UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

JOÃO PAULO TARASCONI RUSCHEL

Parallel Implementations of the Cholesky Decomposition on CPUs and GPUs

Work presented in partial fulfillment of the requirements for the degree of Bachelor in Computer Science

Advisor: Prof. Dr. Philippe Olivier Alexandre Navaux
Coadvisor: Dr. Matthias Diener

Porto Alegre
2016

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitor: Profª. Jane Fraga Tutikian
Pró-Reitor de Graduação: Prof. Vladimir Pinheiro do Nascimento
Diretor do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Ciência de Computação: Prof. Sérgio Luis Cechin
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

"Ryuu ga waga teki wo kurau!"

— SHIMADA, HANZO

ABSTRACT

As Central Processing Units (CPUs) and Graphical Processing Units (GPUs) get progressively better, different approaches and designs for implementing algorithms with high data load must be studied and compared. This work compares several different algorithm designs and parallelization APIs (such as OpenMP, OpenCL and CUDA) for both CPU and GPU platforms. We used the Cholesky decomposition, a high-level arithmetic algorithm used in many linear algebra problems, as the benchmarking algorithm, due to being easily parallelizable, and having a considerable data dependence between elements. We carried out various experiments using the different designs and APIs in order to find the techniques which yield the best performance for each platform. We also compared these implementations with state-of-the-art solutions (such as LAPACK and cuSOLVER), and provided insights into the differences in implementation and performance. Our experiments showed us that parallelization on CPU tends to have a better performance than on GPU for this particular kind of algorithm, due to the intrinsic memory-intensive nature of the algorithm and memory transfer overhead, and that attempts at code micro-optimization do not offer any significant speedup.

Keywords: HPC, parallel programming, OpenMP, OpenCL, CUDA, CPU, GPU, Cholesky.

Implementações Paralelas da Decomposição de Cholesky em CPU e GPU

RESUMO

À medida que Unidades Centrais de Processamento (CPUs) e Gráfico (GPUs) evoluem progressivamente, diferentes abordagens e modelos para implementação de algoritmos com alta carga de dados devem ser estudados e comparados. Este trabalho compara diversos modelos de algoritmos e APIs de paralelização (como OpenMP, OpenCL e CUDA) para as plataformas CPU e GPU. Nós usamos a decomposição de Cholesky, um algoritmo aritmético de alto nível usado em diversos problemas de álgebra linear, como referência, devido a sua fácil paralelização, bem como apresentar alta dependência de dados entre os elementos. Diversos experimentos foram realizados, utilizando os diferentes modelos e APIs a fim de encontrar as técnicas que fornecem a melhor performance em cada plataforma. Também comparamos essas implementações com soluções profissionais (como LAPACK e cuSOLVER), examinando as discrepâncias de implementação e performance. Os experimentos demonstram que a CPU tende a ter melhor performance que a GPU para esse tipo de algoritmo, devido à sua natureza intensiva em memória e o overhead intrínseco da transferência de dados, e que tentativas de micro-otimizações de código não oferecem nenhuma melhora de performance.

Palavras-chave: HPC, programação paralela, OpenMP, OpenCL, CUDA, CPU, GPU, Cholesky.

LIST OF FIGURES

Figure 4.1 Visualizing Cholesky algorithm's data dependencies on a single row. 31
Figure 4.2 Visualizing Cholesky algorithm's data dependencies on a single column. 32
Figure 4.3 Differences between row-wise and column-wise indexing. 34
Figure 4.4 Visualizing transposed Cholesky-Crout iteration. 35
Figure 4.5 Visualizing independence on Cholesky-Crout iterations. 37
Figure 4.6 Visualizing write operations on the Cholesky-Crout algorithm 39

Figure 5.1 OpenMP implementations experiment results for cutoffs. 49
Figure 5.2 OpenMP implementations experiment results for speedup. 50
Figure 5.3 OpenMP implementations experiment results for the different implementations. 51
Figure 5.4 OpenCL (on CPU) implementations experiment results for cutoffs. . 52
Figure 5.5 OpenCL (on CPU) implementations experiment results for the different implementations. 53
Figure 5.6 Comparing speedups for OpenMP, OpenCL (CPU) and LAPACK implementations. 54
Figure 5.7 Comparing overall performance of the best CPU parallelization implementations with a sequential execution. 55
Figure 5.8 OpenCL (on GPU) implementations experiment results for cutoffs. . 57
Figure 5.9 OpenCL (on GPU) implementations experiment results for all implementations. 58
Figure 5.10 CUDA implementations experiment results for different cutoffs. ... 59
Figure 5.11 CUDA implementations experiment results for different TPBs. 60
Figure 5.12 CUDA implementations experiment results for all versions. 61
Figure 5.13 Comparing overall performance of the best GPU parallelization implementations with a sequential execution. 62
Figure 5.14 Performance of the best CPU and GPU implementations, Sequential, LAPACK and cuSOLVER. 64

LIST OF TABLES

Table 4.1 Experiment results for sequential versions when size is 2500. 36
Table 5.1 Experimental Unit Description 48

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
LAPACK	Linear Algebra Package
MKL	Math Kernel Library
OpenCL	(OCL) Open Computing Language
OpenMP	(OMP) Open Multi-Processing
RAM	Random Access Memory
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Threads
VRAM	Video Random Access Memory

CONTENTS

1 INTRODUCTION **10**
1.1 The Cholesky Decomposition **10**
1.2 Motivation **12**
1.3 Objectives **13**
2 CONCEPTS AND TOOLS **14**
2.1 Parallel Processing **14**
2.1.1 CPU Parallelization 16
2.1.2 GPU Parallelization 17
2.1.3 Evaluating Performance 18
2.2 Memory and Caches **19**
2.3 Platforms and APIs **19**
2.3.1 OpenMP 20
2.3.2 OpenCL 22
2.3.3 CUDA 24
3 STATE OF THE ART IMPLEMENTATIONS **25**
3.1 LAPACK **25**
3.2 cuSOLVER **26**
3.3 Related work **26**
4 DEVELOPED IMPLEMENTATIONS **27**
4.1 Sequential Implementations **27**
4.2 General Optimizations **33**
4.3 CPU Parallelization with OpenMP **36**
4.4 CPU and GPU Parallelization with OpenCL **39**
4.5 GPU Parallelization with CUDA **43**
4.6 Using LAPACK **45**
4.7 Using cuSOLVER **45**
5 EVALUATION **46**
5.1 Experiment Design **46**
5.2 Test Machine **47**
5.3 Performance Analysis for CPU Parallelizations **48**
5.3.1 OpenMP 49
5.3.2 OpenCL 52
5.3.3 Sequential x OpenMP x OpenCL x LAPACK 54
5.4 Performance Analysis for GPU Parallelizations **56**
5.4.1 OpenCL 57
5.4.2 CUDA 59
5.4.3 Sequential x OpenCL x CUDA x cuSOLVER 62
5.5 Overall Performance Analysis **64**
6 CONCLUSION **66**
REFERENCES **68**
APPENDIX A — CODE LISTINGS **70**

1 INTRODUCTION

In order to achieve the highest performance during the execution of any algorithm, software engineers need to be equipped with the best possible tools (libraries, APIs and platforms). However, it is not always trivial to determine which ones are the ideal to use in a given situation, therefore making studies that focus on the analysis of such tools instrumental for the development of better and faster software.

The first step is understanding the problem that must be solved. This section will focus on the Cholesky decomposition, its applications and why it was used as benchmarking algorithm, as well as explain what we hope to achieve with this work.

1.1 The Cholesky Decomposition

The Cholesky decomposition (also called Cholesky Factorization) is a well-known linear algebra method for matrix decomposition. Discovered by André-Louis Cholesky, it states that any matrix that is symmetric and positive-definite (all *eigenvalues* are positive) can be decomposed as:

$$A = LL^{\intercal} \tag{1.1}$$

Where A is the input matrix, L is a lower triangular matrix, and $L^{\intercal}$ is its transpose. As an example, consider the following 5x5 matrix A:

$$A = \begin{bmatrix} 29 & 5 & 9 & 5 & 6 \\ 5 & 29 & 10 & 8 & 7 \\ 9 & 10 & 23 & 4 & 5 \\ 5 & 8 & 4 & 26 & 6 \\ 6 & 7 & 5 & 6 & 30 \end{bmatrix}$$

The Cholesky decomposition L for A is (rounded to 2 decimal places):

$$L = \begin{bmatrix} 5.39 & 0 & 0 & 0 & 0 \\ 0.93 & 5.3 & 0 & 0 & 0 \\ 1.67 & 1.59 & 4.2 & 0 & 0 \\ 0.93 & 1.35 & 0.07 & 4.83 & 0 \\ 1.11 & 1.12 & 0.32 & 0.71 & 5.19 \end{bmatrix}$$

Note that every symmetric positive-definite matrix has a unique Cholesky decomposition, and all diagonal values of L are positive.

This concept can be extrapolated to complex matrix as well, by requiring an Hermitian matrix as input, and decomposing it to a lower triangular matrix and its conjugate matrix. However, for this work, only real values were considered.

The following formula defines how an element i, j of the matrix L can be calculated:

$$L_{ij} = \begin{cases} 0 & \text{if } i < j \\ \sqrt{A_{jj} - \sum_{k=1}^{j-1} L_{jk}^2} & \text{if } i = j \\ \frac{1}{L_{jj}} \left(A_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{jk} \right) & \text{if } i > j \end{cases} \tag{1.2}$$

There is ongoing research on different implementations of the Cholesky decomposition, as well as other, modified and complex versions that consider different constraints (FANG; O'LEARY, 2008). However, this work focus primarily on three algorithms:

- **The Cholesky Algorithm** is a recursive, modified version of Gaussian

elimination;

- **The Cholesky-Banachiewicz algorithm** starts from the upper left

corner, and iterates the matrix row by row;

- **The Cholesky-Crout algorithm** starts from the upper left corner,

and iterates the matrix column by column.

The Cholesky decomposition is a tool that can be used on various linear algebra applications, and is available on most numerical computing software (such as Matlab (MATHWORKS, 2016)). Such applications include solving of linear equations (SMITH, 2001) and Monte-Carlo simulations (KROESE; TAIMRE;

BOTEV, 2013) that, although do not require the Cholesky decomposition to be solved (as other methods for solving them exist), benefit greatly from such decomposition, where applicable.

As an example, we can look into how the Cholesky factorization can be used on the solving of linear systems. A linear system $Ax = b$ can efficiently be solved by first decomposing the matrix A into $LL^\intercal$. then solving $Ly = b$ (for y) by forward substitution, and then $L^\intercal x = y$ (for x) by back substitution (RAKOTONIRINA, 2009).

It must be noted, however, that the Cholesky decomposition may not always be used, because, as stated before, the input matrix must be symmetric and positive-definite. That being said, it is possible to use the LU decomposition (a similar factorization method, which factors a matrix into the product of one lower triangular and one upper triangular matrix) for some of the algorithms that would use the Cholesky decomposition. These cases (and the LU decomposition itself) were not considered for this work, and the input matrices used for experimentation were all valid.

1.2 Motivation

The main motivation for this work is to research and benchmark different parallelization and optimization techniques for different platforms, and, by comparing those results to other solutions, expand into the reasons of any differences found, with the hopes of developing better and faster software.

The motivation behind choosing the Cholesky decomposition is twofold: firstly, the algorithms for solving this problem have an interesting data dependence, good for benchmarking; and secondly, as mentioned on the previous item, the method is used to optimize numerically (by decreasing the actual number of calculations needed to be done) various linear algebra problems, which means that when used, a Cholesky decomposition algorithm must be as fast as possible. For this reason, finding faster ways to implement this algorithm is of great importance on other, bigger problems.

1.3 Objectives

The goal of this work is to present concrete benchmark results of different libraries, APIs, platforms, and implementation techniques for the matrix decomposition problem, and further analysis into the ways that such tools improve (or not) the efficiency of the implementation for similar problems. By doing so, we hope to provide useful information regarding which platform, technique or algorithm design should developers use in order to achieve the maximum performance on their implementations.

2 CONCEPTS AND TOOLS

Achieving the best possible performance on a system for a given problem is not an easy task, and it involves several areas of computer science. In this section, we will cover some of these areas, as well as some of the most popular high-performance tools.

2.1 Parallel Processing

Parallel processing is the simultaneous processing of data for a single problem, and its main purpose was best summed by Joseph F. JaJa (JAJA, 1992):

> "the main purpose of parallel processing is to perform computations faster than can be done with a single processor by using a number of processors concurrently".

This work will focus on parallel computing, which is using a single multi-core processor in order to achieve high-performance computing. Parallel computing, however, is not the same as distributed computing. While parallel computing uses a single processor with many cores, distributed computing is defined as using multiple separate (usually physically) machines and processors in order to perform a calculation. While both definitions are of simultaneous data processing, they have vastly different approaches, issues and applications. Note that all the concepts described and discussed on this chapter are simplified mentions of the issues that arise when implementing parallelized algorithms.

Although parallelization can have the potential of increasing the performance, not all algorithms can be parallelized. One of the major issues that prevent some algorithms from being parallelized is data dependence, which is having calculations needing the result of previous calculations. An example of a simple algorithm that cannot be parallelized is shown below:

Listing 2.1: Simple non parallelizable C program.

```
1 int a = 5;
2 int b;
3 int c;
4 b = a  * 2;
5 c = b - 3;
```

The code above cannot be parallelized, as the calculation on line 5 depend on the result of calculation on line 4. Now considering the following code:

Listing 2.2: Simple C program with parallelization potential.

```
1 int a = 5;
2 int b = 2;
3 int c;
4 int d;
5 b = a * 2;
6 c = c * 3;
7 d = b + c;
```

This code can have parallelization on lines 5 and 6, as they have no data dependence between themselves, while line 7 needs both calculations to complete. However, considering the same example, such parallelization is not worth it, due to what is called overhead. Overhead can be defined as a non-functional computation, although necessary for the execution of the algorithm. If we were to parallelize this implementation, some API (such as OpenMP or MPI) would need to be called, which would need to create and manage two threads, and synchronize them once they're over (wait for both to finish), in order to continue the algorithm. This extra computation is the overhead associated with the parallelization API, which results in a trade-off: if the amount of calculations that each thread is tasked is too small, a sequential execution would be faster, as the API overhead would exceed the performance boost from the parallelization; however, there must be a point which the performance increase from parallelization is bigger than the overhead. This concept can also be applied on multithreaded iteration-based algorithms that deal with decreasing data load per iteration, and set a cutoff value, that is, a value in which the algorithm will no longer use parallelization, and rather execute the rest of the computation sequentially, in order to minimize the API overhead.

One last problem that may rise from parallelization is known as race-condition. Consider the code below:

Listing 2.3: Simple non parallalizable C program due to race condition.

```
1 int a = 5;
2 int b = 0;
3 b = a + 1;
4 b = a + 2;
```

Although there is no direct data dependence between lines 2 and 3, both are writing to the same variable, and if these instructions were executed in parallel, the resulting value in *b* is unpredictable. This is known as race-condition, and any parallel algorithm, in order to be correct, must not have this phenomenon.

2.1.1 CPU Parallelization

CPUs (Central Processing Units) are the core of any modern computer, and whose technological advancement was mainly defined by the raw number of cycles per second that it could achieve. In more recent years, however, the main focus for research and development has been on dividing the processor into smaller, independent units called cores, thus providing the ability to have simultaneous computation, and true parallelization. These cores are powerful and can run at very high clock speeds, and most day-to-day applications use only one at a time.

Multi-core processors, that is, processors which have more than a single core, are the standard for modern computing, and are extensively used for high performance computing (AKHTER; ROBERTS, 2006). While consumer-grade processors rarely have more than 4 or 8 cores, professional high-performance processors can contain up to 24 physical cores (48 logical) (INTEL, 2016). Operating Systems can use this to enhance user experience, by having different applications run on different cores, giving the user the ability to have many different apps running simultaneously. At the same time, researchers on all fields can leverage multi-core processors for running complex algorithms faster, by the use of parallel computing APIs.

Given this scenario of natural concurrency on CPUs, and the need for parallel computing capabilities, most programming languages either have native support for parallel computing (like Ada's messaging and monitor systems),

or have had APIs developed for this purpose (such as *std::thread* for C++, and OpenMP for C, C++ and Fortran). These APIs play a vital role on the execution of any algorithm that uses them, therefore studies on their performance and feature set have to be made in order to choose the right API for any given problem.

2.1.2 GPU Parallelization

GPUs (Graphics Processing Unit) were originally developed, as the name suggests, for graphical-intensive applications. Growing in popularity and performance undoubtedly due to the rise of the video game industry in the late 20th century, the use of GPUs for processing of other types of algorithms have recently expanded into many fields, such as biological research, space exploration, and other scientific endeavours (REGE, 2008). The use of GPUs for other types of applications is called GPGPU (General-Purpose Computing on Graphics Processing Units), a term coined by Mark Harris gpgpu.

The rise of GPGPU on the industry and research fields is due to the innate SIMD (Single Instruction, Multiple Data) architecture of a GPU, which consists of many small, independent cores (OWENS, 2007). However, unlike a CPU, which has only a few, powerful cores, GPU contains many (hundreds, or even thousands) of smaller cores, running at slower clock speeds (under 1 GHz). This design was made specifically to deal with algorithms in which there are a lot of simple mathematical operations, on large amounts of data (such as matrix transformations on vectors). Therefore, GPUs are best leveraged on algorithms with high data load, but low data dependence. In more recent years, the term SIMT (Single Instruction, Multiple Threads) has been used to refer architectures for GPGPU (NVIDIA, 2016).

GPUs, however, have an issue with memory. Since GPUs contain many different cores operating at different clock speeds than the CPU, and most of the time physically away from the computer's RAM, there was a need for a special kind of memory that would be closer and dedicated to the GPU, called VRAM (acronym for Video RAM, as it used to be primarily dedicated to storing the frame buffer). As the use of GPUs uses became more general, the VRAM became the memory used by all programs that execute on the GPU. However, since there are two different memories for CPU and GPU, in order for any computation on

the GPU to take place, first the data must be copied from RAM to VRAM, and, after the computation was done, the processed data must be copied back to main memory, which introduces a massive overhead for any computation on the GPU that must be taken into account.

