Universidade do Minho
Escola de Engenharia

Pedro Filipe Araújo Costa

Efficient Computation of the Matrix Square Root in Heterogeneous Platforms

Setembro de 2013

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Pedro Filipe Araújo Costa

Efficient Computation of the Matrix Square Root in Heterogeneous Platforms

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de
Professor Alberto Proença
Professor Rui Ralha

Setembro de 2013

Anexo 3

DECLARAÇÃO

Nome

Pedro Filipe Araújo Costa

Endereço electrónico: pg19830@alunos.uminho.pt Telefone: 913 725 975 / 924 189 979

Número do Bilhete de Identidade: 13775276

Título dissertação

Efficient Computation of the Matrix Square Root in Heterogeneous Platforms

Orientador(es):

Prof. Alberto Proença

Prof. Rui Ralha Ano de conclusão: 2013

Designação do Mestrado ou do Ramo de Conhecimento do Doutoramento:

Mestrado em Engenharia Informática

É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE/TRABALHO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

Universidade do Minho, 01 / 09 / 2013

Assinatura: Pedro Filipe Araújo Costa

Agradecimentos

Apesar do foco de uma dissertação ser o seu conteúdo, o facto deste documento existir deve-se provavelmente mais àqueles que me rodeiam. Deixo aqui, portanto, os devidos agradecimentos, em palavras que dificilmente representam a magnitude das suas contribuições.

Ao meu orientador, o prof. Alberto Proença, pelo rigor e disciplina que me exigiu, pelos desafios e pelas oportunidades. Ao meu co-orientador, o prof. Rui Ralha, pelas oportunidades proporcionadas por esta dissertação e pela disponibilidade sempre que precisei. A ambos e ao prof. João Luís Sobral, por fazerem de Computação Paralela e Distribuída a UCE mais exigente e a mais recompensadora. Aproveito ainda para endereçar um obrigado aos profs. Luís Paulo Santos, José Bernardo Barros, António Luís Sousa, Mário Martins e Orlando Belo, pois se hoje escrevo isto em muito se deve ao impacto que tiveram na minha formação.

I would like to thank the Numerical Algorithms Group for the resources made available for this work. I would also like to emphasize a personal thank you to Edvin Deadman, for being available even during his holidays.

Aos colegas do LabCG, a minha segunda casa durante este ano, pelo ambiente proporcionado, sem o qual esta dissertação nunca teria progredido, e pela disponibilidade quando as ideias faltaram.

Ao professor, e acima de tudo meu amigo, Rui Mendes, pelas conversas e pelas oportunidades a representar a academia em torneios de programação. Aos "pequenitos" do DPUM, obrigado por quererem aprender comigo e pelas horas passadas a conversar e a explorar o mundo da programação.

Aos meus amigos, André, Fábio, Kura e Sampaio, pelos momentos que me permitiram esquecer o trabalho, e um pedido de desculpas pelas várias vezes em que tive de recusar planos para poder manter-me a trabalhar.

Aos meus "irmãos de orientador", André Pereira e Miguel Palhas, pela cumplicidade, pelo companheirismo e pelo apoio que me deram durante estes últimos dois anos. Espero ter conseguido retribuir.

Aos meus pais e à minha irmã, pelas intermináveis divagações que aturaram, mesmo sem perceberem metade. Um particular obrigado à minha mãe, por todos os sacrifícios que fez para me permitir chegar até aqui. Acrescento ainda um humilde obrigado à minha tia Conceição, por toda a ajuda que deu à minha família ao longo destes anos, nem imaginando a magnitude dos seus gestos. Vocês são a razão pela qual escrevo isto.

Last, but certainly not the least, o meu mais profundo obrigado à Catarina, a minha companheira ao longo destes anos de academia, pelo apoio nas horas de maior frustração, pelo carinho quando a motivação me faltou, pela infinita paciência sempre que precisei de desabafar, por todas as horas de que abdicou em prol do meu trabalho. Obrigado por todas as razões para continuar a lutar.

Work developed with the support of the Numerical Algorithms Group and funded by the Portuguese agency FCT, *Fundação para a Ciência e Tecnologia*, under the program UT Austin | Portugal.

nag Results Matter. Trust NAG.
FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA EDUCAÇÃO E CIÊNCIA
UT Austin | Portugal
INTERNATIONAL COLLABORATORY FOR EMERGING TECHNOLOGIES, COLAB

Abstract

Matrix algorithms often deal with large amounts of data at a time, which impairs efficient cache memory usage. Recent collaborative work between the Numerical Algorithms Group and the University of Minho led to a blocked approach to the matrix square root algorithm with significant efficiency improvements, particularly in a multicore shared memory environment.

Distributed memory architectures were left unexplored. In these systems data is distributed across multiple memory spaces, including those associated with specialized accelerator devices, such as GPUs. Systems with these devices are known as heterogeneous platforms.

This dissertation focuses on studying the blocked matrix square root algorithm, first in a multicore environment, and then in heterogeneous platforms. Two types of hardware accelerators are explored: Intel Xeon Phi coprocessors and NVIDIA CUDA-enabled GPUs.

The initial implementation confirmed the advantages of the blocked method and showed excellent scalability in a multicore environment. The same implementation was also used in the Intel Xeon Phi, but the obtained performance results lagged behind the expected behaviour and the CPU-only alternative. Several optimizations techniques were applied to the common implementation, which managed to reduce the gap between the two environments.

The implementation for CUDA-enabled devices followed a different programming model and was not able to benefit from any of the previous solutions. It also required the implementation of BLAS and LAPACK routines, since no existing package fits the requirements of this application. The measured performance also showed that the CPU-only implementation is still the fastest.

Resumo

Computação Eficiente da Raíz Quadrada de uma Matriz em Plataformas Heterogéneas

Algoritmos de matrizes lidam regularmente com grandes quantidades de dados ao mesmo tempo, o que dificulta uma utilização eficiente da cache. Um trabalho recente de colaboração entre o Numerical Algorithms Group e a Universidade do Minho levou a uma abordagem por blocos para o algoritmo da raíz quadrada de uma matriz com melhorias de eficiência significativas, particularmente num ambiente multicore de memória partilhada.

Arquiteturas de memória distribuída permaneceram inexploradas. Nestes sistemas os dados são distribuídos por diversos espaços de memória, incluindo aqueles associados a dispositivos aceleradores especializados, como GPUs. Sistemas com estes dispositivos são conhecidos como plataformas heterogéneas.

Esta dissertação foca-se em estudar o algoritmo da raíz quadrada de uma matriz por blocos, primeiro num ambiente multicore e depois usando plataformas heterogéneas. Dois tipos de aceleradores são explorados: co-processadores Intel Xeon Phi e GPUs NVIDIA habilitados para CUDA.

A implementação inicial confirmou as vantagens do método por blocos e mostrou uma escalabilidade excelente num ambiente multicore. A mesma implementação foi ainda usada para o Intel Xeon Phi, mas os resultados de performance obtidos ficaram aquém do comportamento esperado e da alternativa usando apenas CPUs. Várias otimizações foram aplicadas a esta implementação comum, conseguindo reduzir a diferença entre os dois ambientes.

A implementação para dispositivos CUDA seguiu um modelo de programação diferente e não pôde beneficiar the nenhuma das soluções anteriores. Também exigiu a implementação de rotinas BLAS e LAPACK, já que nenhum dos pacotes existentes se adequa aos requisitos desta implementação. A performance medida também mostrou que a alternativa usando apenas CPUs ainda é a mais rápida.

Contents

List of Figures

List of Algorithms

Glossary

AMD Advanced Micro Devices

AVX Advanced Vector eXtensions

BLAS Basic Linear Algebra Subprograms

CPU Central Processing Unit

CUDA formerly Compute Unified Device Architecture, a parallel computing platform for NVIDIA GPU

DMA Direct Memory Access

DRAM Dynamic Random-Access Memory

DSP Digital Signal Processor

EMU Extended Math Unit

FPGA Field Programmable Gate Array

GCC GNU Compiler Collection

GDDR Graphics Double Data Rate

GPC Graphics Processing Cluster

GPU Graphics Processing Unit

GPGPU General Purpose GPU

HetPlat Heterogeneous Platform

HPC High Performance Computing

icpc Intel C++ Compiler

ILP Instruction-Level Parallelism

ISA Instruction Set Architecture

LAPACK Linear Algebra PACKage

MAGMA Matrix Algebra on GPU and multicore Architectures

MKL Math Kernel Library

MIC Many Integrated Core

MPI Message Passing Interface

MPP Massive Parallel Processing

NAG Numerical Algorithm Group

NUMA Non-Uniform Memory Access

PCI Peripheral Component Interconnect

PCIe PCI Express

pthreads POSIX Threads

RLP Request-Level Parallelism

SFU Special Function Unit

SIMD Single Instruction Multiple Data

SIMT Single Instruction, Multiple Threads

SM Streaming Multiprocessor

SMP Symmetric Multiprocessing

SMX Kepler Streaming Multiprocessor. A redesign of the original SM.

SSE Streaming Single Instruction Multiple Data (SIMD) Extensions

TBB Threading Building Blocks

TLP Thread-Level Parallelism

UMA Uniform Memory Access

USB Unit Stride Blocks

VPU Vector Processing Unit

1 Introduction

For more than half a century, computational systems have evolved at an increasing rate, fueled by a similar demand in computing power. The invention of the digital transistor, in 1947, smaller and faster than its predecessor, made computers smaller and more power efficient. Integrated circuits further reduced the space and power requirements of computers, which in turn led to the emergence of the microprocessor. The evolution of the microprocessor in the following decades followed had two main contributors: the number of transistors and their clock frequency.

A processor's clock frequency is strongly correlated to the rate at which it can issue instructions, which means that a higher frequency is roughly translated into more instructions being processed each second. This meant that the same old applications got faster just with the evolution of the underlying hardware, without any programming effort. On the other hand, increasing the number of transistors in a chip allowed to create complex systems that optimized software execution, mainly with the exploration of Instruction-Level Parallelism (ILP) and the use of memory caches to hide the increasing memory latency times.

In 1965, Gordon Moore, in an attempt to predict the evolution of integrated circuits during the following decade, projected that the number of transistors would double every year [1]. In 1975, adding more recent data, he slowed the future rate of increase in complexity to what is still today known as Moore's Law: the number of transistors in a chip doubles every two years [2, 3].

In 2003, the evolution of the microprocessor reached a milestone. Until then, the increase in clock frequency in microprocessors had followed closely the number of transistors, but the power density in processors was approaching the physical limitations of silicon-based microelectronics with air cooling techniques. Continuing to increase the frequency would require new and prohibitively expensive cooling solutions. With Moore's Law still in effect, and the lack of more ILP to explore efficiently, the key Central Processing Unit (CPU) chip designers turned in a new direction, replacing old single heavy core CPUs with multiple simpler cores working in parallel and sharing common resources.

The advent of multicore architectures also shifted the software development trends, since sequential code is no longer able to efficiently use these multiple computing resources, available in modern systems. It also boosted the popularity of special purpose devices, like Graphics Processing Units (GPUs) and Digital Signal Processors (DSPs). Besides having particular features that might be suited for certain problems, using these devices to perform part of the computation allows the CPU to be focused on other tasks, or to be focused in only a part of the domain, thus increasing efficiency. This can be further extended by interconnecting several computational nodes, each with one or more CPUs and some specialized devices, able to cooperate over a fast network interface. Such systems are called Heterogeneous Platforms (HetPlats).

The Numerical Algorithm Group (NAG) [4] delivers a highly reliable commercial numerical library containing several specialized multicore functions for matrix operations. While the NAG library includes implementations of several algorithms for CUDA-enabled GPUs and the Intel Xeon Phi coprocessor in heterogeneous platforms, it has yet no matrix square root function optimized for these devices [5, 6].

Matrix Algebra on GPU and multicore Architectures (MAGMA) is a project that aims to develop a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures, starting with current CPU+GPU systems. At the moment, MAGMA already includes implementations for many of the most important algorithms in Matrix Algebra but not for the computation of the square root [7]. This feature is also not implemented in any of the major GPU accelerated libraries listed by NVIDIA [8, 9, 10, 11, 12].

In a previous work, Deadman et al. [13] expanded the method devised by Björck and Hammarling [14] to compute the square root of a matrix, implementing an equivalent blocked method in a multicore environment. While blocked approaches are able to make a more efficient use of the memory hierarchy, they are also very well suited for devices designed for vector processing, such as Graphics Processing Units (GPUs) and devices based on Intel Many Integrated Core (MIC) architecture.

1.1 Motivation and Goals

Being the square root of a matrix a common operation to compute in problems from several fields (e.g., Markov models of finance, the solution to differential equations, computation of the polar decomposition and the matrix sign function) [15], creating an optimized implementation would make possible for more complex problems to be studied [16].

Previous work on this algorithm mainly addressed its implementation in a CPU shared memory environment; heterogeneous distributed memory environments are still unexplored. Also, other linear algebra projects oriented at GPUs lack implementations for this algorithm. The resources made available in the recent hardware accelerators hold great potential to improve performance.

This dissertation aims to extend the implementation of the matrix square root algorithm to heterogeneous platforms in order to achieve a higher degree of efficiency. It is particularly interesting to study the performance of this algorithm using massively parallel hardware accelerators.

Throughout this dissertation, three implementations of the core process behind the matrix square root algorithm are proposed and studied. The first, targeted for a multicore environment provides a first approach to the algorithm, the typical naive implementation. It also allows to port the implementations described in previous work to a more familiar open source environment. The second implementation is meant to use the new Intel Xeon Phi coprocessor, testing the device that recently led the *Tianhe-2* supercomputer to the first position in the TOP500 ranking[1]. The third implementation is targeted for CUDA-enabled GPUs.

Each implementation is quantitatively evaluated. The multicore implementation provides a scalability test to help in the analysis the algorithm behaviour, while the other two provide an overview of the computation impact of the algorithm when executed in an hardware accelerator with disjoint memory addressing.

1.2 Document Organization

Chapters 2 and 3 provide the background information required to conveniently contextualize the reader. In particular, Chapter 2 provides an overview on the evolution of High Performance Computing (HPC), the hardware characteristics of heterogeneous platforms and the challenges faced by programmers in this area. It also covers some tools to aid with such issues.

The following chapters focus on the three mentioned implementations. Chapter 4 describes the multicore implementations and further contextualizes the reader with the case study. It also presents the scalability test results and the consequent analysis. Chapter 5 focuses on the implementation targeted for the Intel MIC architecture, the results obtained and optimizations for a better tuning. In a similar fashion, Chapter 6 does the same for the CUDA implementation.

Chapter 7 presents the conclusions of this dissertation and suggestions for future work, including further optimization opportunities and identified unexplored approaches.

---

[1] `http://www.top500.org/lists/2013/06/`

2 Technological Background

Parallelism is far from being a modern concept, much less in the field of HPC. Back in the 1970s, the raw performance of vector computers marked the beginning of modern Supercomputers [17]. These systems, able to process multiple items of data at the same time, prevailed in supercomputer design until the beginning of the 1990s, by which time the Massive Parallel Processing (MPP) architectures became the most popular. MPP systems had two or more identical processors connected to separate memory and controlled by separate (but identical) operating systems. Middle and low-end systems consisted in Symmetric Multiprocessing (SMP) architectures, containing two or more identical processors connected to a single shared main memory and controlled by a single operating system. In the following years cluster systems became the dominating design. In cluster computing multiple separate computers, each having a SMP architecture, are able to cooperate appearing as a single system to the user. This trend has continued up to the present [18, 19].

Hennessy and Patterson [20] define two classes of parallelism:

Data-Level Parallelism consists in many data items being computed at the same time;

Task-Level Parallelism refers to different tasks of work able to operate independently and largely in parallel.

The hardware can exploit these two kinds of parallelism at four different levels. Processors take advantage of ILP without any intervention from the programmer through mechanisms like superscalarity, out-of-order execution, pipelining and speculative execution. Vector processing on the other hand, uses a single instruction to operate over a collection of data in parallel, similar to what happened with vector computers but at a smaller scale. Thread-Level Parallelism (TLP) enables both data and task level parallelism by allowing more than one thread of instructions to be executed at the same time. Threads can also be used to hide memory latencies, by allowing another thread to use the physical resources while the memory request is not fulfilled. Lastly, Request-Level Parallelism (RLP), used in warehouse-scale computing, exploits parallelism among largely decoupled tasks specified by the programmer or the operating system.

For decades programmers did not have to worry with exploring parallelism in their applications but the multicore advent is forcing a deep change in software development. Legacy software, intrinsically sequential, is no longer able to profit from the evolution of computational hardware as new generations of microprocessors work at roughly the same clock frequency (sometimes even less), instead providing extra resources these applications are not prepared to take advantage of. This implies a re-design of old applications, otherwise they will plateau at or near current levels of performance, facing the risk of stagnation and loss of competitiveness, both for themselves and any projects that might depend on these legacy applications [21].

2.1 Heterogeneous Platforms

As the evolution of microprocessors moves towards higher levels of parallelism, several other options exist, from using multiple machines in a cluster, allowing each to perform part of the computation independently, to specific-purpose devices such as DSPs and GPUs.

A given system is said to be heterogeneous when it contains multiple devices of different kinds. Usually, each of these devices is capable of computing several operations in parallel. The most efficient computing systems of the world in the TOP500 list[1] are composed of several interconnected computing nodes, each with multiple multicore CPUs and one or more specialized hardware accelerators. GPUs and Intel Xeon Phi coprocessors are currently the most popular.

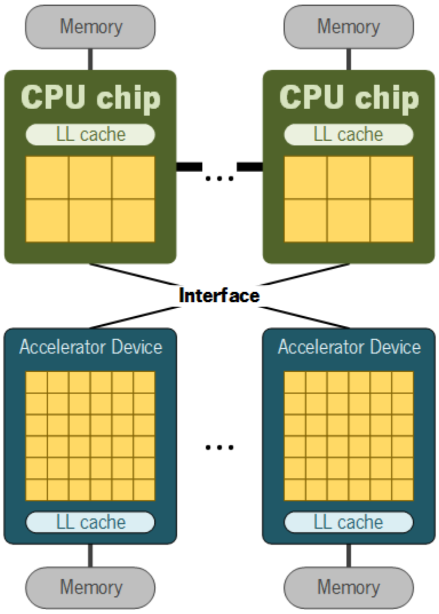


Figure 2.1: Example of a computing node architecture.

