castillo, M., Igual, F. D., Mayo, R., Quintana-Ortí, E. S., and Quintana-Ortí, G. Exploiting the capabilities of modern gpus for dense matrix computations. *Concurrency and Computation: Practice and Experience 21*, 18 (2009), 2457–2477.

[24] Bellens, P., Pérez, J. M., Badía, R. M., and Labarta, J. CellSs: a programming model for the Cell BE architecture. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing – SC2006* (New York, NY, USA, 2006), ACM Press, p. 86.

[25] Bellens, P., Perez, J. M., Cabarcas, F., Ramirez, A., Badia, R. M., and Labarta, J. Cellss: Scheduling techniques to better exploit memory hierarchy. *Sci. Program. 17*, 1-2 (2009), 77–95.

[26] Bientinesi, P. *Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms.* PhD thesis, Department of Computer Sciences, University of Texas at Austin, 2006.

[27] Bientinesi, P., Dhillon, I. S., and van de Geijn, R. A parallel eigensolver for dense symmetric matrices based on multiple relatively robust representations. *SIAM J. Sci. Comput. 27*, 1 (2005), 43–66.

[28] Bientinesi, P., Gunnels, J. A., Gustavson, F. G., Henry, G. M., Myers, M. E., Quintana-Ortí, E. S., and van de Geijn, R. A. The science of programming high-performance linear algebra libraries. In *Proceedings of Performance Optimization for High-Level Languages and Libraries (POHLL-02) , a workshop in conjunction with the 16th Annual ACM International Conference on Supercomputing (ICS'02)* (2002).

[29] Bientinesi, P., Gunnels, J. A., Myers, M. E., Quintana-Ortí, E. S., and van de Geijn, R. A. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software 31*, 1 (Mar. 2005), 1–26.

[30] Bientinesi, P., Igual, F. D., Kressner, D., and Quintana-Ortí, E. S. Reduction to condensed forms for symmetric eigenvalue problems on multi-core architectures. In *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I* (Berlin, Heidelberg, 2010), PPAM'09, Springer-Verlag, pp. 387–395.

[31] Bischof, C., and Loan, C. V. The WY representation for products of Householder matrices. *SIAM Journal on Scientific and Statistical Computing 8*, 1 (1987), s2–s13.

[32] Bischof, C. H., Lang, B., and Sun, X. Algorithm 807: The SBR Toolbox—software for successive band reduction. *ACM Trans. Math. Softw. 26*, 4 (2000), 602–616.

[33] Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C. *ScaLAPACK Users' Guide.* SIAM, 1997.

[34] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: An efficient multithreaded runtime system. In *Journal of parallel and distributed computing* (1995), pp. 207–216.

[35] Blumofe, R. D., and Leiserson, C. E. Scheduling multithreaded computations by work stealing. *J. ACM 46*, 5 (1999), 720–748.

[36] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, H., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H., Luszczek, P., YarKhan, A., and Dongarra, J. Distributed-memory task execution and dependence tracking within dague and the dplasma project. Tech. rep., Innovative Computing Laboratory, University of Tennessee.

[37] Brewer, O., Dongarra, J., and Sorensen, D. Tools to aid in the analysis of memory access patterns for FORTRAN programs. LAPACK Working Note 6, Technical Report MCS-TM-120, June 1988.

[38] Buttari, A., Dongarra, J., Langou, J., Langou, J., Luszczek, P., and Kurzak, J. Mixed precision iterative refinement techniques for the solution of dense linear systems. *Int. J. High Perform. Comput. Appl. 21*, 4 (2007), 457–466.

[39] Castillo, M., Chan, E., Igual, F. D., Mayo, R., Quintana-Orti, E. S., Quintana-Orti, G., van de Geijn, R., and Zee, F. G. V. Making programming synonymous with programming for linear algebra libraries. Tech. rep., The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-08-20, 2008.

[40] Castillo, M., Igual, F. D., Marqués, M., Mayo, R., Quintana-Ortí, E. S., Quintana-Ortí, G., Rubio, R., and van de Geijn, R. A. Out-of-core solution of linear systems on graphics processors. *International Journal of Parallel, Emergent and Distributed Systems 24*, 6 (2009), 521–538.