Nowadays, GPUs have found their way into the desktop and mobile worlds, and are extensively used for both graphical processing, and GPGPU applications. Therefore, much like with CPUs, many different APIs that provide access to GPU execution appeared over the years, although no language has yet came up with native support for GPU offloading. Such APIs include OpenCL and CUDA, which provide general purpose use of the GPU (unlike APIs like OpenGL or Vulkan that focus on rendering and image-based applications).

2.1.3 Evaluating Performance

In order to evaluate and compare the performance of the algorithm designs, techniques and parallelization APIs discussed in this work, we mainly used three criteria: mean time performance of several executions, the speedup and the efficiency. We used *Jaja*'s definition of speedup (JAJA, 1992):

Let P be a given computation problem and let n be its input size. Denote the sequential complexity of P by T(n). That is, there is a sequential algorithm that solves P within this time bound, and, in addition, we can prove that no sequential algorithm can solve P faster. Let A be a parallel algorithm that solves P in time Tp(n) on a parallel computer with p processors. Then, the speedup achieved by A is defined to be:*

$$S_p(n) = \frac{T^*(n)}{T_p(n)} \tag{2.1}$$

For this work, we experimented various sequential algorithms, and used the best implementation as $T^*(n)$, although we make no guarantees that it is, indeed, the best sequential algorithm. Therefore, all the speedups considered in this work are defined as the speedup in relation to this sequential algorithm. Still, the ideal speedup for a parallel algorithm $S_p(n)$ is p.

Lastly, we used the efficiency, which provides an indication of the effective utilization of the *p* processors relative to the algorithm (JAJA, 1992). We used the following definition for efficiency (KARP; FLATT, 1990), where e is the efficiency,

s is the speedup, and p is the number of processors used.

$$e = \frac{s}{p} \tag{2.2}$$

Finally, although we mainly used elapsed time, speedup and efficiency for numerical comparison, these quantitative metrics may not provide a full picture. In this work, we will also have some qualitative discussion on the implementation details for each API, and compare which may be best suited for different applications.

2.2 Memory and Caches

One of the biggest causes of slow execution on both CPUs and GPUs is, and always has been, memory latency. While a consumer-grade Intel processor can reach up to 4.0 GHz clock speeds, the fastests DDR4 RAM can only go as high as 2.1 GHz. This difference between processor and memory speeds mean that, in the trivial case of adding two numbers, a processor would take more time fetching the numbers from the memory, than on actually computing the sum of the values. Therefore, it has become a standard for CPUs and GPUs to have faster, smaller memories physically closer to the cores, called *caches* (SMITH, 1982). These caches store frequently used data so that future requests to this data are faster than having to fetch from the main memory.

As caches are orders of magnitude smaller in bytes than a regular RAM (modern Intel i7 processors have up to 4MB cache), policies have to be implemented in order to manage the information that is stored on the cache (SMITH, 1987). Therefore, similar algorithms may present very different cache behaviour, should the data be accessed in different patterns, meaning that, on modern CPUs and GPUs, the pattern in which the memory is accessed can have an impact on the overall performance of the algorithm.

2.3 Platforms and APIs

This subsection focuses on the three APIs that were used for parallelization on CPU and GPU: OpenMP, OpenCL, and CUDA. These are among most used

libraries for high performance computing.

2.3.1 OpenMP

OpenMP (acronym for Open Multi-Processing) is a collection of compiler directives, library routines and environment variables for CPU parallelism in C, C++ and Fortran (OpenMP Architecture Review Board, 2013). They provide programmers the ability to create and manage parallel programs with high portability. The compiler directives extend those base languages, meaning that the compiler must support the OpenMP API, often activated with a command line option. For this work, we will focus only on OpenMP with C/C++, and will give a brief summary on the most pertinent directives and aspects of the OpenMP specification.

OMP directives are specified with the pragma preprocessing directive, with the following syntax:

#pragma omp directive-name [clause [[,] clause] . . .] new-line

Each OMP executable directive applies to at most one succeeding statement, which must be a structured block. Note that each directive must start with *#pragma omp*.

There are many different constructs and executable directives defined on the OMP API, however one of the fundamental constructs is the *parallel* construct, which marks the start of a parallel execution block. The syntax for the *parallel* construct is as follows:

#pragma omp parallel [clause [[,] clause] . . .] newline Structured-block

The OMP API defines many different clauses, however the most important ones are:

- **If ([parallel :] scalar-expression)** defines that the parallelization will only occur should the expression be evaluated to true, otherwise the directive will be ignored;
- **Num_threads (integer-expression)** defines the maximum number of threads to be used;
- **Private (list)** configures all the variables on the list as private data-sharing attribute. Private variables are the ones which have individual copies on all

threads, and are, therefore, not accessible by any other thread;

- **Firstprivate (list)** configures all the variables on the list as firstprivate data-sharing attribute. Firstprivate variables are the same as private variables, however their values are initialized with the value of the original value when the construct is encountered;
- **Shared (list)** configures all the variables on the list as shared data-sharing attribute. Shared variables are the ones which all the references on the parallel block refer to the same original storage area at the point the directive was encountered. The programmer must, therefore, ensure synchronization;
- **Default (shared | none)** configures, if the argument is 'shared', the default data-sharing attribute to all variables not explicitly determined otherwise (with directives such as with private, firstprivate or shared) as shared. If, however, the argument is 'none', the programmer is required to explicitly determine the data-sharing attribute of all variables.

The *parallel* construct can be combined with a loop construct in order to specify that each iteration of the loop can be executed in parallel. The syntax of the loop construct is as follows:

#pragma omp for [clause[[,] clause] . . .] new-line For-loops

The OpenMP specification provides many different clauses for the loop construct, such as most of the ones defined for the *parallel* construct. However, one of the most important clauses is the *schedule*, which specifies which scheduling policy should the API use in order to manage its threads. There are 5 different scheduling policies:

- **Static** scheduling specifies the iterations to be divided into chunks, and each chunk assigned to the threads in order. This process is mostly done at compile-time, having little to no overhead during execution. This scheduling policy is best used when the work can be divided evenly between the threads (each iteration takes roughly the same amount of time);
- **Dynamic** scheduling specifies the iterations to be divided into chunks similar to static scheduling, however the thread assignment is done in runtime: each thread executes a chunk, then requests for a new chunk, until there are no chunks remaining to be distributed. This method introduces a high runtime overhead, however it is better than static when the load is not evenly

distributed (the iterations take different amounts of time to finish);

- **Guided** scheduling is similar to dynamic scheduling, however automatically varies the size of each chunk allocated to each thread as it executes in order to find the optimal chunk size. This introduces even more runtime overhead, however may deal better than dynamic scheduling for problems with different iteration times;
- **Auto** scheduling specifies the decision regarding scheduling to be delegated to the compiler or runtime system;
- **Runtime** scheduling specifies the decision regarding scheduling to be deferred until run time.

2.3.2 OpenCL

OpenCL (acronym for Open Computing Language) is an open standard for cross-platform parallel programming for heterogeneous platforms called compute devices, such as CPUs, GPUs, FPGAs (Field-Programmable Gate Arrays), and many others (STONE; GOHARA; SHI, 2010). Originally developed by Apple Inc, and currently maintained by the non-profit consortium Khronos Group (with member companies including Apple Inc., Intel Corporation, Nvidia and Qualcomm), OpenCL is an industry standard API for parallelization.

An OpenCL implementation can be divided into two main parts: the host code, and the kernel code (TOMPSON; SCHLACHTER, 2012). The host code is a C/C++ program which uses the OCL API in order to gain access to the compute devices of the machine, and launch functions to be executed on these devices. The kernel code contains the actual functions that will be executed on the devices. OpenCL specifies a programming language based on C99 for programming such kernels. The host can execute functions on the device by enqueuing kernels, and configuring the different parameters for each kernel execution, similar to a regular C function call. Each kernel enqueuing results in the creation of multiple work-items divided into work-groups that will be executed in parallel. A work-item is a single execution of the kernel function, distinguishable from other work-items by its own global and local IDs. Meanwhile, a work-group is a collection of related work-items, and provides data sharing and synchronization

between its own items, although communication between different work-items is not possible. An OpenCL API call to kernel enqueuing provides the programmer with the tools to specify the number of work-items and work-groups, although they may be limited by hardware.

Since the host and device memory may not be the same (for instance, in case of a CPU host and a GPU compute device), all kernel code must use only device memory, and any data transfer from host memory to device memory must be performed by the programmer. OpenCL's memory model for the compute device defines a hierarchy of four levels:

- **Global** memory is shared by all processing elements;
- **Read-Only** memory is writable by the host, but not by the kernels;
- **Local** memory is shared by a group of processing elements;
- **Private** memory is private to each element, but is limited (in size and types).

Each level on this hierarchy is usually slower than the previous (accessing private memory is much faster than accessing local memory, which in turn is faster than accessing global memory). However, due to the nature of having a vast possibility of compute devices, not always will each level be implemented directly in hardware.

In order to provide synchronization in memory reads and writes on kernel code, it is possible to use memory fences, which are limited within work-groups, meaning that it is impossible to have a global barrier in kernel code. In order to have synchronization between all work-items, multiple kernels which break the task into smaller, truly independent chunks must be written, and the host will then enqueue each kernel sequentially.

One of the most interesting and particular features of OpenCL is its high portability, which is achieved by having different platforms as possible compute devices. The downside of this approach, however, is that this requires a considerable amount of setup code (detecting and initializing compute devices, compiling kernels, and so on), which yields a more complex code and higher initialization overhead.

2.3.3 CUDA

CUDA is a computing platform and API created by Nvidia for GPGPU on GPUs (NICHOLLS et al., 2008). Designed to use programming languages such as C and C++, a CUDA code must be compiled with its own LLVM-based C/C++ compiler, and has the limitation of only being able to be executed on CUDA-enabled GPUs. The CUDA platform also provides many libraries (such as cuSOLVER), compiler directives and other computational interfaces (such as OpenCL).

The CUDA workflow is very similar to that of OpenCL: the host code enqueues different CUDA kernels to execute on the GPU, and kernel code can only handle device memory, while host code can only handle host memory, with CUDA API calls for memory transfer. Each CUDA kernel execution is called a thread, and the threads are divided into individual blocks. The total number of blocks and the maximum number of threads per block are limited by hardware. The programmer has full control over the number of threads per block and the number of blocks any given kernel enqueuing will execute, and there are CUDA API calls designed specifically to find the hardware limitations during run-time, in order to prevent invalid executions.

3 STATE OF THE ART IMPLEMENTATIONS

In order to make fair and useful comparisons, it is imperative that we also experiment state of the art implementations of the algorithm. In this section, we will discuss two professional, highly optimized libraries that can solve the Cholesky decomposition problem, one using CPU, and one using GPU parallelization.

3.1 LAPACK

LAPACK, an acronym for Linear Algebra PACKage, is a library for solving the most common numerical linear algebra problems (ANDERSON et al., 1999). First released in 1992 for Fortran 77, then moved to Fortran 90 in later versions, the library has been continually updated since, and was designed to have very high performance, even for today's standards. The provided routines include higher-level problems such as solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems and singular value problems, as well the associated matrix factorizations (such as LU, Cholesky, and QR).

LAPACK routines were also written in a way that uses BLAS (Basic Linear Algebra Subprograms) as much as possible. BLAS is a collection of low-level matrix and vector operations (such as vector addition and multiplication), and its implementations have a very high performance for any given hardware. LAPACK uses Level 3 BLAS, which is a set of specifications for subprograms focused on matrix operations.

The routines defined in the LAPACK specification follow a characteristic naming scheme, which comes the the 6-character limitation of Fortran 77, and remains to this day. The first character of the routine indicates the data type (S for single-precision float, D for double-precision, C for complex, and Z for complex values). The next two letters indicate the type of matrix that the algorithm expects (DI for diagonal, OR for orthogonal, PO for symmetric or Hermitian positive definite, among others). The final three letters indicate the computation that has to be performed.

3.2 cuSOLVER

cuSOLVER is a collection of dense and sparse direct solvers based on CUDA for GPU acceleration in many applications, developed by Nvidia (NVIDIA, 2014a). According to its developer, cuSOLVER can be up to 6 times faster than MKL (Math Kernel Library, a similar high performance library developed by Intel). cuSOLVER is the combination of three libraries:

- **cusolverDN** contains LAPACK-like dense solvers, with algorithms such as

Cholesky and LU factorizations;

- **cusolverSP** contains sparse direct solvers and eigensolvers;
- **cusolverRF** contains sparse refactorization solvers.

It must be noted that the cuSOLVER API is called within a CUDA program, meaning that it requires hardware with CUDA compute capabilities. According to Nvidia, the library is freely available as part of the CUDA 7 RC.

3.3 Related work

There are plenty of research on using CPU and GPU parallelization for solving dense linear algebra problems such as Cholesky decomposition. However, it is most common to use already developed software infrastructure (such as LAPACK and BLAS) for sequential processing. One of such works (LTAIEF et al., 2010) offers unprecedented performance and scalability with a hybrid implementation of the Cholesky decomposition, using LAPACK. Other works have slightly different focus (ABDELFATTAH et al., 2016), dealing with solving large amounts of small matrices, rather than solving a single, large matrix. Using a GPU approach, they investigated various algorithm designs and compared the results with state of the art implementations, obtaining very good results.

There is also a lot of research on providing general but useful comparisons between different parallelization APIs. One of such works (KARIMI, 2015) compares OpenMP and OpenCL as CPU parallelization tools, and argued that OpenCL may have better performance than OpenMP when there is a requirement for a high invocation of threads.

4 DEVELOPED IMPLEMENTATIONS

Once with the necessary knowledge of the problem and the tools available for solving it, the implementation step can begin. This section follows the development process of all the implementations for the Cholesky decomposition algorithm using the discussed libraries and APIs. The full source code for all implementations is available at <https://bitbucket.org/jruschel/parallel-cholesky>.

4.1 Sequential Implementations

Before it is possible to parallelize and optimize any implementation of an algorithm, we need to analyze its sequential version. A "sequential" (or "serial") version is the one which is executed sequentially, that is, from start to end, without interruptions, parallelization or any kind of concurrency, as a single execution thread. Then, the correctness of the implementation (its output is the one intended, and mathematically correct) must be tested, for instance with another implementation of the same algorithm. Only then it is possible to start optimizing the implementation, always checking for correctness. In this chapter, we will discuss the progress on the optimization of the Cholesky decomposition, from the choice of basic method, to the techniques used, and the final implementations.

As previously mentioned, there is a general formula (1.2) for calculating a single L_{ij} of the output matrix L. However, this is the mathematical representation of the decomposition, and implementing this formula (and further optimizing the code) requires an additional effort. We also mentioned the three main basic algorithms for calculating the Cholesky decomposition of a matrix: the Cholesky Algorithm, the Cholesky-Banachiewicz algorithm, and the Cholesky-Crout algorithm. In order to choose which one was more suitable for this work, some study into each of these methods has to be made.

The original Cholesky Algorithm, because of its recursive nature and high data-dependence, was quickly discarded. The Cholesky-Banachiewicz and the Cholesky-Crout algorithms, on the other hand, are very similar, as they use the same formula for calculating the value of each element. They differ, however, on the order of operations, and the way in which the matrix is accessed and calculated: while the Cholesky-Banachiewicz orients itself by row, the Cholesky-Crout

algorithm orients itself by column.

A couple of points have to be considered for this comparison: firstly, the number of operations. Although both algorithms implement the same basic mathematical formula, their implementational differences may result in different asymptotic complexity, therefore resulting in different execution times. It is necessary, then, to compare both implementations (code and execution) in order to know which one is best. The second point is that they may also have different data dependence and access patterns, which, for parallelization, is a very important issue. As discussed before, for good parallelization, the less data dependence between different threads, the faster the execution will be, as there will be fewer threads waiting on the results of other threads.

Before starting to look at some code, a few points regarding the implementations in C:

- By default, the input and output matrices are of single-precision floating points (*floats*);
- On all snippets of code shown below, it is assumed that all both input and output matrices were already allocated (A has the input values, and L is of the appropriate size, and with all elements as zero). For now, the matrix allocation code itself is not important, and all snippets use valid and allocated pointers of type *float* **. Different allocation patterns will be discussed in the next section;
- *dimensionSize* is an integer argument specifying the size of the first dimension of the input and output matrices, and, since they're square matrices, their second dimension;
- *Sqrt* is a function that returns the square root of the double value given as input.

Also, for clarity, all code that follows uses the same names for its parameters and variables as the mathematical formulation, that is:

- A and L are the input and output matrices, which must be of the same size. It must be noted that these matrices are square matrices, and the access pattern is *row, column*.
- I, J, K are the indexes used for accessing the matrices, where I and J are used for row and column (respectively), and K for iterating the previous columns

in a given row.

First, comparing the basic implementation code for both algorithms, starting with the former:

Listing 4.1: C implementation of the Cholesky-Banachiewicz Algorithm

```
1  for (i = 0; i < dimensionSize; i++) {
2      for (j = 0; j < (i + 1); j++) {
3          float sum = 0;
4          for (k = 0; k < j; k++)
5              sum += L[i][k] * L[j][k];
6
7          if (i == j)
8              L[i][j] = sqrt(A[i][i] - sum);
9          else
10             L[i][j] = (1.0 / L[j][j] * (A[i][j] - sum));
11     }
12 }
```

The code starts with a loop (line 1) through all the rows of the input matrix, and a nested loop through all the columns below and including the main diagonal (line 2). With this setup, the actual code only gets executed on the values that must be updated, that is, the values of the main diagonal and the lower triangle. On lines 4 and 5, the iteration through all columns to the left of the current column calculates the sums specified on the formula (note that when $i = j$, it calculates the sum of the squares of the row, while when $i > j$, it calculates the sum of the multiplication between the elements of the current row with the elements of the row that intersects the main diagonal on the current column). On lines 7 to 10, the final values are calculated, a direct translation of the mathematical formulation, using the temporary variable *sum* that was previously calculated.

Note that, although it is required to branch on $i = j$ or *else*, the iterations at lines 1 and 2 make sure that this code will never be executed with $i < j$, meaning that the else is only for the case that $i > j$, and the output matrix will not be changed on its upper triangle.