Popularity of these new specialized devices in HPC, some created and developed in completely separate environments, has been increasing in the last years, triggered by the trend to use the number of cores in computing hardware. GPUs evolution, for example, where execution throughput is more important than execution latency, led to massively parallel architectures, able to process hundreds of pixels at the same time. Devices based on the Intel MIC architecture, on the other hand, are placed between GPUs and CPUs, having the characteristics for massive parallelism while still being able to handle more complex operations (such as running an operating system). Both kinds of devices are explained in more depth in Chapters 5 and 6 where they are explored in the context of this document's case study.

DSPs are another class of accelerators recently made popular for HPC due to new architectures able to perform floating-point computations. Their architecture is very similar to that of a conventional processor, both programming and memory-wise [22]. Alternatively, Field Programmable Gate Arrays (FPGAs) mix a set of configurable logic blocks, digital

---

[1] `http://www.top500.org`

signal processor blocks and traditional CPU cores (optional), all using a configurable interconnect. The key characteristic of these devices is the ability to configure them for a specific type of computation making them extremely versatile [23]. These devices are described here for completeness, not being the scope of this document to explore the case study using them.

Most computing accelerators suffer from the same limitations as conventional processor architectures regarding memory latency (explained in Section 2.2), despite the many strategies each one implements to hide or overcome the problem. Also, since the connection between the CPU and an accelerator is typically performed over a PCI Express (PCIe) interface, using the same memory banks would be a major bottleneck. For this reason, most devices have their own built-in memory hierarchy, which is managed in a space distinct of the CPUs.

2.2 Distributed Memory

CPUs and memories evolution followed very distinct paths. While the former focused on speed, the latter focused on capacity. Throughout the decades, this created and aggravated a performance gap between the processor and the memory, with memory accesses taking much longer than instruction execution (around 100 times more). In CPUs, this limitation was overcome with the creation of a memory hierarchy, with the large main memory in the bottom, and multiple levels of cache memory, each smaller, faster and closer to the computing cores.

In a typical SMP system, all the processors share the same interface to connect to the main memory. These architectures are classified as Uniform Memory Access (UMA), given that all processors are seen as equal by the system and therefore all memory requests have the same latency. Such designs scale poorly with the number of processors, as one processor has to wait until all previous requests from other processors are fulfilled in order to access the main memory. Added to the gap between processor and memory speeds, this further aggravates the memory bottleneck.

Non-Uniform Memory Access (NUMA) architectures arise in response to the interface bottleneck. By moving the memory controller to the CPU chip itself allows for each processor to have its own memory banks, thus parallelizing memory accesses. Yet, this causes memory accesses to take longer when the memory requested lies in another processor's memory, as the request has to go through the connection between the two processors. This increases the importance of controlling thread affinity, similar to what happens with memory cache. Some processors even implement NUMA inside the chip. The Magny-Cours architecture from AMD, for example, consists of two Istambul dies in the same package, each being a NUMA node with two memory channels [24].

In a single multiprocessor computational node, where multiple memories exist each directly connected to one CPU, a single unified address space exists for all the main memory.

This is called a shared memory system, where all the data lies in a single unified address space and every processor has access to all the memory implicitly. In other words, even if a given processor is not directly connected to the memory containing the required data, it can still address that data directly.

Implementing a shared-memory NUMA architectures introduces the complexity of maintaining the cache of multiple processors coherent. This is required in order for the multiple processors to be able to use the same data. When one processor changes a shared data item, the coherency mechanism notifies the remaining processors that the copy in their private cache is outdated and the item must be reloaded from main memory. Maintaining coherency guarantees the correction of the program, but it hampers the scalability of NUMA architectures.

A distributed memory system arises in HetPlats, where each computational node has its own main memory address space. A single computational node may also implement a distributed memory architecture if it contains any hardware accelerator with its own built-in memory (with its own address space). These systems communicate through message passing, having to explicitly transfer all the required data between the multiple address spaces. Between distinct computational nodes, communication is usually done over a network connection using an Message Passing Interface (MPI) library. In the case of accelerators, the solutions to implement communication depend on the tools used for development targeted for such devices. Communication becomes a possible bottleneck with distributed memory. As such, extra effort is required to the distribute the workload efficiently among the available resources in order to achieve the best computation to communication ratio.

2.3 Development Tools

Most developers use conventional sequential programming models, as this is the most natural way to plan the resolution of a given problem, one step at a time. For single-core systems, this worked perfectly, with the only parallelism in applications being extracted by the compiler and the processor at the instruction level. The transition to the multicore era brought together a new programming paradigm, which must be understood in order to fully take advantage of the most modern computing resources available.

Making the transition to parallel programming is not trivial. The ability to concurrently run several execution threads exposes the programmer to new problems: data races, workload balancing, deadlocks, etc. Debugging parallel applications is also harder and it requires smarter approaches, better than simply tracing the code (anything with more than four threads will be very hard to keep track of). The problem becomes even more complex when trying to increase efficiency with HetPlats. Often, a developer must be aware of the underlying architectural details in order to deploy an efficient implementation.

Several tools have been presented to aid developers in taking advantage of the resources available in multicore shared memory environments and HetPlats, namely to distribute data and workloads among all available computing resources and to manage efficient data transfers across private memory spaces. Despite none being explored in the scope of this dissertation, many frameworks have also been developed to abstract the programmer from architectural details and the complexities of adapting code to run in a new platform (like a hardware accelerator).

2.3.1 PThreads, OpenMP, TBB and Cilk

POSIX Threads (pthreads) names the standard C language threads programming interface for UNIX systems. This standard was introduced with the goal of making parallel programming in shared-memory systems portable when hardware vendors implemented their own proprietary versions of threads [25]. This API provides the tools for managing threads, mutual exclusion, condition variables, read/write locks and per-thread context.

OpenMP [26, 27] was formulated under a need similar to the purpose of pthreads: to abstract the different ways operating systems imposed for programming with threads. At the time (1997), UNIX used pthreads, Sun used Solaris threads, Windows used its own API and Linux used Linux threads [28].

OpenMP is a standard API, in C/C++ and Fortran, for parallel programming in a multi-platform shared memory application running on all architectures. The API itself is less verbose than pthreads and is very simple and easy to use (often through compiler directives). It abstracts an inexperience programmer from all the complexity of managing threads, but without lacking the required tools for advanced users to perform fine tuning. It is also portable and scalable. OpenMP only addresses homogeneous systems with conventional CPUs and automatically schedules efficient workloads among all available resources.

Intel Threading Building Blocks (TBB) is a C++ template library created by Intel with a similar purpose to OpenMP. While it is a lot more verbose, and lacks support for other languages, TBB provides algorithms, highly concurrent containers, locks and atomic operations, a task scheduler and a scalable memory allocator [29]. It is harder to program than OpenMP, but Intel claims it achieves equivalent or better performance.

2.3.2 OpenMPC and OpenACC

OpenMPC [30] is an extension of the OpenMP specification to provide translation from regular OpenMP compiler directives to CUDA code. Parallel zone directives delimit the blocks of code candidate for CUDA kernels. Only loop and section directives are considered true parallel sections, which are translated to perform workload distribution among the available threads. Synchronization directives cause the kernels to be split, as this is the only way to force global synchronization among all threads. Directives specifying data properties are interpreted to find the best GPU memory space for the required data.

OpenACC [31] is a standard API, in the same languages as OpenMP, meant to bring the advantages of OpenMP to programming with hardware accelerators. While originally designed only for GPUs, support has been extended for the Intel Xeon Phi coprocessor. It abstracts the programmer from the memory management, kernel creation and the accelerator management. It also allows to execute both on the device and the host at the same time.

Comparing, OpenMPC only provides support only for CUDA-enabled devices, while OpenACC supports NVIDIA and AMD GPUs alike and Intel MIC devices.

3 Case Study: The Matrix Square Root

$$A = X^2 \tag{3.1}$$

The square root of a matrix A is any matrix X that satisfies Equation (3.1). When it exists it is not unique, but when A does not have any real negative eigenvalue it has a unique square root whose eigenvalues all lie in the open right half plane (i.e. have non-negative real parts) [15]. This is the so-called principal square root $A^{1/2}$ and since it is the one usually needed in applications, it is the one the algorithm and associated implementations in this document are focused in computing.

The Schur method of Björck and Hammarling [14] is the most numerically stable method for computing the square root of a matrix. It starts by reducing the matrix A to upper triangular form T. By computing U, the square root of T, also upper triangular, the same recurrence relation allows to compute X, thus solving Equation (3.1). The matrix U is computed by solving Equations (3.2) and (3.3). This method is implemented in MATLAB as the `sqrtm` and `sqrtm_real` functions [32].

$$U_{ii}^2 = T_{ii} \ , \tag{3.2}$$

$$U_{ii}U_{ij} + U_{ij}U_{jj} = T_{ij} - \sum_{k=i+1}^{j-1} U_{ik}U_{kj} \ , \tag{3.3}$$

Deadman et al. [13] devised equivalent blocked methods for this algorithm. Blocking is a typical optimization technique for problems with a very large data set, which improves cache efficiency both in CPUs and hardware accelerators [33, 34, 35]. It preserves and reuses the data in the fastest (but smallest) levels of the memory hierarchy by limiting the computation to a subset of the domain at a time. This improves the application ability to take advantage of locality, both temporal and spatial, thus effectively reducing memory bandwidth pressure.

Given that only multicore environments were explored, the scope of this dissertation focuses on implementing the block method using the available resources in modern heterogeneous platforms.

3.1 Strategies

Equations (3.2) and (3.3) describe an algorithm where each element depends on those at its left in the same row and those below in the same column (Parlett's recurrence [36]). Consequently, the algorithm can be implemented either a column/row or a superdiagonal at a time.

While the first strategy (column/row) is preferred for any serial implementation due to a more efficient use of cache memory (better locality), it presents almost[1] no opportunities for parallelism since no more than one element is ready to be computed at any given time.

On the other hand, computing a super-diagonal at a time allows for several elements to be computed in parallel since all the dependencies were already computed in previous super-diagonals.

3.2 Methods

In the previous work of Deadman et al. [13], the authors devised a blocked algorithm to compute the square root of a matrix. Similar to the original, the blocked algorithm lets U_{ij} and T_{ij} in Equations (3.2) and (3.3) refer to square blocks of dimension $m \ll n$ (n being the dimension of the full matrix).

The blocks U_{ii} (in the main diagonal) are computed using the non-blocked implementation described previously. The remaining blocks are computed by solving the Sylvester equation (3.3). The dependencies can be solved with matrix multiplications and sums. Where available, these operations can be performed using the LAPACK `TRSYL` and Level 3 BLAS `GEMM` calls, respectively.

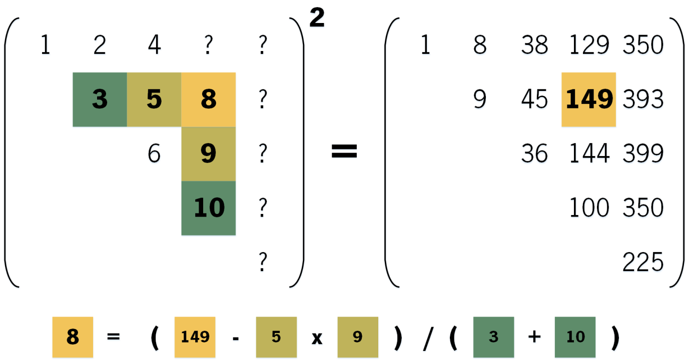
Two blocked methods were devised: a standard blocking method, where the matrix is divided once in a set of well defined blocks; and a recursive blocking method, where blocks are recursively divided into smaller blocks, until a threshold is reached. This allows for larger calls to `GEMM`, and the Sylvester equation can be solved using a recursive algorithm [37].

While the recursive method achieved better results in the serial implementations, when using explicit parallelism the multiple synchronization points at each level of the recursion decreased the performance, favouring the standard blocking method. Given the devices targeted by this document's work, where explicit parallelism is required to take full advantage of the architecture, the recursive method is ignored.

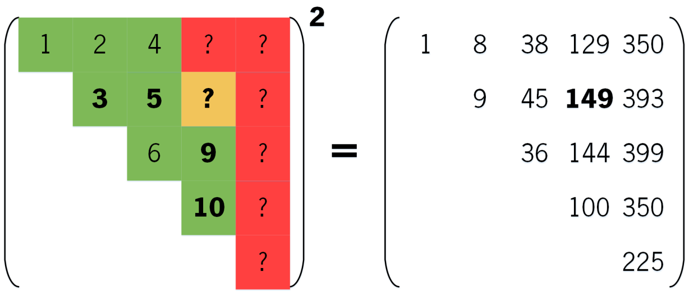
Using the same terminology, the non-blocked and standard blocked methods will be referred to hereafter as the point and block method, respectively.

---

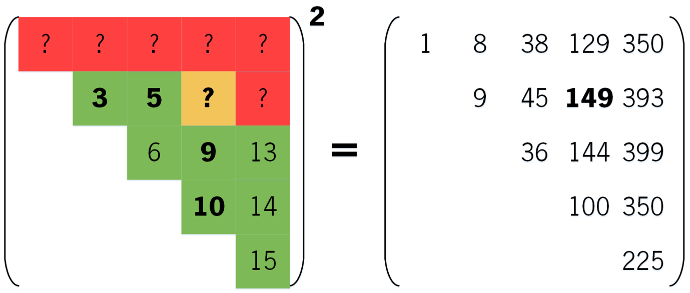
[1] It is possible for this strategy to solve its dependencies in parallel. The following chapters will show this with more detail.



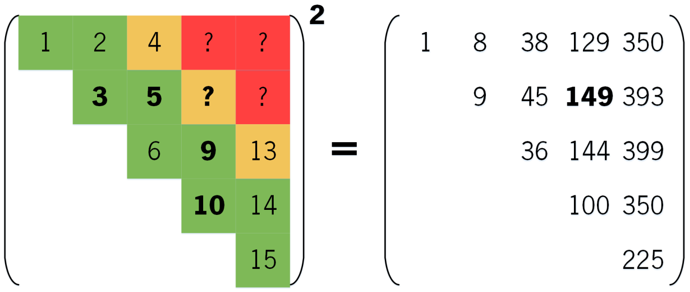
(a) Dependencies.



(b) Column strategy.



(c) Row strategy.



(d) Diagonal strategy.

Figure 3.1: Element/block dependencies and strategies for computing the matrix square root: green is solved, red has unsolved dependencies, yellow is ready to be computed next.

3.3 Evaluation Methodology

All the tests in this dissertation followed the same methodology: the best 3 measurements were considered, where the difference between the best and the worst could not exceed 5%, with a minimum of 10 runs and a maximum of 20. Time measurements were confined to the implementation of the algorithm, disregarding initialization and cleanup steps (such as I/O operations, allocating and freeing memory or interpreting program options). Double precision was used at all times, to emulate the needs of applications with minimal tolerance to precision loss.

Matrices of three different dimensions were used in performance tests (2000, 4000 and 8000). The smallest is meant to fit in the last-level cache of modern CPUs, but being large enough to be relevant when using blocks. The remaining dimensions force the program to use Dynamic Random-Access Memory (DRAM) in all the systems used for performance tests.

The best block dimension in the block method implementations was determined experimentally for each case.

4 Multicore

Most HPC programming nowadays can be resumed to one of two languages: Fortran and C/C++. While the latter is preferred by the majority of programmers, the former is the preference of many mathematicians and physicists to implement their algorithms and simulations.

The reason behind the success of Fortran lies in its evolution. Fortran (FORmula TRANslator) was the first language to abstract the underlying machine, which provided a way for scientists to program their numerical procedures, something that previously required knowledge of binary instructions, or the assembly language [38].

Since its first release, Fortran has been updated several times, each new "version" a language in its own right, while still striving to maintain compatibility with earlier versions. This fact allows for legacy code to keep its compatibility with new projects and provides programmers familiarized with Fortran with better tools, but at the same time it hampers the learning process, as a programmer who learns Fortran 90 will most likely have some trouble interpreting Fortran IV.

Such does not happen with C and C++, as both languages have been vastly enhanced since their first releases but the syntax itself remained very similar. Yet, these do not seem to be so attractive for scientists. Other alternatives are currently being used by the scientific and academic world, such as MATLAB and the free-software equivalent GNU Octave, which provide an even friendlier language and environment targeted for math and still focusing on performance.

Some features in Fortran are particularly useful for linear algebra algorithms, such as the array slicing notation. These features have long been incorporated in MATLAB and GNU Octave. As for C++, while the language does not support these features, this behaviour can be emulated using the Armadillo library. Intel Cilk Plus also provides an array notation eespecially targeted for identifying data parallelism.

The Armadillo library is a C++ linear algebra library with an API deliberately similar to MATLAB. Its complex template system (meta-programming) is targeted for speed and ease of use. In particular, it uses complex meta-programming mechanisms available in C++ to minimize memory usage. It also works as an abstract interface, allowing to use a high performance replacement for BLAS and LAPACK such as Intel MKL, AMD ACML or OpenBLAS. The interface is mostly agnostic during compilation and only during linkage does it require the replacement package to be available.

The original implementation developed in [13] was coded in Fortran 90 and it is under the licensing restrictions of the NAG Fortran library. In order to port the implementation to an open source environment, C++ is preferred due to familiarity with the language. Also, it has similar potential for HPC, it is more flexible and using the Armadillo library provides the same features for easier and faster development.

4.1 Column/Row

The implementation of the matrix square root algorithm was based on a MATLAB implementation of the algorithm for real upper triangular matrices from [13] and, as such, holds the same assumptions:

1. The input matrix is already in triangular form, being a perfect upper triangular real matrix;
2. The principal square root of the input matrix exists.

