Cranfield University - School of Engineering

GPGPU applied to linear algebra for finance: example of Cholesky decomposition

Msc Thesis
Academic Year 2010-2011

Dominique CLAUSE

Supervisor: Andre Nedelcoux

September 2011
This thesis is submitted in partial fulfilment of the requirements for the Degree of Master of Science.

Acknowledgments

I would like to thank the company Excelian for supporting this thesis and also my supervisor there Andre Nedelcoux.

I would like to thank Dr. Stuart Barnes for his notes about GPU programming.

Finally, I particularly thank Professor Chris Thompson for his help and guidance. Our numerous meetings were very helpful for my work.

Abstract

Since the release of the Nvidia CUDA architecture for General Purpose Programing on Graphical Processor Unit (GPGPU), a lot of projects using the parallel capacities of those GPU have been undertaken. Through many studies, GPGPUs proved to be interesting for a large class of problems.

The aim of this project is to study the impact of GPGPU in the context of finance, starting with the example of Cholesky decomposition, useful for linear problem solving and in Monte Carlo correlated option pricer.

After a short review of GPGPU history, purpose, interest and architecture details, we will study the various parameters involved in Cholesky decomposition which affects performances when implemented on GPUs. To do so, we propose a brief review of the mathematical background around Cholesky decomposition followed by implementation of a high performance algorithm for dense matrices. Then, performance test results will be given and analysed. Using a blocked algorithm, we have achieve an interesting speed-up, reaching a factor 40 compared to an i7 CPU computation for a matrix size of 2500.

Finally Cholesky decomposition will be used in a real-life scenario of quantitative finance: a Monte-Carlo pricer for a basket of European options. The speed-up achieve is a factor of 2000 for 128 options computed using over a million of Monte Carlo paths.

This report will emphasises the importance of various factors taking place on GPU such as problem the size, parallelism level or memory access. Every parameter will be tested independently to show its importance in the computation, but more generally in GPGPU algorithms.

Keywords: GPU, GPGPU, CUDA, parallel, Cholesky decomposition, finance, Monte Carlo

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

The recent release of CUDA for General Purpose Computing on Graphical Processor Units (GPGPU), has brought developers to explore this new way to program, having use of the extraordinary capacities of GPUs to deal with parallel computation. By various aspect, GPGPU has shown its many benefits over classic programming in the past few years. The performance gain reached with this technique makes it used widely in high performance scientific simulation: from fluid-dynamics to finance.

In such a perspective, the company Excelian[1] is exploring how GPGPU affect quantitative finance calculation performances. Excelian is a worldwide company with headquarters in London, providing consulting services, project delivery and managed services with a strong focus in the investment banking world. For this reason, Excelian is associated with Cranfield University for this study review about GPGPU. As a vendor, their aim is to determine whether if it would be interesting to convert existing programs or algorithms into GPU versions. This not only includes reporting the performance gain, but also investigate how easy it is to use tools, the time it takes to discover and use CUDA C language, what parameters are involved and which problems can occur in the process.

As a response to those interrogations, this thesis -and therefore its report- will explore how and in which extend GPGPU with CUDA affect a cholesky decomposition problem. To do so, we will use the CUDA GPU technology with the CUDA language. The choice of CUDA is justified by the following arguments.

- CUDA is a well known and extended technology for GPGPU. Multiple examples and documentation can be found
- CUDA is compatible with C and therefore relatively easy to integrate.
- CUDA is very close to OpenCL and therefore easy to port if OpenCL becomes a well defined standard.
- CUDA can support usage of multiple GPUs for even greater performances.
- within the investment banking community, CUDA seems to be currently the preferred technology for GPGPUs[1] .

As a literature review, we will remind briefly the historical background on GPUs as well as the general architecture it uses. This will be the key element to have to understand parameter testing. Cholesky decomposition algorithm will also be reminded in the following.

Then, we will expose the particular work of this thesis: the implementation of the Cholesky decomposition on GPGPU. More importantly every parameter involved will be studied separately in order to give an idea of its influence on the general performance.

Finally, as I finished this as a couple of weeks before deadline, Excelian has asked me to explore Cholesky in a real use case: a Monte Carlo option pricer for a basket of

[1] Andre Nedelcoux, Excelian

European option pricing. Unfortunately, as time was short and shared with the writing of this report, I could not lead the analysis as deep as I would have liked to. However, this report presents my preliminary results and conclusion, as well as a summary of what is option, option pricing and its many variables. This knowledge about finance was actually quite difficult to clarify in a short time.

2 Technical background on GPU

2.1 On the way to GPU

2.1.1 Introduction

A computer is a programmable machine able to execute sets of instructions on data. The general architecture of a computer can be summed up in four distinct parts described by the Von Neuman model: [25]

- An arithmetic logic unit performing basic operations.
- A control unit in charge of sequencing operations.
- A memory which stores both the software to tell the control unit which operation to perform and the data on which to perform operations.
- Input-output devices to communicate with operator.

In computers, the CPU (Central Processing Unit) is the component in charge of every performed calculation. However, recent years have seen the appearance of more and more resource-intensive applications, especially in term of graphics. Therefore, the evolution of CPU has followed two main pathways:

- The need to speed-up calculation, done by increasing the internal frequency of CPUs, reaching nowadays a few GHz in personal computers.
- The need to perform many computations at the same time, resulting in the development of multi-cores processors and multi-threading technology.

2.1.2 What is a GPU?

In the late 1980s and early 1990s, the demand in video and image processing for screen display has boomed and hardware accelerators have emerged. This component has evolved into graphic cards whose main computational component is the GPU, a Graphicals Processor Unit. Military and scientific simulations, cinematic FX effects, video games, more specifically recent 3D games and Doom like games -first person shoot'em up in more realistic 3D environment- : all those applications have contributed to popularise 3D graphics in computers [37]. Those greedy applications have contributed to develop the GPU, and GPU programming.

2.1.3 Why use a GPU?

How to compute the appearance of a 3D realistic environment controlled in real-time by a player? How to compute and insert in a movie some outstanding FX ? The answer is obviously by being able to compute one by one the color, light, shadow, texture and finally the showing of each pixels.

Let us give an idea of the size of such a computation. Presently, a full HD video called 1080p resolution is defined with a native resolution of at least 1920x1080 pixels, ie more than 2 million pixels per frame with a 30-50 frames/second rate. A complex application such as a game will require a thousand of operations per second, including lighting, shadowing, texturing, mapping, volumetric effect computation, filtering, and others. This can lead us to up to 10^{11} operations per second performed by the GPU in an interactive game. Such a heavy computation is obtained by massive parallelism: modern GPUs (after 2009) have a massive parallel architecture called *Hardware Acceleration* and therefore can basically be seen as *large number of fine-grained parallel processors* [12].

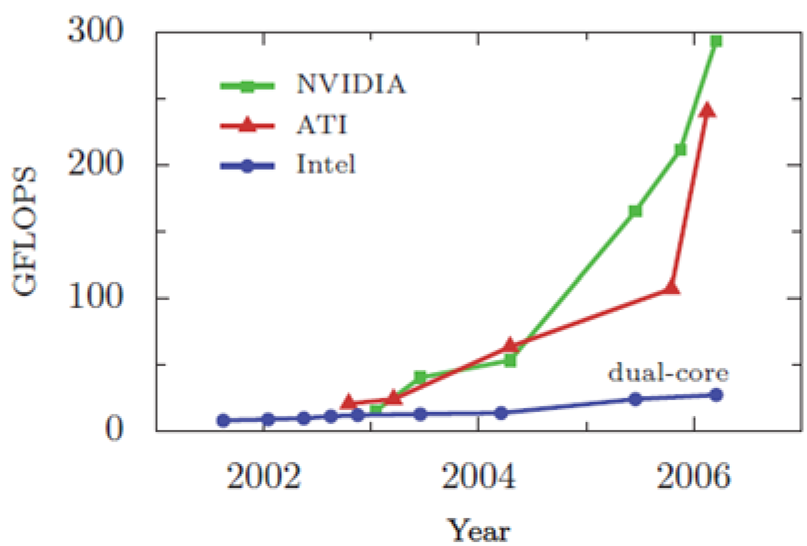

Figure 1: Evolution of performances on CPUs and GPUs over time [34]
The calculation speed of GPU explains their interest in scientific computing.

As any computation unit, GPU requires an execution environment (hardware) and a driver (software). Typically, any program using this GPU based computation will be separated into two parts:

- The core program, running on the host processor (CPU). Its role is to gather data from the host machine, to configure the GPU and to dispatch computation to GPU units.

- One or more calculation nodes or algorithms being executed on the GPU environment.They are called *device kernels*.

One of the issues is to deal with the communication between the CPU and GPU in order to configure the GPU, transfer data and execute computation. We can notice that the paradigm used here is SPMD (Single Program, Multiple Data). Indeed, the same operations (ie same code or same *kernel*) will be executed concurrently in multiple threads (potentially in units of ten thousands) on different pieces of data from the general data decomposition of the problem (in image processing, those pieces of data will be pixels).

2.2 How to program GPU for Graphics?

In this context, GPU drivers had to be created to give developers a programming environment to program graphic computation. First GPU drivers appeared in the form of Graphics API such as Direct3D from Microsoft[21] and OpenGL from the Kronos Consortium [23]. Obviously, those API were graphics oriented, using vertex on pipelines. A pipeline, in this context, is a step by step process to render the vertex. Both OpenGL and Direct3D use similar pipelines. This process can be explained as following [22] :

Basically the process is performed on a vertex buffer: a list of vertexes composing basic geometric figures such as triangles or lines composing the image. Those vertexes have various attributes such as coordinates, colour, textures, etc...

A first step called *vertex shader* is applied independently to each vertex. It consists of various transformations to compute coordinates and on screen appearance. The *rasterizer* stage then computes pixels covered by primitive figures to determine surfaces from vertexes.

The *pixel shader* stage computes the resulting colour of each of those pixels depending on light, shadows, textures... In other words, on a massive parallel modern card, each pixel will be treated in parallel in its own thread by programmable arithmetic operations called *rendering*.

Here is a simple trick: what if on a pixel, we replace colours in formation (texture, light, shadow,...) by any general data. In theory, you could then *fool* the GPU by rendering non graphical but any general computation. This was originally the idea of GPGPU.

GPGPU, General-Purpose computing on Graphics Processing Units, is a diversion of GPU -usually handling graphics computation- to compute more general calculations, usually for scientific purpose. The challenge then was to adapt general algorithms into graphical algorithms fitting vertexes and pipeline paradigm. This was done successfully for many of those; such has reduction, matrix calculus, sorting algorithms and others. Owens et al. presents an overview of those techniques [34]. OpenGL and Direct3D were helpful tools to do so but this had still some important limitations: As GPU were designed for graphics computation, they were not able to compute with floating-point data. This huge resource computation had to be tricked by software implementation of floating point variables with doubles or integers. Only shading languages could be used (DirectX or OpenGL) and algorithms had to be transposed into graphic-like using shaders. Development using those techniques was consequently difficult and the need was to have a more general framework to extend standard programming for GPU.

2.3 Modern GPGPU programming

The creation of CUDA by Nvidia -released the 15 February 2007- opened new opportunities for high performance computation. The idea behind it was to give developers a low level C-compatible framework to develop for GPUs. Moreover, both hardware and software improved this framework's capacities to offer general programming:

- memory addresses of the GPU
- mechanisms for data transfer between CPU and GPU and between GPU units
- Internal thread communication mechanisms
- Single precision floating-point arithmetic (double precision today)

GPU programmers would no longer have to disguise their algorithms as graphic computation, and the behaviour of floating-point arithmetic now follows the IEEE specifications.