Now, for the Cholesky-Crout algorithm:

Listing 4.2: C implementation of the Cholesky-Crout Algorithm

```
1  for (j = 0; j < dimensionSize; j++) {
2      float sum = 0;
3      for (k = 0; k < j; k++) {
4          sum += L[j][k] * L[j][k];
5      }
6      L[j][j] = sqrt(A[j][j] - sum);
7
8      for (i = j + 1; i < dimensionSize; i++) {
9          sum = 0;
10         for (k = 0; k < j; k++) {
11             sum += L[i][k] * L[j][k];
12         }
13         L[i][j] = (1.0 / L[j][j] * (A[i][j] - sum));
14     }
15 }
```

The code starts, contrary to the Cholesky-Banachiewicz algorithm, with a loop through all the matrix's columns (line 1), and doing, for each, two things: calculating the main diagonal value (lines 3 to 6), and all the elements below the main diagonal (lines 8 to 14). For the main diagonal, the sum of the squares of the elements of that row is made (line 4), and then the simple formula is applied (line 6) for setting the value. Then, a loop through all the elements below the main diagonal (line 8) calculates the value for each individually, by making a similar sum (lines 10 to 12), and applying the formula (line 13).

Note that, just as the Cholesky-Banachiewicz algorithm, the loops are made in such a way that the upper triangle of the matrix is left untouched, which prevents unnecessary calculations, since it is known that those values will always be zero. The only condition for both of these implementations to remain correct with such simple optimization is requiring that the output matrix is initialized zero-ed, that is, with the value of zero on all its elements. This is not a problem for most languages and applications, and certainly not in C (for instance, using 'calloc' instead of 'malloc' automatically initializes the allocated dynamic memory with zeroes).

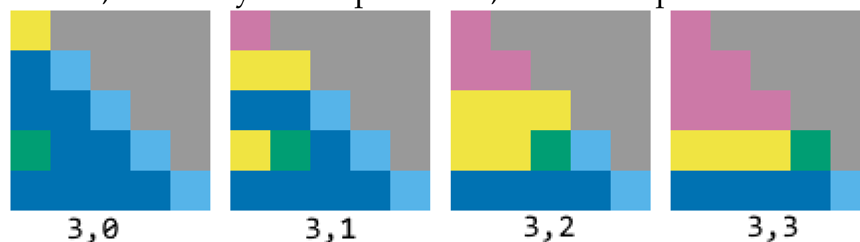
Both codes presented above are simple, and perform the calculation of the Cholesky decomposition accurately, for any sized, valid matrix. Now, we need to compare the two implementations with the metrics we established earlier: number of operations and data dependency.

For the first point, a quick analysis of both snippets (more importantly, their loops) shows that they do not differ on this aspect. Both the Cholesky-Banachiewicz and the Cholesky-Crout algorithms only iterate through the values that should be calculated (main diagonal and lower triangle). Therefore, number of operations is not a contributing factor.

For the second point, however, both algorithms are vastly different. As mentioned before, the Cholesky-Banachiewicz is row-oriented (noticeable on the code as the first loop is an iteration through all rows of the matrix), while the Cholesky-Crout is column-oriented (noticeable on the code as the first loop is an iteration through all columns of the matrix). It is worth remembering that the difference of data dependency being discussed here is not a mathematical one, as the data needed for each individual element is the same for both implementations; however, the difference lies in the order in which those values are calculated.

To better visualize this issue, we can compare the dependencies of a single row, and the dependencies of a single column, side-by-side. Note that for each element of the matrix, the dependency is the same regardless of the algorithm.

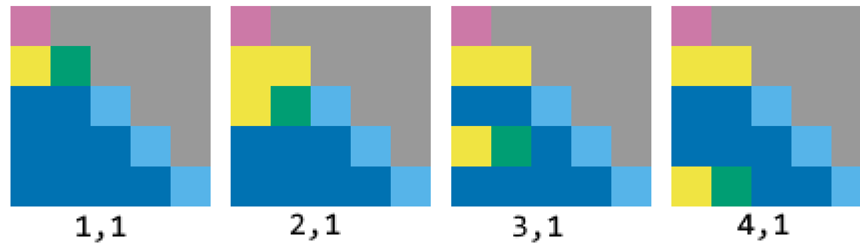
Figure 4.1: Example of data dependencies of a single row on a 5x5 matrix. Green elements are the active elements for each iteration, which direclty depend on the yellow elements, indirectly on the pink ones, and not dependent of the blue ones.



Source: Author

Figure 4.1 clearly shows how each element of a row depend on all the rows that come before it (except for the element on the first column), as well as some columns before it, but never a row or column after it. This is the pattern for the Cholesky-Banachiewicz algorithm.

Figure 4.2: Example of data dependencies of a single column on a 5x5 matrix. Green elements are the active elements for each iteration, which direclty depend on the yellow elements, indirectly on the pink ones, and not dependent of the blue ones.



Source: Author

Figure 4.2 clearly shows how each element of a column depend only of its row and the row of the main diagonal intersecting this column (element (3,1) requires row 3 and row 1, but not row 2; element (4,1) requires row 4 and row 1, but not rows 2 and 3). This is the pattern of the Cholesky-Crout algorithm.

The point of this analysis is to find which of these patterns is more suitable for parallelization and further optimization, and the images presented above have the answer. While iterating on rows has a complicated, increasing dependence of values of previous iterations, looping through columns has a clear, constant dependence on controlled values. The calculation of each row of a column does not need the value of any other row of that same column, only on the columns that come before it. It does, however, need the value for the diagonal value (which, then, must be calculated before). These characteristics make this algorithm easier to parallelize, with the only constraints being:

1. A column c can only be computed after all columns $j < c$ were computed; and
2. All rows below the main diagonal of a column can only be computed after the main diagonal element of this column was computed.

The algorithms on this work are based, therefore, on the Cholesky-Crout algorithm, parallelizing the calculation of the rows of a single column, then moving to the next column.

On a final note, an interesting feature of the Cholesky algorithm, regardless of which orientation is followed, is how the greater the column, the more elements are needed to calculate a value, as it has to iterate through all of the columns before it. This means that this is a very memory-intensive algorithm,

but with very simple calculations (sums and multiplications; square root being the most complicated). Therefore, it is expected that the greatest bottleneck will be memory fetching, and should be a priority for optimizations by using the cache as best as possible.

4.2 General Optimizations

Now that a basis for the algorithm has been chosen, an analysis of the code can be made in order to find optimization points, before even discussing parallelization. These optimization techniques are slight changes to the sequential implementation discussed on the previous section, and will attempt to increase the overall performance of the algorithm. All the techniques discussed in this section are C-related.

Starting from where we left off, consider the code listing 4.2, which, in order to manipulate a value i, j of a given matrix A, uses a reference pattern in the form of:

$A[i][j]$

This pattern uses an 'array of arrays' to represent the matrix: for each element of the first dimension of the array (rows), there's a pointer to another array (columns). Although this method represents a matrix correctly, it does mean that, when executing this code, the program must first get the pointer of the first dimension, then access the array of the second dimension, and then go to the element it needs (two memory look-ups). The code for this method uses a double-pointer to floats (*float* **), which is basically a pointer to a list of pointers. Each pointer of that list points to a row of the matrix, and there are *number of columns* pointers on the list. This approach may also produce bad cache performance, as the rows may be allocated on separate, distant places in memory.

There is a better way to represent a matrix, and that is to allocate it *contiguously*. Contiguous allocation means that all of the matrix is represented on a single dimension, with a simple function used to map a 2-dimensional index to a single-dimensional index. The array holding such matrix must be allocated with size *size_of_first_dimension* * *size_of_second_dimension*, since this single array contains all of the matrix. The transformation from a 2D point i, j, to a 1D offset *index*, on a matrix with *dimensionSize* as the size of one of its dimensions can be

written in two ways:

Listing 4.3: Row-wise indexing of element (i j).

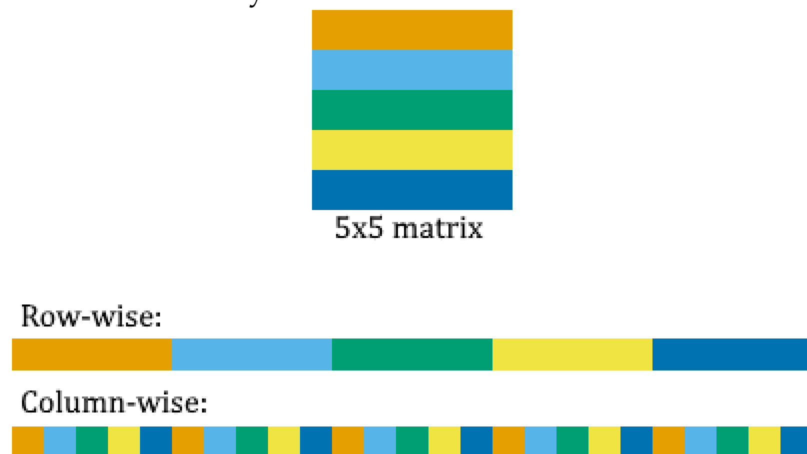
```
1 index = i * dimensionSize + j
```

Listing 4.4: Column-wise indexing of element (i j).

```
1 index = j * dimensionSize + i
```

Although code listing 4.3 and 4.4 are similar, their behaviour is massively different. Row-wise means that, in memory, the elements of a single row are placed side-by-side, and the next row is placed after it, and so on. Meanwhile, column-wise means that the elements of a single column are placed side-by-side, with the next column placed after it, and so on. The following image exemplifies the two different patterns:

Figure 4.3: Difference between row-wise and column-wise indexing on a contiguous array representing a 2D 5x5 matrix. Each row of the example matrix is coloured to visualize memory behaviour.



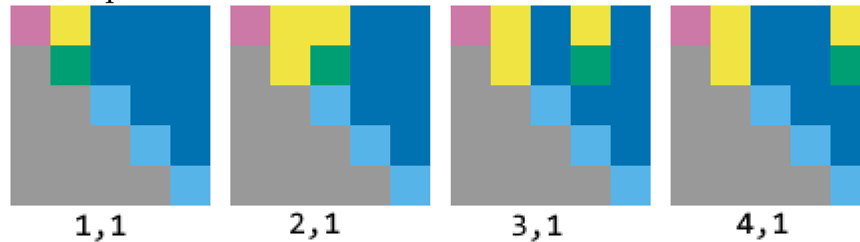
Source: Author

Another advantage of using a single array (instead of an array of arrays) is the reduced use of pointers (since only a single pointer is needed in order to use the array). This feature will be very useful when discussing GPU parallelization. A modified version of the algorithm that uses contiguous memory allocation is listed on the Appendix (A.1).

As previously discussed, the Cholesky decomposition is memory-intensive, and therefore the cache plays a crucial role on the performance of any implementation. One interesting way we could demonstrate this is to use column-wise memory access on the contiguous arrays used, instead of row-wise, and compare

the results. This change is effectively transposing the matrices, and may result in differences cache behavior, as different regions of memory will be on cache.

Figure 4.4: Example of a single column iteration of the Cholesky-Crout algorithm with transposed memory access. Green elements are the active elements for each iteration, which direclty depend on the yellow elements, indirectly on the pink ones, and not dependent of the blue ones.



Source: Author

This new access pattern can be used both for the input and output matrices by changing the access pattern of an element (i, j) to (j, i), with the following considerations:

- The input matrix is symmetric (a restriction of the Cholesky decomposition), therefore transposing it yields the exact same matrix; and
- The output matrix will be upper-triangle, instead of lower-triangle. If the final output must be lower-triangle, a simple matrix transpose operation at the end of the computation will result in the correct output.

So far, we have considered that the input and output matrices are two separate arrays in memory, however this is not mandatory for this algorithm. It is possible to calculate the Cholesky decomposition successfully on a single matrix, that is, having only one matrix with the input at the start, and having the same matrix with the correct output at the end. This can be called “in-place calculation”, and may have significant improvements of spatial and time performance. Considering only one matrix (therefore, one array), the cache will not be “fought over” by the two different matrices (input and output), and will contain information about only one, reducing the number of cache misses.

Finally, in order to choose which of the sequential implementations of the algorithm we would start optimization via parallelization, we performed a simple experiment and analysed the results. In this experiment, we executed the following 6 algorithms:

1. Cholesky-Banachiewicz;

2. Cholesky-Crout with double reference (*A[i][j]*);
3. Cholesky-Crout with contiguous memory allocation (*A[i *d + j]*);
4. Cholesky-Crout contiguous, and transposed (*A[j * d + i]*);
5. Cholesky-Crout contiguous, and in-place;
6. Cholesky-Crout contiguous, in-place, and transposed.

The experiment was made for a fixed input matrix size of 2500x2500 on an *Intel Xeon E5-2650 v3*. Below are the results:

Table 4.1: Experiment results for sequential versions when size is 2500.

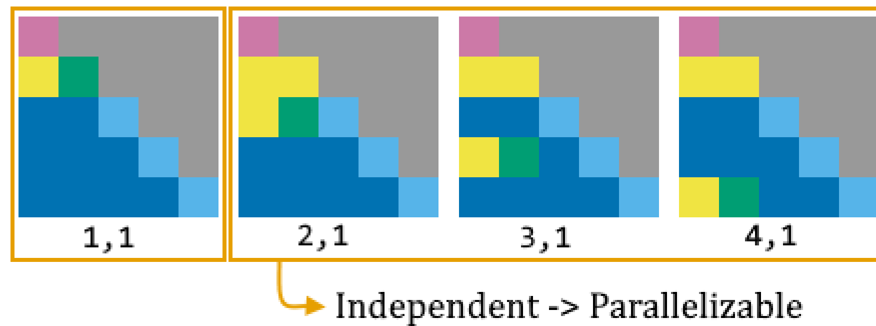
Algorithm	**Mean Time**	**Standard Error**
Banachiewicz	2.662720	0.02379097
Crout Double Reference	2.695829	0.01107501
Crout Contiguous	2.663613	0.02046345
Crout Contiguous Tranposed	4.390176	0.08827412
Crout Contiguous In-Place	2.645548	0.02150006
Crout Contiguous In-Place Transposed	4.368486	0.07353618

These results demonstrate how simple changes on the algorithm design can have expressive impact on the performance of an algorithm. Both transposed executions had a significantly worse performance (almost twice slower) than their non-transposed counterparts, proving the importance of a cache-aware implementation. In contrast, Cholesky-Banachiewicz, and Cholesky-Crout contiguous with and without in-place calculations were statistically equivalent (their executions were within each other's error range). However, since the former is not easily parallelizable (as previously discussed), we selected the Cholesky-Crout algorithm for parallelization, using both versions with and without in-place calculation.

4.3 CPU Parallelization with OpenMP

In this section, we will take the sequential version of the Cholesky-Crout algorithm (with the optimizations discussed previously), find its parallelization points, and modify its code to implement such parallelization with OpenMP. As was previously discussed, the Cholesky-Crout may be parallelized on its calculation of rows, for each column:

Figure 4.5: Example of a single column iteration of the Cholesky-Crout algorithm, with noted dependencies between iterations. Green elements are the active elements for each iteration, which direclty depend on the yellow elements, indirectly on the pink ones, and not dependent of the blue ones. The main diagonal element (1,1) must be calculated first, however all the other elements ((2,1), (3,1) and (4,1)) are independent, and can therefore be executed in parallel.



Source: Author

There is a simple OpenMP directive that can be used to achieve this parallelization: *parallel for*. This directive must be placed on the line before the *for* that should be parallelized. The full implementation using the OpenMP directive can be found on the Appendix A.2. The directive used is as follows:

Listing 4.5: OpenMP directive used to parallelize a 'for' loop.

```
1 #pragma omp parallel for private(i,k,sum) shared (A,L,j,
     diagonal_value) schedule(static)  if (j < dimensionSize
     - cutoff)
```

The directive *"#pragma omp parallel for"* is marking the 'for' just below it to be made parallelizable. Ths means that each iteration of the loop will be executed concurrently, as separate threads. The number of threads that will be created varies depending on the current column, and is given by the expression $dimensionSize - (j + 1)$, where j is the index of the current column.

The directive also contains a few other settings, used to configure the parallelization:

- **schedule(static)** marks the scheduling policy of the thread pool as static. The choice for this scheduling policy (instead of, for instance, dynamic scheduling) is due to the fact that all of the iterations have the exact same data load, meaning that the data is evenly distributed between all threads, and should take roughly the same time finish. Using static scheduling on such situations reduces operational overhead, and is one major advantage

of this implementation;

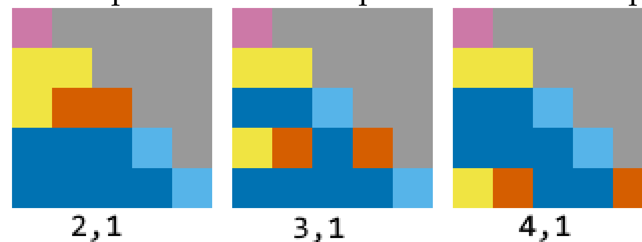
- **if (j < dimensionSize - cutoff)** is a conditional statement, implementing a cutoff value for the parallelization. This command is simply stating that for columns *0* to *dimensionSize-cutoff*, the execution will be parallelized as described. However from *dimensionSize-cutoff* to *dimensionSize*, the program will execute sequentially;
- **private(i, k, sum)** marks the variables *'i'*, *'k'*, and *'sum'* as being private, that is, each individual thread will have its own copy of each of these variables. This is needed as each thread has its own loops and temporary values;
- **shared(A, L, j, diagonal_value)** marks the variables *'A'*, *'L'*, *'j'* and *'diagonal_value'* as being shared, that is, all threads share these values between themselves. Marking as 'shared' may be dangerous in case one of the threads would change its value, however it is not the case for any of theses variables. The variables *'j'* (the current column) and *'diagonal_size'* are an integer and a float (respectively), and their values will never change. On the case of A and L, they are merely the pointers, and the area of memory that they point to is shared, meaning that the code must make sure that, when a thread writes to a memory position, no other thread is reading or writing to that same position. This is achieved simply by having each thread write only to one position (its index i).

On top of the "Simple" parallelization discussed above, we also implemented a version that attempts to fasten the execution time by removing the sum of squares used for the calculation of the main diagonal element. This can be achieved by distributing this sum as the final calculation for each element of the lower-diagonal. After the calculation of an element n, it calculates its square and adds to the main diagonal element at position (n, n) on the output matrix. Since we are considering that the output matrix starts blank (with 0 on all elements), by the time the columns iteration reaches a column j, its main diagonal (j, j) will already contain the sum of the squares of its row, therefore removing the need for a loop.

This modification, however, changes the write pattern for each thread, by adding an additional write to each thread. The following image represents the write operations for each thread (following the example matrix and color codes defined previously) on this modified version of the Cholesky-Crout algorithm,

proving that this modification is still valid.