In other words, this implementation does not support complex arithmetic, neither does it support quasi-triangular matrices. It also does not compute the eigenvalues of the input matrix in order to check if the principal square root exists. These assumptions allow to focus the efforts towards implementing the core of the process using the available resources in heterogeneous systems, instead of implementing and validating multiple full featured solutions.

This reference implementation uses the column strategy, i.e., computes the square root matrix one column at a time. Algorithm 1 shows the algorithm for this strategy: columns are swept from left to right and (in each column) rows are traversed from the main diagonal up. The main diagonal element has no dependencies, so it is immediately computed. All the other elements depend on those on its left and below. Solving the dependencies for a given element U_{ij} outside the main diagonal consists in multiplying the sub-row in i, from the main diagonal to the column $j-1$, with the sub-column in j, from the row $i+1$ down to the main diagonal.

A block implementation of this algorithm consists in expanding the indices i and j to ranges. Given the assumptions for these implementations, the blocks can be thought of as a regular grid of squares where only the blocks in the last row and column may have smaller dimensions (if the block size is not a multiple of the matrix dimension). In fact, due to the upper triangular form of the matrix, only the blocks in the last column may be smaller (the last row is mostly zeros) and all the blocks in the main diagonal are squared.

Algorithm 2 shows the blocked algorithm using the column strategy. It seems quite more complex than the point method but it is mostly due to the expansion of indices to ranges. As such most of the algorithm can be directly associated with the previous method, with some small exceptions:

Algorithm 1: Matrix Square Root (column, point)

```
input  : A real upper triangular matrix $T$
output: A real upper triangular matrix $U$, where $U^2 \approx T$
n ← dimension of $T$
fill $U$ with zeros
for $j \leftarrow 0$ to $n-1$ do
    $U_{jj} \leftarrow \sqrt{T_{jj}}$
    for $i \leftarrow j-1$ to $0$ do
        $s \leftarrow 0$
        if $j-1 > i+1$ then
            $k \leftarrow$ range$(i+1,\ j-1)$
            $s \leftarrow U_{ik} \times U_{kj}$
        end if
        $U_{ij} \leftarrow \frac{T_{ij}-s}{U_{ii}+U_{jj}}$
    end for
end for
```

- Two variables x and y are added to store the blocks coordinates, and are used to reach the dependency blocks;
- Dependencies can no longer be solved with a vector-vector multiplication (instead, the sums and multiplications are done explicitly);
- The scalar arithmetic is replaced with linear algebra functions; in particular, computing the resulting block after solving the dependencies implies solving a Sylvester equation.

A row strategy is very similar to the column alternative. It consists mainly in swapping the two outer loops, so that the algorithm sweeps the matrix rows, from the bottom upward, and traverses each row from the main diagonal to the element in the last column. These two strategies are equivalent and fit the two methods for storing multidimensional arrays in linear memory: the column strategy takes advantage of a column-major order used in Fortran, MATLAB and GNU Octave; the row strategy has better locality in a row-major order, typically used in C, C++ and Python.

While the row-major order is typically used in C++, the Armadillo library uses column-major order by default to improve its compatibility with the standard Fortran interfaces used in BLAS and LAPACK packages. Some libraries, such as CUBLAS, do not even provide a way to configure this behaviour [10]. For convenience, all the implementations described in this document assume column-major order.

4.2 Super-diagonal

Given the dependencies of each element, using the column/row strategy does not allow to for more than one element to be computed at a time. On the other hand, since the

Algorithm 2: Matrix Square Root (column, block)

```
input  : A real upper triangular matrix $T$
input  : The dimension of a full block $b$
output: A real upper triangular matrix $U$, where $U^2 \approx T$
$n \leftarrow$ dimension of $T$
fill $U$ with zeros
$x \leftarrow 0$
$j_0 \leftarrow 0$
while $j_0 < n$ do
    $j_1 \leftarrow$min($j_0 + b$, $n$)$-1$
    $j \leftarrow$range($j_0$, $j_1$)
    $U_{jj} \leftarrow$sqrtm($T_{jj}$)
    $G = U_{jj}$

    $y \leftarrow x$
    $i_0 \leftarrow j_0$
    while $i_0 > 0$ do
        $y \leftarrow y - 1$
        $i_1 \leftarrow i_0$
        $i_0 \leftarrow i_0 - b$
        $i \leftarrow$range($i_0$, $i_1$)
        $F \leftarrow U_{ii}$
        $C \leftarrow T_{ii}$
        for $z \leftarrow y + 1$ to $x - 1$ do
            $k_0 \leftarrow z \cdot b$
            $k_1 \leftarrow (z + 1) \cdot b - 1$
            $k \leftarrow$range($k_0$, $k_1$)
            $C \leftarrow C - U_{ik} \times U_{kj}$
        end for
        $U_{ij} \leftarrow$sylvester($F$,$G$,$C$)
    end while

    $x \leftarrow x + 1$
    $j_0 \leftarrow j_0 + b$
end while
```

Algorithm 3: Matrix Square Root (diagonal, point)

```
input  : A real upper triangular matrix $T$
output: A real upper triangular matrix $U$, where $U^2 \approx T$
$n \leftarrow$ dimension of $T$
fill $U$ with zeros
for $d \leftarrow 0$ to $n-1$ do
    for $e \leftarrow 0$ to $n-d-1$ do
        if $d = 0$ then
            $U_{ee} \leftarrow \sqrt{T_{ee}}$
        else
            $i \leftarrow e$
            $j \leftarrow e + d$
            $s \leftarrow 0$
            if $j - 1 > i + 1$ then
                $k \leftarrow$range($i+1$,$j-1$)
                $s \leftarrow U_{ik} \times U_{kj}$
            end if
            $U_{ij} \leftarrow \frac{T_{ij}-s}{U_{ii}+U_{jj}}$
        end if
    end for
end for
```

dependencies of each element lie in those below and at its left, all the elements in the same diagonal can be computed in parallel.

Extending the column/row algorithm to a strategy that sweeps diagonals, starting with the main diagonal and going up towards the top-right corner of the matrix, is not trivial but becomes simple when indexing each of the diagonals and their elements. Algorithm 3 shows the algorithm for computing the square root of a matrix using the point method and the (super-)diagonal strategy. The diagonals are numbered from 0 (main diagonal) to $n-1$ (the last diagonal, containing only the top-right corner element). Inside each diagonal, the elements are indexed in top-down order, starting at zero. Notice that this simple index system fits the needs of the algorithm perfectly, allowing to compute easily how many elements each diagonal has. The core of the algorithm, specifically the operations performed in each element, remains the same as with the column/row strategy.

The extension to the block method is shown in Algorithm 4. In this algorithm, besides expanding i and j into ranges, d and e index a whole diagonal of blocks and a whole block, respectively. The expansion of indices to ranges is similar to what happens in the column/row strategy, with the exception that there is no need for the block coordinates x and y. These are replaced with the diagonal and element indices, which in turn slightly simplifies the procedure of solving the dependencies.

Algorithm 4: Matrix Square Root (diagonal, block)

```
input  : A real upper triangular matrix $T$
input  : The dimension of a full block $b$
output: A real upper triangular matrix $U$, where $U^2 \approx T$
$n \leftarrow$ dimension of $T$
#blocks $\leftarrow \lceil n/b \rceil$
fill $U$ with zeros
for $d \leftarrow 0$ to #blocks $- 1$ do
    for $e \leftarrow 0$ to #blocks $- d - 1$ do
        $i_0 \leftarrow e \cdot b$
        $i_1 \leftarrow$min($(e+1) \cdot b,\ n$)$-1$
        $i \leftarrow$range($i_0,\ i_1$)
        if $d = 0$ then
            $U_{ii} \leftarrow$sqrtm($T_{ii}$)
        else
            $j_0 \leftarrow (e+d) \cdot b$
            $j_1 \leftarrow$min($(e+d+1) \cdot b,\ n$)$-1$
            $j \leftarrow$range($j_0,\ j_1$)
            $F \leftarrow U_{ii}$
            $G \leftarrow U_{jj}$
            $C \leftarrow T_{ij}$
            for $z \leftarrow 1$ to $d - 1$ do
                $k_0 \leftarrow (e+z) \cdot b$
                $k_1 \leftarrow (e+z+1) \cdot b - 1$
                $k \leftarrow$range($k_0,\ k_1$)
                $C \leftarrow C - U_{ik} \times U_{kj}$
            end for
            $U_{ij} \leftarrow$sylvester($F$,$G$,$C$)
        end if
    end for
end for
```

4.3 Implementation

Implementing the algorithms described in this chapter using C++ relied on heavy use of the Armadillo library. The MATLAB-like syntax made available by this library allowed for straightforward translation from the point method algorithms to the working code. See Listing 1 for a working example (stripped of header inclusion and namespace specification).

As for the block methods, the implementation requires a little more explanation. `span` is the type used by Armadillo to represent ranges. Giving a range as a coordinate when accessing elements of the matrix actually retrieves a sub-matrix, which can then be used as a matrix on its own. The BLAS routine `GEMM` is interfaced using the standard multiplication operator (`*`) between two matrices. Lastly, the Sylvester equation is interfaced by the `syl()` function in Armadillo. After compilation, the library requires linkage to a BLAS and a LAPACK library.

The parallelization of the diagonal strategy implementations was done using OpenMP. Once again, the index system simplifies the process, allowing to parallelize the algorithm using only the `parallel for` directive.

4.4 Validation

Higham [39] describes the extension of the method devised by Björck and Hammarling, which requires complex arithmetic, to compute the real square root of a real matrix in real arithmetic. In his work, the author analyses the real Schur method, and concludes that it is stable, provided $\alpha_F(X)$ is sufficiently small, where

$$\alpha_F(X) = \frac{\|X\|_F^2}{\|A\|_F^2} \ .$$

The two blocked methods presented by Deadman et al. [13] for performing the same computation were both concluded to satisfy the same backward error bounds as the point algorithm.

Validating an implementation is based on measuring the relative error. Yet, the variation of this value is tied to the stability of the algorithm, which in turn was proved to be tied to the matrices used as input. The matrices used to validate and profile these implementations were randomly generated, and first attempts of validation showed great variations in the relative error as the dimension increased. To make it even more confusing, MATLAB uses its own implementation of BLAS and LAPACK, which translated in different relative errors for the same operations between the implementations described in this chapter and the reference. For these reasons, the relative error control was reduced to checking its order of magnitude.

```
typedef unsigned long ulong;

template<typename T>
void sqrtm (const Mat<T>& t, Mat<T>& u) {
    const ulong n = t.n_rows;
    u = zeros<Mat<T> >(n,n);
    for (ulong d = 0; d < n; ++d) {
        const ulong m = n - d;
#pragma omp parallel for
        for (ulong e = 0; e < m; ++e) {
            if (d == 0)
                u(e,e) = std::sqrt( t(e,e) );
            else {
                // find the index
                const ulong i = e;
                const ulong j = e + d;
                // solve dependencies
                const ulong z[2] = { i + 1, j - 1 };
                T s = T(0);
                if (z[1] > z[0]) {
                    const span k(z[0], z[1]);
                    const Mat<T> tmp = r(i,k) * r(k,j);
                    s = tmp(0,0);
                }
                r(i,j) = (t(i,j) - s) / (r(i,i) + r(j,j));
            }
        }
    }
}
```

Listing 1: C code for the matrix square root algorithm, using the point method and the diagonal strategy.

4.4.1 Control Matrices

Randomly generated matrices may not have a principal square root. To ensure the existence of such a matrix, after one being generated it is multiplied by itself and the resulting matrix is the one used to test the implementations. Yet, this process already introduces some rounding errors through the matrix multiplication operation, which worsens the difficulties in properly comparing relative errors.

To ease the validation of new implementations, control matrices are generated instead of random ones. These matrices are composed by integer numbers, consecutive along the columns. See for example the left hand side of Equation (4.1). The square of such a matrix is a very well defined integer matrix, and as such it is reasonable to expect that any working implementation would be able to revert the process with minimal loss of precision. The consecutive elements of these matrices make them very easy to confirm visually, which aids in confirming progress during the development process. It allows to confirm correction of very large matrices by checking specific elements since the expected first element in a given column can be easily calculated through the sequence of triangular numbers.

$$\underbrace{\begin{bmatrix} 1 & 2 & 4 & 7 & 11 \\ 0 & 3 & 5 & 8 & 12 \\ 0 & 0 & 6 & 9 & 13 \\ 0 & 0 & 0 & 10 & 14 \\ 0 & 0 & 0 & 0 & 15 \end{bmatrix}^2}_{U} = \underbrace{\begin{bmatrix} 1 & 8 & 38 & 129 & 350 \\ 0 & 9 & 45 & 149 & 393 \\ 0 & 0 & 36 & 144 & 399 \\ 0 & 0 & 0 & 100 & 350 \\ 0 & 0 & 0 & 0 & 225 \end{bmatrix}}_{T} \tag{4.1}$$

4.5 Results

Performance measurements are required to verify and quantify the improvements from using the block method over the point one and to properly compare the algorithms for the two presented strategies. This section presents the results obtained for these algorithms with tests running in a single node of the 701 group in the SeARCH[1] cluster. In addition to what was described in Section 3.3, the methodology for these tests also included power-of-two equivalent dimensions (2048, 4096 and 8192). A block dimension of 64 was experimentally found to be the most efficient.

Nodes in this group run Linux CentOS 6.2 with two 8-core Intel Xeon E5-2650 processors and 64GB of shared DRAM (NUMA). Each of its processors runs at a clock frequency of 2.00 GHz and has hardware support for 16 simultaneous threads with Intel Hyper-Threading. Further information regarding the hardware in these nodes is shown in Table 4.1.

---

[1] `http://search.di.uminho.pt`

Tests were built using Intel Composer XE 2011 (compiled with Intel C++ Compiler (icpc) 12.0.2 and linked with Intel Math Kernel Library (MKL), for optimized BLAS and LAPACK) and Armadillo C++ Linear Algebra library (version 3.800.2).

Clock frequency	2.00 GHz
Cores	8
SIMD width	256-bit (AVX)
Memory size	64 GB
Peak DP FLOPs	128 GigaFLOP/s
Peak Memory Bandwidth	51.2 GB/s

Table 4.1: Hardware details for SeARCH group 701 nodes (further information available in [40, 41]).

Preliminary tests showed identical results for the column and the row strategy. Consequently, results will are shown only for one of these strategies. As expected, they do not scale at all, unlike the diagonal strategy that reaches its maximum performance when using all the hardware supported threads (Figures 4.1 and 4.2).

The block method also scales well (Figure 4.3), with the maximum speedup versus the point-row implementation being reached when using 32 threads (the 16 cores with Hyper-Threading). Using power-of-two matrices it is possible to see that not only is the block method faster than the point alternative, it is also less sensitive to small variations in the matrix size (Figure 4.5).

Blocking already shows significant speedup on its own, when compared to the point method using the same number of threads (Figure 4.4). Nonetheless, Figure 4.6 shows the scalability of the block method compared with single-threaded point-diagonal achieves impressive speedup values (over 128 using 16 or 32 threads). These values are even higher for particular matrix sizes (Figure 4.7).

4.6 Analysis

The results presented in this chapter confirm those described in [13]. The explicit parallelism made available by the diagonal strategy easily overcomes the lack of locality when compared with the colum/row strategy, and both methods scale very well, reaching the peak performance when all the available resources are used.

Using blocks is also clearly more efficient than the point method, although there is less speedup from blocking as the number of threads is increased. Still, the accumulated speedup shows a clear "bathtub" curve, with the peak performance being reached when using all the hardware supported threads in the CPU.

The sensitivity of the point method is especially strange. There is a particular case, with the matrix dimension of $n = 2048$, where the execution time is the double of $n = 2047$

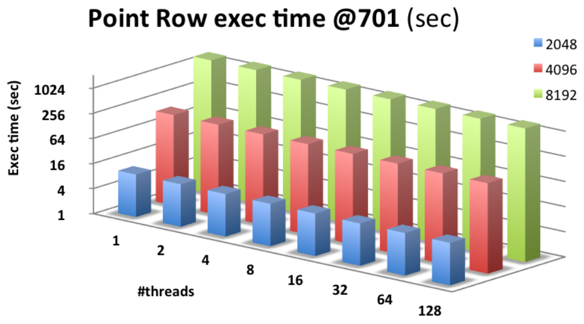


Figure 4.1: Execution times for point-row.

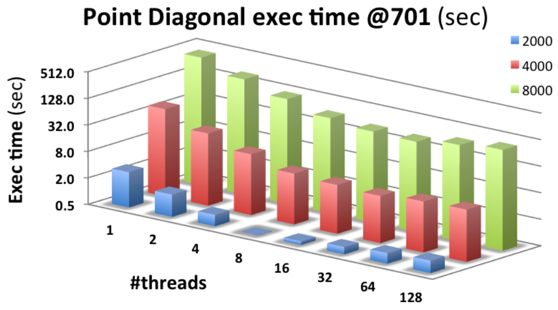


Figure 4.2: Execution times for point-diagonal.

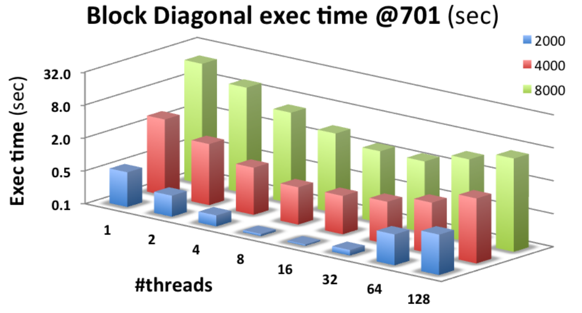


Figure 4.3: Execution times for block-diagonal.