Many scientific fields can therefore use the advantages of GPGPU. Such interesting projects include:

- Biological imaging: as one example, we could cite the TechniScan Svara systems which relies on an Nvidia Tesla card to process the 35Gb of a 15 minutes scan [38].
- Physics simulation: Many physical phenomena can take advantage of a massive parallel processing to solve its sets of equations. In the case of fluid dynamics for example.
- Environmental sciences: weather forecasting, global warming, pollution effect on environment are just examples among others.
- Financial modelling: Market forecast algorithm take advantage of GPGPU for simulation computation.

Following Nvidia's example, the competition also developed GPGPU frameworks. We can now consider the following main frameworks or libraries:

- DirectX, from Microsoft
- ATI Stream, from AMD
- OpenCL, from Khronos Group
- CUDA, from Nvidia

2.4 Brief overview of main GPGPU tools

2.4.1 DirectX

Microsoft extended its DirectX to GPGPU with DirectX11 in 2009 [21]. DirectX is a collection of tools, traditionally for 3D real-time programming for video games, and more recently also for general programming on GPU. DirectX is a widely used and developed library in video games but less so in the usage of GPGPU. The reason is that the library exclusively works on windows, while most scientific applications work on Linux servers. Also, it is not as "general purpose" as CUDA can be although its *Compute Shaders* are direct concurrent of ATI Stream and nVidia CUDA.

2.4.2 Stream

ATI Stream, formerly *Close to Metal* is the AMD framework for GPGPU on ATI graphic cards since 2007 [4]. ATI Stream is property of AMD and only works on ATI graphic cards. This library is not as widely used as CUDA or OpenCL can be in scientific computation field.

2.4.3 OpenCL

OpenCL is developed by the Khronos Group [13] composed of Nvidia, AMD, Intel and Apple. This framework has no hardware part and therefore is made to be compatible with any modern graphic cards and any platform. As such, it is the more generic way to do GPGPU, plus, it has been an open standard since 2008. In addition to the compatibility of any GPUs, OpenCL is also compatible with CPUs. In theory, OpenCL is therefore able to select any combination of eligible component and to offer a unified view of it to perform calculations using every processing resource. The advantage is obviously to delegate the calculation of each part of a program to the most suitable component: CPUs for sequential task, GPUs for massive parallel computation. As a standard, writing an OpenCL program today is therefore the assurance of it running faster and faster in the future as GPUs and CPUs performances rise, without having to rewrite any code. This has a back side: OpenCL is not as optimized for a special platform as other library can be, plus, it is not yet as mature as CUDA.

2.4.4 CUDA

In 2007, Nvidia introduced CUDA [32]. Although this API does only work on Nvidia GPUs, it seems to be the most used technology in scientific GPGPU-based high performance computation. A great point of CUDA is to be compatible with standard C development. Also CUDA dispose of a much extended SDK and quality and diverse documentation and examples can be found. Moreover, as CUDA was the earliest stable and functional released tool for GPGPU, it is now one of the most extensively used.

2.5 Expected gain of GPGPU over CPU programming

2.5.1 Performance gain

Studying performance gain on the particular problem of Cholesky decomposition is one of the aims of this study. This is motivated by the existence of many examples of diverse subjects showing that high expectations can be made on the gain performance of GPGPU over CPU. As an example, the Figure 2 [20] shows an order computation capabilities of various chipsets.

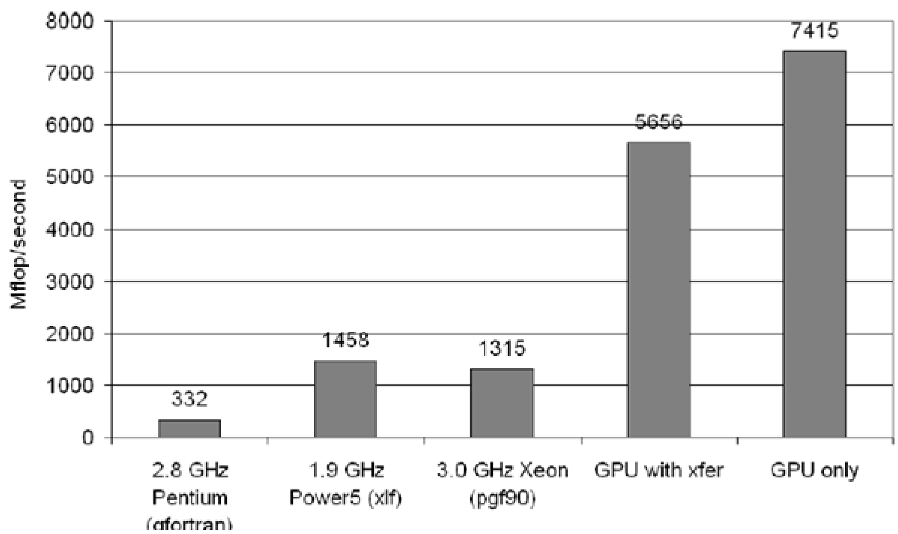

Figure 2: Performance comparison of Pentium, Power5, Xeon and GPU
A priori, the higher the flop rate is, the faster the computation will be. In this way, the figure omens a nice speed-up using GPU when no data dependencies occurs.

However, due to data dependencies in the Cholesky decomposition algorithm, I would not expect a speed-up ratio above five to ten for matrix size $N \geq 2000$ [41] [42] [44].

2.5.2 Watt per GFLOPS

The next table, Figure 3, aims to show the benefit of using GPU over CPU in terms of computation power versus the energy consumption. It shows that the electrical cost of running a multi-GPU bay is far less than a standard CPU-based cluster.

	GLOPS	TDP	Watt/GLOPS
CPU			
Intel Pentium 4 - 2.8 Ghz - 1 cœur - x86	2,7	70	25,93
Intel Centrino Duo - 1.66 Ghz - 2 cœurs - x86	3,1	30	9,68
Intel Xeon 5410 - 2.33 Ghz - 4 cœurs - x64	9,5	100	10,53
Intel Core I7 980x - 3.33 Ghz - 6 coeurs - x64	20	130	6,50
AMD Athlon 64 x2 - 3 Ghz - 2 cœurs - x64	5,9	65	11,02
AMD Phenom II x6 - 3.0 Ghz - 6 cœurs - x64	17,8	100	5,62
GPU			
NVidia 9800 GT - 112 cœurs - 600 Mhz	370	230	0,62
NVidia GT240 - 96 coeurs - 550 Mhz	260	180	0,69
NVidia GTX 285 - 240 cœurs - 648 Mhz	700	370	0,53
NVidia GTX 480 - 480 cœurs - 700 Mhz	1350	410	0,30
ATI HD 3870 - 320 cœurs - 775 Mhz	490	240	0,49
ATI HD 4890 - 800 cœurs - 850 Mhz	1360	290	0,21
ATI HD 5970 - 3200 cœurs - 725 Mhz	4640	430	0,09

Figure 3: Comparison of Watt/GFLOPS of CPU/GPU (data from [8])
The table suggest that comparatively to their computational capacities, GPU are more energy efficient than CPUs.

2.5.3 General performance comparison

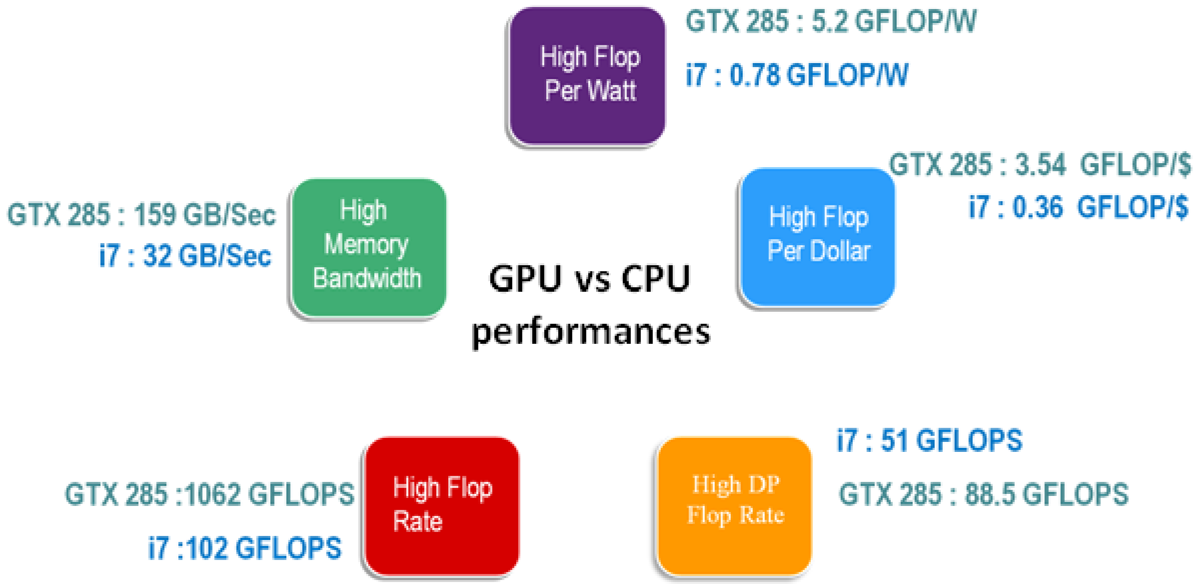

Figure 4: General performance comparison on GPU vs CPU (datas from [33])

The Figure 4 sums up how GPUs are overtaking CPU capacities in many different ways: speed, energy efficiency, price compared to speed, transfer bandwidth,...

2.6 CUDA technical overview

2.6.1 General architecture

In this section, we will attempt to overview the CUDA Architecture, so as to give the reader the basic vocabulary for the understanding of the following thesis work.

The CUDA framework defines two execution platforms: *host* and *device*. Obviously host code will refer to a portion of code executed on the CPU and device code to a portion of code executed on a device: any CUDA capable GPU. CUDA programming uses a SIMD (Single Instruction Multiple Data) paradigm in a parallel and efficient way. In other words, a same portion on code -in a C-style function- is defined to run on the GPU. This function is therefore known as a kernel. A *kernel* is a function simultaneously executed by a large number of *threads*, each one processing different data in parallel.

A *thread* executes a *kernel* using some data. It is referred using its coordinates (or indexes) on the *block* it belongs to.

A *block* is simply a group of threads. It can gather up to 512 threads represented in a 3D-array. Threads from a same block shares a *shared memory* pool (see 2.6.2) and can be synchronized together. Using synchronization and shared memory, threads for a same block can cooperate. A block is accessible using its coordinates (or indexes) on a *grid*.

A *grid* is a group of blocks of threads. It can contain up to 65535 blocks per dimension in a 3D-array. Threads from different blocks of the grid cannot be synchronized together, and therefore are independent and cannot cooperate. A grid is associated to a kernel in the way that every threads from a grid run the same kernel.

Figure 5[2] sums this up. Notice that the size of each block/grid groups allow a kernel to be executed massively in parallel at the scale of 10^{15} threads simultaneously.