Figure 4.6: Example of write operations for each row of a single column of the Cholesky-Crout algorithm with distributed sum. The active elements for each iteration direclty depend on the yellow elements, indirectly on the pink ones, and are not dependent of the blue ones. The orange elements are the elements in which each thread must perform a write operation on the output matrix.



Source: Author

As a third and final OpenMP implementation, we took the concept of in-place calculation discussed previously on *General Optimizations* and parallelized it. The OMP pragma is very similar to the one mentioned earlier, however there is only one matrix for input and output. Note that this modification also yields the correct result, as the memory access patterns remained unchanged. Also, most of the implementations found for the Cholesky decomposition use this method for memory management (eg LAPACK uses only one contiguous array for input and output).

4.4 CPU and GPU Parallelization with OpenCL

In this section, we will look into how to implement the Cholesky decomposition using OpenCL, which can then be executed both on the CPU and on the GPU. As previously mentioned, it is possible to run the same OpenCL code with different so-called 'compute devices', that is, different platforms in which the code will be executed, and this platform can either be a CPU or a GPU. However, this also means that the OpenCL implementation is vastly more complex than other APIs (such as OpenMP).

An OpenCL program has two parts: the host code and the kernel code. While the host code will be executed sequentially on the CPU, the kernel code is executed on the compute device. The kernel code contains the code for the processing of a single element of the data, and is executed when the host enqueues kernels to be executed. This enqueuing has a few parameters, such as the number

of work-items and the data to be processed.

For parallelization of the Cholesky decomposition with OpenCL, we used the Cholesky-Crout algorithm as basis, but a few modifications had to be made. In order for code on the compute device to be able to access and manipulate any data, it first needs to be copied over from the main memory (RAM) to its own memory (which, in case of GPU, is the VRAM) - and that process is very slow. Then, after the execution, if the data has been changed, it needs another copy, from its own memory to RAM - again, a very slow process. Therefore, we want to minimize the number of memory copies between the memories, ideally only once in each direction (one to copy the input matrix to the device memory, and one to copy the result back to the RAM). This means that all of the computation must be done with kernel code, keeping the communication between CPU and the device to a minimum. Moreover, all other parallelization problems discussed previously also apply here (read-write and overhead trade-off).

Reviewing the Cholesky-Crout algorithm, the parallelization is on its columns - for each column, each element on its lower-diagonal (not on the main diagonal) is independent of each other, and therefore parallelizable. However, the columns are not independent of each other. There are, therefore, two synchronization points:

1. A column 'j' can only be calculated once all of column 'j - 1' has been calculated; and
2. The elements of a column (except the main diagonal) can only calculate

their values after the main diagonal value has been computed.

For the implementations discussed here, the host code iterates through all of the columns, enqueuing kernels to process the current column. We implemented three different ways for this enqueuing, which will be discussed further on this section.

OpenCL presents a very low-level API, and the initialization step provides the programmer with a lot of different options. The initialization step for all three of our OpenCL implementations is basically the same, and is responsible for:

- Configuring the correct Platform and Device IDs (with *clGetPlatformIDs* and *clGetDeviceIDs*). This is the configuration of which compute device will the kernels be executed in;

- Creating an OpenCL context (with *clCreateContext*);
- Creating the command queue (with *clCreateCommandQueue*). Note that only one queue is needed, and is configured to be executed in-order (that is, kernels will be executed on the same order as they are queued), which is what maintains one of the synchronization points that were made earlier. It may also be created with the *CL_QUEUE_PROFILING_ENABLE* flag, in order to get the kernel execution time;
- Creating two RAM and VRAM memory buffers (with *clCreateBuffer*) for input and output matrices. Configured with the flags:
  - *CL_MEM_READ_ONLY* on the input matrix A, specifying that the kernels can read, but not write to;
  - *CL_MEM_READ_WRITE* on the output matrix L, specifying that the kernels can read and write;
  - *CL_MEM_COPY_HOST_PTR* on both matrices A and L, specifying that the contents from the host must be copied to the device prior to any execution.
- Loading and building the program (with *clCreateProgramWithSource* and *clBuildProgram*). The CL code, although it could be embedded into the main source as a string, is being loaded as a separate .cl file, and is built in runtime;
- Creating the execution kernel (with *clCreateKernel*);
- Configuring kernel arguments (with *clSetKernelArg*). These arguments are the inputs for the host code, and are the input and output matrix pointers (in device memory), the dimension size (width of the matrices) and the current column being processed.

The calculation step is a single host iteration of the columns, and at each iteration one or more kernels are enqueued. On the device side, each work-item is responsible for a single element of that column. This is implementing the first synchronization point made above - the code for column j will execute only after all the code for column $j-1$ was executed, as OpenCL guarantees that the kernels will execute in-order. As discussed before, we implemented three different kernel enqueuing methods, yielding three different algorithms. They were:

1. **Simple Single Kernel** is the simplest implementation, using a single big kernel to calculate each value of the column by enqueuing, at each column iteration, $dimensionSize - j$ work-items. Since there is no synchronization between the main diagonal element and the others, all items are required to recalculate that value locally. Although this may have impact on performance, it is necessary for the algorithm to be correct. Also, since the same kernel is used for both the main diagonal and the other elements (which have very different algorithms for calculating their value), there's the need for an additional conditional statement on the start of the kernel, which in turn may also result in loss of performance. The kernel for this implementation can be found at the Appendix (A.3);
2. **Simple Multiple Kernels** builds upon the *simple single kernel*, which has the need for each work-item to recalculate the values of the main diagonal. The only way we can remove this rather slow process is to make sure that by the time the values of the lower-diagonal are to be calculated, the main diagonal element was already calculated and had its value updated to global memory. As discussed before, this is not possible in OpenCL with barriers, since barriers work only in their own work-groups, and it is not guaranteed (nor good for performance) that all work-items are on the same work-group. Therefore, we implemented this synchronization by having multiple kernels: one for calculating only the main diagonal, which will be queued first; and one for the other elements, which will be queued after. The first kernel is enqueued with only 1 work-item, while the second kernel is enqueued with $dimensionSize - j - 1$ work-items (except for the last column of the matrix). The problem with this implementation is the overhead of enqueuing twice the number of kernels;
3. **In-Place Multiple Kernels** applies the concept of in-place calculation discussed previously by having the kernels only use one matrix for both input and output, which decreases the memory use on the device, and may improve cache performance. Note, however, that this change does not improve on the memory transferring problem inherent to OpenCL, as the matrix still needs to be copied to and from the device. We had to use multiple kernels for this version, since it is impossible to have a single kernel and in-place calculation (the work-item responsible for the main diagonal would

write to the same position that the other work-items would be reading), which may dampen the performance boost given by using in-place calculation.

4.5 GPU Parallelization with CUDA

In this section, we will discuss our implementations of the Cholesky decomposition in GPU using CUDA. As mentioned before, CUDA, although having some similarities to OpenCL, has a very unique API. Programs are written as .cu files, which contains both the host and the kernels, and compiled with a special compiler, called NVCC. CUDA, although having a much easier to use API than OpenCL, does not offer same low-level control, which even though can be a problem for some applications, was not a hindrance for this work.

The base algorithm was the Cholesky-Crout, and in order to implement it in CUDA, a very similar overall strategy to the OpenCL implementations was made. First, the input matrix of the appropriate size is loaded to the host memory, and then copied to the GPU memory (VRAM), which is a slow process. Then, a host loop goes through all columns. At each iteration, one or more kernels is launched, and each one is executed once per element of the lower-triangle of the column (one kernel execution is one element of the matrix). We tested three different techniques for launching the kernels:

1. In the **Normal Single Kernel** version, the kernel code for each element is the same for both the main diagonal element and all the elements below it, with a conditional statement checking which of the two the current execution is. A conditional statement such as this one may be very damaging to performance, specially on platforms such as GPUs, which take advantage of serial operations on large sets of data. This version also suffers from lack of synchronization between the threads, which means that the lower-triangle threads must re-calculate the value of the main diagonal value in order to process their own, which may also result in loss of performance. The kernel for this implementation can be found at the Appendix (A.4);
2. In the **Normal Multiple Kernel** version, there are two kernels: one calculates the main diagonal element of the column, and the other is used to

calculate each element below it. The first kernel is executed only by one thread, and the other kernel for the rest of the column, and synchronization between them is guaranteed, meaning that the main diagonal element will always be calculated before any of the other elements are executed, removing the need for re-recalculation. However, this version may suffer from higher overhead, since it needs to launch two times more kernels than the Single Kernel version;

3. The **In-Place Multiple Kernel** version also uses multiple kernels, that is, two kernels for each column (one for main diagonal, one for the rest). However, this version uses the same matrix for its input and output. This small change can yield a big performance boost, as only half as many memory is needed, therefore more of useful data can be saved on cache. The only drawbacks of this version come from the fact that it requires the use of two kernels, as it's the only way that global synchronization of CUDA threads can be achieved, and it is not possible to perform the re-calculation that the normal single kernel version uses.

Regardless of the approach used for kernel launching, a CUDA kernel call needs to know how many threads to execute. This number however, is not mapped directly to the GPU, as CUDA works in blocks (similar to Local Groups in OpenCL). Each block is independent and has a fixed number of threads, and both the number of total blocks and the number of threads per block are limited by the GPU being used. The code needs to calculate these values on run-time, based on the number of elements in needs to calculate. For testing purposes, we made the number of threads per block (TPB) be a parameter for benchmarking, and the total number of blocks was calculated based on the TPB and the total number of elements that need to be calculated, and can be described by the formula: $NumberOfBlocks = ceil(ElementsToCalculate/TPB)$.

Note that this setup may result in more threads being launched than necessary, possibly resulting in performance loss. Also, the kernel code must ensure that it was tasked to calculate a valid element of the matrix, meaning an additional if-statement must be performed, which in turn may also result in performance loss. In order to minimize this effect, as well as maximizing concurrency, a bigger number of blocks, with as few threads per block as possible, is crucial. However, as mentioned before, the number of blocks, as well as the number of

threads per block, is limited by hardware. This means, for instance, that for a 2000x2000 matrix, on a GPU that has a maximum of 512 blocks, and 512 threads per block, the TPB can be no less than 4.

4.6 Using LAPACK

In order to benchmark CPU optimizations of the Cholesky decomposition, we wrote a simple program that makes the appropriate external calls to the LAPACK implementation and measured its elapsed time. LAPACK is a high performance library for linear algebra problems.

Once we had a simple testing program, which initialized matrices on the same way as previously discussed (contiguous, float array) the LAPACK call corresponding to Cholesky decomposition was made. It's worth noting that the calculation is also done in-place (the input matrix is also the output matrix). The call itself is SPOTRF (where the 'S' stands for single-precision), and it had as arguments the size of the matrix, the input/output matrix, and which diagonal would the result be placed.

4.7 Using cuSOLVER

In order to benchmark the GPU optimizations, we used cuSOLVER's Cholesky decomposition calls. cuSOLVER is a CUDA library that provides useful LAPACK-like features.

The testing program creates matrices the same way as previously discussed, and is copied to the GPU memory the same way as any CUDA program does. There are two cuSOLVER functions that have to be called in order to perform the calculation: *cusolverDnSpotrf_bufferSize* and *cusolverDnSpotrf* (note the *"Spotrf"* on the function names, which is a LAPACK standard). At the end, the memory buffer is copied back to the host. Just as every other GPU implementation discussed in this work, the calculated elapsed time includes the memory transfers.

5 EVALUATION

In this final section, we will discuss the evaluation process of the implementations discussed so far. In order to have meaningful and correct experiment results, we followed a scientific and systematic approach (JAIN, 2015).

This section will detail the design of experiments, as well as how their executions were conducted. Finally, we will analyse the results of these experiments, comparing algorithm designs, implementation parameters, and execution platforms, in order to determine which techniques yield the best performance.

5.1 Experiment Design

A total 7 experiments were conducted, with the only shared parameter being the input matrix size. For all implementations, the matrices were initialized in the same way, with same values (although this wouldn't change the elapsed time). All experiments were executed for 6 different matrix sizes (250, 500, 1000, 2500, 5000 and 7500) in order to measure different data loads.

- For the sequential implementations, we tested the six different sequential implementations (Banachiewicz, Crout with double reference, Crout with contiguous allocation, transposed Crout with contiguous allocation, in-place Crout with contiguous allocation, and in-place transposed Crout with contiguous allocation) in order to find the best known sequential implementation. All parallel algorithms were developed based from this implementation, as the results from this experiment were used for Table 4.1. We also used this result as the basis for calculating the speedup of the other algorithms;
- For the OpenMP implementations, we executed all three different OpenMP implementations (simple, distributed sum and in-place) with varying cutoff values (0, 128, and 512) and number of threads (1, 2, 4, 8, 16, and 32) in order to measure how the speedup would change depending on the number of threads used;
- For the OpenCL implementations on CPU, we executed all three OpenCL implementations (simple single kernel, simple multiple kernel, and in-place

multiple kernel) using the machine's CPU as the compute device, and varied the cutoff value (0, 128, and 512);

- For LAPACK, which was performed in order to compare the performance of a state-of-the-art implementation with our own algorithms, we varied the number of maximum threads (1, 2, 4, 8, 16, and 32) to compare the speedups with our OpenMP implementations;
- For the OpenCL implementations on GPU, we executed all three OpenCL implementations (simple single kernel, simple multiple kernel, and in-place multiple kernel) using the machine's GPU as the compute device, and varied the cutoff value (0, 128, and 512);
- For the CUDA implementations, we executed all three CUDA implementations (normal single kernel, normal multiple kernel, in-place multiple kernel) with varying Threads per Block (1, 2, 4, 8, 128, 512, and 1024) and cutoff values (0, 128, and 512);
- For cuSOLVER, which was performed in order to compare the performance of a state-of-the-art implementation with our own algorithms, we simply executed to all different input sizes.

All experiments were generated using a DoE (Design of Experiments) package in R (GROEMPING; AMAROV; XU, 2016), which generated full factorial experiment design (SAS, 2016), with 15 replications for each argument permutation. The experiments are randomized in order to prevent any kind of execution bias. Bash scripts were used in order to automatically execute the experiments. The measured times in all experiments were taken **only** from the elapsed algorithm time, not counting the initialization process for the APIs and the environment. For the OpenCL and CUDA implementations, however, the data transfer procedure was measured. All experiment data (design, scripts and results) are available at <https://bitbucket.org/jruschel/parallel-cholesky>.

5.2 Test Machine

All experiments were executed on two identical machines, with the following configurations for the CPU (INTEL, 2014) and GPU (NVIDIA, 2014b):

Table 5.1: Experimental Unit Description

Property	Machine
Processor (CPU)	2 x Intel Xeon E5-2650 v3 (10 cores)
Hyperhreading	Yes
Total Compute Units	40 (2 x 2 x 10)
Max. Core Frequency	2.3 GHz
Cache	25 MBytes
GPU	Tesla K80
Max. Frequency	823 MHz
CUDA cores	4992
VRAM	24 GByte GDDR5
Max. Threads per Block	1024
OS	Linux Ubuntu 16.04

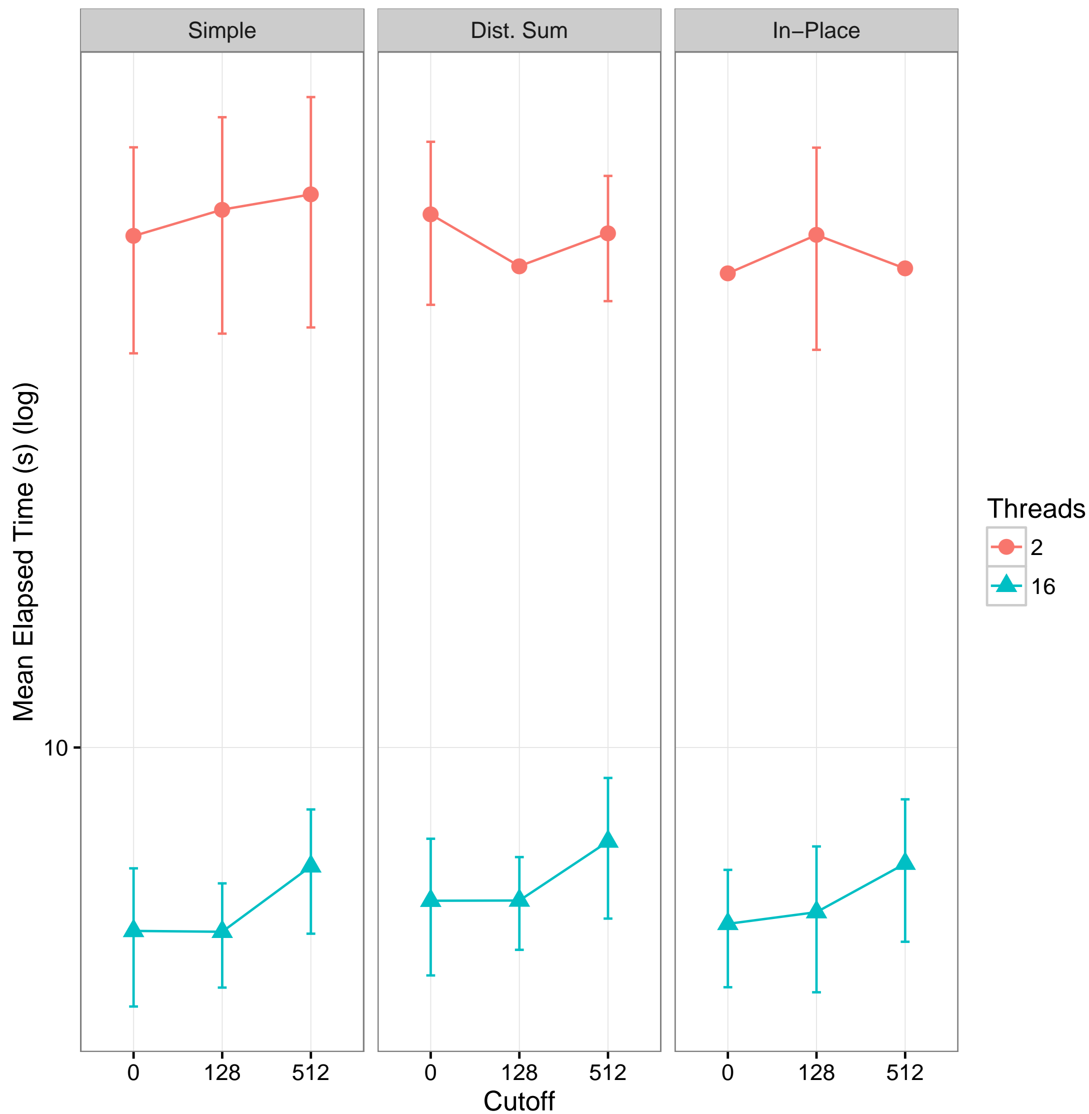
Note that the total number of compute units (threads) is 40 due the machine having two Intel Xeon E5-2650 v3, with *hyperthreading* enabled. *Hyperthreading* is an Intel technology which allows for a single core to act as two, virtually doubling the number of threads for a single processor. The machines also had 125Gbytes of DDR4 RAM.