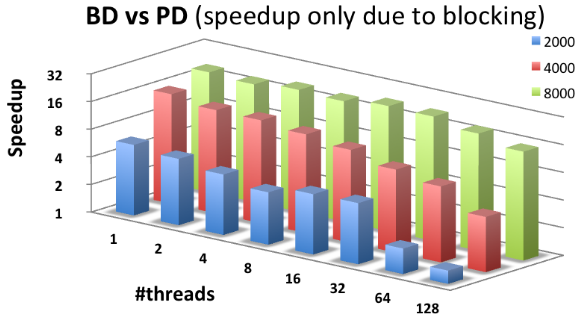


Figure 4.4: Speedups achieved with blocking for diagonal strategy.

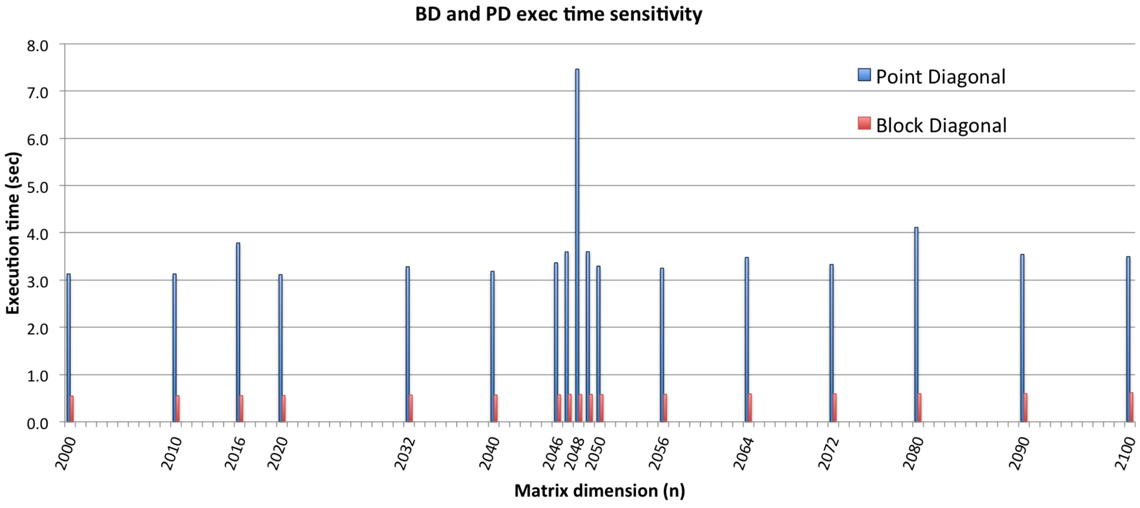


Figure 4.5: Execution time sensitivity for both point-diagonal and block-diagonal.

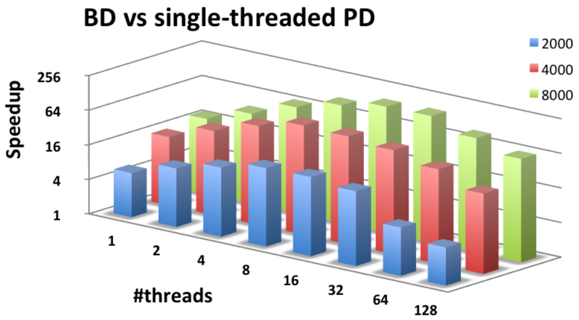


Figure 4.6: Accumulated speedup (blocking and parallelism) achieved with diagonal strategy.

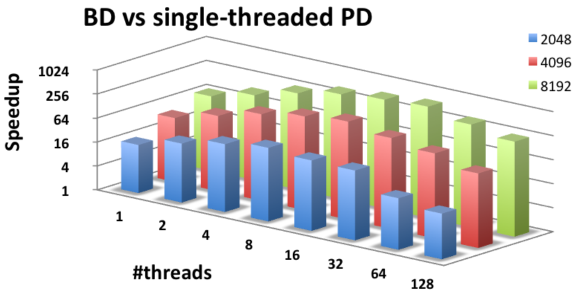


Figure 4.7: Accumulated speedup (blocking and parallelism) achieved with diagonal strategy (power of 2 sizes).

or $n = 2049$. Such strange behaviour does not happen with the block method, which implies that something happens at cache level with this particular size. In fact, research shows similar cases where the authors call this effect "cache resonance"[2] [42]. Basically, what happens is that this particular size causes the access stride to reach only a small group of cache lines. This rapidly saturates the cache ways available, causing capacity misses.

The scalability of the algorithm shown in these results increase the expectations of using the massive parallelism made available by the hardware accelerators studied in this dissertation.

---

[2] `http://stackoverflow.com/a/10364901/664321`

5 Intel MIC

The MIC architecture [43] is Intel's response to the increased demand for General Purpose GPUs (GPGPUs) as massively parallel hardware accelerators. The conceptual design of these coprocessors is distinct from GPGPUs and it follows Intel's trend to increase the number of cores in its products. Initially these devices were targeted for memory bound problems, unlike GPUs, but Intel has also launched a different version of the chip especially tuned for compute bound problems.

5.1 Architecture

The Intel Xeon Phi Coprocessor contains up to 61 fully functional in-order Intel MIC Architecture cores running at 1GHz (up to 1.3GHz), each containing 64KB of L1 cache (evenly split for data and instructions) and 512KB of L2 cache.

The device can support 8 memory controllers with two GDDR5 channels each. These support a total of 5.5 GT/s, corresponding to a theoretical aggregate bandwidth of 352 GB/s. So far, the maximum memory size available is 16GB[1].

A high performance on-die bidirectional ring connects the L2 caches, the memory controllers and the PCIe interface logic. The connected L2 caches allow for requests to be fulfilled from other cores' L2 caches faster than it would be from memory, thus implementing a last-level cache with over 30MB. Cache coherency is preserved across the entire coprocessor through a distributed tag directory using a reversible one-to-one hashing function.

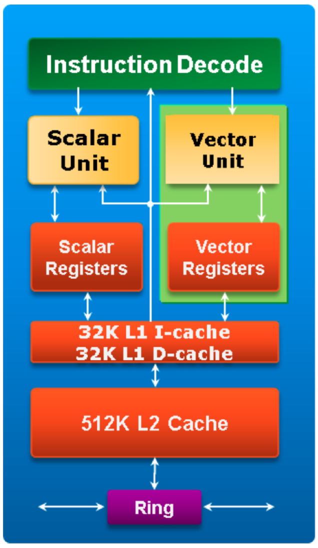


Figure 5.1: Intel MIC Architecture core diagram..

Intel MIC Architecture is based on the x86 Instruction Set Architecture (ISA), extended with 64-bit addressing and 512-bit wide SIMD vector instructions and registers. Yet, it does not support other SIMD ISAs (MMX, Intel Streaming SIMD Extensions (SSE) and Intel Advanced Vector eXtensions (AVX)).

Each coprocessor core supports up to 4 hardware threads and can execute 2 instructions per clock cycle, one on the U-pipe and one on the V-pipe (not all instructions can be executed

---

[1] Intel Xeon Phi Coprocessor 7120X [44]

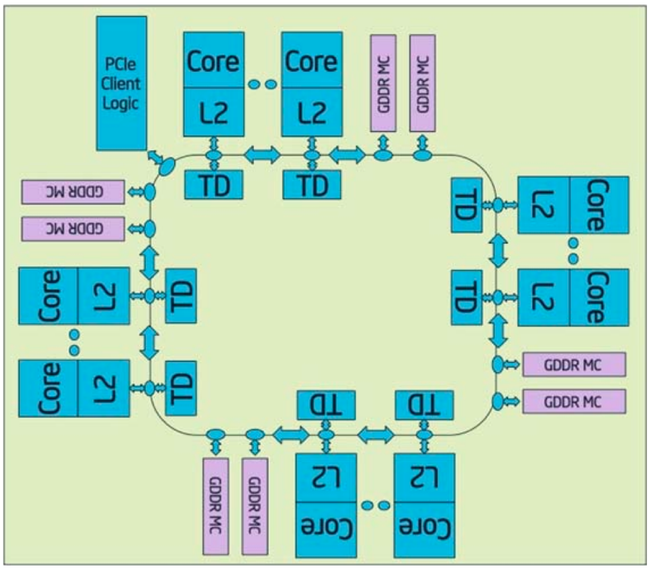


Figure 5.2: Knights Corner Microarchitecture.

in the latter). Each hardware thread has a "ready-to-run" buffer comprising two instruction bundles, each bundle representing two instructions that can be executed simultaneously.

The Vector Processing Unit (VPU) contains 32 vector registers, each 512-bit wide. It includes the Extended Math Unit (EMU) and is able to execute up to 16 integer or single-precision floating-point operations per cycle (half for double-precision). Additionally, each operation can be a floating-point multiply-add, thus doubling the number of operations in each cycle. Fully utilizing the VPU is critical to achieve high performance with the coprocessor.

5.2 Programming model

Intel MIC devices [43] have a Micro Operating System, a Linux*-based operating system, as opposed to what happens with most accelerator devices. This enables the coprocessor to work as a separate remote network node, independent of the host system.

These devices are able to operate in three different modes:

Native The application is run solely on the coprocessor, as if it was a remote network node;

Offload The host system offloads work to the coprocessor, as is usually done when using hardware accelerators;

Message Passing Using MPI, the coprocessor is treated as another peer in the network.

The native mode is the only one that allows all the cores to be used, since it is not necessary for the OS to manage the system, something that requires one of the cores to be exclusive in other modes. Running an application natively in the coprocessor requires building specifically for its architecture, which in icpc is done by providing the `-mmic` flag to both in the compiling and linking stages.

Native applications also require libraries specifically built for the Intel MIC architecture. While the Intel libraries are made available by default in the Intel Composer XE Suites, third-party libraries like Boost have to be especially built for this architecture. These libraries are then required in the linking phase of the building process and while running the application. This implies that they must be copied to the device together with the application.

Offload mode treats the device as a typical hardware accelerator, similar to what happens with a GPU, using compiler directives (`pragma offload` in C/C++) to control the application behaviour. Code for both the host and the coprocessor are compiled in the host environment. During the execution of the first offloaded code, the runtime system loads the executable and the libraries linked with the code into the coprocessor, and these remain on the device memory until the host program terminates (thus maintaining state across offload instances).

The offload code regions may not run on the coprocessor, depending on whether any device is present and it has any resources available. In these cases the offload code regions are executed on the host.

As happens with other hardware accelerators, offloading work to the device requires moving data between the host and the coprocessor. Using the offload directive this is explicitly done as directive clauses. An `in` clause defines the data that must be transferred from the host to the device before executing the offload region, while `out` transfers the data from the device to the host at the end of the offloaded code. Additionally, `inout` merges the functionality of both clauses, avoiding duplication. Using this memory copy model the data must be scalar or bitwise copyable structs/classes, i.e., arrays or structs/classes containing pointers are not supported. Exceptionally, variables used within the offload construct but declared outside its scope are automatically synchronized before and after execution in the coprocessor.

Alternatively, data can be implicitly transferred to the device using two new Intel Cilk Plus keywords: `_Cilk_shared` and `_Cilk_offload`. The former is used to declare a variable "shared" between the host and the coprocessor. This data is synchronized at the beginning and end of offload functions marked with the `_Cilk_offload` keyword. This implicit memory copy model surpasses the limitations of the explicit model in the offload directive, allowing for complex, pointer-based data structures.

Working in message passing mode is possible through three MPI programming models. The most common model, symmetric, creates MPI processes on both the host and the coprocessor, which are able to communicate over the PCIe bus. Coprocessor-only places all the MPI ranks on the coprocessor, similar to what happens with the native execution mode (the only difference lies in using MPI). Lastly, the host-only model confines all processes to the host, where the coprocessor can be used through offload pragmas. Symmetric and host-only models allow for hybrid OpenMP/MPI programming, offering more control over the parallelism.

Shared memory parallel programming in the coprocessor is possible using pthreads, OpenMP, Intel TBB and Intel Cilk. By default, for intra-node communication MPI also uses a shared memory network fabric.

Intel MKL has been extended to offer special support for the Intel Xeon Phi coprocessor since version 11.0. Some of its functions are especially optimized for the wider 512-bit SIMD instructions, and future releases will provide a wider range of functions.

The coprocessor makes available three distinct usage models for MKL: automatic offload, compiler assisted offload and native. Automatic offload requires no change in the source code, with the exception of enabling MKL in the coprocessor (may also be done in the environment). The runtime may automatically transfer data to the Xeon Phi and perform computations there. By default, the library decides when to offload and also tries to determine the optimal work division between the host and the devices (MKL supports multiple coprocessors), but this can be manually set in the source code or in the environment.

Native and compiler assisted offload usage happen when MKL is used in native execution and inside an offloaded code region, respectively. In comparison with automatic offload, compiler assisted offload has the advantage of allowing for data persistence on the coprocessor.

5.3 Native execution

The native execution mode provides several advantages over its alternatives. To start, it makes one extra core available. Given that the coprocessor's cores architecture is based on the x86 ISA, it also considerably reduces the development time, as a CPU functional implementation requires only to be rebuilt targeting the MIC architecture in order for the device to be able to run it natively. It also skips the communication necessary in an offload-based implementation, which is a potential bottleneck for many applications. For these reasons, this mode was selected for the first attempt to use the Intel Xeon Phi coprocessor.

According to Intel [45], to measure the readiness of an application for highly parallel execution, one should examine how the application scales, uses vectors and uses memory. Examining the scalability of the algorithm is performed by charting the performance of the implementation as the number of threads increases, something that was previously done in Section 4.5 with encouraging results.

Examining the vectorization consists in turning it on and off to check the differences, yet math routines such as those provided by Intel MKL remain vectorized no matter how the application is compiled. Since this includes the BLAS and LAPACK routines that provide the heavy-lifting in the algorithm, vectorization is assumed to be as good as it can get.

Memory is left unexamined, as it requires using hardware events. Memory can be a major issue, especially considering that the increased parallelism of these devices only makes sense using the explicit parallelism of the diagonal strategy, which presents no unit stride whatsoever (with the exception of half of the dependencies for each element). Yet, the series of Intel Xeon Phi coprocessor used for this dissertation was designed to be ideal for memory bound workloads [46].

As previously stated, no change was required to the code developed in the previous chapter, the only change being in the build process (the `-mmic` flag). However, the previous build system was not prepared for the Intel Xeon Phi coprocessor, since it implied cross-compilation, and had to be adapted.

5.3.1 Results

Although the Intel Xeon Phi coprocessor is able to run the same code as a regular Intel Xeon processor, the question remains whether it is able to achieve similar or higher efficiency without extra effort. This section shows the results obtained with performance tests using the diagonal strategy in the coprocessor, following the methodology described in Section 3.3 with a block dimension of 32.

Measurements for this section were performed using a single computational node of the 711 group in the SeARCH cluster, containing two Intel Xeon E5-2670 CPUs sharing 64GB DRAM (NUMA) and an Intel Xeon Phi Coprocessor 5110P (see Table 5.1 for the hardware details). These nodes run Linux CentOS 6.4 and provide Intel Composer XE 2013. Tests were built with icpc 13.1.2, MKL, and Armadillo (3.900.7).

Figures 5.3 and 5.4 show the obtained execution times using the diagonal strategy for both methods as described in Chapter 4. The scalability of the algorithm is near perfect for both, with the execution time being cut almost by half until the number of threads matches the number of cores in the device. After that, the speedup slows down, with the point method reaching its peak performance when using 4 hardware threads per core (full Hyper-Threading) and the block method plateauing at 2 hardware threads per core. The

	CPU	**MIC**
Clock frequency	2.60 GHz	1.053 GHz
Cores	8	60
SIMD width	256-bit (AVX)	512-bit
Memory type	—	GDDR5
Memory size	64 GB	8 GB
Memory speed	—	5.0 GT/s
Peak DP FLOPs	166.4 GigaFLOP/s	1.01 TeraFLOP/s
Peak Memory Bandwidth	51.2 GB/s	320 GB/s

Table 5.1: Hardware details for SeARCH node 711-1 (further information available in [41, 47, 48]).

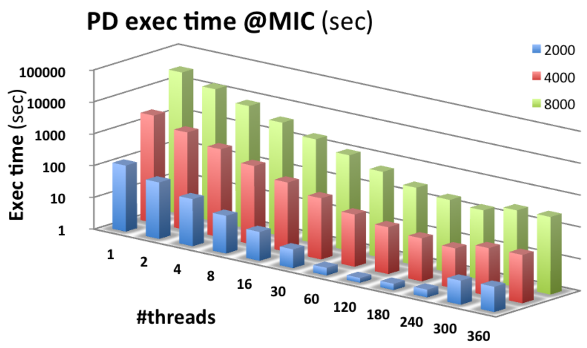


Figure 5.3: Execution times for point-diagonal in the Intel Xeon Phi.

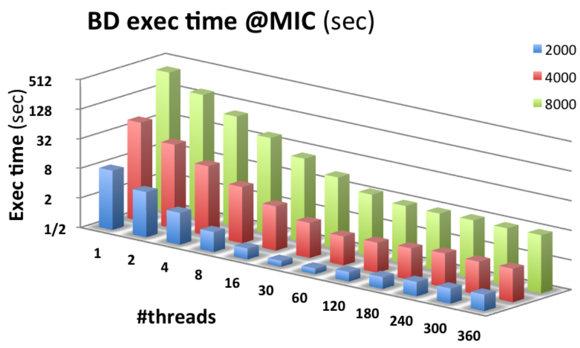


Figure 5.4: Execution times for block-diagonal in the Intel Xeon Phi.