[41] Chan, E. *Application of Dependence Analysis and Runtime Data Flow Graph Scheduling to Matrix Computations*. PhD thesis, The University of Texas at Austin, 2010.

[42] Chan, E., and Igual, F. D. Runtime data graph scheduling of matrix computations with multiple hardware accelerators. FLAME Working Note #50. Technical Report TR-10-36, The University of Texas at Austin, Department of Computer Sciences, Oct. 2010.

[43] Chan, E., Quintana-Ortí, E. S., Quintana-Ortí, G., and van de Geijn, R. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *SPAA '07: Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures* (San Diego, CA, USA, June 9-11 2007), ACM, pp. 116–125.

[44] Chan, E., van de Geijn, R., and Chapman, A. Managing the complexity of lookahead for lu factorization with pivoting. In *SPAA '10: Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures* (New York, NY, USA, 2010), ACM, pp. 200–208.

[45] Chan, E., Van Zee, F. G., Bientinesi, P., Quintana-Ortí, E. S., Quintana-Ortí, G., and van de Geijn, R. SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. In *ACM SIGPLAN 2008 symposium on Principles and Practices of Parallel Programming – PPoPP 2008* (2008), pp. 123–132.

[46] Chan, E., Van Zee, F. G., Quintana-Ortí, E. S., Quintana-Ortí, G., and van de Geijn, R. Satisfying your dependencies with SuperMatrix. In *Proceedings of IEEE Cluster Computing 2007* (September 2007), pp. 91–99.

[47] Chapman, B., Jost, G., and Pas, R. v. d. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[48] Choi, J., Dongarra, J., Ostrouchov, S., Petitet, A., Walker, D. W., and Whaley, R. C. A proposal for a set of parallel basic linear algebra subprograms. In *PARA '95: Proceedings of the Second International Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science* (London, UK, 1996), Springer-Verlag, pp. 107–114.

[49] Choi, J., Dongarra, J. J., Pozo, R., and Walker, D. W. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation* (1992), IEEE Comput. Soc. Press, pp. 120–127.

[50] ClearSpeed. http://www.clearspeed.com.

[51] Demmel, J., Dongarra, J. J., Du Croz, J., Greenbaum, A., Hammarling, S., and Sorensen, D. Prospectus for the development of a linear algebra library for high-performance computers. Tech. Rep. MCS-TM-97, Sept. 1987.

[52] Dhillon, S., Parlett, N., and Vomel. The design and implementation of the MRRR algorithm. *ACM Trans. Math. Softw. 32*, 4 (2006), 533–560.

[53] Dongarra, J., Hammarling, S. J., and Sorensen, D. C. Block reduction of matrices to condensed forms for eigenvalue computations. LAPACK Working Note 2, Technical Report MCS-TM-99, Sept. 1987.

[54] Dongarra, J., van de Geijn, R., and Walker, D. Scalability issues affecting the design of a dense linear algebra library.

[55] Dongarra, J., and Walker, D. Lapack working note 58: "the design of linear algebra libraries for high performance computers. Tech. rep., Knoxville, TN, USA, 1993.

[56] Dongarra, J. J., Bunch, J. R., Moler, C. B., and Stewart, G. W. *LINPACK Users' Guide*. SIAM, Philadelphia, 1979.

[57] Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw. 16*, 1 (Mar. 1990), 1–17.

[58] Dongarra, J. J., van de Geijn, R. A., and Whaley, R. C. Two dimensional basic linear algebra communication subprograms. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing* (Mar. 1993).

[59] Du Croz, J., Mayes, P., and Radicati, G. Factorization of band matrices using level 3 BLAS. LAPACK Working Note 21, Technical Report CS-90-109, July 1990.

[60] Duato, J., Igual, F. D., Mayo, R., Peña, A. J., Quintana-Ortí, E. S., and Silla, F. An efficient implementation of gpu virtualization in high performance clusters. In *Euro-Par Workshops* (2009), pp. 385–394.