5.3 Performance Analysis for CPU Parallelizations

There were a total of 3 experiments focusing on parallelization on CPU cores: the OpenMP implementations, the OpenCL implementations with CPU as the compute device, and the implementation using LAPACK. The results presented here are also being compared with the sequential execution on the same hardware.

5.3.1 OpenMP

Figure 5.1: Comparing the elapsed time in seconds (Y axis, log) for different cutoffs (X axis) for all three OMP implementations (horizontal grid), with 2 and 16 threads (vertical grid), and a fixed input size of 7500.



As figure 5.1 demonstrates, changing the cutoff value (the number of columns at the end of the matrix that will be executed sequentially) is statistically irrelevant, and yielded no increase in performance. Regardless of threads, algorithm and input size, the elapsed time are within the standard error of each other.

Figure 5.2: Comparing speedups (Y axis) for all three OMP implementations (horizontal grid), sizes 250, 5000, and 7500 (vertical grid), and 2, 8 and 32 threads (X axis). The dotted line represents the ideal speedup (which is equal to the number of used threads).

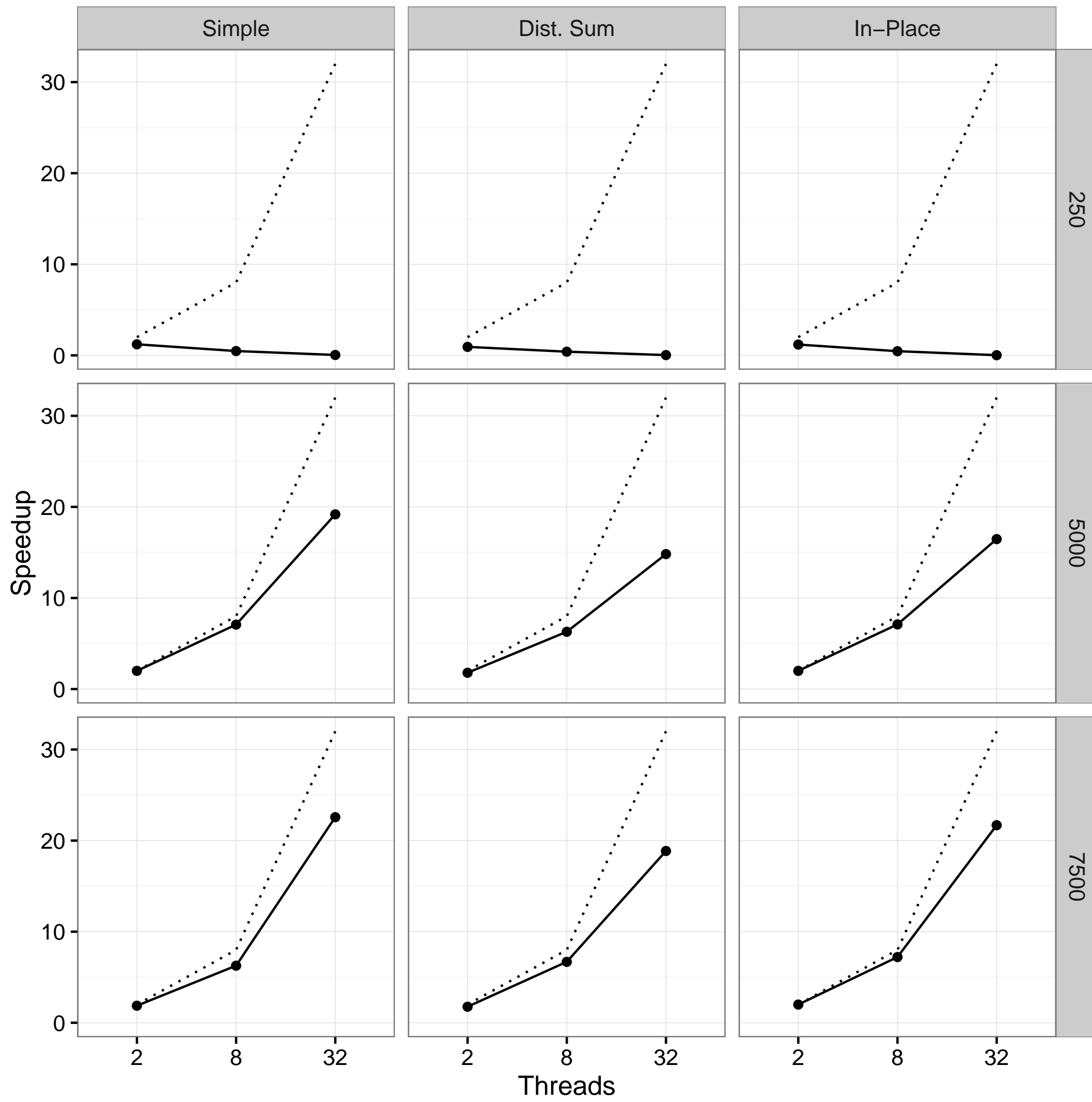
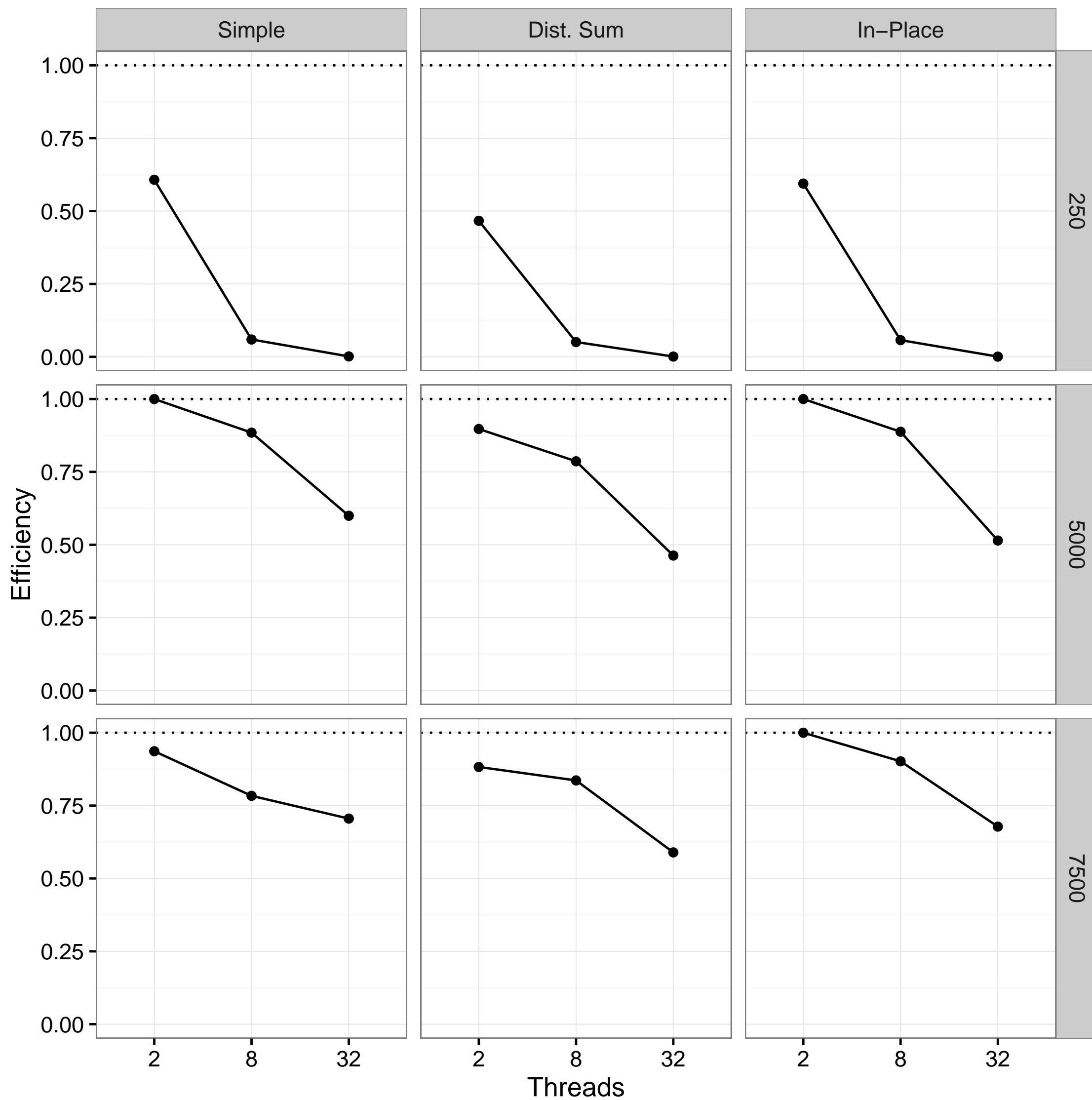


Figure 5.3: Comparing efficiency (Y axis) for all three OMP implementations (horizontal grid), sizes 250, 5000, and 7500 (vertical grid), and 2, 8 and 32 threads (X axis). The dotted line represents the ideal efficiency (1).



As figures 5.2 and 5.3 demonstrate, using a parallel implementation for small sizes (like 250) offers no speedup, as the number of calculations is too small, and the overhead of creating and managing threads is too big when compared with the time spent on actual computation. As the input size increases, though, the speedup begins approaching its ideal, as more time is spent on actual parallel computing, not on managing the threads. As for the algorithms, they offer similar speedups in all cases, although the In-Place implementation has a slightly better overall result. For all experiments, the efficiency decreases as the number of threads increases, which may be attributed to the fact that the more threads there are, the more memory access is being made at the same time at very distant

places in memory, which in turn could hurt the cache performance.

5.3.2 OpenCL

Figure 5.4: Comparing the elapsed time in seconds (Y axis, log) for different cutoff values (X axis) for all three OpenCL implementations executing on CPU, for sizes of 250, 2500, and 7500.

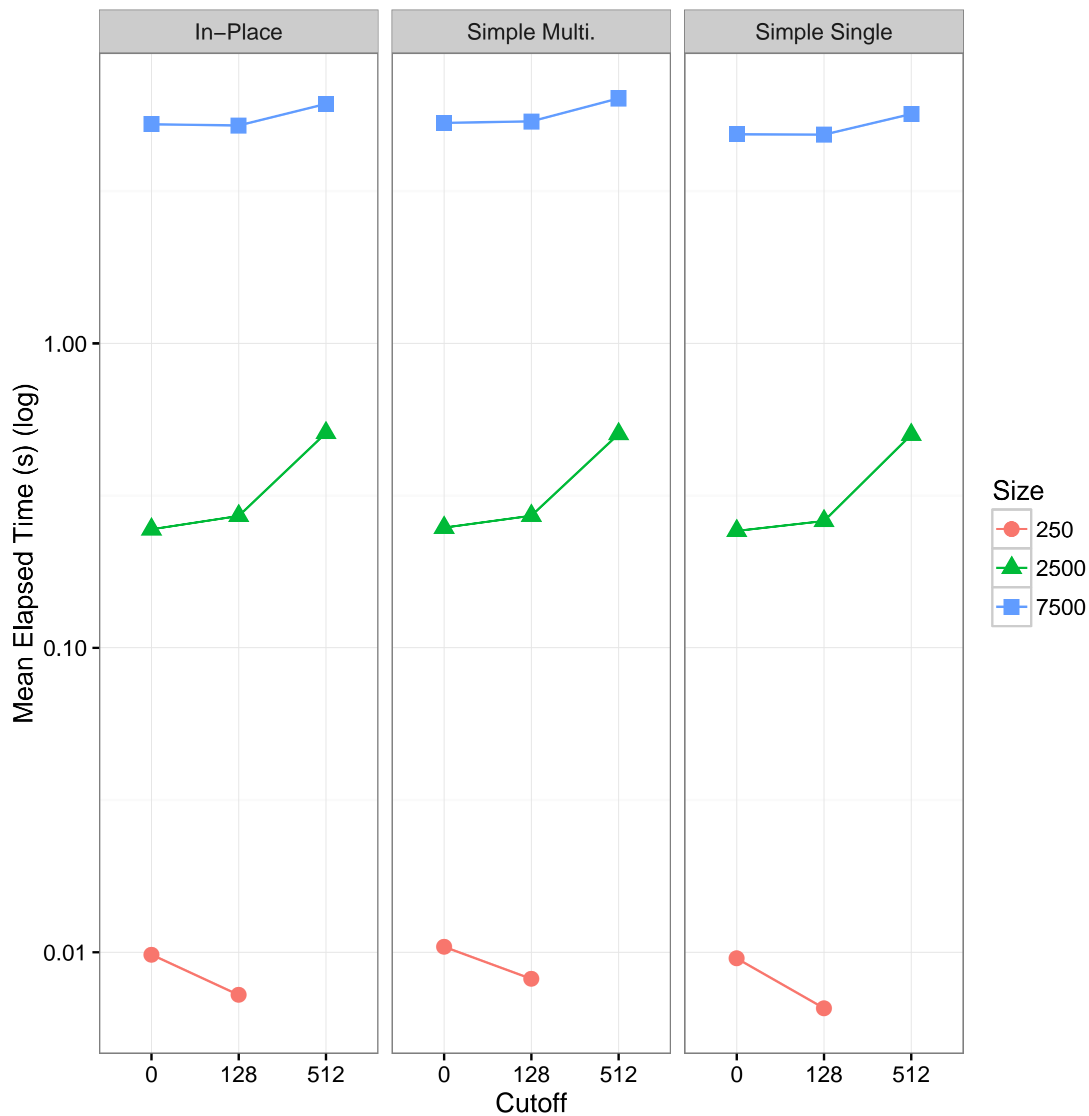
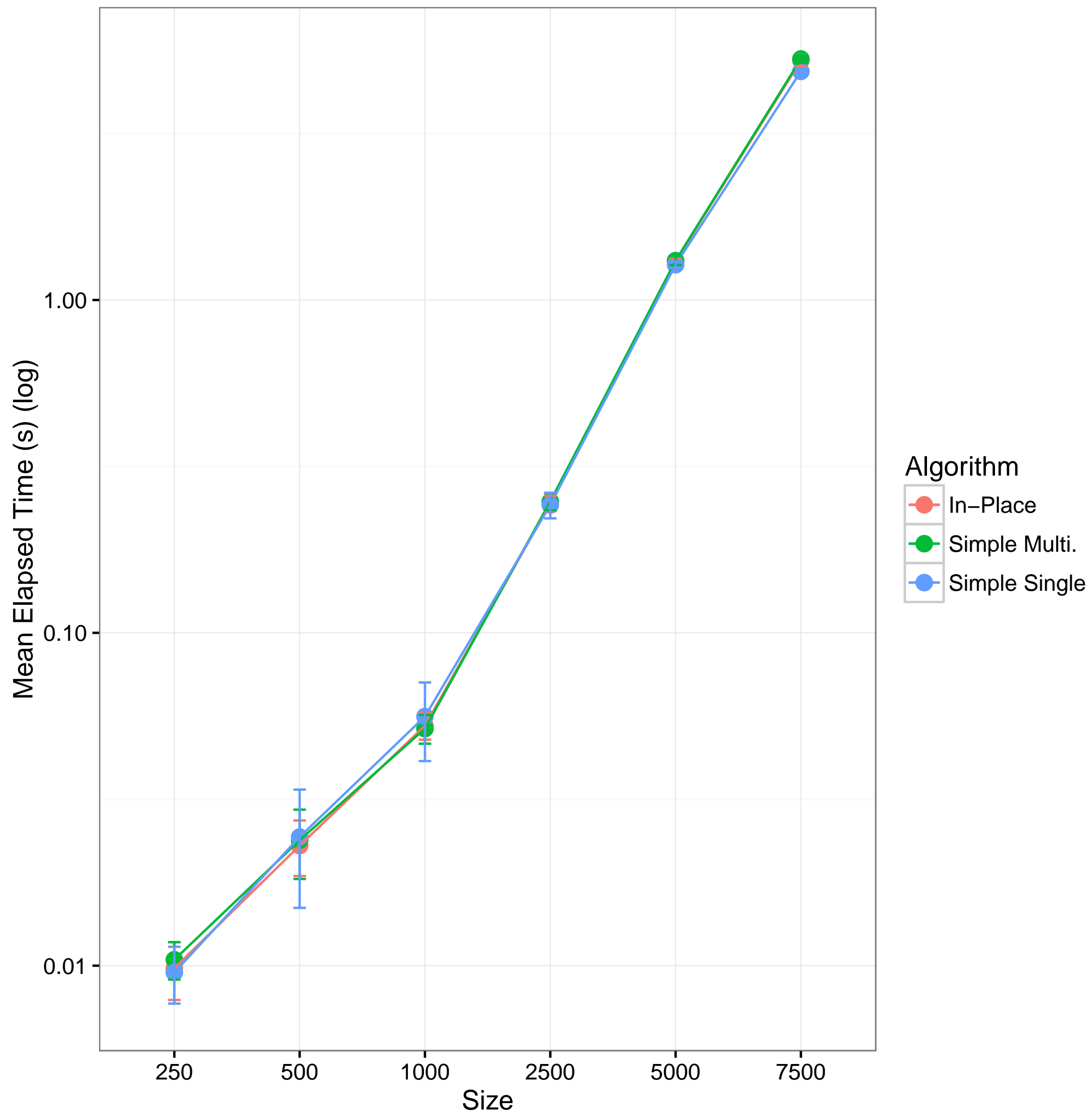


Figure 5.4 shows the mean elapsed time in seconds (in log scale) for the smallest, biggest and medium input size values. For small matrices, a bigger cut-off yields a better result, as the overhead for the communication CPU <-> Device (such as kernel enqueuing) is too big, when compared to the actual computation that needs to be made. As the matrices size increase, however, this overhead gets

relatively smaller, and having a smaller cutoff value is better, as more calculations can be done in parallel.