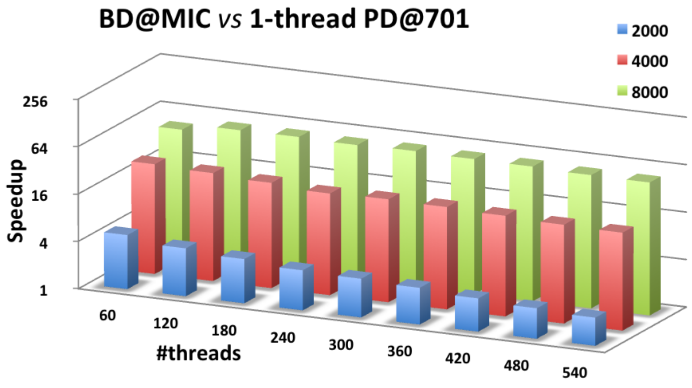


Figure 5.5: Accumulated speedup from block-diagonal in the Intel Xeon Phi versus point-diagonal in the CPU.

block method performance only leaves this plateau when 8 or more threads per core are used. Although the accumulated speedup of executing the block-diagonal in the Intel Xeon Phi is still significant when compared to single-threaded point method in the CPU, it does not even reach half of the accumulated speedup achieved in the CPU (Figure 5.5). In fact, when comparing the accumulated speedups in both platforms, the best value obtained in the Intel Xeon Phi is about three times slower than the best value obtained in the CPU.

5.4 Optimization Techniques

The results presented in Section 5.3.1 are surprising, given the success of those obtained with the multicore implementation in Chapter 4 and the resources available in the Intel Xeon Phi coprocessor. The discrepancy is so large that a decision was made at this point to improve the performance of this implementation before exploring any other execution modes or programming models.

Documents from Intel state that the best way to prepare for Intel Xeon Phi coprocessors is to fully exploit the performance that an application can get on Intel Xeon processors first. Trying to use the coprocessor without maximizing the use of parallelism on the processor will almost certainly be a disappointment [45]. As such, the optimizations presented in this section are focused on a deeper analysis of the implemented algorithm and profiling the application running on a multicore environment, as doing so allows to use the tools made available in Intel Parallel Studio XE 2013.

5.4.1 Massive Parallelism

Many-core devices like the Intel Xeon Phi coprocessor make hundreds of parallel computing resources available to applications. When the degree of parallelism in such applications is too low, some of these resources remain idle during execution, hampering efficiency. This is a corollary from Amdahl's Law [49], which explains that the maximum theoretical speedup an application might achieve in a given architecture is limited by the amount of time a sequential processor would spend in the parallel part of the code in comparison to the sequential part.

The degree of parallelism explored so far in the matrix square root algorithm is quite limiting. For any matrix of dimension n, there will be at most n elements to be computed in parallel, which happens only once. After the main diagonal, every other diagonal has one less element to compute, until the last diagonal containing only one element. Consequently, the last diagonals are unable to take advantage of a high quantity of parallel resources.

Yet, this can be compensated when solving dependencies. Analysing Equation (3.3) shows a sum of multiplications that translates into a dot product between the elements at the left and below the one being computed (excluding the main diagonal). A dot product is a highly parallel operation implemented in Level 1 BLAS, and the number of dependencies increases as the algorithm progresses. For the last diagonal's only element, this operation can perform $n - 2$ multiplications in parallel and sum all the products with a reduction.

For the point method, introducing parallelism when solving the dependencies requires a nested `parallel for` OpenMP directive with a reduction clause. As for the block method, this extension is not trivial as most OpenMP libraries do not allow reductions to be performed using non-scalar types such as Armadillo matrices. This can be circumvented by separating the directives: inside the parallel zone, each thread initializes a private matrix with the size of a full-block; a parallel loop then iterates over the dependencies, each thread subtracting a block from its copy; after the loop, an OpenMP lock allows each thread to add the computed block to the final result without creating a race condition.

This technique does not solve the application bottlenecks, but it allows for a better usage of resources in massively parallel devices (useful for both Intel Xeon Phi and GPUs).

5.4.2 Loop Unrolling

Revisiting Chapter 3 and Algorithms 3 and 4, in particular the description of the algorithm dependencies, a deeper analysis leads to the conclusion that it is logical to unroll the diagonal loop. In both algorithms, the first and second diagonals act differently from the rest. The first diagonal has no dependencies and, as such, recursively applies the square root on the focused element/block (standard `sqrt` in the point method, which in turn is used by the block method).

On the other hand, the elements/blocks in the second diagonal depend only of those in the main diagonal. As such, there are no dependencies to solve, as the main diagonal elements/blocks are used directly to compute the final result.

The following diagonals perform additional work, having to compute how the elements/blocks on the left and below affect the input value, where this affected value is the one used to compute the final result.

Algorithm 5: Matrix Square Root (diagonal, point)

```
input  : A real upper triangular matrix $T$
output: A real upper triangular matrix $U$, where $U^2 \approx T$
$n \leftarrow$ dimension of $T$
fill $U$ with zeros
sqrtm_d0($T$, $U$)
sqrtm_d1($T$, $U$)
for $d \leftarrow 2$ to $n-1$ do
    sqrtm_dn($d$, $T$, $U$)
end for
```

Algorithm 5 shows the unrolled algorithm for the point method, using three distinct functions, one for each case. `sqrtm_d0` handles the main diagonal ($d = 0$), `sqrtm_d1` handles the first super-diagonal ($d = 1$) and `sqrtm_dn` handles all the other diagonals, (d is provided as an argument in the function call). These functions are described in Algorithms 6 to 8, respectively. Algorithms 10 to 12 show the corresponding algorithms for the block method, following the same index expansion logic described in Chapter 4.

Algorithm 6: Matrix Square Root – main diagonal (point)

```
input  : A real upper triangular matrix $T$
output: A real upper triangular matrix $U$, where $U^2 \approx T$
$n \leftarrow$ dimension of $T$
for $e \leftarrow 0$ to $n-1$ do
    $U_{ee} \leftarrow \sqrt{T_{ee}}$
end for
```

5.4.3 Armadillo

The information gathered by the Basic Hotspot analysis available in Intel VTune Amplifier XE 2013 allows to identify the most time-consuming source code regions in an application. Results of this analysis run against the code implemented so far are shown in Table 5.2,

Algorithm 7: Matrix Square Root – first super-diagonal (point)

```
input  : A real upper triangular matrix $T$
output: A real upper triangular matrix $U$, where $U^2 \approx T$
$n \leftarrow$ dimension of $T$
for $e \leftarrow 0$ to $n - 2$ do
    $i \leftarrow e$
    $j \leftarrow e + 1$
    $U_{ij} \leftarrow \frac{T_{ij}}{U_{ii}+U_{jj}}$
end for
```

Algorithm 8: Matrix Square Root – other super-diagonals (point)

```
input  : The diagonal index $d$
input  : A real upper triangular matrix $T$
output: A real upper triangular matrix $U$, where $U^2 \approx T$
$n \leftarrow$ dimension of $T$
for $e \leftarrow 0$ to $n - d - 1$ do
    $i \leftarrow e$
    $j \leftarrow e + d$
    $r \leftarrow$ sub-row in $i$ from $i + 1$ to $j - 1$
    $c \leftarrow$ sub-column in $j$ from $i + 1$ to $j - 1$
    $s \leftarrow r \times c$
    $U_{ij} \leftarrow \frac{T_{ij}-s}{U_{ii}+U_{jj}}$
end for
```

Algorithm 9: Matrix Square Root (diagonal, block)

```
input  : A real upper triangular matrix $T$
input  : The dimension of a full block $b$
output: A real upper triangular matrix $U$, where $U^2 \approx T$
$n \leftarrow$ dimension of $T$
#blocks $\leftarrow \lceil n/b \rceil$
fill $U$ with zeros
sqrtm_d0($T$, #blocks, $b$, $U$)
sqrtm_d1($T$, #blocks $- 1$, $b$, $U$)
for $d \leftarrow 2$ to #blocks $- 1$ do
    sqrtm_dn($d$, $T$, #blocks $- d$, $b$, $U$)
end for
```

Algorithm 10: Matrix Square Root – main diagonal (block)

```
input : A real upper triangular matrix $T$
input : The number of blocks in this diagonal #blocks
input : The dimension of a full block $b$
output: A real upper triangular matrix $U$, where $U^2 \approx T$
$n \leftarrow$ dimension of $T$
for $e \leftarrow 0$ to #blocks $- 1$ do
    $i_0 \leftarrow e \cdot b$
    $i_1 \leftarrow$min($(e+1) \cdot b$, $n$)$-1$
    $i \leftarrow$range($i_0$, $i_1$)
    $U_{ii} \leftarrow$sqrtm($T_{ii}$)
end for
```

Algorithm 11: Matrix Square Root – first super-diagonal (block)

```
input : A real upper triangular matrix $T$
input : The number of blocks in this diagonal #blocks
input : The dimension of a full block $b$
output: A real upper triangular matrix $U$, where $U^2 \approx T$
$n \leftarrow$ dimension of $T$
#blocks $\leftarrow \lceil n/b \rceil$
for $e \leftarrow 0$ to #blocks $- 1$ do
    $i_0 \leftarrow e \cdot b$
    $i_1 \leftarrow$min($(e+1) \cdot b$, $n$)$-1$
    $i \leftarrow$range($i_0$, $i_1$)
    $j_0 \leftarrow (e+1) \cdot b$
    $j_1 \leftarrow$min($(e+2) \cdot b$, $n$)$-1$
    $j \leftarrow$range($j_0$, $j_1$)
    $U_{ij} \leftarrow$sylvester($U_{ii}$, $U_{jj}$, $T_{ij}$)
end for
```

Algorithm 12: Matrix Square Root – other super-diagonals (block)

```
input : A real upper triangular matrix $T$
input : The number of blocks in this diagonal #blocks
input : The dimension of a full block $b$
output: A real upper triangular matrix $U$, where $U^2 \approx T$
$n \leftarrow$ dimension of $T$
#blocks $\leftarrow \lceil n/b \rceil$
for $e \leftarrow 0$ to #blocks $- 1$ do
    $i_0 \leftarrow e \cdot b$
    $i_1 \leftarrow$min($(e+1) \cdot b,\ n$)$-1$
    $i \leftarrow$range($i_0,\ i_1$)
    if $d = 0$ then
        $U_{ii} \leftarrow$sqrtm($T_{ii}$)
    else
        $j_0 \leftarrow (e+d) \cdot b$
        $j_1 \leftarrow$min($(e+d+1) \cdot b,\ n$)$-1$
        $j \leftarrow$range($j_0,\ j_1$)
        $F \leftarrow U_{ii}$
        $G \leftarrow U_{jj}$
        $C \leftarrow T_{ij}$
        for $z \leftarrow 1$ to $d - 1$ do
            $k_0 \leftarrow (e+z) \cdot b$
            $k_1 \leftarrow (e+z+1) \cdot b - 1$
            $k \leftarrow$range($k_0,\ k_1$)
            $C \leftarrow C - U_{ik} \times U_{kj}$
        end for
        $U_{ij} \leftarrow$sylvester($U_{ii},\ U_{jj},\ C$)
    end if
end for
```

with the two most time-consuming code regions happening in MKL. Consequently, these are ignored as the optimized library is left in charge of these operations. After BLAS and LAPACK, the most time-consuming function belongs to Armadillo, and is meant for copying the blocks to be used as independent matrices.

Function	Time Spent (s)
`dgemm`	43.311
`dtrsyl`	23.398
`arrayops::copy_big`	15.555
`[OpenMP Worker]`	6.303
`dgees`	5.811

Table 5.2: Basic Hotspot analysis results (in development environment).

Armadillo tries to minimize matrix allocations and copies whenever possible. For example, chaining matrix addition operations uses a complex system of template "glues" that solve these additions without performing additional memory allocations. It also cares to use BLAS operations without allocating a matrix to store the result whenever possible. Nevertheless, it is unable to perform optimizations like these when blocks are isolated because these have to be treated as standalone matrices from then on. Also, Armadillo lacks an interface for the LAPACK function `TRSYL` which does not the result matrix is not allocated. Consequently, it is necessary to remove Armadillo from the implementation, replacing it with standard arrays and manual calls to BLAS and LAPACK.

For simplicity, implementation is bound to the Intel MKL BLAS interface. In the functions arguments (Algorithms 5 to 12), Armadillo matrices are replaced with standard memory pointers (and the matrix dimension). In the point method, $s \leftarrow r \times c$ is replaced with a call to Level 1 BLAS `DOT`, which does not require the c and r arrays to be isolated by using increments of 1 and n, respectively.

For the block method, $C \leftarrow C - U_{ik} \times U_{kj}$ is replaced with a call to Level 3 BLAS `GEMM`, and $U_{ij} \leftarrow \texttt{sylvester}(U_{ii}, U_{jj}, C)$ is replaced with a call to LAPACK `TRSYL`. Both these calls overwrite one of the operands, effectively removing any need for allocations and copy operations.

Note that Armadillo is removed from the computation but it is still used for I/O operations (loading the matrix from a file and result output).

5.4.4 Unit Stride Blocks

After removing Armadillo, it becomes clear how BLAS and LAPACK calls access submatrices using the leading dimensions of the whole matrix. This simple approach is, however, error prone and better locality can be achieved if it is not required to jump n elements from one block column to the next.

The matrices can be reorganized so each independent block is contiguous in memory, effectively making it an independent matrix. See Equation (5.1) for example. A is a regular column-major matrix, with the elements in the same column contiguous in memory, and each column also contiguous in memory. When trying to access the sub-matrix A_{11}, corresponding to the first two rows and columns, three elements of the first column must be skipped. In matrices where the dimension is large enough this translates into one memory access per block column. Converting A to Unit Stride Blocks (USB) format generates B where this does not happen because each block is now a column-major matrix, with all the blocks in the same column contiguous in memory, and the same being true for all columns of blocks.

$$\underset{A}{\left[\begin{array}{c|c|c|c|c} 1 & 2 & 4 & 7 & 11 \\ 0 & 3 & 5 & 8 & 12 \\ 0 & 0 & 6 & 9 & 13 \\ 0 & 0 & 0 & 10 & 14 \\ 0 & 0 & 0 & 0 & 15 \end{array}\right]} \Rightarrow \underset{B}{\left[\begin{array}{cc|cc|c} 1 & 2 & 4 & 7 & 11 \\ 0 & 3 & 5 & 8 & 12 \\ \hline 0 & 0 & 6 & 9 & 13 \\ 0 & 0 & 0 & 10 & 14 \\ \hline 0 & 0 & 0 & 0 & 15 \end{array}\right]} \tag{5.1}$$

This conversion operation does add some overhead to the initialization and to the cleanup (to revert the result to standard format), but it improves cache usage through spatial locality. This overhead may or may not be worth depending on how good is this improvement and how it affects the computation.

5.4.5 Overwrite

All the implementations so far assume at least two distinct matrices are used in the algorithm, one for T and another one for U. Aside from dependencies, this implies that when a block U_{ij} is being computed, another block T_{ij} must also be present. The memory footprint becomes even larger with the USB format where the conversion generates a second re-organized matrix, which is then used as the input matrix for the algorithm. The algorithm computes a third matrix with the result, also in USB format, which then has to be re-organized into a fourth matrix with the final result in the standard format.

BLAS and LAPACK routines minimize the memory footprint by overwriting one of the operands with the result of the operation. In the case of the matrix square root algorithm this is also possible since only one element/block in the input matrix T is used in the computation of each element/block in U. Consequently, the memory footprint can be easily reduced by overwriting T with U.

5.5 Results

The techniques described in Section 5.4 can not be considered optimizations until their impact in the application performance is properly quantified and evaluated. This section discusses the measurements performed for this purpose, using the same environment and methodology as already described in Section 5.3.1.

Performance tests focused mainly on the the techniques described in Sections 5.4.4 and 5.4.5 (USB and OW). Preliminary tests revealed that the first technique (massive parallelism) did not decrease the execution time of the algorithm, and the benefit from the second (loop unrolling) was too little to be significant. Although this was not the case with the third (removed Armadillo) it was used as the base for USB and OW, since it required a major change to the implementation. When reading the following results, it is important to consider that a significant fraction of the improvements shown by these techniques comes from that.

Theoretically, the parallelization of the dependency solving step compensates for the lack of parallelism in the more advanced stages of the algorithm. However, tests showed this not to be the case, showing how the memory access pattern hampers further scalability. As the algorithm progresses, the number of dependencies for each element increase, but half of this dependencies lie in distinct cache lines. The lack of locality seems to hamper the algorithm so much that it would nullifies any improvement from increased parallelism.

On the other hand, the lack of performance improvements from loop unrolling did not come as a surprise, serving the purpose of proving how the Intel Xeon Phi coprocessor is superior to a GPU when dealing with conditional branches.

The execution time for USB is shown in Figure 5.6 and, comparing with Figure 5.4, the speedup is clear even for the sequential time (reduced to less than half). The peak performance remained near the same number of threads (30 for $n = 2000$, 60 for $n = 4000$ and 120 for $n = 8000$), but when more threads than those supported by the hardware are demanded, the performance now drops more intensely, with the optimized implementation taking twice as long with 360 threads.

While not so good, OW also improves the efficiency of the implementation, as shown in Figure 5.7. Contrary to what happens with USB, this optimization practically removes the penalty of demanding more threads than what the hardware supports, maintaining the execution time near peak performance even above 8 threads per core. Yet, the minimum execution time this optimization achieves is not as low.

Although USB exceeds OW in speedup, the highest efficiency is achieved when using both optimizations at the same time (Figure 5.8). The sequential time and the time obtained with 360 threads are practically the same as for USB only, and the same happens for the peaks. However, the value obtained with these peaks is reduced when OW is also applied.

Figure 5.9 shows the best execution times for the naive implementation and each opti-

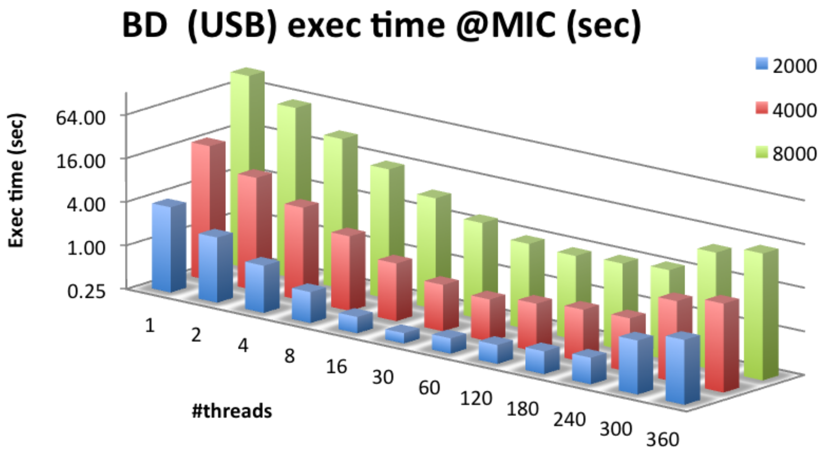


Figure 5.6: Execution times for USB in the Intel Xeon Phi coprocessor.