[2]Diagram from Nvidia CUDA Programming Guide

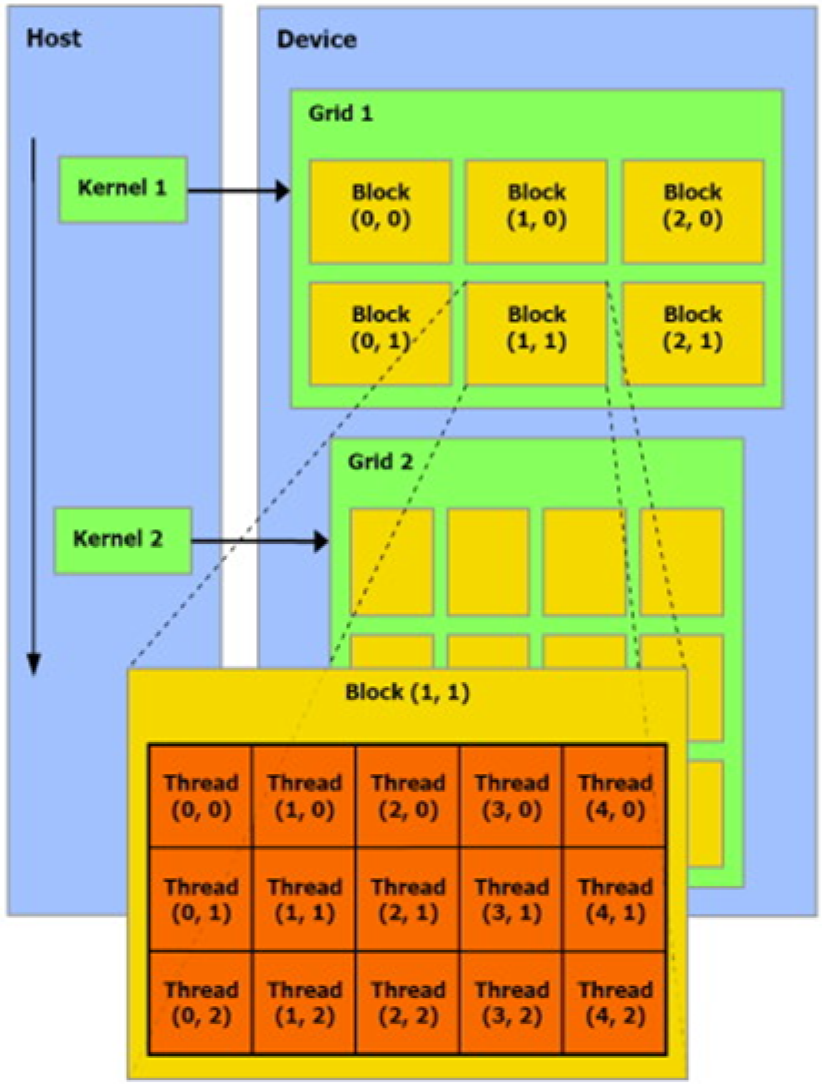

Figure 5: General CUDA architecture view
For easier comprehension blocks and grid have been represented in 2D but threads can be organised up to 3D.

Threads within a block are processed by a collection of multiprocessors. Each multiprocessor is made of eight single stream processors running instructions from a thread within a single block. Each instruction takes four clock cycles, so a single stream processor can process up to four instructions at the same time in a pipeline. This make $8 \times 4 = 32$ identical instructions that can be processed at a same time by a single multiprocessor. So if 32 threads of a single block are at the same step (same instruction to execute), they can perform their instruction on the same multiprocessor. This group of 32 homogeneous threads is called a *warp*. If the threads of a same block are not homogeneous (or does not execute the same instruction at the same time) they will have to be split into smaller bunch of homogeneous threads running on different multiprocessors. Maintaining homogeneous threads is a great factor of optimization. Moreover, if the instruction to be performed is an access to 32 contiguous data items in memory, then this access can be done in one data transfer instead of 32 different ones. This is referred as *coalescent memory access*. As global memory of the GPU is quite slow (see 2.6.2), the speed-up of coalescent memory access over non-coalescent one is non negligible.

2.6.2 Memories

Various memories (such as *texture memory*, *constant cache* or *registers*) are used with CUDA. We will explicitely use in this thesis the following ones.

Global memory is the main graphic card memory, accessible by every thread of every grid. Made from SDRAM chips connected with the GPU this memory is large (up to a few giga-Bytes) but has slow access (400-600 clock cycles) [6].

Shared memory is shared by every thread from the same block. This memory is small (a few kilo-bytes) but fast access (5 clock cycles) [6].

Local memory is the memory specific to a thread for its local variables.

Figure 6[3] sums this up.

[3] Diagram from Nvidia CUDA Programming Guide

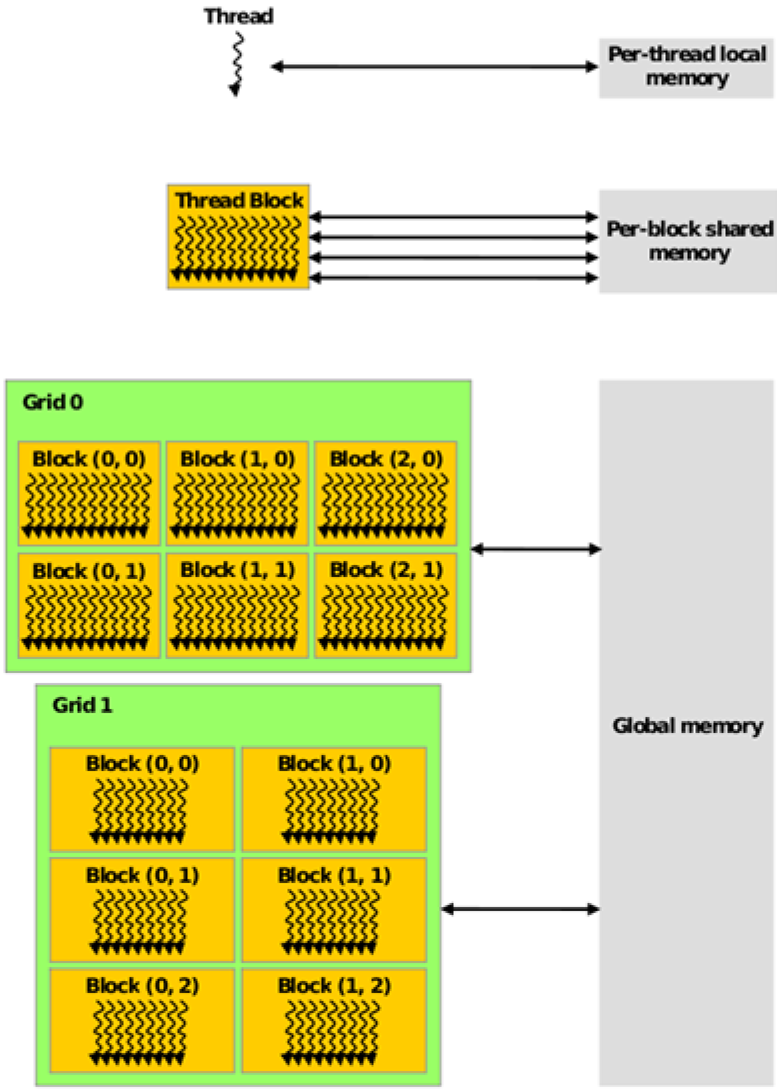

Figure 6: General CUDA memory view

3 Test procedure

The aim of this section is to describe the test procedure that will be employed in the following. Next section will indeed describe various algorithms and performance tests using the following general schema:

- Description of the algorithm
- Preliminary results
- Various possible optimization and there effect
- Eventually some more analysis if needed

To benchmark the algorithms and their possible optimization, it takes to describe here a strong testing procedure. First of all, let's notice that the test will be performed on square symmetric definite positive matrix according to the requirements of the Cholesky decomposition algorithm, the first algorithm we will implement (see §4. MatrixMarket is a *visual repository of test data for use in comparative studies of algorithms for numerical linear algebra*[2]. Those matrices are commonly used in literature papers for tests (in [14], [17], [35] among others). Unfortunately, MatrixMarket does not present enough dense matrix for testing the influence of Matrix size on performance. Therefore, we will generate our how dense matrices, following an idea from [35]. The Algorithm 1 shows this generation. It is made by summing of a great number p ($p \gg N$) of outer-products of randomly generated vectors with themselves. A quick analysis on it let us recognise the generation and summing of symmetric definite-positive tensors giving a matrix of inertia. Thus, this matrix have the particularity of being an inertia matrix and a tensor. Plus its coefficients are such as for $i, j \in [\![1, N]\!]$, $0 \leq a_{ij} \leq N$

Algorithm 1 Generator of a particular dense matrix

Input: N the size of desired matrix.
Output: A random coefficient dense symmetric definite-positive matrix

for $a = 1$ to $5N$ **do**
  for $b = 1$ to N **do**
    $tmp[b]$ = random float between $[0, 1]$
  end for
  for $i = 1$ to N **do**
    for $j = 1$ to N **do**
      $a_{ij} += tmp[i] * tmp[j]$
    end for
  end for
end for

Performance test will therefore be done with matrix size N={10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000}. The choice of these sizes is arbitrary but corresponds to the *idea of size* of matrices present in MatrixMarket. For a question of time of availability of the test-platform, bigger matrix up to $N = 10000$ have been generated and tested only when necessary. The great number of different matrix size help to observe how the algorithms behave for tiny ($10 \leq N \leq 100$), small ($100 \leq N \leq 1000$) and big ($1000 \leq N \leq 4000$) matrices.

The program to test an algorithm perform operations in the following order:

- read the matrix from file stored in MM_Format (using the mmio library from MatrixMarket) [18].
- Do k times:
  1. Allocate memory on GPU (cudaMalloc)
  2. Copy the matrix to GPU (HostToDevice)
  3. Perform the computation (Computation)
  4. Copy the result back to CPU (DeviceToHost)
  5. Check the accuracy of computation
- Return the average time and detailled time of the k calculations.

In the last chapter on Monte-Carlo simulation, the same idea will be applied. Options data will be generated randomly in a range of value.

From now, the speed-up will refer to the gain of performance on GPU against CPU computation. This ratio is given by t_{CPU}/t_{GPU}.

The accuracy of the computation will now be referred as E_∞. Its calculation is not included in the time measurement. E_∞ is the infinite matrix norm of the difference between the reference matrix and LL^T where L is the computed Cholesky factor (see §4 for details on Cholesky):

$$E_\infty = \|A - LL^T\|_\infty$$

The test itself is made automatically using simple C command-line program which aim is to read the INPUT directory of matrix files and to run the testing program for each of these files. It redirects the output of the testing-program to a file using the command: `system("./prog filename.mtx -c 100 > filename.out")`. The parameter `-c` forces the accuracy test to be performed (not included in time measurement) and `100` is the number k of repetition of the calculation.

Last detail about the test itself: k have arbitrary be chosen to 100. This number seems big enough to flatten any artefact that could happen but small enough not to monopolize the testing-platform. In the later result, we will refer to the name `init` as being the result of the first computation, and to `average` as the average results of the next k computation, `init` not being include. We will indeed show that the `init` scenario behave slightly differently then the next ones.

The testing platform is composed of two possible devices: a `GeForce 9800 GT` and a `Tesla C2050`. We will use the second device: the Telsa C2050 for our performance test. The development environment is accessible only on computers with a GeForce 9800 GT and is composed of Visual Studio 2008 and CUDA SDK. No in-site GPU debugger nor other helper tools can be installed due to student IT restriction, this would slow down my development.

The Figure 7 details the capacities of the device for performance test. Details are obtained through the code `enum_gpu.cu` from Chapter 3 page 33 of [37].

— General Information for device 0 —	
Name:	Tesla C2050
Compute capability:	2.0
Clock rate:	1147000
Device copy overlap:	Enabled
Kernel execution timeout :	Disabled
— Memory Information for device 0 —	
Total global mem:	2817982464
Total constant Mem:	65536
Max mem pitch:	2147483647
Texture Alignment:	512
— MP Information for device 0 —	
Multiprocessor count:	14
Shared mem per mp:	49152
Registers per mp:	32768
Threads in warp:	32
Max threads per block:	1024
Max thread dimensions:	(1024, 1024, 64)
Max grid dimensions:	(65535, 65535, 1)

Figure 7: Details of Tesla C2050 capacities

When talking about CPU performances, we will refer to a sequential mono-thread program running on an Intel CPU 6 Cores i7-980X clocked at 3.33Ghz [16].