[61] Elmroth, E., Gustavson, F., Jonsson, I., and Kagstrom, B. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review 46*, 1 (2004), 3–45.

[62] ESSL, I. http://www-03.ibm.com/systems/p/software/essl/index.html.

[63] Fatahalian, K., Sugerman, J., and Hanrahan, P. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2004), ACM, pp. 133–137.

[64] Fernando, R., and Kilgard, M. J. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003.

[65] Fogue, M., Igual, F. D., Quintana-Orti, E. S., and van de Geijn, R. A. Retargeting PLAPACK to clusters with hardware accelerators. In *International Conference on High Performance Computing and Simulation (HPCS 2010)* (2010), pp. 444 –451.

[66] Galoppo, N., Govindaraju, N. K., Henson, M., and Manocha, D. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing – SC2005* (Washington, DC, USA, 2005), IEEE Computer Society, p. 3.

[67] Girkar, M., and Polychronopoulos, C. D. Extracting task-level parallelism. *ACM Trans. Program. Lang. Syst. 17*, 4 (1995), 600–634.

[68] Göddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P. S., Buijssen, S. H., Grajewski, M., and Turek, S. Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. *Parallel Computing 33*, 10–11 (Sept. 2007), 685–699.

[69] Göddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P. S., Wobker, H., Becker, C., and Turek, S. Using GPUs to improve multigrid solver performance on a cluster. *International Journal of Computational Science and Engineering 4*, 1 (Nov. 2008), 36–55.

[70] Golub, G. H., and Loan, C. F. V. *Matrix Computations*, 3rd ed. The Johns Hopkins University Press, Baltimore, 1996.

[71] Goto, K. `http://www.tacc.utexas.edu/resources/software`.

[72] Goto, K., and Geijn, R. A. v. d. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw. 34*, 3 (2008), 1–25.

[73] Grama, A., Karypis, G., Kumar, V., and Gupta, A. *Introduction to Parallel Computing (2nd Edition)*, 2 ed. Addison Wesley, January 2003.

[74] Gropp, W., Lusk, E., and Skjellum, A. *Using MPI*. The MIT Press, 1994.

[75] Gunnels, J. A., Gustavson, F. G., Henry, G. M., and van de Geijn, R. A. Flame: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software 27*, 4 (Dec. 2001), 422–455.

[76] Gunnels, J. A., Henry, G. M., and van de Geijn, R. A. Formal Linear Algebra Methods Environment (FLAME): Overview. FLAME Working Note #1 CS-TR-00-28, Department of Computer Sciences, The University of Texas at Austin, Nov. 2000.

[77] Gustavson, F. G., Karlsson, L., and Kagstrom, B. Three algorithms for Cholesky factorization on distributed memory using packed storage. In *Applied Parallel Computing. State of the Art in Scientific Computing*. Springer Berlin / Heidelberg, 2007, pp. 550–559.

[78] Hendrickson, B. A., and Womble, D. E. The torus-wrap mapping for dense matrix calculations on massively parallel computers. 1201–1226.

[79] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Pub., San Francisco, 2003.

[80] Hennessy, J. L., and Patterson, D. A. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[81] Howard, J., Dighe, S., Hoskote, Y., Vangal, S., Finan, D., Ruhl, G., Jenkins, D., Wilson, H., Borkar, N., Schrom, G., Pailet, F., Jain, S., Jacob, T., Yada, S., Marella, S., Salihundam, P., Erraguntla, V., Konow, M., Riepen, M., Droege, G., Lindemann, J., Gries, M., Apel, T., Henriss, K., Lund-Larsen, T., Steibl, S., Borkar, S., De, V., Van Der Wijngaart, R., and Mattson, T. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International* (feb. 2010), 108 –109.

[82] Igual, F. D., Mayo, R., and Quintana-Ortí, E. S. Attaining high performance in general-purpose computations on current graphics processors. *High Performance Computing for Computational Science - VECPAR 2008: 8th International Conference, Toulouse, France, June 24-27, 2008. Revised Selected Papers* (2008), 406–419.