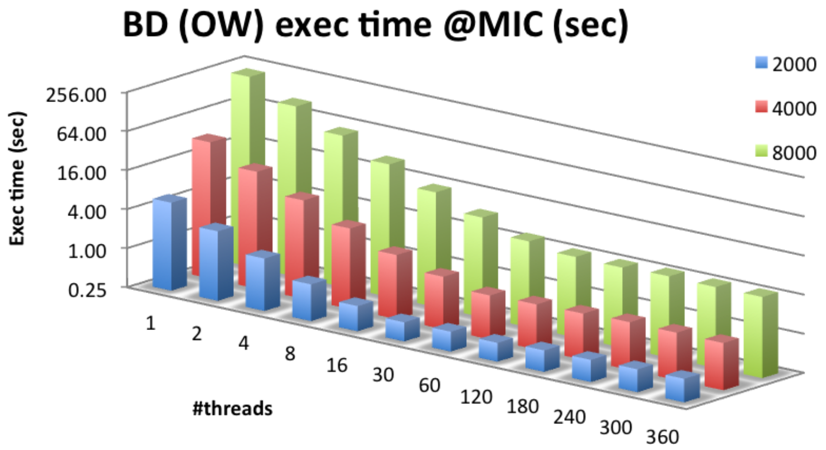


Figure 5.7: Execution times for OW in the Intel Xeon Phi coprocessor.

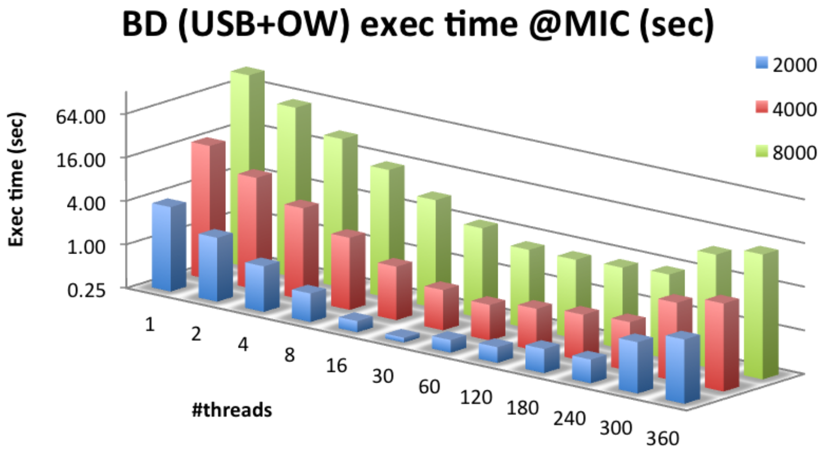


Figure 5.8: Execution times for USB and OW in the Intel Xeon Phi coprocessor.

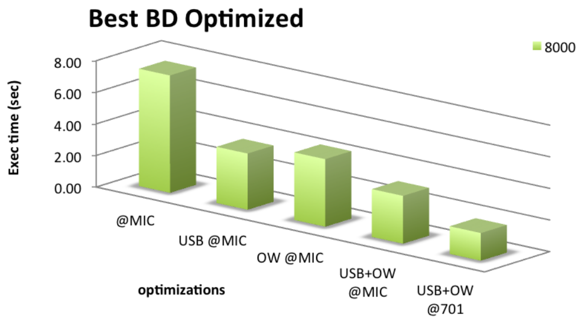


Figure 5.9: Best execution times for the optimizations.

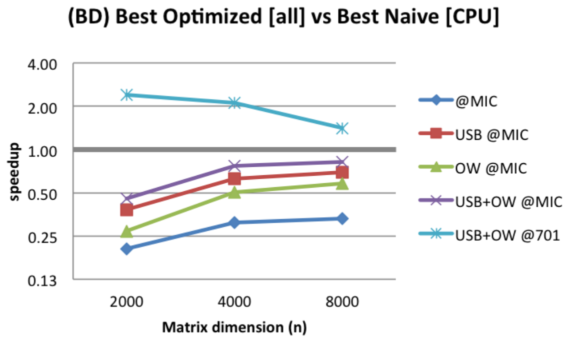


Figure 5.10: Speedups for the optimizations in the Intel Xeon Phi coprocessor versus multicore in group 701 (from Chapter 4).

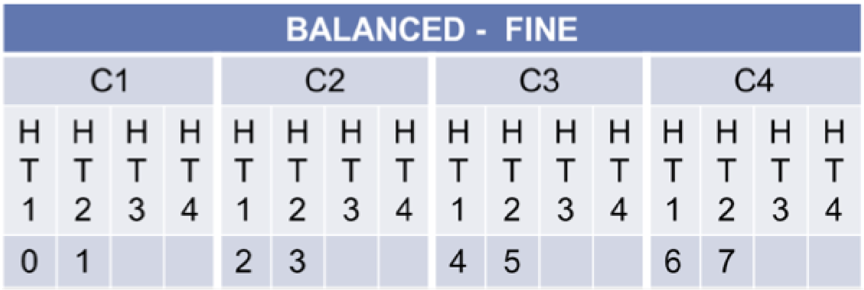
mization running in the Intel Xeon Phi and the best execution time for USB and OW together running on the multicore environment. While it is clear the benefit of both optimizations, there is the problem of these optimizations being common to both the coprocessor and the multicore environment. However, Figure 5.9 shows that these optimizations significantly reduce the speedup gap between the CPU and the device.

5.6 Further Optimizations

The results obtained in the previous section prove that achieving higher efficiency with the coprocessor is not as trivial as porting the code for it. However, there are still unexplored paths, which will remain so in the context for this dissertation due to its natural time constraints. The most promising of these paths is the control of thread affinity in the Intel Xeon Phi.

One of the most problematic issues when using a NUMA system lies in the fact that a process that starts in one processor may at some time be scheduled out and moved to another. The problem behind this lies in the memory hierarchy. When a process starts in a specific processor, it populates the cache and fills the memory directly linked to that processor with the data it requires. If moved to another processor, its data is no longer available in cache, which leads to a memory access, but the data it requires no longer lies in the closest memory. This also happens at the core level, with threads being scheduled to different cores inside the same processor, losing the advantage of locality in the core's private cache. For the specific case of Intel MIC devices, it is important for threads working with consecutive elements or in the same block to be in the same core for cache efficiency to be maximized.

Although OpenMP (specification 3.1) does not include any way to control thread affinity, Intel's OpenMP library contains a mechanism for it through an environment variable (`KMP_AFFINITY`) [50, 51]. Figure 5.11 shows the most promising affinity policy available in



BALANCED - FINE															
C1				C2				C3				C4			
HT1	HT2	HT3	HT4	HT1	HT2	HT3	HT4	HT1	HT2	HT3	HT4	HT1	HT2	HT3	HT4
0	1			2	3			4	5			6	7		

Figure 5.11: Example of balanced thread affinity policy (Intel OpenMP) in a 4 core coprocessor for 8 threads with fine granularity.

the coprocessor, where the threads are spread among the cores but grouped sequentially by ID when the number of demanded threads exceeds the number of cores. This policy is expected to achieve higher efficiency (due a better usage of the first levels of cache) with any granularity.

Despite this mechanism requiring minimal changes in the source code (or none at all), rerunning the performance tests at this point with the correct affinity setup would hamper the development of a CUDA implementation (described in the next chapter). Consequently, it is left for future work.

6 CUDA

Nowadays, GPUs are the most popular hardware accelerator being used in HPC. These devices evolved in the field of computer graphics, where each pixel is usually independent of those around. For these reasons, GPUs were designed from scratch to be able to perform the same simple operation using huge amounts of data.

For over a decade, computer scientists and domain scientists have been using these devices to execute code produced for other purposes besides image rendering – the advent of GPGPUs [52]. Programming these accelerators is not a trivial task since it requires knowledge of the underlying architecture in order to be able to take full advantage of the device capabilities. The GPU implementations described throughout this document will be targeted for NVIDIA devices using the NVIDIA's CUDA framework, since it is the dominant proprietary framework for GPGPU programming. For this reason, the architectural characteristics of GPGPUs will be described using CUDA terminology.

6.1 Programming Model

CUDA provides an extension to the C language (CUDA C) allowing the programmer to define functions – kernels – that are executed multiple times in parallel by as many different CUDA threads. Kernels must be declared using a new keyword, which tells the compiler the function will be executed in the GPU but launched from the CPU (host). A kernel is launched using a new execution configuration syntax, and once executing each thread is assigned a unique identifier accessible from within the kernel.

Threads are grouped in blocks, which in turn compose the grid executing the kernel. The maximum number of threads per block is quite limiting (1024 in recent generations [53]), but a kernel can be executed by multiple equally-shaped thread blocks. These blocks are distributed to the available Streaming Multiprocessors (SMs) in undefined order, in parallel or in series, and, consequently, must be independent. On the other hand, threads belonging to the same block are able to cooperate by sharing data and by synchronizing their execution.

There are three levels of memory available to the programmer: first and fastest, every thread has its own private local memory; each block then has shared memory visible to all its threads; and, finally, all the threads have access to the same device global memory, which is persistent across multiple kernel executions.

6.2 Architecture

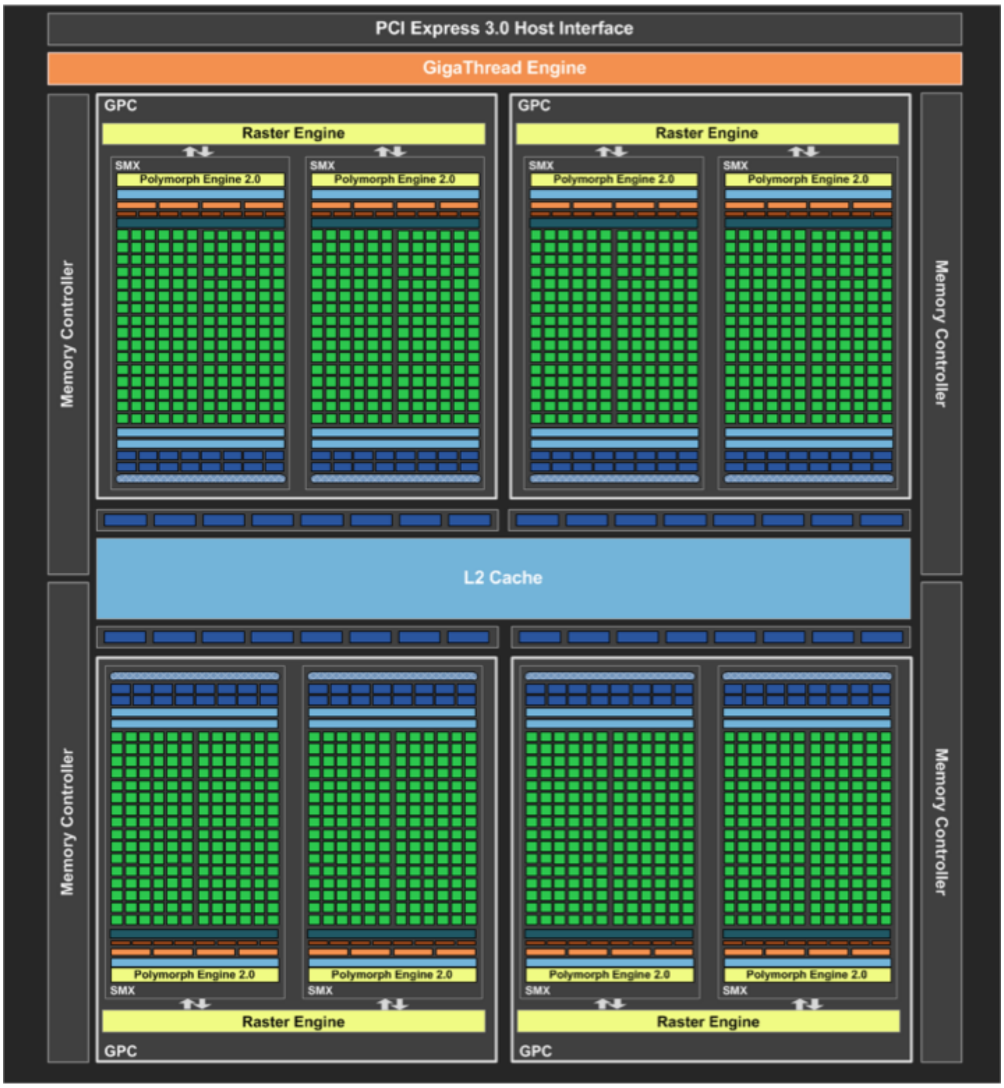


Figure 6.1: Overview of the GeForce GTX 680 Kepler Architecture [54].

CUDA-enabled GPUs are composed by several building blocks called Graphics Processing Clusters (GPCs), each with multiple multithreaded SMs connected to the global device memory (GDDR5 DRAM). Each SM contains

- a large set of CUDA cores, the processing units that perform the arithmetic operations;
- a much more limited number of Special Function Units (SFUs) and Load/Store units;
- a Register File, big enough to provide each thread with a many registers (255 in recent generations [53]);

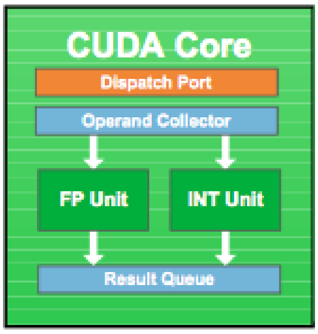


Figure 6.2: CUDA core diagram.

- Level 1 data cache, shared among all cores;
- shared memory;
- schedulers to map threads to the cores for execution;
- instruction cache shared among the schedulers;

In CUDA, a kernel represents a set of instructions to be executed as a parallel task. These parallel tasks are constituted by a set of CUDA threads, which execute the same instructions on different data (follow both SIMD and Single Instruction, Multiple Threads (SIMT) approaches). CUDA threads are organized in a hierarchy: blocks aggregate threads assigned to the same SM, and the set of all the blocks running the same kernel is a grid.

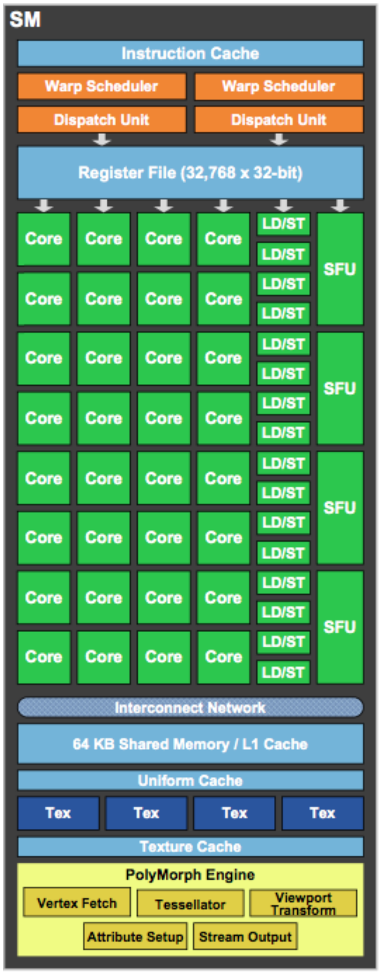


Figure 6.3: Streaming Multiprocessor diagram for the GF100 architecture.

Inside a SM, the scheduler groups up to 32 threads from the same block into warps, which are then set to run on the SM at a given time. Since warps group threads running the same instruction of the kernel at any given time, conditional jumps are very expensive. When a conditional jump is met, if divergence occurs, it causes the two conditional branches to be executed consecutively, doubling the warp execution time.

While scheduling warps for execution, the scheduler holds them in a scoreboard waiting for data and issues warps containing those ready for execution with very low switching time. For this reason, these devices benefit from having a lot more threads than those able to run concurrently, as it helps hiding the memory latency.

When accessing memory in a CUDA kernel, coalesced memory accesses are required in order to achieve an efficient memory usage. Coalesced accesses happen when the threads in a warp access global memory at the same time asking for contiguous addresses. Since the load units are able to retrieve data from memory in blocks, this results in more data being fetched with less accesses. Coalesced accesses also help the memory controller to find the best grouping of threads to merge the requests into fewer memory accesses.

GPUs implementing the G80 architecture, the first CUDA-enabled devices, had a memory bandwidth of 86.4 GB/s. On the other hand, modern GPUs using PCIe Generation 3 interface can transfer data between global memory and the system main memory at 8 GB/s in each direction (at the same time) [55]. Since communication with the CPU is so expensive it must be kept to a minimum in order to maximize performance.

6.2.1 NVIDIA Kepler Architecture

Kepler devices contain up to 15 SMX, an improved version of SM, with more and smaller cores, working at half the clock frequency. L2 cache size was doubled to 1536 KB. Each individual SMX contains up to 192 cores and 64K registers, with a maximum number of registers per thread of 255. In comparison with older architectures like Fermi, the Kepler architecture adds a new 48 KB read-only data cache and a new 32KB+32KB configuration for the L1 cache and shared memory.

Some of the new features of this architecture are mainly targeted at programming, such as the introduction of dynamic programming and Hyper-Q. Dynamic programming allows the device to generate work for itself, therefore enabling it to adapt to the amount and form of parallelism throughout the program's execution. Hyper-Q enables the cores from the same CPU to launch kernels in the same GPU. Multiple kernels launched in the same GPU will be scheduled to different SMX.