Figure 5.5: Comparing the elapsed time in seconds (Y axis, log) for all three OpenCL implementations executing on CPU, for a fixed cutoff value of 0.



As figure 5.5 demonstrates, the three OCL implementations have no significant statistical difference in performance. This may indicate that using dual kernels for the calculation, instead of a single kernel, doesn't decrease the performance in any significant amount, although this could be because of the need for double calculation when using a single kernel. These results also show that having in-place calculation on OCL offers no speedup (while having this same technique with OMP did provide some performance boost). Although the algorithms are statistically equivalent, we will consider the Simple Single implementation as having the best OCL performance, as the mean times were slightly smaller than

the other implementations.

5.3.3 Sequential x OpenMP x OpenCL x LAPACK

Figure 5.6: Comparing speedups with 32 threads for the best OMP implementation (In-Place, with cutoff value of 0), the best OCL implementation on the CPU (Simple Single, with cutoff value of 0), and LAPACK. The dotted line represents the ideal speedup (32).

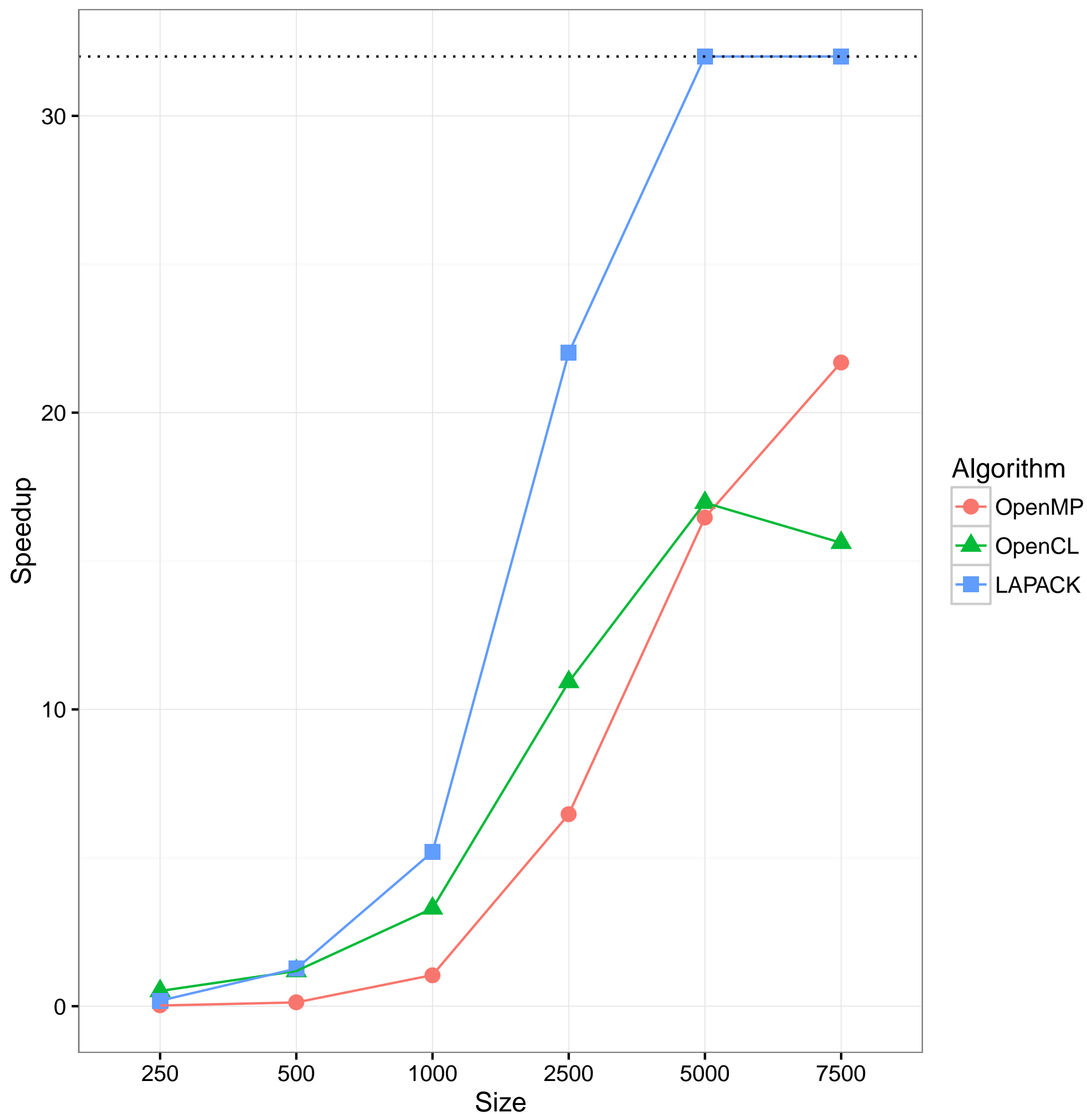


Figure 5.6 compares the speedups for the best Cholesky decomposition implementations for CPU parallelization (In-Place for OMP, Simple Single for OCL), and an implementation using LAPACK. For small (less than 1000) input sizes, the speedups are very similar (and small, as was mentioned earlier). However, as the matrices size increases, LAPACK fastly approaches ideal speedup, while the

implementations presented in this work reach only as high as 22 (OMP implementation for size 7500). This was to be expected, as LAPACK is a professional library that has been developed for over two decades.

Figure 5.7: Comparing elapsed times (Y axis, log) for the best OMP implementation (In-Place, with cutoff value of 0), the best OCL implementation (Simple Single with cutoff value of 0), LAPACK and the best sequential time (Contiguous Crout In-Place), considering 32 threads.

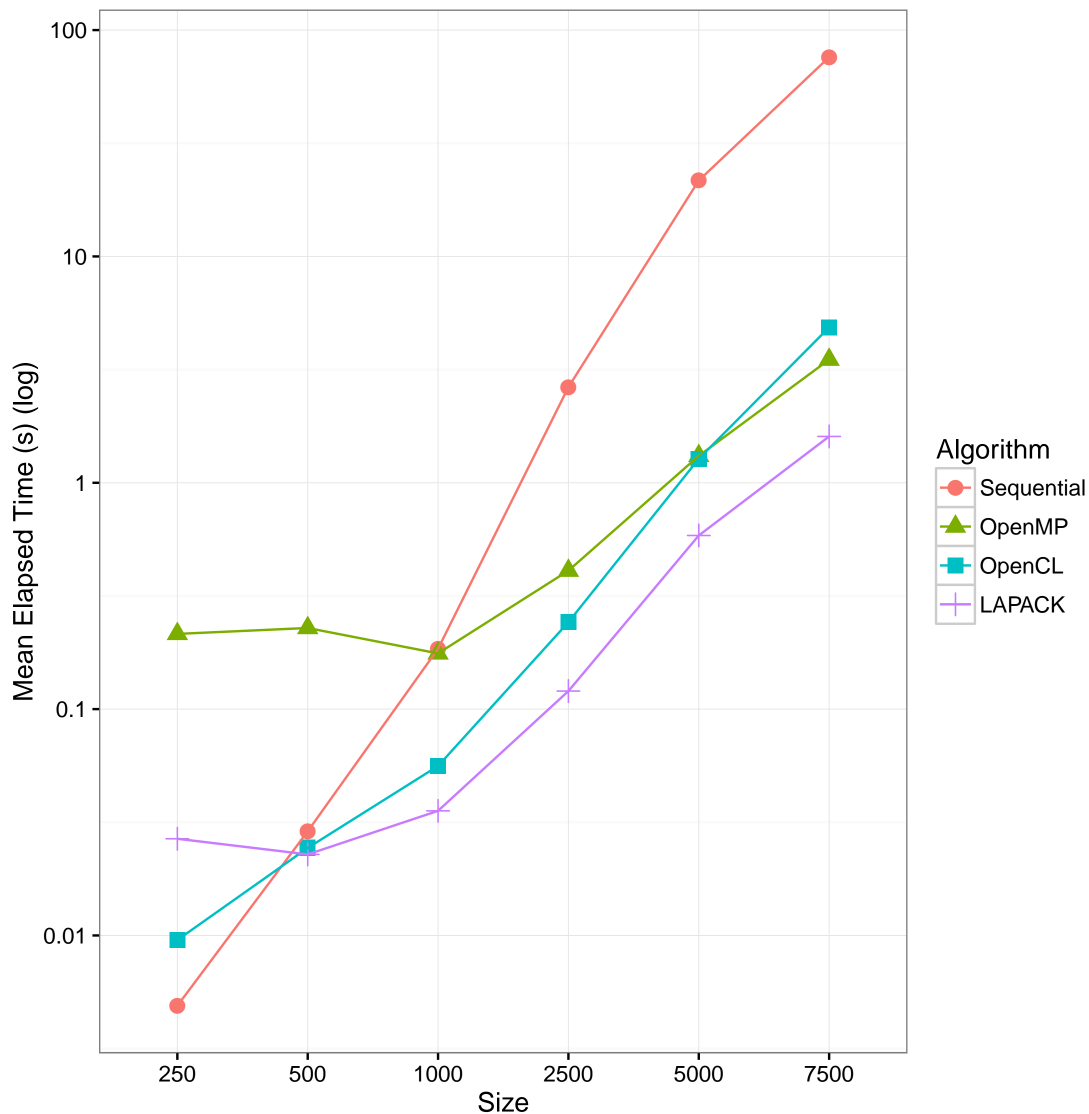


Figure 5.7 compares the sequential, the best parallel versions discussed in this work, and the implementation using LAPACK for various sizes. For small sizes, a sequential execution is faster than any parallel implementation, while for sizes larger than 1000, it is slower. The OCL implementation was faster than LAPACK for small matrices (250 and 500), however LAPACK outperformed both OCL and OMP for all sizes larger than that.

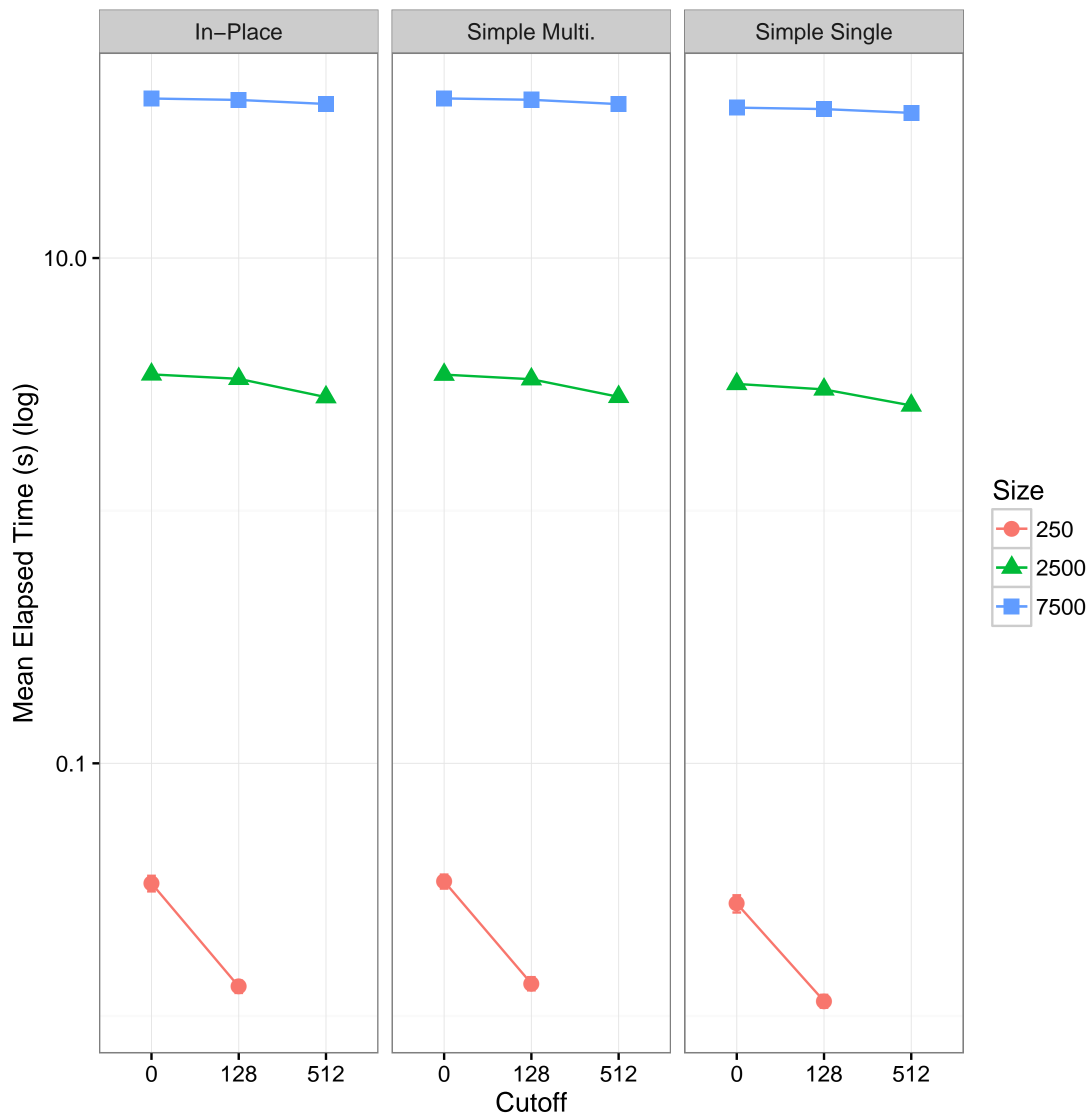
Between the OCL and OMP implementations, the former performed better than the later on most of the input sizes. One of the reasons this may be is due to different overhead properties on both APIs: OpenMP may have a worse overall overhead, but one which has a good scalability (given how it has bad performance for small sizes, but rapidly increases speedup for bigger sizes); while OCL may have a smaller overall overhead, but one which doesn't have a good scalability (given how it has the best performance for small sizes). Also, the machine in which we made these experiments has a very powerful CPU (40 cores at 2.3 GHz). Finally, OpenCL had an interesting decline in performance for very large sizes (specially for 7500), which may be a result of the fact that, as this OCL implementation is using the CPU both as host and as device, the matrices may be doubled on CPU memory, which would decrease the cache performance.

5.4 Performance Analysis for GPU Parallelizations

There were a total of 3 experiments focusing on parallelization on the GPU: the OpenCL implementations with GPU as the compute device, the CUDA implementations, and the implementation using cuSOLVER. The results presented here are also being compared with the sequential execution on the same hardware.

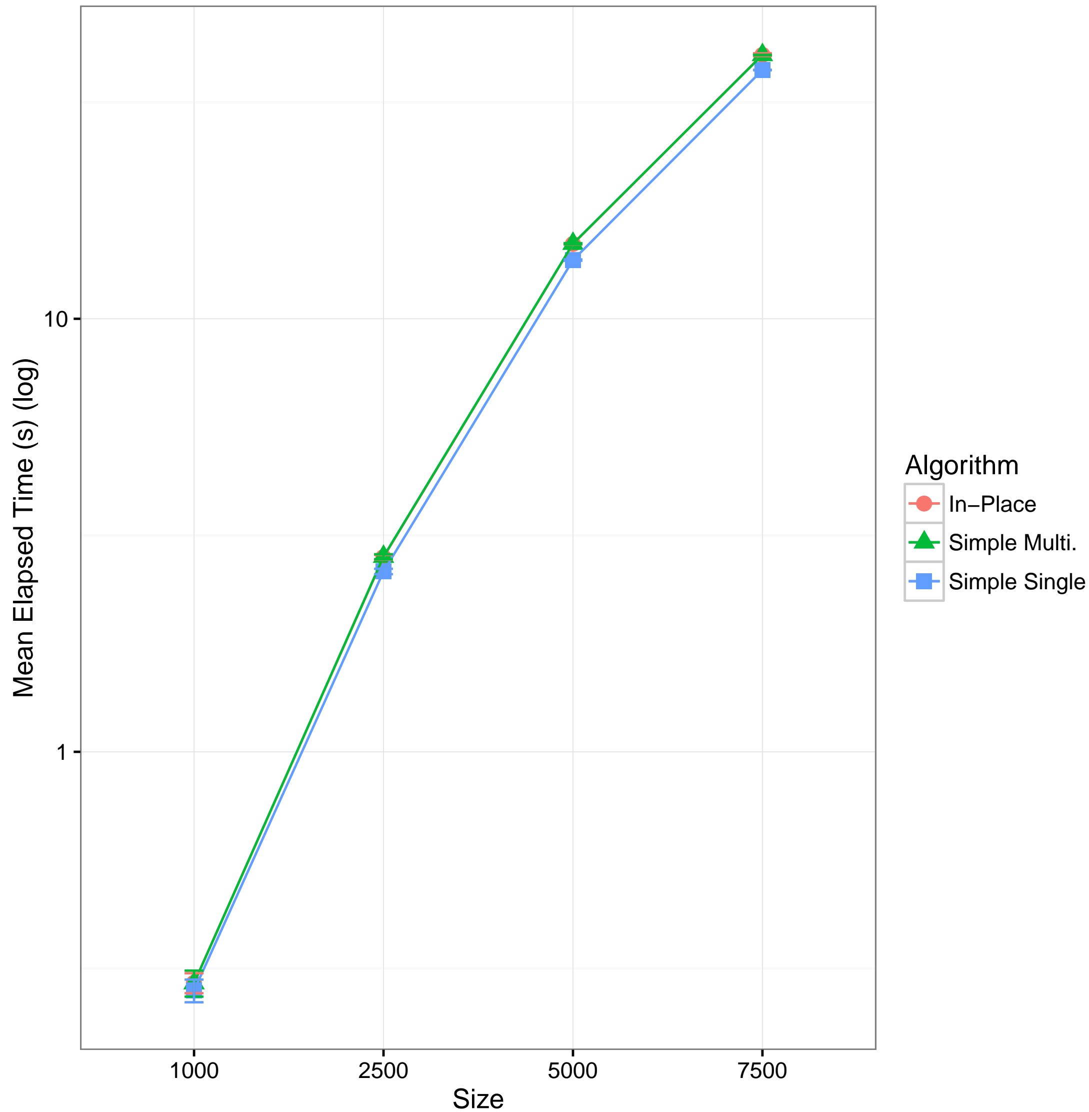
5.4.1 OpenCL

Figure 5.8: Comparing elapsed times in seconds (Y axis, log) for different cutoff values for all three OCL implementations on GPU, for sizes of 250, 2500, and 7500 (X axis).