[83] Igual, F. D., Quintana-Ortí, G., and van de Geijn, R. Level-3 BLAS on a GPU: Picking the low hanging fruit. In *ICCMSE 2010: Proceedings of the Eighth International Conference of Computational Methods in Sciences and Engineering* (2011), AIP Conference Proceedings. (To appear. Also published as FLAME Working Note 37).

[84] Ino, F., Matsui, M., Goda, K., and Hagihara, K. Performance study of lu decomposition on the programmable gpu. In *High Performance Computing - HiPC 2005*, D. Bader, M. Parashar, V. Sridhar, and V. Prasanna, Eds., vol. 3769 of *Lecture Notes in Computer Science*. Springer Berlin. Heidelberg, 2005, pp. 83–94.

[85] Intel MKL. http://software.intel.com/en-us/intel-mkl/.

[86] Jiang, C., and Snir, M. Automatic tuning matrix multiplication performance on graphics hardware. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 185–196.

[87] Jung, J., and O'Leary, D. Cholesky decomposition and linear programming on a gpu. *Scholarly Paper University of Maryland* (2006), 1–21.

[88] Kågström, B., Ling, P., and Loan, C. V. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Softw. 24*, 3 (1998), 268–302.

[89] Kirk, D. B., and Hwu, W.-m. W. *Programming Massively Parallel Processors: A Hands-on Approach*, 1 ed. Morgan Kaufmann, February 2010.

[90] Krüger, J., and Westermann, R. Linear algebra operators for gpu implementation of numerical algorithms. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers* (New York, NY, USA, 2003), ACM, pp. 908–916.

[91] Kurzak, J., and Dongarra, J. Implementation of mixed precision in solving systems of linear equations on the cell processor: Research articles. *Concurr. Comput. : Pract. Exper. 19*, 10 (2007), 1371–1385.

[92] Lang, B. Efficient eigenvalue and singular value computations on shared memory machines. *Parallel Comput. 25*, 7 (1999), 845–860.

[93] Langou, J., Langou, J., Luszczek, P., Kurzak, J., Buttari, A., and Dongarra, J. Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 2006), ACM, p. 113.

[94] Larsen, E. S., and McAllister, D. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)* (New York, NY, USA, 2001), Supercomputing '01, ACM, pp. 55–55.

[95] Low, T. M., and van de Geijn, R. An API for manipulating matrices stored by blocks. Tech. Rep. TR-2004-15, Department of Computer Sciences, The University of Texas at Austin, May 2004.

[96] Ltaief, H., Tomov, S., Nath, R., Du, P., and Dongarra, J. A scalable high performant cholesky factorization for multicore with gpu accelerators. *Proc. of VECPAR'10 (to appear)*.

[97] Luna, F. D. *Introduction to 3D Game Programming with DirectX 10*. Jones and Bartlett Publishers, 2008.

[98] Marker, B. A., Van Zee, F. G., Goto, K., Quintana-Ortí, G., and van de Geijn, R. A. Toward scalable matrix multiply on multithreaded architectures. In *European Conference on Parallel and Distributed Computing – Euro-Par 2007* (February 2007), pp. 748–757.

[99] Markoff, J. Writing the fastest code, by hand, for fun: A human computer keeps speeding up chips. *The New York Times* (2005).

[100] Marqués, M. *Una Aproximacion de Alto Nivel a la Resolucion de Problemas Matriciales con Almacenamiento en Disco*. PhD thesis, Universitat Jaume I, 2009.

[101] Marqués, M., Quintana-Ortí, G., Quintana-Ortí, E. S., and van de Geijn, R. A. Using graphics processors to accelerate the solution of out-of-core linear systems. In *ISPDC* (2009), pp. 169–176.

[102] Martin, R. M. *Electronic Structure: Basic Tehory and Practical Methods*. Cambridge University Press, Cambridge, UK, 2008.

[103] Mattson, T. G., Van der Wijngaart, R., and Frumkin, M. Programming the intel 80-core network-on-a-chip terascale processor. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (Piscataway, NJ, USA, 2008), IEEE Press, pp. 1–11.