Load units in the Kepler architecture are capable of getting blocks of 256 bytes from shared memory [53].

6.3 Implementation

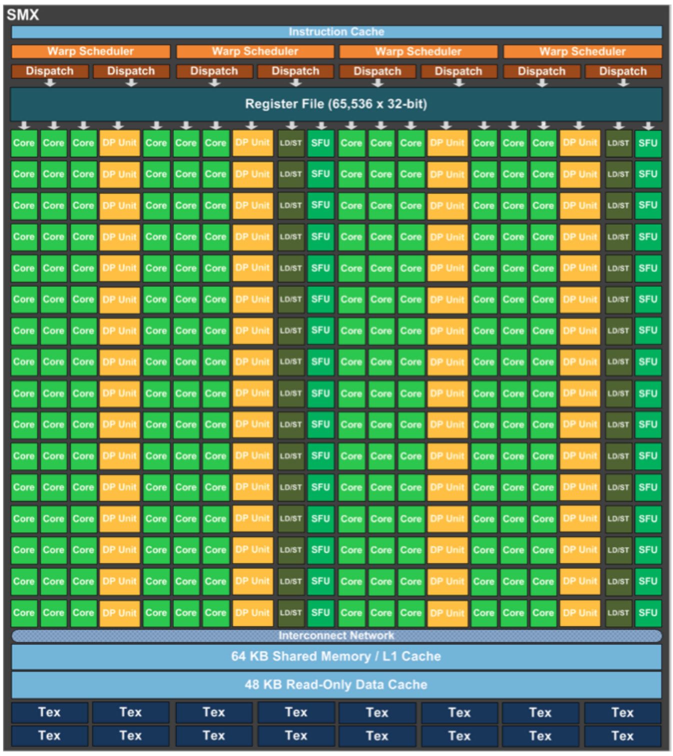
Unlike MIC devices, GPUs differ greatly from CPUs. The distinct programming model for this kind of devices requires a shift in the way the programmer thinks about the algorithm. Consequently, little of the code implemented in Chapters 4 and 5 is reusable in a CUDA implementation of the matrix square root algorithm.

First experiments have shown that the NVIDIA compiler is incompatible with recent versions of the GNU compiler, which prevents the usage of modern features in the C++ language used by the Armadillo library. Although the usage of this library was reduced to loading the matrix file and outputting the result in Section 5.4, and the incompatibility was isolated and found not to be related with these input/output operations, the Armadillo library is prepared to have all its headers used simultaneously. This very tight coupling results in having to remove any trace of the library from the CUDA implementation.

Removing Armadillo implied that the code to load the matrix files had to be ported to a compatible implementation. To ease the task, `ARMA_ASCII` was selected as the default format. This is the simplest text format in Armadillo, with the files having a small header (meant to identify the data type and the dimensions of the matrix) immediately followed by the matrix content.

Figure 6.4: Overview of the Kepler GK110 architecture.

(a) Full chip block diagram.



(b) SMX diagram.

Contrary to the solutions in the previous chapters, a CUDA implementation of this algorithm can not take advantage of optimized BLAS and LAPACK libraries. The available packages assume that its kernels will have the entire device available, and most LAPACK packages do not even implement `TRSYL`. Experience from Chapter 5 show that this is not the case since both methods contain independent parallel calls to BLAS and LAPACK functions. This implies having to reimplement each of these functions so that they can be used by all the threads in a single CUDA block.

Synchronization is also different for this implementation. In previous chapters, the parallel zones were confined to the computation of each diagonal, which were iterated over sequentially. This introduced the synchronization necessary to prevent that any diagonal were computed without its dependences being ready. In the context of a CUDA kernel, it is not possible to synchronize the entire device, consequence of blocks having to be independent. Therefore, the only way to implement this synchronization (for both methods) is to have each diagonal computed by a different kernel, at the expense of having to wait for the kernel to return from the device before launching a new one.

Kernels were implemented following an approach similar to OpenMPC by translating the regions of parallel execution (delimited by OpenMP `for` directives) in Chapters 4 and 5. This allowed for the improvements described in Section 5.4 to affect how the kernels were implemented.

Since only one diagonal of elements can be computed in parallel at any time, the point method is implemented using one-dimensional grid and blocks. This is the simplest method since linearising the indices, from both the threads and the blocks, each element in the diagonal can be computed by one thread. There is a trade-off: using one thread per element allows to take advantage of the larger parallelism among elements in the beginning, but suffers from lack of parallelism after some point since it does not concurrently solve the dependencies. On the other hand, using a whole block per element would hamper performance in the beginning and improve as the algorithm advances since it would be able to solve the dependencies in parallel. These implementations are named coarse point-diagonal (cPD) and fine point-diagonal (fPD) in Section 6.5.

With the block method this trade-off disappears and gives place to the lack of optimized BLAS and LAPACK libraries. It also introduces the possibility for using two-dimensional blocks since each block works as a standalone matrix, yet the kernels are kept with one-dimensional blocks for compatibility with the point method. Blocks in the main diagonal are solved using a single-block implementation of the point method. For the remaining diagonals, the entire thread block computes the indices (effectively isolating the required blocks for the computation) and calls the implemented single-block BLAS and LAPACK functions.

6.4 Single-block BLAS and LAPACK

For the previous chapters, MKL provided optimized routines for BLAS and LAPACK, thus extracting high efficiency from well known linear algebra operations. These packages exist for practically every language and CUDA is not an exception, with many alternatives listed by NVIDIA. However, the solutions provided in these packages for CUDA aim to use all the resources available in the device efficiently and, consequently, are not suited to be used inside another kernel.

Implementing the block method for CUDA requires using BLAS and LAPACK routines confined to the scope of a single-block. This section describes how these routines were implemented to fit the problem restrictions in the absence of an optimized alternative.

GEMM

$$C = \alpha AB + \beta C \tag{6.1}$$

The general matrix-matrix multiplication function was reimplemented based on the rowwise block-striped parallel algorithm [56, pp. 277-281] using a function signature similar to the one used by MKL. It solves Equation (6.1) by iterating over the columns in C and having each thread responsible for a row. For each column, every thread applies β to the respective element in C. It then iterates over the rows in B (or columns in A), computing the first parcel in the right side of the equation.

GEMV

$$y = \alpha Ax + y \tag{6.2}$$

This function implements the general matrix-vector multiplication. It is a simplified version of `GEMM`, iterating over the columns in A and having a thread assigned to each row. Each thread then computes the respective element in y.

TRPAISV

$$(A + \alpha I)x = b \tag{6.3}$$

`TRPAISV` does not exist implemented in any BLAS library. It is based on the triangular solve function (`TRSV`) function, which solves the equation $Ax = b$, with the small change of adding α to the elements in the main diagonal of A when these are used.

Equation (6.3) is solved by implementing the row-oriented parallel back substitution algorithm as described in [56, pp. 293-295]. The algorithm iterates backwards over the columns of A, with each row assigned to one thread. For each column c, it starts by having a single thread compute the final value of the c-th element in x (adding α), after which each thread updates its respective element in x.

To minimize memory allocations and copy operations, b is overwritten with x.

TRSYL

$$AX + XB = C \tag{6.4}$$

Lastly, this function solves the Sylvester equation (Equation (6.4)) using the Bartels-Stewart algorithm [57, pp. 367-368] (Algorithm 13). It iterates over the columns in C, with the first column calling only `TRPAISV`. The remaining ones need a call to `GEMV` before so `TRPAISV` is able to compute the final column.

Similar to what happens in `TRPAISV`, C is overwritten with X to minimize memory allocations.

Algorithm 13: Bartels-Stewart

```
input : $A$: upper triangular matrix $m \times m$
input : $B$: upper triangular matrix $n \times n$
input : $C$: square matrix $m \times n$
output: $X$: square matrix $m \times n$
for $k \leftarrow 0$ to $n - 1$ do
    $i \leftarrow$range(0,$m - 1$)
    $j \leftarrow$range(0,$k - 1$)
    $X_{ik} \leftarrow C_{ik} + C_{ij} \times B_{jk}$
    solve(($A - B_{kk}$)$X_{ik} = X_{ik}$)
end for
```

6.5 Results

This section presents the efficiency measurements performed for the described CUDA implementation to evaluate how GPUs stand in comparison with the already studied CPU-only and Intel Xeon Phi alternatives. This evaluation follows the methodology described in Section 3.3, with a block dimension of 64 (experimentally found to be the best).

Results in this section were obtained in SeARCH node 711-1 (already described in Section 5.3.1), which, in addition to the Intel Xeon Phi coprocessor, also contains a NVIDIA Tesla K20m board (details in Table 6.1). GNU Compiler Collection (GCC) 4.4.7 and CUDA 5.0 were used to build the test executables.

Although this implementation was able to profit in some measure from the experience gathered in the previous chapters, it was not able to match, much less exceed, the

GPUs	1× GK110
Multiprocessors	13× SMX
CUDA cores	192 per SMX
Double-precision units	64 per SMX
SFUs	32 per SMX
Load/Store units	32 per SMX
Memory size	5 GB
CUDA capability	3.5
Peak DP FLOPS	1.17 TeraFLOP/s
Peak Memory bandwidth	208 GB/s

Table 6.1: Hardware details for the NVIDIA Tesla K20m board in SeARCH node 711-1 (further information available in [53, 58]).

performance obtained in a multicore environment, even before the optimization techniques described in Section 5.4. Figure 6.6 shows the execution times achieved by the three CUDA implementations (cPD, fPD and BD) in comparison with the fully optimized multicore implementation. The superiority of multicore is clear, being around 15 times faster than the best result for $n = 8000$ in the GPU.

However, there are several reasons for this to happen. First, the multicore implementation has been the target of several optimizations in this document, while this is an initial completely functional CUDA implementation. Given the time already invested in the Intel Xeon Phi coprocessor, it is not feasible to perform optimizations targeting the CUDA environment. Next, the absence of optimized BLAS and LAPACK packages suiting the needs of the implementation. MKL provides it for both multicore environment and coprocessor, while manual (also naive) implementations had to be devised for CUDA. Lastly, the synchronization requirement between diagonals forces the runtime to return to the CPU between two consecutive diagonals, creating a bottleneck when the amount of elements/blocks no longer provides enough parallelism to overcome the communication cost.

What comes as a surprise in these results is the fact that the fine point-diagonal (cPD) implementation achieved a higher efficiency than the block alternative. The increased parallelism significantly compensates for the less efficient cache usage of the point method.

6.6 Further Optimizations

While further optimizations were not implemented using CUDA, it is still relevant to study how this implementation could be improved.

NVIDIA's command-line profiler `nvprof` allows to profile the application in the same environment as it was tested, exporting the results for visualization using the Visual Profiler. In its turn, this tool allows to perform a Guided GPU utilization analysis, which reveals a lack of overlapping in operations.

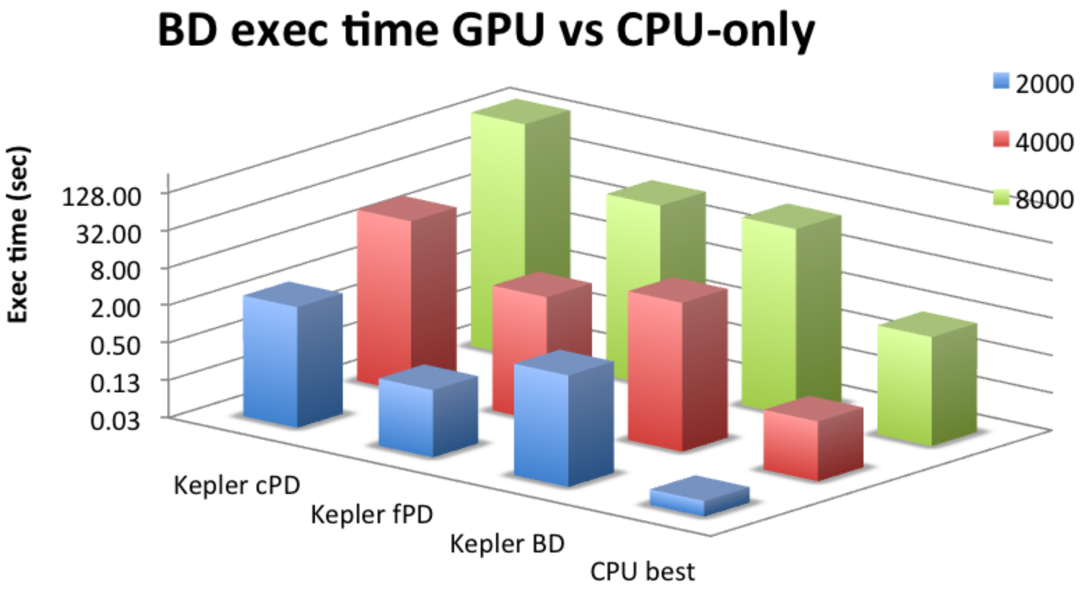


Figure 6.6: Execution times for the CUDA implementation in a Tesla K20m (Kepler architecture).

6.6.1 Page-Locked Host Memory

Page-locked memory, also known as pinned memory, has the important property of never being paged out to the disk. This means that the operating system can safely allow for an application to access the physical address of the memory required.

Knowing the physical address of a buffer in the host memory allows for a GPU to use Direct Memory Access (DMA) to copy data to or from the host. DMA allows these transfers to be performed without intervention from the CPU, which in turn leaves it free to be paging out these buffers or relocating their physical address. When a memory copy is performed using pageable memory, the CUDA driver first copies the data to a page-locked "staging" buffer, and then it performs the copy from that buffer to the GPU using DMA [59].

Yet, it has the consequence of disabling virtual memory for those pages in the host memory. This would cause the host to run out of available memory much faster, not only failing in machines with smaller amounts of memory but also affecting the performance of other applications in the same system. Fortunately, none of these issues would be problematic in the system used for performance tests due to the very large amount of memory and the care for not having any other application running on the system at the same time to minimize interferences with time measurements.

In order to use page-locked memory, the CUDA runtime offers the function `cudaHostAlloc`, which is meant to replace the standard C library routine `malloc`. It also offers the respective replacement for `free` as `cudaFreeHost`.

Although changing an application to use page-locked host memory should only require replacing the routines for memory allocation, using C++ objects it becomes more complicated. In particular, two different objects are used in the initialization of the implemented program. First, a matrix object loads the content of the specified file, being responsible for reimplementing I/O operations compatible with Armadillo. Second, a CUDA array object is in charge of performing the required memory allocations in the device, copy operations between it and the host and cleaning up the allocated resources on destruction, abstracting the original CUDA API with a friendlier C++ version.

Changing the CUDA implementation described in this chapter to use page-locked host memory would require merging these two objects. Upon loading the matrix from file, the memory on the host would have to be allocated using the routine for page-locked memory. At this time, it would have to allocate the memory in the device. This object would also have to be responsible for freeing all the allocated memory, both for the host and the device, on destruction. Memory transfers between the device and the host would have to be performed upon request.

6.6.2 Streams

CUDA streams represent queues of GPU operations, such as kernel launches and memory copies, that get executed in the same order they are added. Streams work with asynchronous operations, which in case of memory transfers requires the host memory to be page-locked. This allows for the operations to be executed by the device without interference from the CPU, thus reducing the time between successive kernel launches. They also behave like tasks on a CPU, allowing for parallelism by overlapping operations in different streams, restricted by the resources available in the device. For example, it is possible to overlap two memory copies between the host and the device, one in each direction, while also computing a kernel.

Extending the existing implementation to use streams would first require for the page-locked optimization to be implemented. After this, copying the next diagonal to be computed to the device could be overlaped with copying the last computed diagonal back to the host. At the same time, the device would be computing the current diagonal.

This optimization would maximize the overlap of operations in the device, thus increasing its efficiency.

7 Conclusions

This dissertation was focused on the main goal of achieving an efficient implementation of the matrix square root algorithm using hardware accelerators in an heterogeneous platform. Three cases were studied with this goal in mind. The first, a port of the algorithm already described in a previous work running in a shared-memory multicore environment, increased the familiarity with the algorithm and served as the base for the other implementations. The second, targeted for devices of the Intel MIC architecture, allowed to put the new Intel coprocessors to the test, as well as its programming model. The third, an implementation targeted for CUDA enabled NVIDIA GPUs, put the algorithm to the test using the most popular accelerator device nowadays.

The multicore implementation not only validated the results presented in [13], it also allowed to conclude that the algorithm has a near perfect scalability with the explicit parallelism of the diagonal strategy, despite its lack of locality. Additionally, the block method was found to eliminate a cache resonance effect triggered specifically by power-of-two matrix dimensions.

Porting the multicore implementation to the Intel Xeon Phi coprocessor was confirmed to be trivial, although such was found not be the case with achieving high performance using these devices. The similar programming models inspire to similar practices, but the truth is that it requires a distinct way of thinking, a lot more targeted for vectorization than what is required when programming for CPUs.

The weak operating system in the coprocessor also forces an adaptation of the methodology since it does not support any major featured script language. This is overcome by running the programs through a remote session that breaks all the automatic mechanisms previously prepared to aid in running in collecting all the required data. Consequently, the time required for performing performance tests greatly increased, preventing further work due to time constraints.

Initial results using the Intel Xeon Phi shown that the multicore code, although functional, is less efficient in the coprocessor. Following the recommendations in Intel documentation stating that optimizations should be focused initially on the CPU implementation, the code was profiled in the development environment using Intel VTune Amplifier XE 2013 and performance was found to be hurt by the usage of the Armadillo library. Five optimization techniques were applied with the purpose of achieving higher performance in the Intel Xeon Phi coprocessor: (a) massive parallelism; (b) loop unrolling; (c) replacement of Armadillo;

(d) reorganization of the matrices by blocks; (e) and replacement of the output matrix with the input overwrite.

Massive parallelism was found by solving elements/blocks dependencies in parallel, which is theoretically able to compensate for the decreasing parallelism as the number of elements/blocks per diagonal decreases with the progress of the algorithm. However, the obtained results did not show any improvements from parallelizing the dependency solving step, proof of how the lack of locality in this strategy prevents the algorithm from achieving higher efficiency.