4 Cholesky decomposition for dense matrix

4.1 Definition

Let be a system of linear equations to be solved. In its matrix form, such a system can be written as:

$$Ax = b, \tag{1}$$

where A is an $N \times N$ matrix and b a vector of N known coefficients.

Such a system can be solved by two types of methods. The *direct* methods give an exact solution of the system while *iterative* ones give an approximated solutions [40].

A common trick for direct methods is to decompose the matrix A into a product of two matrices. The system is then solved by a double substitution. In the particular case where A is a symmetric definite positive matrix, a decomposition called Cholesky Factorization can be performed more efficiently then any other decomposition. The overall complexity may be similar to other decompositions but it will require fewer operations (addition, division, multiplication,..) [36].

Cholesky factorisation is defined as such:

Theorem 1 (*Cholesky factorization [39]*)

Let $A \in \mathbb{R}(N \times N)$ be an symmetric definite positive matrix, then it has a Cholesky factorization

$$A = LL^T$$

where L is a lower triangular matrix.

Solving the system (1) is therefore made easy using a forward and backward substitution. Let be the linear system $Ax = b$:

1. a forward-substitution $Ly = b$ can be setted and solved to find the vector y.
2. a backward-substitution $L^T x = y$ easily help to find x and finally solve the system.

4.2 Understanding the coefficients

To understand the basic operations taking place in the decomposition, let's retro-engineer our problem. Let be L a lower triangular matrix with positive diagonal entries, and L^T be its transposed matrix. Therefore, the Cholesky decomposition can be written:

$$LL^T = \begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{pmatrix} \begin{pmatrix} l_{11} & l_{21} & \cdots & l_{n1} \\ 0 & l_{22} & \cdots & l_{n2} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & l_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{n1} \\ a_{12} & a_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ a_{n1} & \cdots & \cdots & a_{nn} \end{pmatrix}$$

Therefore, a coefficient a_{ij} of matrix A is given by the relation:

$$\left| \begin{array}{lcl} a_{ii} = l_{i1}l_{i1} + \cdots + l_{ii}l_{ii} & \Longrightarrow & l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{lik}^2} \\ a_{ij} = l_{i1}l_{j1} + \cdots + l_{ii}l_{ji} & \Longrightarrow & l_{ji} = \dfrac{a_{ij} - \sum_{k=1}^{i-1} l_{ik}l_{jk}}{l_{ii}} \end{array} \right.$$

From this, we can deduce that the calculation of the Cholesky decomposition will include 3 main parts (later implemented as kernels):

1. Square root on every diagonal coefficient
2. Subracting by the sum of products (we will call it sub-matrix update)
3. The division of result by diagonal element of the same column (we will call it column normalization)

4.3 Algorithm

From what we have seen in the previous paragraph, we can deduce that the Cholesky decomposition will be obtained through a triple nested loop. Here is a basic algorithm to solve it.

Algorithm 2 Numerical Cholesky factorisation

Input: The matrix A is symmetric positive definite.
Output: A lower triangular matrix L such as $A = LL^T$

for $i = 1$ to n **do**
    $a_{ii} := \sqrt{a_{ii}}$
    for $j = i + 1$ to n **do**
        $a_{ij} = \frac{a_{ij}}{a_{ii}}$
    end for
    for $j = i + 1$ to n **do**
        for $k = i + 1$ to j **do**
            $a_{kj} = a_{kj} - a_{ij}a_{ik}$
        end for
    end for
end for

Cholesky decomposition		
Operation:	# of frags:	Complexity:
Square root	n	n
Division	$\frac{n(n+1)}{2}$	$O(n^2)$
Submatrix update	$\frac{1}{2}(\frac{n(n-1)(2n+1)}{6})$	$O(\frac{n^3}{6})$

Table 1: Arithmetic operation counts

The complexity of such an algorithm is therefore in $O(\frac{n^3}{6})$ but requiring fewer operations than any other decomposition [9].

In the following, we will store matrices as a vector. This will be done column by column (see Figure 8). Therefore, what we will refer as $A[i,j]$ or a_{ij} is the element on the i^{th} lign and j^{th} column and can be found in memory in the vector at position $i + j \times n$. The reason for this choice can be explained by the literature review on Cholesky. In the case where specific optimization is made on sparse matrices using trees, a column-based representation for the matrix is more efficient [14] and [5].

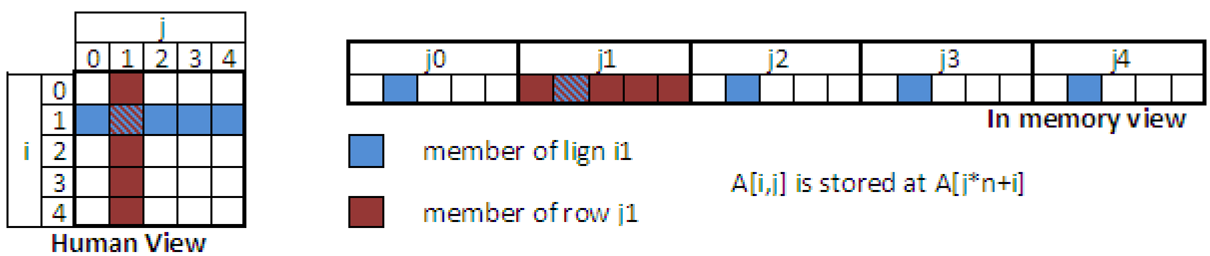

Figure 8: In memory representation of a Matrix
Elements from a single column are contiguous in memory. Therefore, raw based algorithms will be slow down by paging issues.

Depending on how the loop iterations are ordered, we can define six different variations of this algorithm [10][4]. These are shown in Figure 9. Each of those algorithms have the same complexity but the data access is differently order and this strongly changes the performance on a CPU (see Figure 10).

As the matrix is represented column by column, a line by line access is expensive [11] due to paging issues. For this reason, we won't study algorithms A, D and E of figure 9.

[4] page 11 of this paper shows the six algorithms.
Algorithm 2 is wrong as the external loop should not be external but should be closed before the square root operation. This mistake is however corrected as the algorithm is transformed into a Fan-In on Figure 7 page 13.

Algorithm C computes an element from external loop i by using a triangular line by line access on upper elements. This leads to dependencies on elements from different column that will not be transferable as a whole blocks (ie dependant elements are not contiguous in the vector representation).

This lets algorithms B and F which are column version. Those algorithms will now be respectively referred to as *Fan-In* and *Fan-Out* algorithms. This means that for an iteration on the external loop, *Fan-In* will modify elements on the same column while *Fan-Out* will modify elements from different columns.

A

```
FOR j=1 TO n
   FOR i=j TO n
       FOR k=1 TO j-1
          A[i,j]=A[i,j]-A[i,k]*A[j,k]
       ENDFOR
       IF (i=j) THEN A[i,i]=SQRT(A[i,i])
              ELSE A[i,j]=A[i,j]/A[j,j]
       ENDIF
   ENDFOR
ENDFOR
```

B

```
FOR j=1 TO n
   FOR k=1 TO j-1
       FOR i=j TO n
          A[i,j]=A[i,j]-A[i,k]*A[j,k]
       ENDFOR
   ENDFOR
   A[j,j]=SQRT(A[j,j])
   FOR i=j+1 TO n
      A[i,j]=A[i,j]/A[j,j]
   ENDFOR
ENDFOR
```

C

```
FOR i=1 TO n
   FOR j=1 TO i
       FOR k=1 TO j-1
          A[i,j]=A[i,j]-A[i,k]*A[j,k]
       ENDFOR
       IF (i=j) THEN A[i,i]=SQRT(A[i,i])
              ELSE A[i,j]=A[i,j]/A[j,j]
       ENDIF
   ENDFOR
ENDFOR
```

D

```
FOR i=1 TO n
   FOR k=1 TO i
      IF (k=i) THEN A[i,i]=SQRT(A[i,i])
              ELSE A[i,k]=A[i,k]/A[k,k]
      ENDIF
      FOR j=k+1 TO i
         A[i,j]=A[i,j]-A[i,k]*A[j,k]
      ENDFOR
   ENDFOR
ENDFOR
```

E

```
FOR k=1 TO n
   A[k,k] = SQRT(A[k,k])
   FOR i=k+1 TO n
       A[i,k]=A[i,k]/A[k,k]
       FOR j=k+1 TO i
          A[i,j]=A[i,j]-A[i,k]*A[j,k]
       ENDFOR
   ENDFOR
ENDFOR
```

F

```
FOR k=1 TO n
   A[k,k] = SQRT(A[k,k])
   FOR i=k+1 TO n
      A[i,k]=A[i,k]/A[k,k]
   ENDFOR
   FOR j=k+1 TO n
      FOR i=j TO n
         A[i,j]=A[i,j]-A[i,k]*A[j,k]
      ENDFOR
   ENDFOR
ENDFOR
```

Figure 9: Different versions on how to access data
Algorithm B and F are expected to be faster as they are full column based.

4.4 Performance test on CPU

As expected from previous analysis, we see from Figure 10 that algorithms B and F are the most efficient ones. This comes from the ordering in the access of elements in the matrix. As this one is stored column by column in memory (column-major), an access on row by row produces paging issues [19] and slow down the computation.

As Excelian -the company supporting this thesis- works in the finance field, let's notice that QuantLib -an open source library used in Quantitative Finance [3]- performs the operation through algorithm A but using row-major matrices. JQuanLib, the port of QuantLib to Java [1] uses the algorithm B on column-major matrices.

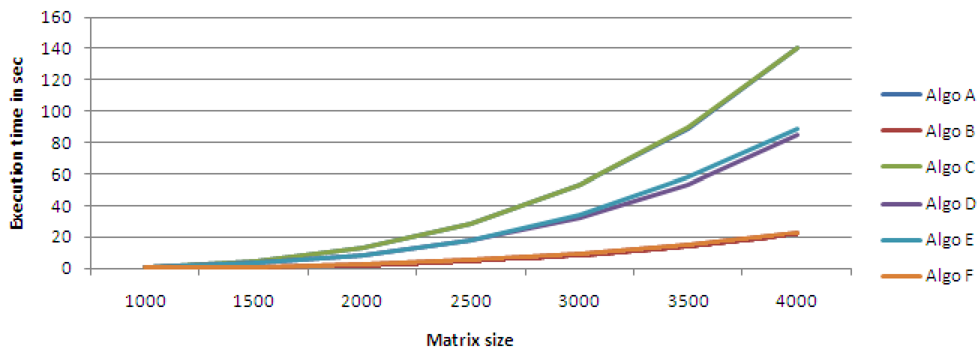

Figure 10: Execution time on CPU (the lower the better)
The figure experimentally verifies previously defined ordering issues: Algorithm B and F are the fastest are expected.

In conclusion to Figure 10, algorithm B will now be employed when talking about CPU algorithm performance. Indeed, to analyse speed-up of our implementations, it would make no sense to compare with an inefficient algorithm.

5 Cholesky on GPU for dense matrix: pointed algorithm

5.1 Algorithm

The first idea for implementing a parallel Cholesky decomposition for GPU is to use a naive algorithm. This algorithm is called naive as it consists of a translation of Algorithm B into a CUDA version, exploiting GPU parallelism as far as possible. Therefore, it is not naive in term of *stupid* but naive as *the first idea that comes in mind*. As a matter of fact, this algorithm is proposed in [17] (Algorithm 2) under the name *Outer Product Form*. For some reasons, it uses two streams of data S and T successively bounded to the active kernel. We propose here to use a single stream to avoid copying or binding data from one stream to another during the computation.

The following Algorithm 3 presents the general computation while Algorithm 4, 5 and 6 are the different CUDA kernels used for the calculation.