For varying cutoff values, figure 5.8 demonstrates that increasing the number of iterations running on the host actually increases, even if slightly, the overall performance of the algorithm. This result is a sharp contrast to the same OCL implementation when executed on the CPU (figure 5.4), which showed that increasing cutoff values makes for a worse execution time. An explanation for this result is the fact that the communications overhead when dealing with CPU and GPU is far greater than when dealing only with the CPU.

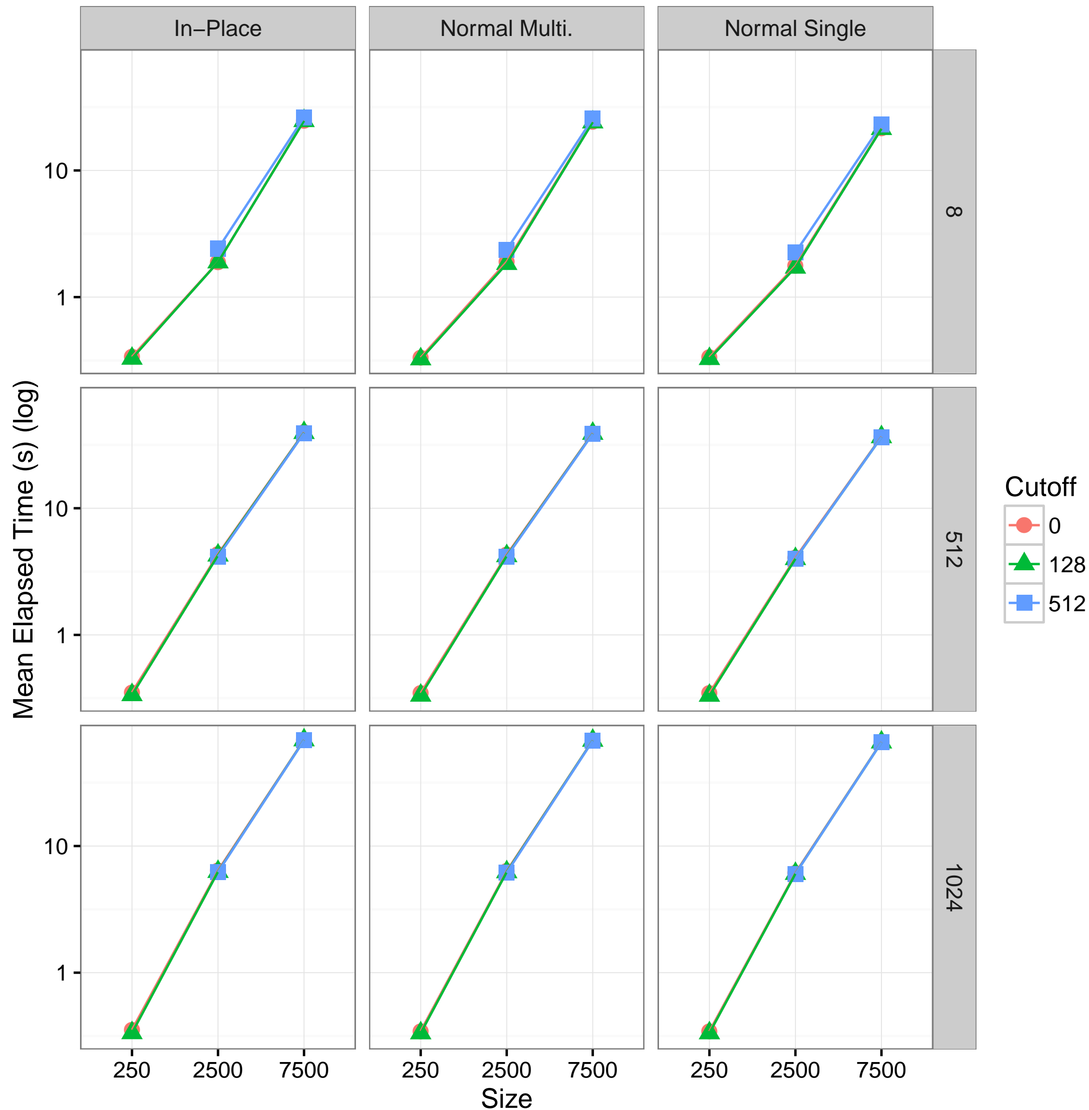
Figure 5.9: Comparing elapsed times in seconds (Y axis, log) for all three OpenCL implementations executing on GPU, for a fixed cutoff value of 512.



Similar to the results of the OpenCL implementations when executed on the CPU (figure 5.9), figure 5.9 shows that all three of the implementations behave in a very similar way, offering little to no speedup when compared to each other. However, the Simple Single version behaves slightly better than all others for bigger sizes (2500 and up), which can be the result of having only a single kernel enqueuing per iteration (unlike the other two, which have two kernel enqueuings per iteration), as the communication between host and kernel is far slower when dealing with GPUs.

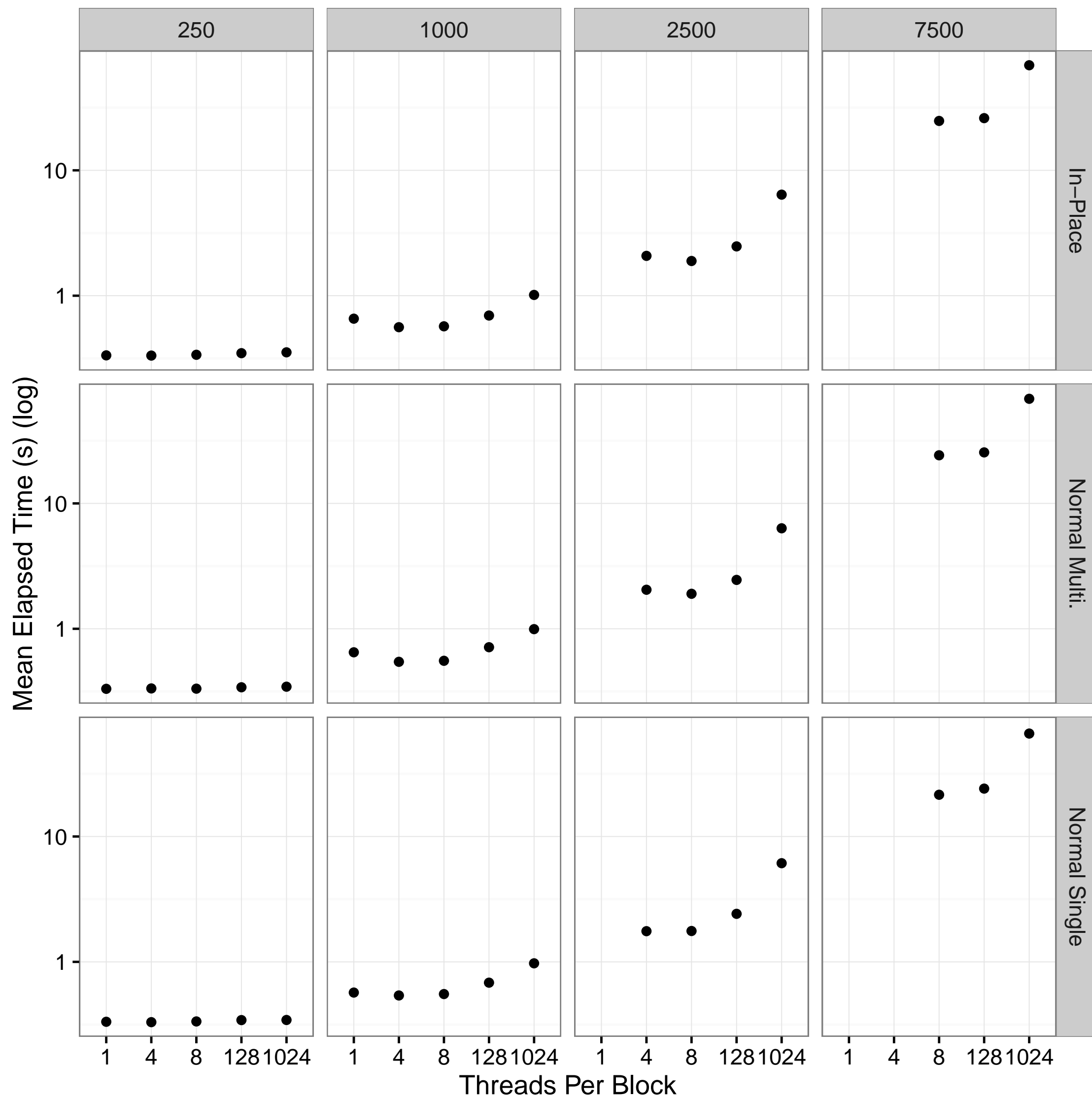
5.4.2 CUDA

Figure 5.10: Comparing elapsed times in seconds (Y axis, log) for different cutoff values for all three CUDA implementations (horizontal grid), and varying TPB values (vertical grid).



As figure 5.10 clearly demonstrates, different cutoff values offer no speedup for any of the CUDA implementations, which is a major difference from the OCL implementations on GPU (figure 5.8), which had a benefit from bigger cutoff values. CUDA, being developed by NVIDIA, will naturally have a better performance with NVIDIA GPUs than any other API on the same hardware, and such performance increase may come in the form of reduced communication overhead between host and kernel code, which was the main reason a cutoff value would be used.

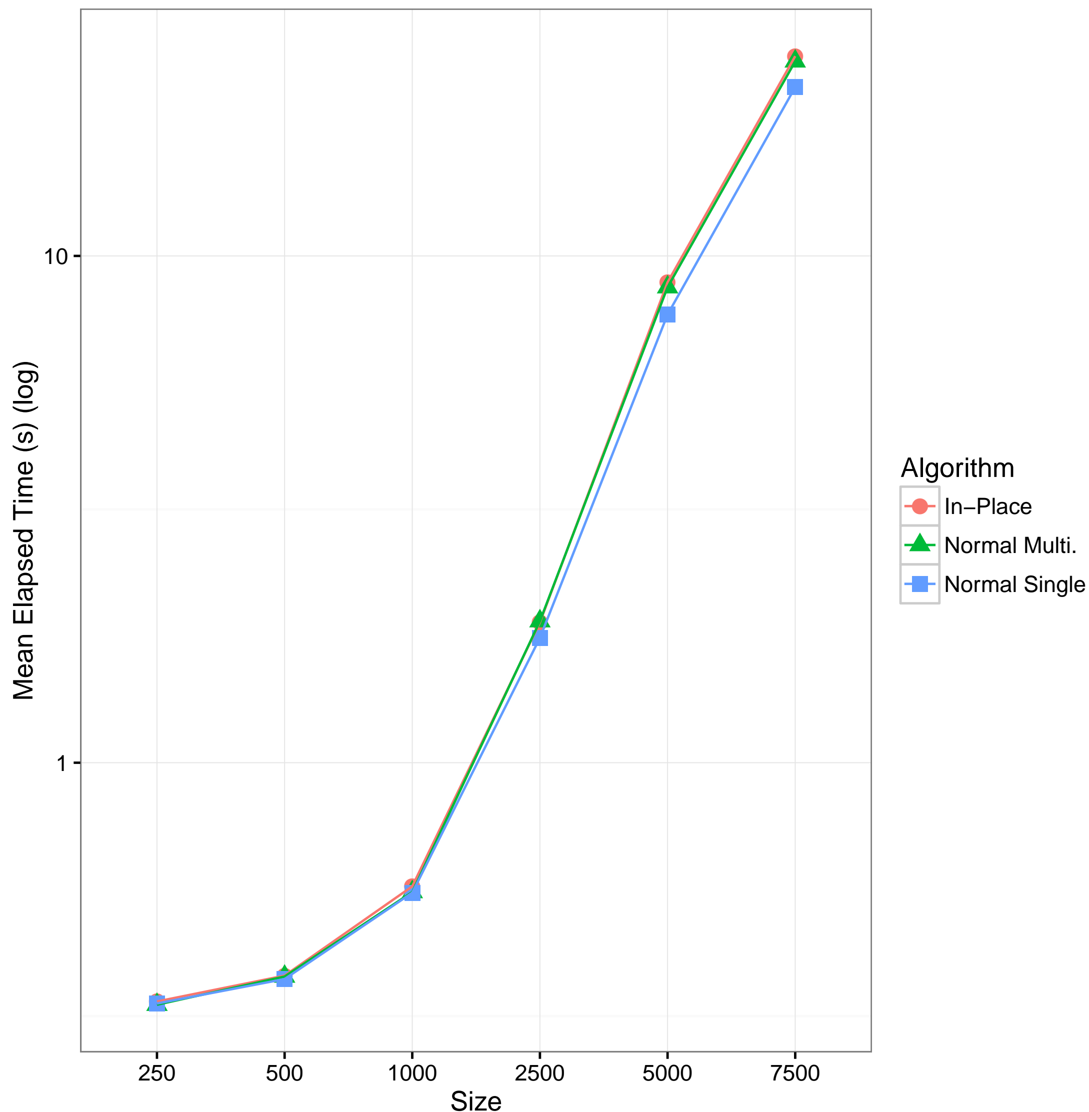
Figure 5.11: Comparing elapsed times in seconds (Y axis, log) for different TPBs (Threads per Block) for all three CUDA implementations (vertical grid), and varying input sizes (horizontal grid).



As figure 5.11 shows, varying the number of Threads per Block yields different results depending on the size of the matrix being calculated. For small sizes (250), different TPBs had no noticeable effect on the overall performance of any of the implementations. However, as the input size increases, smaller TPBs result in faster execution. Note that some permutations of the parameters are not on this graph (such as TPB of 4 for an input size of 7500) due to hardware limitations (the Tesla K80 has a maximum of 1024 total blocks, while a matrix of size 7500 and 4 TPB would require 1875 (7500/4) blocks). For all input sizes in which it made a significant difference on the overall performance, the smallest TPBs were the fastest in execution time, although the optimal TPB was of 8, even

when it was possible to be smaller. This was to be expected, as less threads per block means more blocks are created, meaning more of the GPU is actually used, and more block parallelization is achieved. Therefore, we considered a TPB of 8 as the optimal value for all implementations.

Figure 5.12: Comparing elapsed times in seconds (Y axis, log) for all three different CUDA implementations for varying input sizes, considering 8 Threads per Blocks and a cutoff value of 0.



The experiments being shown on figure 5.12 demonstrated how the three different CUDA implementations behaved very similarly for all input sizes, although the Normal Single version had slightly faster execution times than the other two. One of the reasons that the Normal Single implementation performed better has to do with having only a single kernel enqueuing per iteration (similar to the way that the Simple Single version was better on the OCL experiments),

unlike the other two, which have two kernel enqueing per iteration.

5.4.3 Sequential x OpenCL x CUDA x cuSOLVER

Figure 5.13: Comparing elapsed times in seconds (Y axis, log) for the best OpenCL implementation executed on the GPU (Simple Single with cutoff value of 128 for sizes 250 and 500, and 512 for all other sizes), the best CUDA implementation (Normal Single with 8 TPB and a cutoff value of 0), cuSOLVER and the best sequential algorithm (Contiguous Crout In-Place).

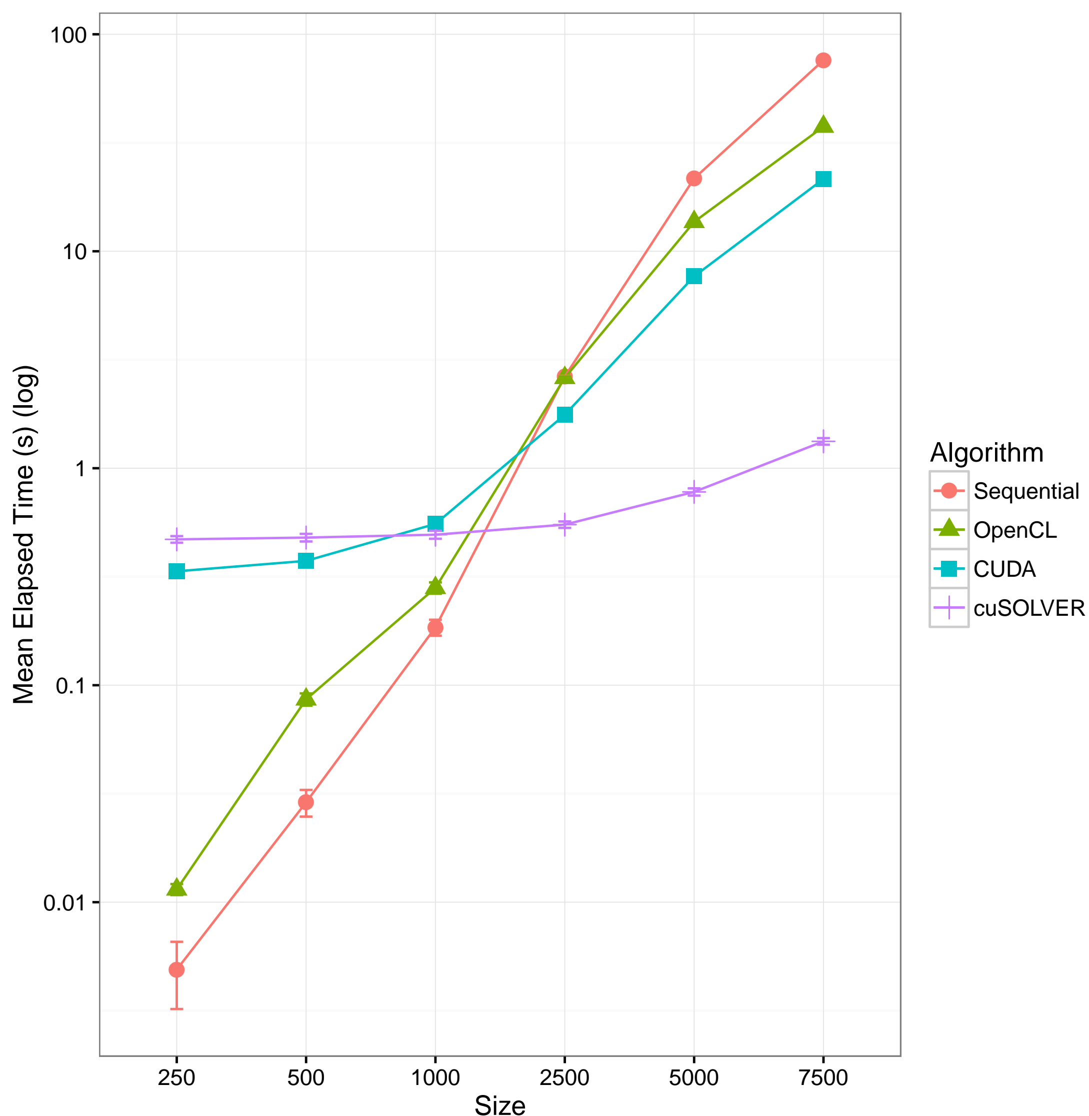


Figure 5.13 compares the best GPU implementations presented in this work with a sequential execution. For small matrix sizes (less than 1000), executing the algorithm sequentially had better results than for any GPU implementation (including cuSOLVER), which can be easily explained by the fact that, for small ma-

trices, there is a big combined overhead involved in memory transfers between RAM and VRAM, plus the communication needed for kernel enqueuing. However, for bigger matrix sizes, this overhead pays off, and all GPU implementations show some amount of speedup. For these large matrices, the CUDA implementation outperformed OpenCL's, which, given how similar both implementations are, may indicate that CUDA's API produces less overhead than OpenCL's when dealing with large data sets. cuSOLVER was still significantly better than both, which was to be expected. For small matrices, both CUDA and OCL implementations had better performances than cuSOLVER, which may be due to both implementations having smaller overheads than the state of the art CUDA implementation, as additional environment variables and function calls have to made in order to use the state of the art implementation.