Obtained results also showed the absence of significant improvement with loop unrolling, proving that conditional branching is not as harmful in the coprocessor as it is in a GPU.

The Armadillo library was found to be very useful during for the multicore implementation, significantly reducing the development time. However, despite being planned for HPC it still lacks mechanisms to avoid extra memory allocations and copies in some situations. For this reason, its usage had to be limited to I/O operations.

The two last optimizations, both based on an implementation without Armadillo, shown significant improvements, especially together. The accumulated speedup of these optimizations reduced the execution time for the larger matrix dimensions to less than half. Nevertheless, since these optimizations also apply to the multicore environment, the performance achieved in the coprocessor did not reach the values of a CPU implementation, despite reducing the gap significantly. Further optimizations were not pursued to allow for a CUDA implementation within the time constraints.

Reimplementing the algorithm for CUDA-enabled devices proved to be the most tricky. Incompatibilities with the compiler and a tight coupling in the library prevented the already minimal presence of Armadillo, forcing to a reimplementation of the I/O operations. As for the development of the algorithm itself, the massive parallelism explicitly made available by the CUDA programming model significantly eased up the expression of the algorithm parallelism, despite the paradigm change. However, the absence of BLAS and LAPACK packages targeted to be used inside a single block of threads implied that these routines had to be manually implemented.

Results obtained with (naive) CUDA implementations also did not match the performance of the multicore environment, but they showed that a point method implementation using the the massive parallelism (unveiled in Section 5.4.1) is able to surpass the efficiency obtained by the better cache usage of the block method. The absence of time to proceed optimizing this implementation left very promising paths unexplored.

Finally, even not having achieved higher performance using any of the hardware accelerators studied during this dissertation, the CUDA implementation may be used as a way for the processor to delegate part of its workload, allowing it to be used for other tasks. The same is true for the coprocessor implementation, although it would require changing the execution mode to offload, something that is not expected to be complex.

7.1 Future Work

As is typical in research projects, several paths, either available from the start or unveiled with the progress, were not taken during this dissertation due its natural time constraints. Those paths, left for future work, are described in this section.

First, all the implementations presented in this document could be integrated in a single software package (such as BLAS library) supporting the three studied environments. In order for such a package to be useful though, the restrictions imposed by the assumptions presented in Section 4.1 would have to be lifted by extending the implementations, both for complex arithmetic and to allow for quasi-triangular matrices, and by adding the Schur decomposition to allow for any matrix to be used.

Given the success of the results obtained with the multicore implementation and, in contrast, the lack of performance found in the implementations for the two hardware accelerators, an MPI implementation of the algorithm might prove itself more efficient. Distributing the matrix through the available nodes, having each compute part of a diagonal using the multicore implementation and communicate only that part to the remaining nodes, and reducing the number of nodes involved gradually (as the size of the diagonal decreases) has the potential to replicate the results obtained with the benefits of increasing the parallelism through a HetPlat.

Alternatively, using only one computational node for iterating over the diagonals but having the remaining nodes cooperating in the computation of the dependencies could also prove to be efficient, but it would more complex. An hybrid solution would also be interesting, by having less nodes computing the diagonal as the algorithm progresses but having the increasing idle nodes cooperating to solve the dependencies.

As for the implementation using Intel MIC devices, further profiling, now using the command-line tool for Intel VTune Amplifier, would reveal why the achieved performance was not able to surpass the multicore implementation. In particular, it would be interesting to use hardware events to examine how memory is used by the algorithm. If confirmed to be the bottleneck, a diagonalized rearrangement of the matrix could make the algorithm more efficient, at the expense of a preparation and cleanup step that, unlike what happens with the USB format, would almost certainly not be useful for any other linear algebra routines.

Regarding both Chapters 4 and 5, the importance given to thread affinity in [60] shows that performing experiments with thread affinity properly defined would be interesting enough to make it a priority. The balanced thread affinity policy with Intel's OpenMP library is expected to be the decisive step to take the implementation to equivalent levels in both Xeon processors and Intel Xeon Phi coprocessors.

Given that there was only opportunity to explore the native execution mode of the Intel

Xeon Phi coprocessor, it would be interesting to explore offload in the future. Similarly, if an MPI implementation proves to be efficient the question remains whether using the coprocessor, either as another node or as an offload device, improves efficiency. The usage models for Intel MKL are also intriguing unexplored paths. The compiler assisted offload, in particular, due to the advantage of allowing for data persistence, is expected to improve performance because the device would be focused entirely on executing routines already optimized for it. On the other hand, the multiple parallel calls to these routines from the host when computing an entire diagonal of blocks in parallel could cause most of the work to be performed in host due to the unavailability of resources in the device, thus reducing the advantage of using the coprocessor.

Specifically for the CUDA implementation, the optimizations described in Section 6.6 are left for future work due since these would require the reimplementation of some components, which was not viable within the time constraints. Additionally, an optimized BLAS package for single-block to replace the routines implemented in Section 6.4 would improve the efficiency of the implementations using GPUs. Although these routines were implemented with performance in mind, there was no opportunity for deep profiling and improvements.

Lastly, for both devices studied during this dissertation, implementations executing in the CPUs and the accelerator at the same time would hardly achieve higher speedups because of the synchronization required between the diagonals. Nevertheless, an hybrid implementation starting in the device and moving to the CPU when the algorithm lacks the required parallelism could merge the best performance of both worlds. Alternatively, if offloading only the BLAS and LAPACK routines proved to be efficient using the Intel Xeon Phi coprocessor, NVIDIA GPUs could use the same model, allowing for already existing optimized packages to be used [7, 9, 10].

Bibliography

[1] G.E. Moore. “Cramming More Components Onto Integrated Circuits”. In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85. ISSN: 0018-9219. DOI: `10.1109/JPROC.1998.658762`. URL: `http://www.cs.utexas.edu/~fussell/courses/cs352h/papers/moore.pdf`.

[2] G.E. Moore. “Progress in digital integrated electronics”. In: *Electron Devices Meeting, 1975 International*. Vol. 21. 1975, pp. 11–13. URL: `http://www.eng.auburn.edu/~agrawvd/COURSE/E7770_Spr07/READ/Gordon_Moore_1975_Speech.pdf`.

[3] *1965 - "Moore's Law" Predicts the Future of Integrated Circuits*. URL: `http://www.computerhistory.org/semiconductor/timeline/1965-Moore.html` (visited on Aug. 13, 2013).

[4] The Numerical Algorithms Group. *NAG Numerical Components*. URL: `http://www.nag.co.uk` (visited on Aug. 31, 2013).

[5] The Numerical Algorithms Group. *NAG Numerical Routines for GPUs Manual*. Apr. 2012. URL: `http://www.nag.com/numeric/GPUs/naggpu_doc_0.6.pdf`.

[6] The Numerical Algorithms Group. *NAG Library Manual, Mark 23*. Feb. 2011. ISBN: 978-1-85206-209-5. URL: `http://www.nag.com/numeric/fl/nagdoc_Intel_MIC_FS23.3/xhtml/FRONTMATTER/manconts.xml` (visited on Aug. 31, 2013).

[7] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. “Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects”. In: *Journal of Physics: Conference Series* 180.1 (2009).

[8] AccelerEyes. *SQRTM - Jacket Wiki*. URL: `http://wiki.accelereyes.com/wiki/index.php/SQRTM` (visited on Aug. 31, 2013).

[9] EM Photonics. *CULAPACK Function List*. URL: `http://www.culatools.com/dense/lapack/` (visited on Aug. 31, 2013).

[10] NVIDIA. *cuBLAS Library - User Guide*. Version 5.0. 2012-10. URL: `http://docs.nvidia.com/cuda/pdf/CUDA_CUBLAS_Users_Guide.pdf`.

[11] NVIDIA. *cuSPARSE Library - User Guide*. Version 5.0. Oct. 2012. URL: `http://docs.nvidia.com/cuda/pdf/CUDA_CUSPARSE_Users_Guide.pdf`.

[12] N. Bell and M. Garland. *Cusp Library Features*. Version 0.3.0. Mar. 2012. URL: `http://code.google.com/p/cusp-library/wiki/Features`.

[13] Edvin Deadman, Nicholas J. Higham, and Rui Ralha. “Blocked Schur Algorithms for Computing the Matrix Square Root”. In: *Applied Parallel and Scientific Computing: 11th International Conference, PARA 2012, Helsinki, Finland*. Ed. by P. Manninen and P. Öster. Vol. 7782. Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2013, pp. 171–182. DOI: `10.1007/978-3-642-36803-5_12`.

[14] Åke Björck and Sven Hammarling. “A Schur method for the square root of a matrix”. In: *Linear Algebra and its Applications* 52–53 (1983), pp. 127–140.

[15] Nicholas J. Higham. *Functions of Matrices: Theory and Computation*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2008. ISBN: 978-0-898716-46-7.

[16] John L. Gustafson. “Reevaluating Amdahl’s law”. In: *Commun. ACM* 31.5 (May 1988), pp. 532–533. ISSN: 0001-0782. DOI: `10.1145/42411.42415`. URL: `http://doi.acm.org/10.1145/42411.42415`.

[17] Erich Strohmaier. *20 Years Supercomputer Market Analysis*. May 2005. URL: `http://tclark.ittc.ku.edu/eecs739fall2007/papers/strohmaierSC2005.pdf` (visited on Aug. 31, 2013).

[18] National Security Agency, ed. *The Next Wave*. Vol. 20. 1. 2013. URL: `http://www.nsa.gov/research/tnw/tnw201/articles/pdfs/TNW_20_1_Web.pdf`.

[19] *Development over Time*. Aug. 2013. URL: `http://www.top500.org/statistics/overtime/`.

[20] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X, 9780123838728.

[21] Rob Farber. “Redefining What is Possible”. In: *Scientific Computing* (Jan. 2011). URL: `http://www.scientificcomputing.com/printpdf/articles/2011/01/redefining-what-possible` (visited on Aug. 31, 2013).

[22] Francisco D. Igual, Murtaza Ali, Arnon Friedmann, Eric Stotzer, Timothy Wentz, and Robert van de Geijn. *Unleashing DSPs for General-Purpose HPC. FLAME Working Note #61*. Technical Report TR-12-02. The University of Texas at Austin, Department of Computer Sciences, Feb. 2012.

[23] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. “State-of-the-art in heterogeneous computing”. In: *Sci. Program.* 18.1 (Jan. 2010), pp. 1–33. ISSN: 1058-9244. URL: `http://dl.acm.org/citation.cfm?id=1804799.1804800`.

[24] AMD. *Introduction to "Magny-Cours"*. Aug. 2013. URL: `http://developer.amd.com/resources/documentation-articles/articles-whitepapers/introduction-to-magny-cours/`.

[25] Blaise Barney. *POSIX Threads Programming*. Ed. by Lawrence Livermore National Laboratory. URL: `https://computing.llnl.gov/tutorials/pthreads/`.

[26] OpenMP Architecture Review Board, ed. *About the OpenMP ARB and OpenMP.org*. URL: `http://openmp.org/wp/about-openmp/`.

[27] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. Version 3.1. July 2011. URL: `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`.

[28] Andrew Binstock. *Threading Models for High-Performance Computing: Pthreads or OpenMP?* Ed. by Intel Corporation. URL: `http://software.intel.com/en-us/articles/threading-models-for-high-performance-computing-pthreads-or-openmp`.

[29] Intel. *Intel Threading Building Blocks Documentation*. URL: `http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm`.

[30] Seyong Lee and Rudolf Eigenmann. "OpenMPC: Extended OpenMP Programming and Tuning for GPUs". In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. ISBN: 978-1-4244-7559-9. DOI: `10.1109/SC.2010.36`. URL: `http://dx.doi.org/10.1109/SC.2010.36`.

[31] Michael Wolfe. *The OpenACC Application Programming Interface*. Version 2.0. June 2013. URL: `http://www.openacc.org/sites/default/files/OpenACC%202%200.pdf`.

[32] Nicholas J. Higham. *The Matrix Function Toolbox*. URL: `http://www.ma.man.ac.uk/~higham/mftoolbox`.

[33] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. "The cache performance and optimizations of blocked algorithms". In: *SIGPLAN Not.* 26.4 (Apr. 1991), pp. 63–74. ISSN: 0362-1340. DOI: `10.1145/106973.106981`. URL: `http://doi.acm.org/10.1145/106973.106981`.

[34] Rajib Nath, Stanimire Tomov, and Jack Dongarra. "An Improved Magma Gemm For Fermi Graphics Processing Units". In: *Int. J. High Perform. Comput. Appl.* 24.4 (Nov. 2010), pp. 511–515. ISSN: 1094-3420. DOI: `10.1177/1094342010385729`. URL: `http://dx.doi.org/10.1177/1094342010385729`.

[35] Wendy Doerner. *Cache Blocking Techniques*. Ed. by Intel Corporation. Aug. 23, 2012. URL: `http://software.intel.com/en-us/articles/cache-blocking-techniques` (visited on Aug. 31, 2013).

[36] B.N. Parlett. "A recurrence among the elements of functions of triangular matrices". In: *Linear Algebra and its Applications* 14.2 (1976), pp. 117–121. ISSN: 0024-3795. DOI: http://dx.doi.org/10.1016/0024-3795(76)90018-5. URL: http://www.sciencedirect.com/science/article/pii/0024379576900185.

[37] Isak Jonsson and Bo Kågström. "Recursive blocked algorithms for solving triangular systems – Part I: one-sided and coupled Sylvester-type matrix equations". In: *ACM Trans. Math. Softw.* 28.4 (Dec. 2002), pp. 392–415.

[38] IBM. *FORTRAN – The Pioneering Programming Language*. URL: http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/fortran/ (visited on Aug. 31, 2013).

[39] Nicholas J. Higham. "Computing real square roots of a real matrix". In: *Linear Algebra and its Applications* 88-89 (Apr. 1987), pp. 405–430. ISSN: 0024-3795. DOI: 10.1016/0024-3795(87)90118-2. URL: www.maths.manchester.ac.uk/nareports/narep89.pdf.

[40] Intel Corporation, ed. *Intel Xeon Processor E5-2650*. URL: http://ark.intel.com/products/64590/ (visited on Aug. 31, 2013).

[41] Intel Corporation, ed. *Intel® Xeon® Processor E5-1600/ E5-2600/E5-4600 Product Families*. May 2012. URL: http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-1600-2600-vol-1-datasheet.pdf.

[42] Cleve Moler. *MATLAB Incorporates LAPACK*. Ed. by MathWorks. 2000. URL: http://www.mathworks.com/company/newsletters/articles/matlab-incorporates-lapack.html (visited on Aug. 31, 2013).

[43] Sudha U. Thiagarajan, Charles Congdon, Sumedh Naik, and Loc Q. Nguyen. *Intel® Xeon Phi^{TM} Coprocessor Developer's Quick Start Guide*. Version 1.5. Dec. 2012. URL: http://software.intel.com/sites/default/files/article/335818/intel-xeon-phi-coprocessor-quick-start-developers-guide.pdf.

[44] Intel Corporation, ed. *Intel® Xeon Phi^{TM} Coprocessor*. June 2013. URL: http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-phi-coprocessor-datasheet.pdf.

[45] James Reinders. *An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors*. Ed. by Intel Corporation. 2012. URL: http://software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors.pdf.

[46] Intel Corporation, ed. *Introducing the Intel® Xeon Phi^{TM} Coprocessor. Architecture for Discovery*. URL: http://www.intel.com/content/dam/www/public/us/en/documents/presentation/xeon-phi-architecture-for-discovery-presentation.pdf.

[47] Intel Corporation, ed. *Intel Xeon Processor E5-2670.* URL: `http://ark.intel.com/products/64595/` (visited on Aug. 31, 2013).

[48] Intel Corporation, ed. *Intel Xeon Phi Coprocessor 5110P.* URL: `http://ark.intel.com/products/71992/Intel-Xeon-Phi-Coprocessor-5110P-8GB-1_053-GHz-60-core`.

[49] Gene M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference.* AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: `10.1145/1465482.1465560`. URL: `http://doi.acm.org/10.1145/1465482.1465560`.

[50] Michaela Barth, Mikko Byckling, Nevena Ilieva, Sami Saarinen, and Michael Schliephake. *Best Practice Guide Intel Xeon Phi v0.1.* Ed. by Volker Weinberg. Mar. 2013. URL: `http://www.prace-project.eu/IMG/pdf/Best-Practice-Guide-Intel-Xeon-Phi.pdf`.

[51] José Carlos Mouriño Gallego, Carmen Cotelo Queijo, Andrés Gómez Tato, and Aurelio Rodríguez López. *Evaluation of Intel® Xeon Phi^{TM} to execute easily scientific applications.* Tech. rep. July 2013. URL: `https://www.cesga.es/es/biblioteca/downloadAsset/id/732`.

[52] NVIDIA. *What is GPU Computing?* URL: `http://www.nvidia.com/object/what-is-gpu-computing.html` (visited on Aug. 31, 2013).

[53] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110.* Tech. rep. 2012.

[54] NVIDIA. *NVIDIA GeForce GTX 680.* Tech. rep. 2012. URL: `http://international.download.nvidia.com/webassets/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf`.

[55] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach.* 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012-12. ISBN: 0124159923, 9780124159921.

[56] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP.* Ed. by McGraw-Hill. International. ISBN: 007-123265-6.

[57] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* Ed. by The Johns Hopkins University Press. Third. ISBN: 0-8018-5414-8.

[58] NVIDIA, ed. *Tesla Kepler GPU Accelerators.* URL: `http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf`.

[59] Jason Sanders and Edward Kandrot. *CUDA by Example. An Introduction to General-Purpose GPU Programming*. Ed. by NVIDIA Corporation. 2011. ISBN: 978-0-13-138768-3, 0-13-138768-5.

[60] Jim Jeffers and James Reinders. *IntelXeon PhiCoprocessor High Performance Programming*. 2013. ISBN: 978-0-12-410414-3.