Algorithm 3 Cholesky decomposition for CUDA: Outer Product form

Input: N the size the matrix.
Input: a pointer to a 2D $N \times N$ array: the matrix to decompose
Output: A lower triangular matrix L such as $A = L^T$

1: allocate an $N \times N$ array on GPU
2: copy matrix A to GPU global memory
3: **for** $(k = 0$ to $N - 1)$ **do**
4: Perform SquareRootKernel on element k,k [using 1 thread]
5: Peform NormalizeColumnKernel on column k [using $N - k$ thread]
6: Perform UpdateSubMatrixKernel on sub-matrix under pivot k,k [using $(N - k)^2$ threads]
7: **end for**
8: Perform SquareRootKernel on element N,N

Algorithm 4 SquareRootKernel : compute $\sqrt{a_{k,k}}$

Input: k the index of element to compute.
Input: a pointer to a 2D $N \times N$ array: the matrix to decompose
Output: the square root of diagonal element k,k

1: $a_{k,k} = \sqrt{a_{k,k}}$

Algorithm 5 NormalizeColumnKernel : normalize column k $col(k) = col(k)/a_{k,k}$

Input: k the index of column to normalize.
Input: N the size of the matrix
Input: a pointer to a 2D $N \times N$ array: the matrix to decompose
Output: the normalization of column k

threadId = threadIdx.x + blockIdx.x $*$ blockDim.x
while (k<threadId<N) **do**
  $a_{threadId,k} = a_{threadId,k}/a_{k,k}$
  threadId += blockDim.x * gridDim.x;
end while

Algorithm 6 UpdateSubMatrixKernel : update sub-matrix under pivot k,k

Input: k the index of pivot.
Input: N the size of the matrix
Input: a pointer to a 2D $N \times N$ array: the matrix to decompose
Output: the update of the sub-matrix

$i = threadIdx.x + blockIdx.x * blockDim.x$
$i = threadIdx.y + blockIdx.y * blockDim.y$
while ((k<i<N) and (k<j<N)) **do**
  $a_{i,j} = a_{i,j} - a_{i,k} * a_{j,k}$
  i += blockDim.x * gridDim.x
  j += blockDim.x * gridDim.x
end while

Here are some remarks on the algorithm:

- The whole matrix $N \times N$ is transmitted to the matrix but only the lower triangular part is used as the matrix is symmetrical
- SquareRootKernel can be done by one thread only.
- NormalizeColumnKernel is done on a single column. Therefore, $N - k$ threads runs in parallel in 1D blocks. The number of blocks used is the one required to have the correct number of threads according to blockDim. CUDA architecture define blockDim to be a maximum of 512 threads.
- UpdateSubMatrixKernel is done on a square sub-matrix. Therefore, $(N - k)^2$ threads are used in 2D blocks. The number of threads per block is defined as being $blockDim.x \times blockDim.y$. Therefore, because of the limit on CUDA architecture, $blockDim.x \times blockDim.y$ is a maximum of 22×22.
- In NormalizeColumnKernel and UpdateSubMatrixKernel, the while loop is there to provide a general case for virtually any matrix size. For real, it is usefull only is every thread/block and every block of CUDA architecture is used. This happens in NormalizeColumnKernel if $N > 33553920$ and in UpdateSubMatrixKernel if $N > 5792$.

5.2 Performance test and analysis for case 512-22

The case called 512-22 is the following: 512 threads/block are used for NormalizeColumnKernel and 22×22 threads are used in UpdateSubMatrixKernel. This correspond to the maximum of resources authorized by CUDA architecture.

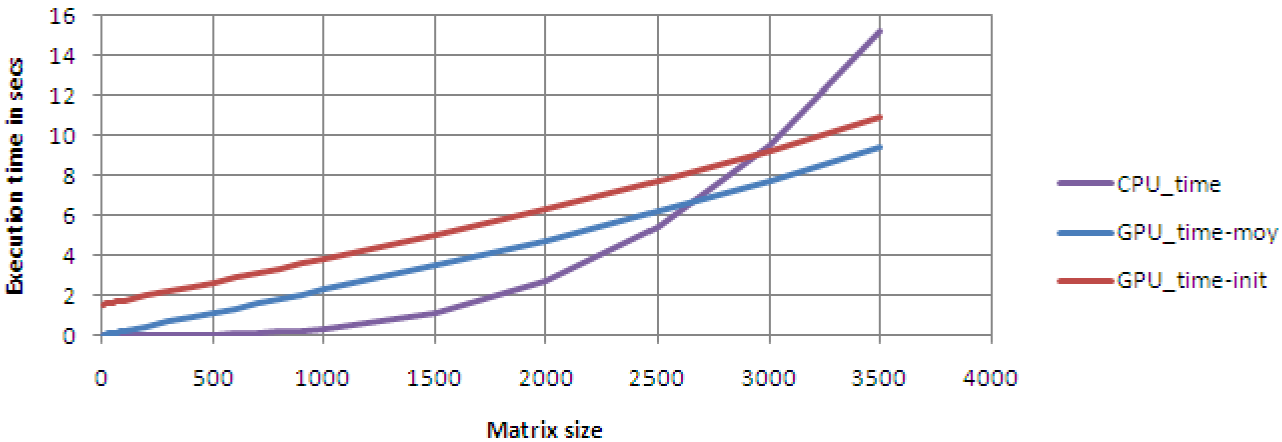

Figure 11: Performance CPU vs GPU for case 512-22
Notice how the GPU calculation is made $O(N)$ compared to CPU $O(N^3)$. The difference between init *and* average *scenarios is explained later on.*

This first test aims to study the influence of the matrix size on the execution time. As expected, Figure 11 reminds us that the GPU computation time is proportional to the number of elements in the matrix. This comes from the data dependencies. The square root kernel has to compute the diagonal elements one by one sequentially. The column normalization and sub-matrix update kernels can deal with their coefficients all at the same time. The general complexity is therefore O(N), ie proportional to the number of elements.

We can also notice that `init` and `average` scenarios have the same parallel graphs, but the init case is higher. We will show later why (see Paragraph 5.3).

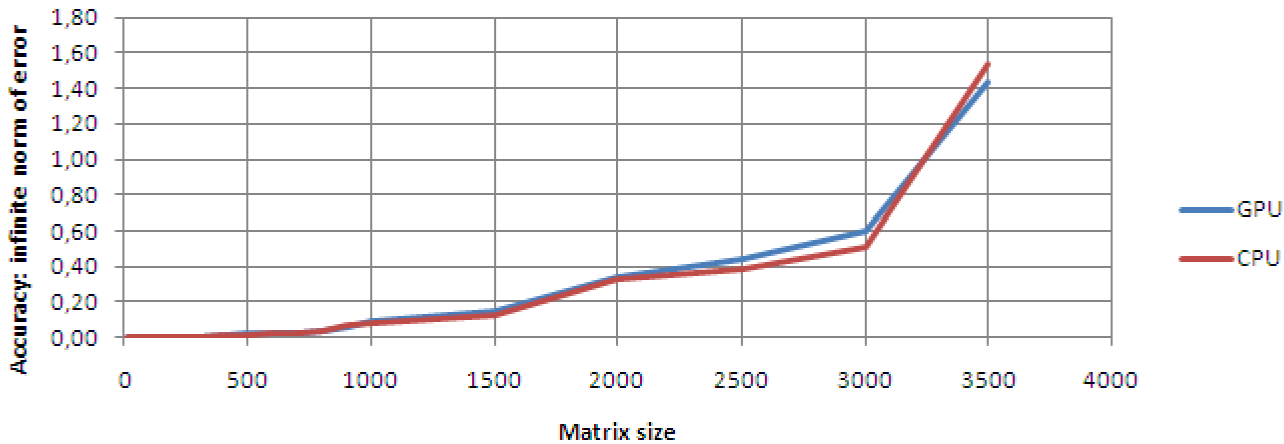

Figure 12: Accuracy CPU vs GPU for case 512-22
Accuracy of GPU and CPU calculation should ideally be equal as the same operations are performed in the same order on each elements.

As both CPU and GPU version performs strictly the same operations on every coefficient, it is interesting to notice that -although an artefact for N=2500- the accuracy of both is identical. This accuracy is determined by the precision of the device. In the case of our test environment, the Telsa C2050 card can perform double precision operations like a CPU does. In the development environment however, the GeForce 9800 is single precision operation. The accuracy of the result would have been much lower using that second card.

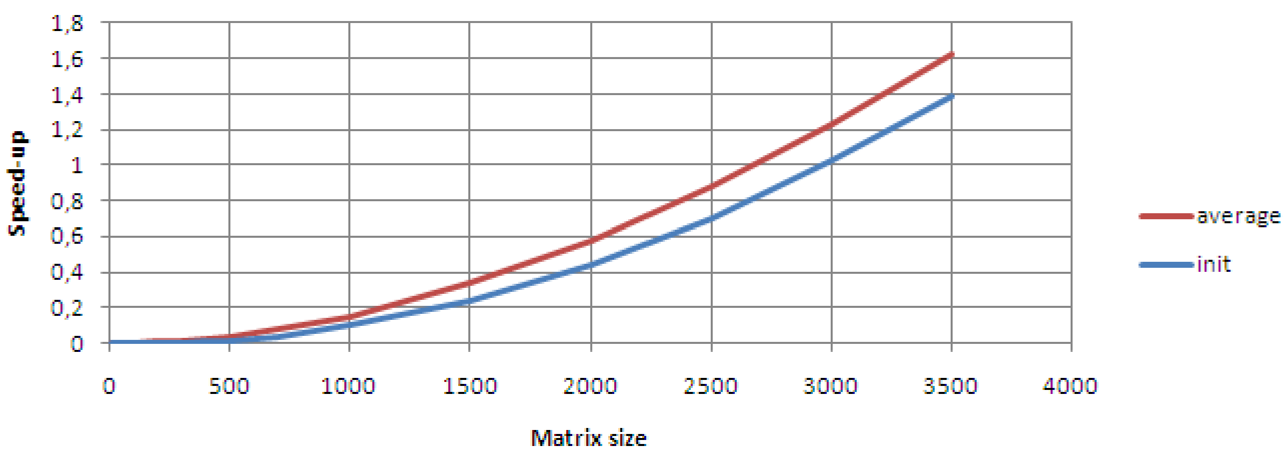

Figure 13: Speed-up CPU vs GPU for case 512-22
Programmed and setted without taking care of GPU specification, this case usage exposes a weak speed-up.

The decomposition on a CPU is fast for tiny and small matrices so the speed-up is not. For large matrices the situation is reversed. The speed-up ratio however is not high. A speed-up ratio above 1 means that GPU computation is faster than CPU. This only happens for matrices above $N = 2600$. Is something slowing down the computation or is this the best we can have? To have a better understanding of the bottleneck in the computation, lets analyse what GPU does.

5.3 GPU utilization

5.3.1 Allocate memory

We have seen earlier that the execution time of the `init` scenario is different from the one in the `average` scenario. Now is the time to explore further to understand what exactly happens. Figures 14 and 15 shows a deeper investigation of `cudaMalloc` operation. This aims at allocating memory on GPU and is an equivalent of `malloc` in C.