[104] Moler, C. B. Iterative refinement in floating point. *J. ACM 14*, 2 (1967), 316–321.

[105] Moore, G. Cramming more components onto integrated circuits. *Proceedings of the IEEE 86*, 1 (Jan. 1998), 82–85.

[106] Netlib.org. http://www.netlib.org/blas.

[107] Netlib.org. http://www.netlib.org/eispack.

[108] Netlib.org. http://www.netlib.org/lapack.

[109] Perez, J. M., Badia, R. M., Labarta, J., Perez, J. M., Badia, R. M., and Labarta, J. Technical report 03/2007 a flexible and portable programming model for smp and multi-cores bsc-upc, 2007.

[110] Perez, J. P., Bellens, P., Badia, R. M., and Labarta, J. Cellss: making it easier to program the cell broadband engine processor. *IBM J. Res. Dev. 51*, 5 (2007), 593–604.

[111] Performance Library, S. http://developers.sun.com/sunstudio/.

[112] Petitet, A. P., and Dongarra, J. J. Algorithmic redistribution methods for block-cyclic decompositions. *IEEE Trans. Parallel Distrib. Syst. 10*, 12 (1999), 1201–1216.

[113] Poulson, J., Marker, B., and van de Geijn, R. Elemental: A new framework for distributed memory dense matrix computations. FLAME Working Note #44. Technical Report TR-10-20, The University of Texas at Austin, Department of Computer Sciences, June 2010.

[114] Quintana-Ortí, G., Igual, F. D., Quintana-Ortí, E. S., and van de Geijn, R. A. Solving dense linear systems on platforms with multiple hardware accelerators. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2009), ACM, pp. 121–130.

[115] Quintana-Ortí, G., Quintana-Ortí, E. S., Chan, E., van de Geijn, R., and Van Zee, F. G. Design and scheduling of an algorithm-by-blocks for LU factorization on multithreaded architectures. FLAME Working Note #26 TR-07-50, The University of Texas at Austin, Department of Computer Sciences, September 2007.

[116] Quintana-Ortí, G., Quintana-Ortí, E. S., Chan, E., van de Geijn, R., and Van Zee, F. G. Design of scalable dense linear algebra libraries for multithreaded architectures: the LU factorization. In *Workshop on Multithreaded Architectures and Applications – MTAAP 2008* (2008). CD-ROM.

[117] Quintana-Ortí, G., Quintana-Ortí, E. S., Chan, E., Van Zee, F. G., and van de Geijn, R. A. Scheduling of QR factorization algorithms on SMP and multi-core architectures. In *16th Euromicro International Conference on Parallel, Distributed and Network-based Processing – PDP 2008* (2008), F. S. D. El Baz, J. Bourgeois, Ed., pp. 301–310.

[118] Quintana-Ortí, G., Quintana-Ortí, E. S., Remón, A., and van de Geijn, R. Supermatrix for the factorization of band matrices. FLAME Working Note #27 TR-07-51, The University of Texas at Austin, Department of Computer Sciences, September 2007.

[119] Quintana-Ortí, G., Quintana-Ortí, E. S., Remon, A., and van de Geijn, R. A. An algorithm-by-blocks for supermatrix band cholesky factorization. In *Eigth International Meeting on High Performance Computing for Computational Science* (2008). submitted.

[120] Quintana-Ortí, G., Quintana-Ortí, E. S., van de Geijn, R., Zee, F. V., and Chan, E. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software*. To appear.

[121] Quintana-Ortí, G., Quintana-Ortí, E. S., van de Geijn, R. A., Van Zee, F. G., and Chan, E. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software 36*, 3 (July 2009), 14:1–14:26.

[122] Reinders, J. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[123] Rice, J. R. *Matrix Computations and Mathematical Software*. New York, NY, USA, 1981.

[124] Sebastien, S.-L., and Wolfgang, E. *The Complete Effect and HLSL Guide*. Paradoxal Press, 2005.

[125] Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W., and Dongarra, J. *MPI: The Complete Reference*. The MIT Press, 1996.