The OpenCL version that is shown on this graph is one with the largest cutoff values that were tested, which may the reason why its shape follows the overall curve of the sequential execution. As seen on figure 5.8, higher cutoff values yielded better results for the OpenCL on the GPU, which means that the library was not able to fully use the power of the hardware. Given by how similar the CUDA and the OpenCL implementations are, it is likely that either the API presents worse performances on the GPU, or the hardware in which these results were generated provides bad support for OpenCL.

5.5 Overall Performance Analysis

Figure 5.14: Comparing elapsed times in seconds (Y axis, log) for the best OpenMP implementation (In-Place with cutoff value of 0), the best OpenCL implementation (Simple Single with cutoff value of 0 on CPU, and 512 on GPU), the best CUDA implementation (Normal Single with 8 TPB, and a cutoff value of 0), LAPACK, cuSOLVER and the best sequential implementation (contiguous Crout in-place) with various input sizes.

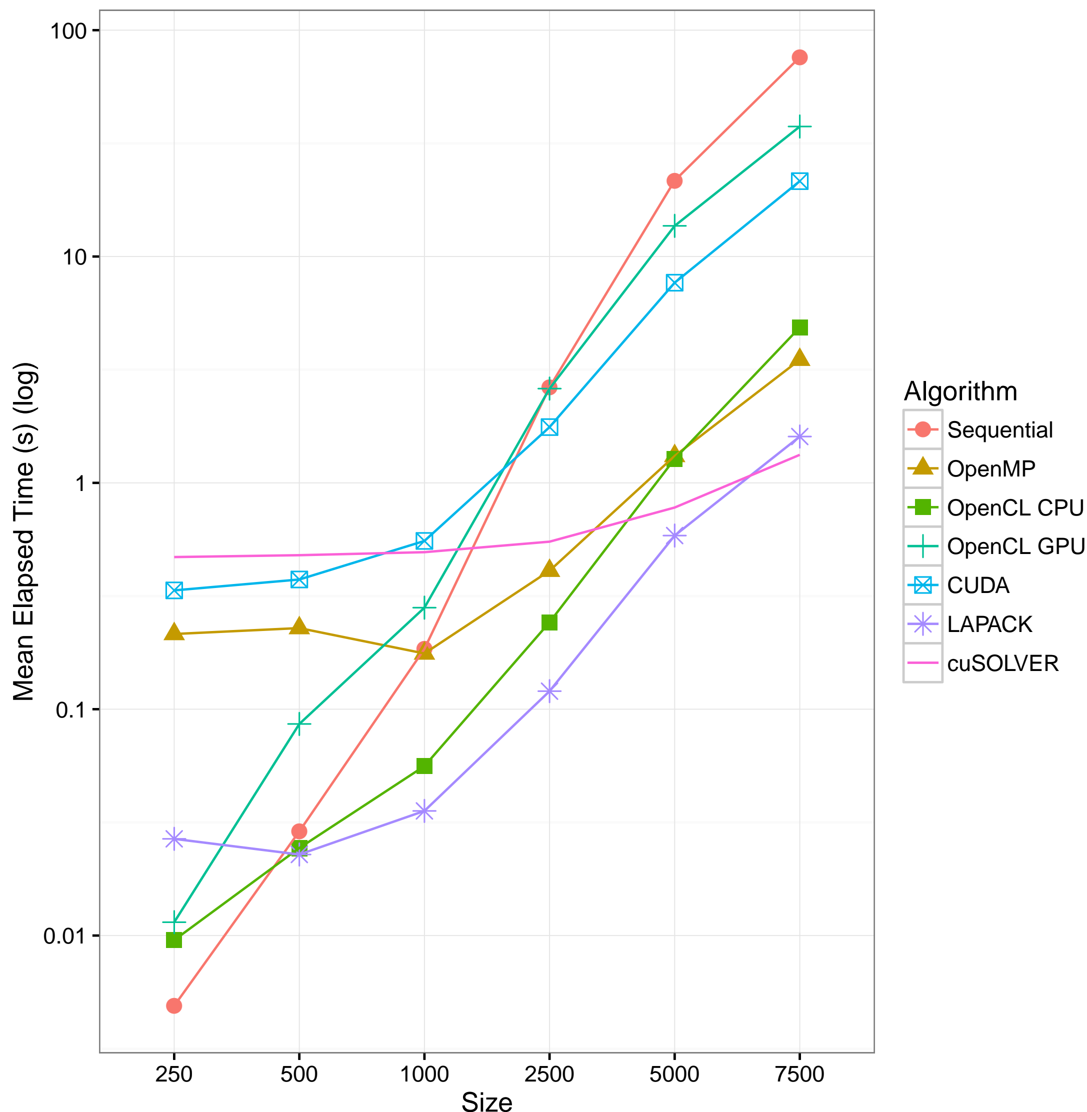


Figure 5.14 is a summary of all the experiments conducted for this work, presenting the results for the best parameters for each of the APIs used. The graph shows, for each matrix size, whether is best to use parallelization on the CPU, on the GPU, or execute sequentially. Already at matrix sizes of 500, LAPACK and OpenCL on the CPU started presenting some speedup, while GPU implementations only between sizes 1000 and 2500, which can can be explained

by the fact that the GPU implementations have a much larger base overhead than CPU parallelization.

OpenCL (CPU) and LAPACK had the best performance for medium-sized matrices, and a GPU execution was only able to be better than both at the largest size we experimented. We expect that this trend would continue should we have continued to larger matrix sizes.

6 CONCLUSION

This work presented and compared different algorithms and implementations for the Cholesky decomposition, using several different well-known libraries and APIs for parallelization on the CPU and the GPU. The experiments conducted and presented in this work may also be used in order to help determine the best techniques to optimize the implementation for similar problems.

Our findings can be split into three groups, depending on the size of the input matrix: for very small matrices, an optimized sequential execution is both easier and faster than one with any type of parallelization; for medium-sized matrices, parallelization on the CPU had very good speedups and is the best option; for large-sized matrices, execution on the GPU began outperforming all other implementations. Note that the definition of “small”, “medium”, and “large” are very volatile, and may depend heavily on the hardware in which the implementations are executed. We experimented on a very powerful CPU, meaning that a higher performance on implementations that use CPU parallelization was to be expected.

When dealing with similar problems, the optimal strategy is to break the problem into Level 3 BLAS routines and use a parallelized LAPACK implementation for medium-sized matrices, and a GPU implementation (such as cuSOLVER) for large-sized matrices. However, if the problem cannot be specified as a series of BLAS calls, the best approach is to implement the algorithm in OpenCL, and execute it with the CPU as the compute device. If the CPU doesn’t support OpenCL, the next best option is OpenMP, which is also easier to use. For slow CPUs connected to modern, powerful GPUs, however, CUDA or OpenCL may have better performances even for medium-sized matrices.

Finally, as computers are tasked with the processing of larger datasets, it is the job of the software engineer to choose the right tools in order to implement highly optimized solutions. This work analysed the performance and implementation details of some of these tools on a mathematical context, and provided concrete results that may be used to guide programmers towards writing the best possible software.

The results reported in this study were generated in virtue of the agreement between Hewlett Packard Enterprise (HPE) and the Federal University of

Rio Grande do Sul (UFRGS), financed by resources in return for the exemption or reduction of the IPI tax, granted by Brazilian Law nº 8248, 1991, and its subsequent updates.

REFERENCES

ABDELFATTAH, A. et al. Performance tuning and optimization techniques of fixed and variable size batched cholesky factorization on gpus. **Procedia Computer Science**, 2016.

AKHTER, S.; ROBERTS, J. **Multi-core programming**. [S.l.]: Intel press Hillsboro, 2006. v. 33.

ANDERSON, E. et al. **LAPACK Users' Guide**. Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999. ISBN 0-89871-447-8 (paperback).

FANG, H.-r.; O'LEARY, D. P. Modified cholesky algorithms: a catalog with new approaches. **Mathematical Programming**, v. 115, n. 2, p. 319–349, 2008. ISSN 1436-4646.

GROEMPING, U.; AMAROV, B.; XU, H. **Package 'DoE.base'**. 2016. <https://cran.r-project.org/web/packages/DoE.base/DoE.base.pdf>. Accessed: 2016-04-24.

INTEL. **Xeon E5-2650 v3 Specifications**. 2014. <https://ark.intel.com/products/81705/Intel-Xeon-Processor-E5-2650-v3-25M-Cache-2_30-GHz>. Accessed: 2016-30-11.

INTEL. **Xeon E7-8890 v4 Specifications**. 2016. <http://ark.intel.com/products/93790/Intel-Xeon-Processor-E7-8890-v4-60M-Cache-2_20-GHz>. Accessed: 2016-30-11.

JAIN, R. **Art of Computer Systems Performance Analysis: Techniques for Experimental Design Measurements Simulation and Modeling**. 2nd. ed. [S.l.]: John Wiley & Sons, 2015.

JAJA, J. **Introduction to Parallel Algorithms**. 1. ed. [S.l.]: Addison-Wesley Professional, 1992. Paperback. ISBN 0201548569.

KARIMI, K. The feasibility of using opencl instead of openmp for parallel CPU programming. **CoRR**, abs/1503.06532, 2015.

KARP, A. H.; FLATT, H. P. Measuring parallel processor performance. **Commun. ACM**, ACM, New York, NY, USA, v. 33, n. 5, p. 539–543, maio 1990. ISSN 0001-0782.

KROESE, D. P.; TAIMRE, T.; BOTEV, Z. I. **Handbook of Monte Carlo methods**. [S.l.]: John Wiley & Sons, 2013. v. 706.

LTAIEF, E. et al. A scalable high performant cholesky factorization for multicore with gpu accelerators. In: **High Performance Computing for Computation Science - VECPAR**. [S.l.: s.n.], 2010.

MATHWORKS. **Cholesky Factorization (MatLab Documentation)**. 2016. <https://www.mathworks.com/help/matlab/ref/chol.html>. Accessed: 2016-30-11.

NICHOLLS, J. et al. Scalable parallel programming with cuda. **Queue - GPU Computing**, ACM New York, NY, USA, New York, USA, v. 6, p. 40–53, 2008. ISSN 1542-7730.

NVIDIA. **cuSOLVER Developer Page**. 2014. <https://developer.nvidia.com/cusolver>. Accessed: 2016-11-29.

NVIDIA. **Tesla K80 Specifications**. 2014. <http://www.nvidia.com/object/tesla-k80.html>. Accessed: 2016-30-11.

NVIDIA. **CUDA 8.0 Programming Guide**. 2016. <https://docs.nvidia.com/cuda/cuda-c-programming-guide>. Accessed: 2016-11-30.

OpenMP Architecture Review Board. **OpenMP Application Program Interface Version 4.0**. 2013. <http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf>. Accessed: 2016-11-29.

OWENS, J. Gpu architecture overview. In: **SIGGRAPH Courses**. [S.l.: s.n.], 2007. p. 2.

RAKOTONIRINA, C. On the cholesky method. **Journal of Interdisciplinary Mathematics**, v. 12, n. 6, p. 875–882, 2009.

REGE, A. An introduction to modern gpu architecture. **En ligne]**, 2008.

SAS. **Full Factorial Designs**. 2016. <http://www.jmp.com/support/help/Full_Factorial_Designs.shtml>. Accessed: 2016-04-24.

SMITH, A. J. Cache memories. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 14, n. 3, p. 473–530, set. 1982. ISSN 0360-0300.

SMITH, A. J. **Design of CPU Cache Memories**. [S.l.], 1987.

SMITH, S. P. Likelihood-based analysis of linear state-space models using the cholesky decomposition. **Journal of Computational and Graphical Statistics**, v. 10, n. 2, p. 350–369, 2001.

STONE, J. E.; GOHARA, D.; SHI, G. Opencl: A parallel programming standard for heterogeneous computing systems. **IEEE Des. Test**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 12, n. 3, p. 66–73, maio 2010. ISSN 0740-7475.

TOMPSON, J.; SCHLACHTER, K. An introduction to the opencl programming model. 2012.

APPENDIX A — CODE LISTINGS

The code snippets in this Appendix contain only the relevant code for the Cholesky decomposition implementations discussed in this work, in C. The full source code for all implementations discussed in this work is available at <https://bitbucket.org/jruschel/parallel-cholesky>.

Listing A.1: Modified sequential Cholesky-Crout implementation using row-wise contiguous memory allocation.

```
1  for (j = 0; j < dimensionSize; j++) {
2          float sum = 0;
3          for (k = 0; k < j; k++) {
4                  sum += L[j * dimensionSize + k] * L[j *
                      dimensionSize + k];
5          }
6          L[j * dimensionSize + j] = sqrt(A[j * dimensionSize
               + j] - sum);
7
8          for (i = j + 1; i < dimensionSize; i++) {
9                  sum = 0;
10                 for (k = 0; k < j; k++) {
11                         sum += L[i * dimensionSize + k] * L
                              [j * dimensionSize + k];
12                 }
13                 L[i * dimensionSize + j] = (1.0 / L[j *
                      dimensionSize + j] * (A[i *
                      dimensionSize + j] - sum));
14         }
15 }
```

Listing A.2: Simple parallel implementation of the Cholesky-Crout algorithm with OpenMP.

```
1  for (j = 0; j < dimensionSize; j++) {
2          float sum = 0;
3          for (k = 0; k < j; k++) {
4                  sum += L[j * dimensionSize + k] * L[j *
                      dimensionSize + k];
5          }
6          L[j * dimensionSize + j] = sqrt(A[j * dimensionSize
               + j] - sum);
7
8          #pragma omp parallel for private(i,k,sum) shared (A
              ,L,j,diagonal_value) schedule(static) if (j <
              dimensionSize - cutoff)
9          for (i = j + 1; i < dimensionSize; i++) {
10                 sum = 0;
11                 for (k = 0; k < j; k++) {
12                         sum += L[i * dimensionSize + k] * L
                              [j * dimensionSize + k];
13                 }
14                 L[i * dimensionSize + j] = (1.0 / L[j *
                      dimensionSize + j] * (A[i *
                      dimensionSize + j] - sum));
15         }
16 }
```

Listing A.3: Kernel for the parallel implementation "Simple Single Kernel" of the Cholesky-Crout algorithm with OpenCL.

```
1  __kernel void choleskyColumnSimple(__global float *A,
     __global float *L,
2          const unsigned int dimensionSize, const unsigned
              int col) {
3          const int g_id = get_global_id(0);
4          int row = col + g_id;
5          int k;
6          float sum = 0;
7          float value;
8          float sum_d = 0;
9          if (g_id == 0) {
10                 for (k = 0; k < col; k++) {
11                         sum_d += L[col * dimensionSize + k]
                               * L[col * dimensionSize + k];
12                 }
13                 L[col * dimensionSize + col] = sqrt(A[col *
                       dimensionSize + col] - sum_d);
14         }
15         else {
16                 for (k = 0; k < col; k++) {
17                         sum   += L[row * dimensionSize + k]
                               * L[col * dimensionSize + k];
18                         sum_d += L[col * dimensionSize + k]
                               * L[col * dimensionSize + k];
19                 }
20                 value = sqrt(A[col * dimensionSize + col] -
                       sum_d);
21                 L[row * dimensionSize + col] = (1.0 / value
                       * (A[row * dimensionSize + col] - sum))
                      ;
22         }
23 }
```

Listing A.4: Kernel for the parallel implementation "Normal Single Kernel" of the Cholesky-Crout algorithm with CUDA.

```
1  __global__ void choleskyKernel_NormalS(float* A, float* L,
     int dimensionSize, int col) {
2          const unsigned int tid = blockIdx.x * blockDim.x +
              threadIdx.x;
3          unsigned int row = col + tid;
4          int k;
5          float sum = 0;
6          float value;
7          float sum_d = 0;
8  
9          if (tid == 0) {
10                 for (k = 0; k < col; k++) {
11                         sum_d += L[col * dimensionSize + k]
                              * L[col * dimensionSize + k];
12                 }
13                 L[col * dimensionSize + col] = sqrtf(A[col
                      * dimensionSize + col] - sum_d);
14         }
15         else {
16                 if (row < dimensionSize) {
17                         for (k = 0; k < col; k++) {
18                                 sum   += L[row *
                                      dimensionSize + k] * L[
                                      col * dimensionSize + k
                                      ];
19                                 sum_d += L[col *
                                      dimensionSize + k] * L[
                                      col * dimensionSize + k
                                      ];
20                         }
21                         value = sqrt(A[col * dimensionSize
                              + col] - sum_d);
```

```
22
23                          L[row * dimensionSize + col] = (1.0
                               / value * (A[row *
                              dimensionSize + col] - sum));
24                  }
25          }
26  }
```