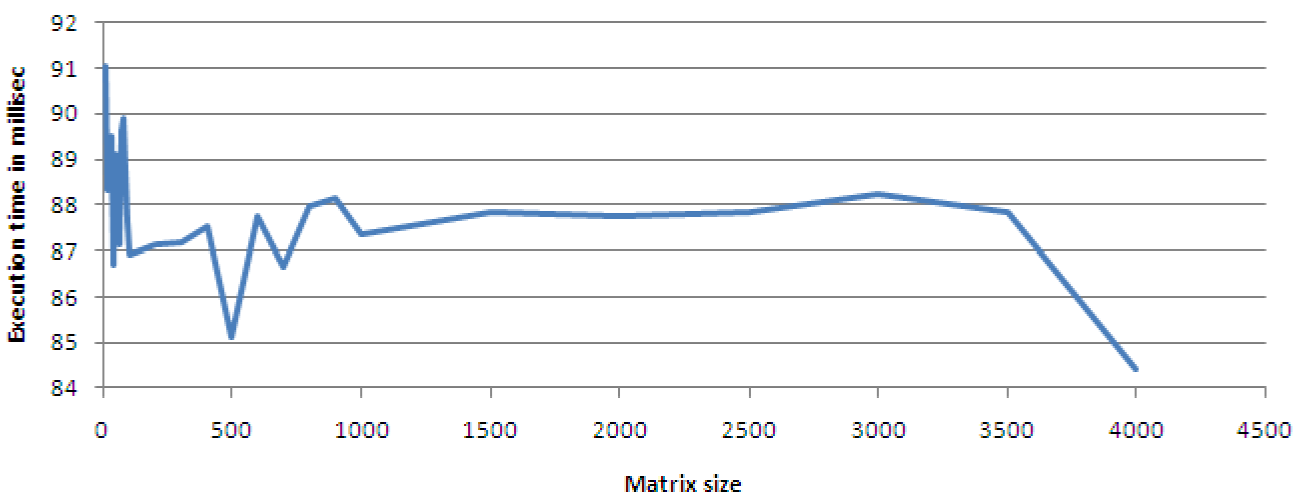

Figure 14: cudaMalloc execution time for init scenario
The context initialization taking place in the init *scenario makes it long and incoherent. Unfortunately, it is a CUDA required that cannot be avoided.*

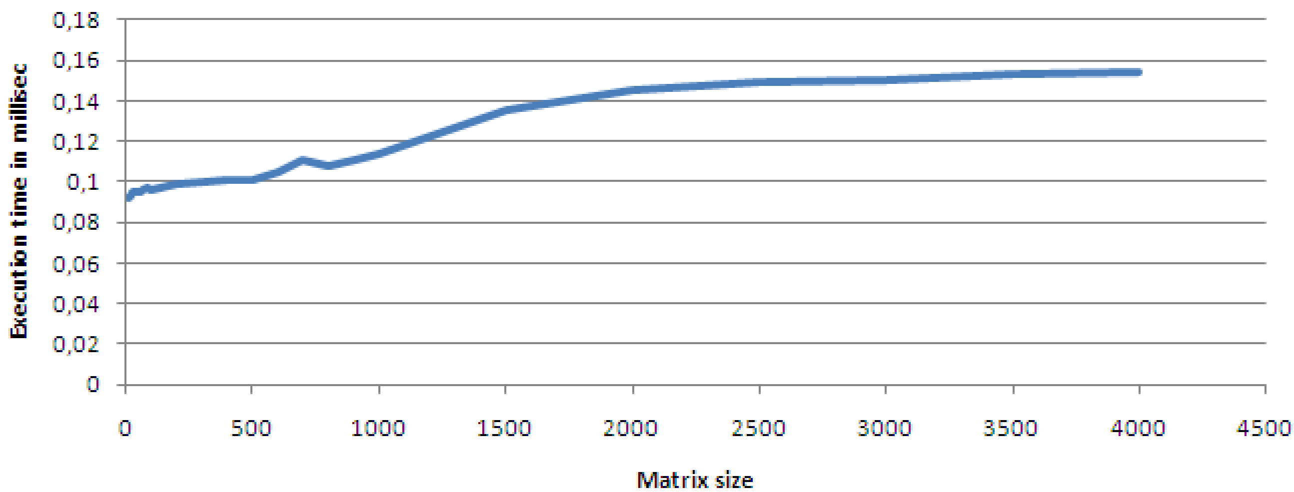

Figure 15: cudaMalloc execution time for average scenario
A none perturbed memory allocation is much faster (notice the scale difference with previous Figure) and roughly proportional to the number of elements to allocate.

The average time for the `cudaMalloc` operation seems to follow a well determined trend. During this small amount of time, the GPU finds room in its memory for the desired quantity of data to transfer. This is done similarly to the `malloc` C operation on CPU [37]. However, in the initial case, a much longer operation is performed. This extra time does NOT belong to `cudaMalloc` although in our case it is called by it. It actually comes from a *CUDA context initialization* that is performed on the device by the first CUDA function called, no matter which. In our programs the cudaMalloc operation is called first so this is the one creating the CUDA context [26].

> "A CUDA context is analogous to a CPU process. All resources and actions performed within the driver API are encapsulated inside a CUDA context, and the system automatically cleans up these resources when the context is destroyed."[27]

5.3.2 Transfer data

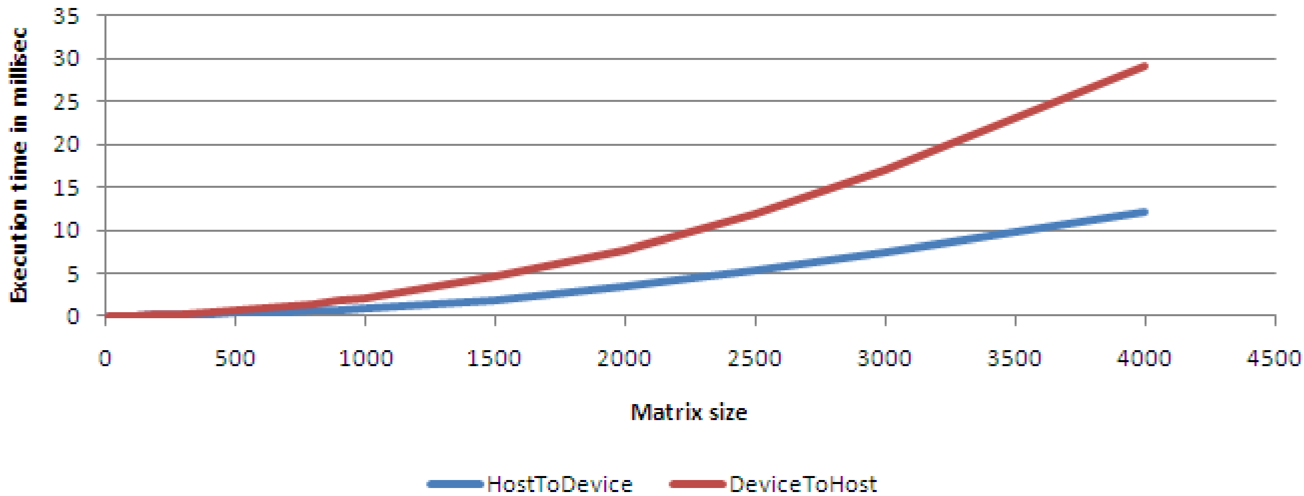

Figure 16: Data transfer between Host and Device
The bandwidth transfer is limited by hardware specification. The difference of bandwidth from HostToDevice and DeviceToHost is also explained by hardware writing and reading speeds.

Nvidia documentation [30] about Telsa C2050 exposes a bandwidth transfer of 144GB/sec. The CUDA SDK bandwidth test tool applied to our testing environment gives the results in Figure 17 These results are really slower then what indicated by Nvidia! In our simulation, the transfer time HostToDevice of a N=2500 matrix takes 4.885 ms. The amount of data transferred is `N*N*sizeof(float)` bytes. A `float` in a double precision environment is 4 bytes. This gives an actual bandwidth of 5.11GB/sec.

```
Host to Device Bandwidth, 1 Device(s), Paged memory
  Transfer Size (Bytes)    Bandwidth(MB/s)
       33554432                 4724.2

Device to Host Bandwidth, 1 Device(s), Paged memory
  Transfer Size (Bytes)    Bandwidth(MB/s)
      33554432                  4035.3

Device to Device Bandwidth, 1 Device(s)
  Transfer Size (Bytes)    Bandwidth(MB/s)
      33554432                  84511.6
```

Figure 17: Bandwidth test results (from CUDA SDK)

The bandwidth announced by Nvidia is the theoretical bandwidth of the GPU card. The reason why it is much slower is the same one which explains why the data transfer from Host to Device is faster then from Device to Host. From a GPU perspective, the bandwidth is identical in both directions. However, writing on the Device is faster than writing on the Host, and the writing on the Host is slower than the bandwidth from the Device. That's why data transfer in this direction is slower: the bottleneck is the transfer bandwidth speed of the Host.

We can note that we could easily divide the number of elements to transfer by two. Instead of copying the whole $N \times N$ matrix, we can transfer a vector $A = \{ a_{11}, a_{21}, a_{22}, ..., a_{i1}, ..., a_{ii}, ..., a_{NN} \}$. Indeed, A being symmetric, we don't need the upper triangular part. An element (i,j) would be found at index $\frac{(i+1)i}{2} + j =$ `((i+1)i)»1+j` . Unfortunately, the relative small gain of time during memory transfer would be lost during the computation due to the extra operations required to find the index of the element in the matrix.

5.3.3 Computation

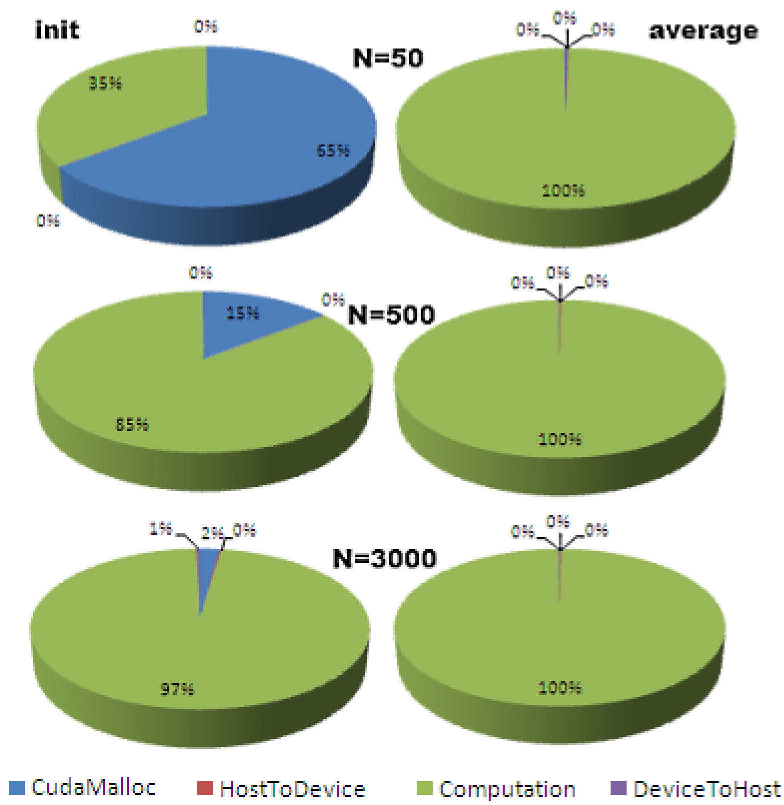

Figure 18: GPU usage: relative time comparison of activities
Context initialization introduces a lack of performance for small matrices but the real interest of the Figure is to prove the importance of having a good and efficient algorithm.

The Figure 18 shows the percentage of time required by every step in the calculation. At the scale of the problem, `DeviceToHost` and `HostToDevice` memory transfers are negligible. The conclusion of this section is that the computation itself has to be improved, more than any other parts. Three different improvement could do this:

1. Using shared memory instead of global memory.
2. Using the optimal number of threads/block for each kernel.
3. Using coalescent memory access.

The first point can't be achieved with this algorithm. Indeed, each kernel performs one read on global memory and one write. This can't be reduced with such separated kernels: a single copy to shared memory would take the same number of accesses to global memory: one to read and one to write back. The only possibility for this algorithm would be to join the three kernels into one. Then, the usage of shared memory would be useful. However, in such a situation, it would need a global synchronism system to deal with the data dependencies, but CUDA only does support per block synchronism. Implementing a global barrier would be possible using atomic operations. However those are not supported before architecture 4.0 and our one is only 2.0 (see Figure 7).