[126] SPARK. http://www.cs.utexas.edu/users/flame/Spark/.

[127] Strazdins, P. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. Tech. Rep. TR-CS-98-07, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia, 1998.

[128] Sunderam, V. S. Pvm: a framework for parallel distributed computing. *Concurrency: Pract. Exper. 2*, 4 (1990), 315–339.

[129] Susan L. Graham, Marc Snir, C. A. P., Ed. *Getting up to speed: the future of supercomputing*. The National Academies Press, 2004.

[130] Tomasulo, R. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. of Research and Development 11*, 1 (1967).

[131] Tomov, S., Dongarra, J., and Baboulin, M. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parallel Comput. 36* (June 2010), 232–240.

[132] Tomov, S., Nath, R., and Dongarra, J. Accelerating the reduction to upper hessenberg, tridiagonal, and bidiagonal forms through hybrid gpu-based computing. *Parallel Comput. 36* (December 2010), 645–654.

[133] Tomov, S., Nath, R., Ltaief, H., and Dongarra, J. Dense linear algebra solvers for multicore with gpu accelerators. *Proc. of IPDPS'10*.

[134] Top500.org. http://www.top500.org.

[135] van de Geijn, R., and Watts, J. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience 9*, 4 (April 1997), 255–274.

[136] van de Geijn, R. A. Representing linear algebra algorithms in code: The FLAME API. *ACM Trans. Math. Softw.*. submitted.

[137] van de Geijn, R. A. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.

[138] van de Geijn, R. A., and Quintana-Ortí, E. S. *The Science of Programming Matrix Computations*. www.lulu.com, 2008.

[139] Volkov, V., and Demmel, J. Lu, qr and cholesky factorizations using vector capabilities of gpus. Tech. rep.

[140] Volkov, V., and Demmel, J. Using gpus to accelerate the bisection algorithm for finding eigenvalues of symmetric tridiagonal matrices. Tech. Rep. UCB/EECS-2007-179, EECS Department, University of California, Berkeley, Dec 2007.

[141] Volkov, V., and Demmel, J. LU, QR and Cholesky factorizations using vector capabilities of GPUs. Tech. Rep. UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.

[142] Volkov, V., and Demmel, J. W. Benchmarking gpus to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (Piscataway, NJ, USA, 2008), IEEE Press, pp. 1–11.

[143] von Neumann, J. First draft of a report on the EDVAC. *IEEE Ann. Hist. Comput. 15*, 4 (1993), 27–75.

[144] Watkins, D. S. *Fundamentals of Matrix Computations*, 2nd ed. John Wiley and Sons, inc., New York, 2002.

[145] Whaley, R. C., and Dongarra, J. J. Automatically tuned linear algebra software. In *Proceedings of SC'98* (1998).

[146] Wu, Y.-J. J., Alpatov, P. A., Bischof, C., and van de Geijn, R. A. A parallel implementation of symmetric band reduction using PLAPACK. In *Proceedings of Scalable Parallel Library Conference, Mississippi State University* (1996). PRISM Working Note 35.

[147] Wulf, W. A., and McKee, S. A. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News 23*, 1 (1995), 20–24.

[148] Zafont, M. J., Martin, A., Igual, F., and Quintana-Orti, E. S. Fast development of dense linear algebra codes on graphics processors. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing. Workshop on High-Level Parallel Programming Models & Supportive Environments* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 1–8.

[149] Zee, F. G. V. `libflame`*: The Complete Reference*. www.lulu.com, 2009.

[150] Zhang, Y., and Sarkar, T. K. *Parallel solution of integral equation-based EM problems in the frequency domain; electronic version*. Wiley Series in Microwave and Optical Engineering. Wiley, Hoboken, NJ, 2009.

[151] Zhang, Y., Sarkar, T. K., van de Geijn, R. A., and Taylor, M. C. Parallel MoM using higher order basis function and PLAPACK in-core and out-of-core solvers for challenging EM simulations. In *IEEE AP-S & USNC/URSI Symposium* (2008).