About the second point, the optimal number of threads can be found by a further analysis.

Last is the third point: [6] teaches us that if half a warp of threads executes a Global Shared Memory access, it requires one memory transaction per thread unless if those threads perform a coalescent memory read. In this case the transfer can be done in one single transaction. As a transaction with Global Shared Memory takes 400 to 600 cycles, coalescence would greatly improves efficiency. A coalescent memory access occurs when ([6] page 21):

1. The width of the thread block is a multiple of a half warp.
2. The width of the array is a multiple of a half warp.

The number of threads should also be chosen not to overcome the total number of registers and shared memory allowed per block (see Figure 7). Our algorithm requires 8 registers and 28 bytes of shared memory per thread at most (result obtained using CUDA Profiler). This is low enough not to worry about it: even with the maximum threads per block we won't exceed the limit.

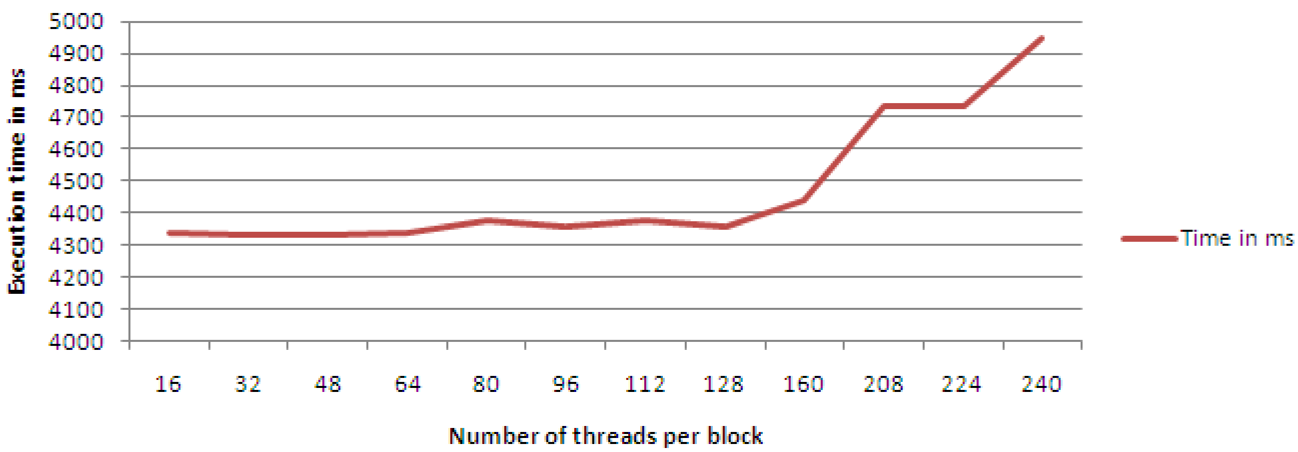

Figure 19: Optimum number of threads/block for normalization kernel
Optimum performance is reached at $N = 48$ corresponding to theoretical expectation in term of warp size correlation.

The figure 19 suggests that the optimal number of threads per block in the normalization kernel is 48. This is 3*16 or 3 times the size of half warp. This corresponds to the criteria of coalescence stated before.

Figure 20 suggests that the optimum number of threads per block to use in the submatrix kernel is one thread per block. This could seem surprising. But as the update of every coefficient in the sub matrix is independent, an interesting fact comes up. It is more efficient to use one block per element than having many elements computed by different threads within the same block as GPU have enough calculation to do to keep busy.

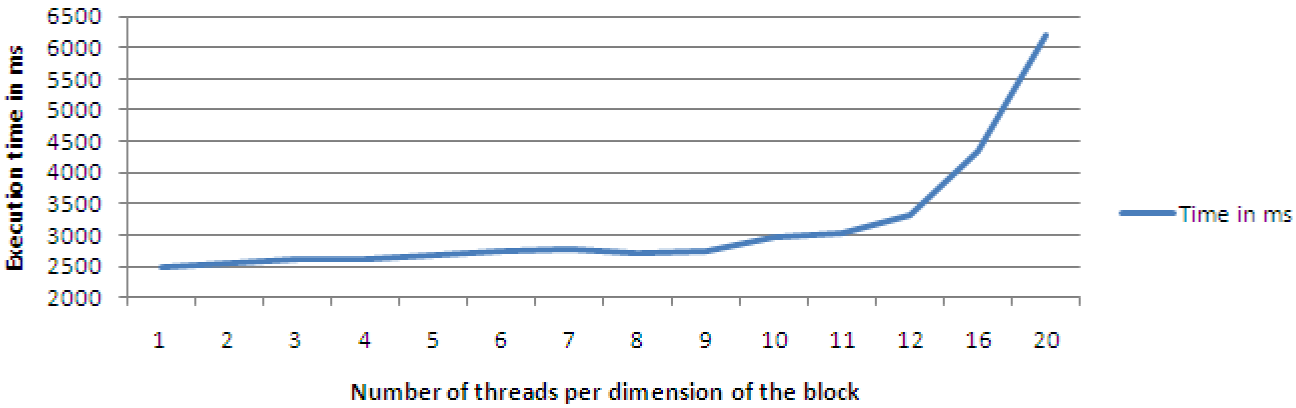

Figure 20: Optimum number of threads per dimension of 2D block for sub-matrix update kernel
Having one thread/block is enough to keep the GPU busy.

5.4 Performance test and analysis for case 48-1

The aim of this case is to obtain the speed-up with the optimal settings determined for this algorithm. This speed-up will be the optimal one obtainable for this algorithm on this particular plateform. Similarly to 512-22, the case called 48-1 means: 48 threads/block are used for NormalizeColumnKernel and $1 \times 1 = 1$ threads is used in UpdateSubMatrixKernel.

The accuracy in the results of this case and the previous one is strictly identical as both do the exact same operations.

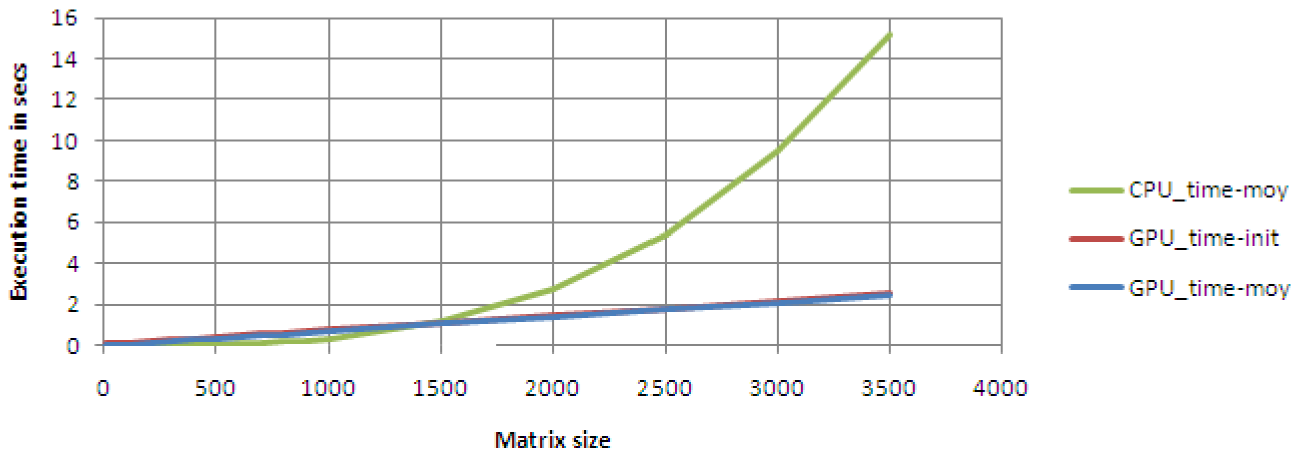

Figure 21: Performance CPU vs GPU for case 48-1
At the scale of the computation, the context init operation influence is now reduced.

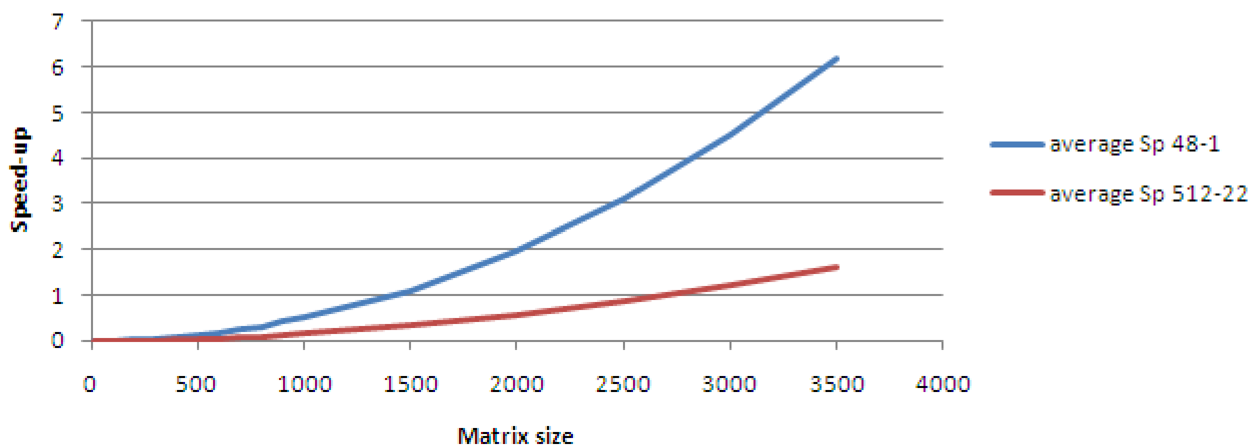

Figure 22: Speed-up CPU vs GPU for case 48-1
Simply setting the GPU resources accordingly is enough to ensure a consequent performance gain.

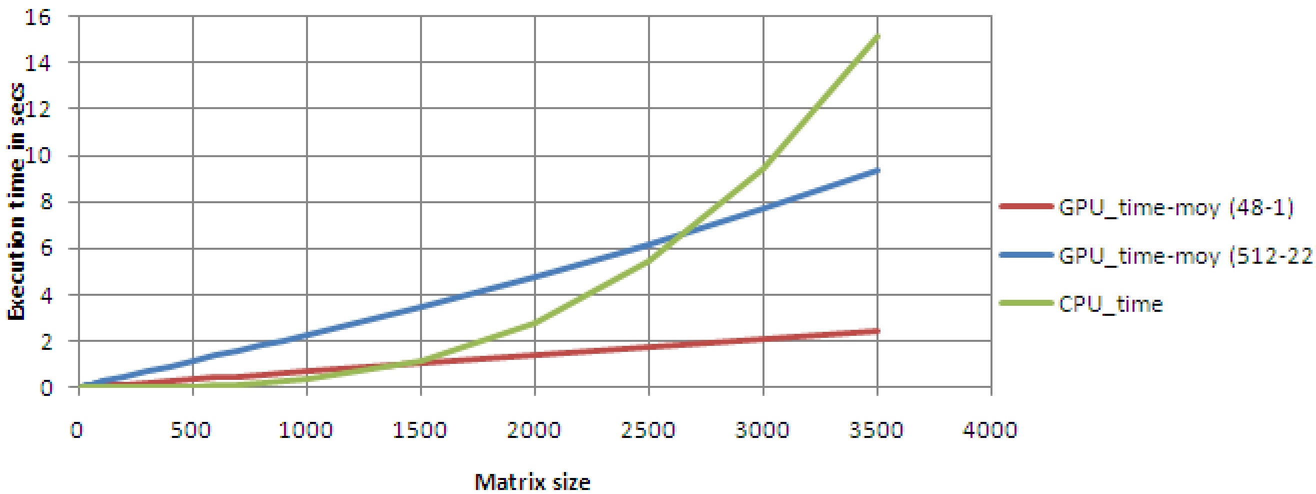

Figure 23: Comparison between case 48-1 and case 512-22
This figure emphasis what is shown on previous one.

5.5 Discussion and Conclusion

To conclude the study of this first algorithm, we have shown that it is rather simple to port Cholesky decomposition to a GPU using a naive technique. The transfer from a C implementation to a naive CUDA C one is quite straight forward. However, we also shown that the control of tricky situation require some deeper understanding of the CUDA architecture. Indeed, when the C algorithm is mastered and when CUDA programming principles are understood (about a week using [37]), only a day or so is needed to convert CPU program to this pointed algorithm.

This being said, the speed-up gained with such an implementation is not enough to justify the investment for most of the applications. Indeed, for a problem size requiring a matrix of less than 1500×1500, the speed-up is nil. For bigger matrix, the speed-up is superior to one which make it beneficial but not as high as GPU would let suppose. This comes from the high number of data dependencies, thus global memory access and synchronization required for the computation of each element. Finally, let's remember that those results are available in the double precision environment of a Tesla card. The accuracy of the solution and the speed-up would be degraded using the GeForce card from the development environment.

Although we sound a bit negative in this conclusion, the real interesting part of this was to determine conditions and parameters for optimal performances. It also has aimed for me to understand more deeply what GPU is doing and how CUDA works: that was my first CUDA code! On Figure 7-8 of [41] is shown a similar performance using only a GeForce GTX280. As this card is not as good as ours, the only conclusion is that the algorithm is improvable. That is where the next section comes in.

6 Cholesky on GPU for dense matrix: Blocked algorithm

6.1 Algorithm

We have seen previously that the three kernels of our first naive GPU implementation could not be merged to compute the decomposition at the scale of the matrix size. Indeed, no threads synchronization is possible within different CUDA blocks. The idea of this algorithm is to decompose the matrix A into blocks having the size of CUDA blocks [15]. In other word, we will process the matrix block per block using the Cholesky decomposition on each block. This means that the size of the matrix to decompose has to be a multiple of the size of a CUDA block.

Therefore, in the linear problem $Ax = b$ where A is an $N \times N$ matrix with $N = 3 \times size\_of\_CUDA\_block$, the decomposition $A = LL^T$ can be written as:

$$\begin{pmatrix} A_{1,1} & A_{2,1}^T & A_{3,1}^T \\ A_{2,1} & A_{2,2} & A_{3,2}^T \\ A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} = \begin{pmatrix} L_{1,1} & (0) & (0) \\ X_{2,1} & L_{2,2} & (0) \\ X_{3,1} & X_{3,2} & L_{3,3} \end{pmatrix} \begin{pmatrix} L_{1,1}^T & X_{2,1}^T & X_{3,1}^T \\ (0) & L_{2,2}^T & X_{3,2}^T \\ (0) & (0) & L_{3,3}^T \end{pmatrix} =$$

$$\begin{pmatrix} L_{1,1}L_{1,1}^T & L_{1,1}X_{2,1}^T & L_{1,1}X_{3,1}^T \\ X_{2,1}L_{1,1}^T & X_{2,1}X_{2,1}^T + L_{2,2}L_{2,2}^T & X_{2,1}X_{3,1}^T + L_{2,2}X_{3,2}^T \\ X_{3,1}L_{1,1}^T & X_{3,1}X_{2,1}^T + X_{3,2}L_{2,2}^T & X_{3,1}X_{3,1}^T + X_{3,2}X_{3,2}^T + L_{3,3}L_{3,3}^T \end{pmatrix} \tag{2}$$

The equation (2) is the case of a 3-partition, ie the matrix is composed of 3 blocks. Algorithm 12 shows the general idea of solving a k-partition matrix.

Algorithm 7 Computation of $A = LL^T$

Input: A an $N \times N$ matrix with n a multiple of CUDA block size.
Output: the computation of the Cholesky factor L.

1: calculate $L_{k,k}$ = CholeskyOnBlock($A_{k,k}$)
2: calculate column $X_{i,k} = A_{i,k}(L_{k,k}^T)^{-1}$ with $i > k$
3: update sub-matrix $A_{i,j} = A_{i,j} - X_{i,k}X_{j,k}^T$ with $i > k$ and $j \geq i$

The interesting aspect of this computation is that it follows the rules we have point earlier. No synchronization of threads within different blocks is required so we will be able to compute CholeskyOnBlock in a single kernel taking advantage of shared memory. The size of matrix will have to be a multiple of block size. If moreover we choose the block size as being a multiple of a half warp, coalescent memory access will be able to occurs. In other word, this algorithm has the opportunity to be much faster than previous one. At this point, the speed-up is however speculative as the paper [15] in which the idea was found has not any performance test.

One last point before we start implementation: I think that the solver should be adapted to the problem we have and not the opposite. In other word, I want our algorithm to be able to solve the decomposition for any matrix size, not only the ones which size if a multiple of block size. To do so, we introduce a prequel operation: matrix augmentation. Let us consider A an $N \times N$ matrix and s being the size of a block. If N is not a multiple of s, we can find d such as $N' = N + d$ is a multiple of s. Therefore, our algorithm can be performed on A' the $N' \times N'$ augmented matrix of A. A' as the following structure:

$$A' = \begin{pmatrix} A & (0) \\ (0) & Id_d \end{pmatrix}$$

The cholesky factor L' of A' restricted to $N \times N$ will therefore be equal to L, the factor A.

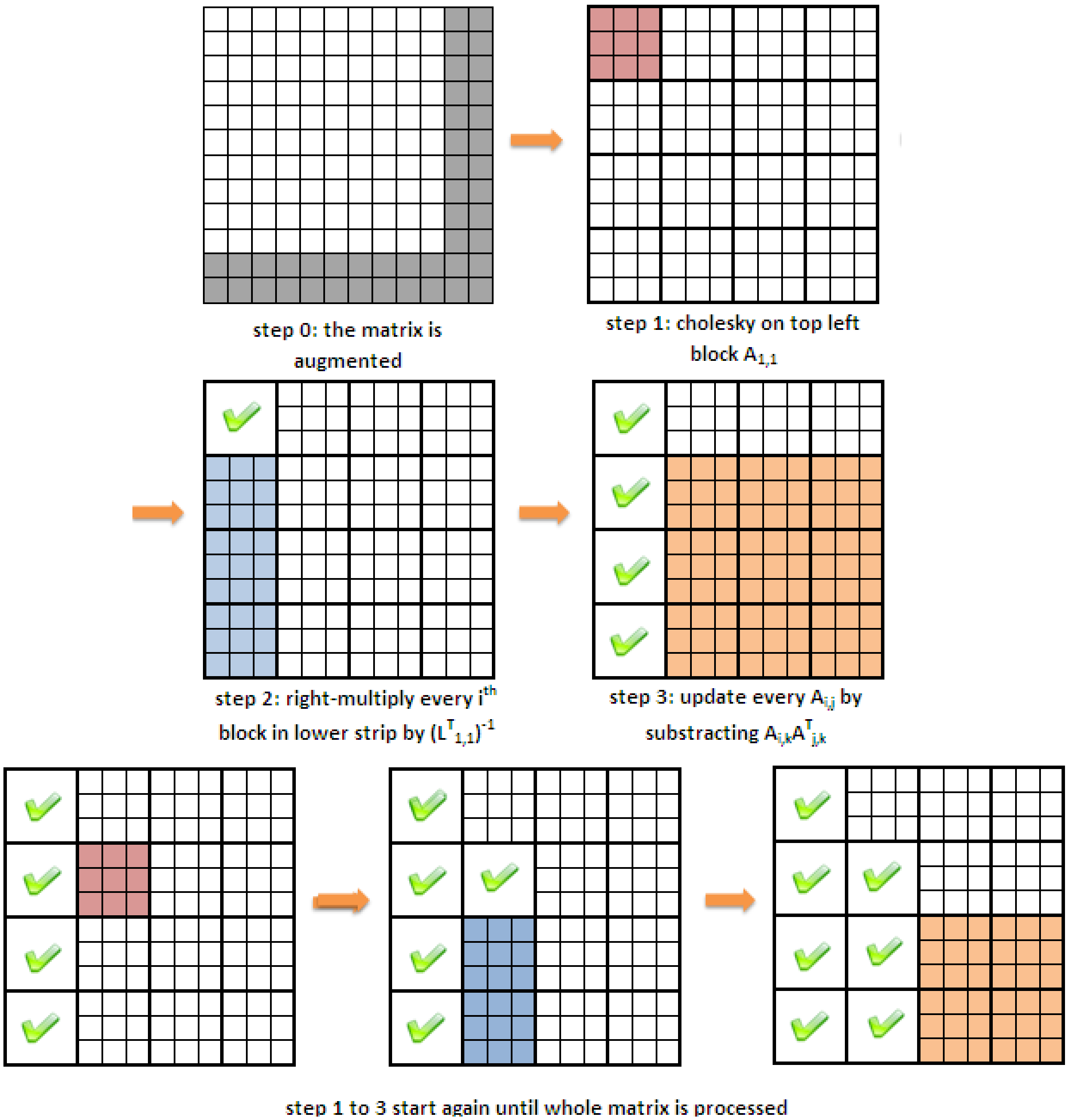

Figure 24: Step by step process for *per block algorithm* (step given in Algo 12)

Algorithm 8 Cholesky algorithm performed on a diagonal block (Algo 12 step 1)

Input: $A_{k,k}$ the block from matrix A to factorize
Output: $L_{k,k}$ the Cholesky factor of the diagonal block k

```
x = threadIdx.x // the thread coordinate x in the CUDA block
y = ThreadIdx.y // the thread coordinate y in the CUDA block
sharedBuff(x,y) = A_{x,y} // copy the block in shared memory
for k = 0 to blockSize − 1 do
    SynchronizeThreads()
    if y=k then
        /* Compute the diagonal elements root square */
        diagElement = sqrt(sharedBuff(k,k))
        /* Update the column k */
        if x ≥ y then
            sharedBuff(x,y) = sharedBuff(x,y) / diagElement
        end if
    end if
    SynchronizeThreads()
    /* Submatrix update */
    if (x ≥ y > k ) then
        sharedBuff(x,y) -= sharedBuff(x,k) × sharedBuff(y,k)
    end if
end for
SynchronizeThreads()
A_{x,y} = sharedBuff(x,y) // copy the block back to global memory
```

Algorithm 9 Normalisation algorithm performed on lower strip k (Algo 12 step 2)

Input: $A_{i,k}$ the i^{th} block in the lower strip under block $A_{k,k}$
Input: boffset : the equivalent coordinate j of the first column of this block is the whole matrix A (block offset).
Output: $X_{i,k}$ the i^{th} block under diagonal block k in Cholesky factor L

blockSize = blockDim.y // *the dimension of the block*
bx = (blockIdx.x+1) × blockSize + boffset // *the coordinate of the first line of this block in the whole matrix A*
x = threadId.x // *the thread coordinate x in the CUDA block*
y = ThreadId.y // *the thread coordinate y in the CUDA block*
sharedBuff(x,y) = $A_{bx+x,y+boffset}$ // *copy current block into shared memory*
trTopLeft(x,y) = $A_{y+boffset,x+boffset}$ // *copy the transposed top left block calculated previously in Algo 8 :* $L_{k;k}^{T}$
SynchronizeThreads()
/* *Compute* $(L_{k,k}^{T})^{-1}$ *the inverse of top left transpose block*/
if x=0 **then**
    for k=blockSize-1 to 0 decreasing **do**
        dotprod = 1 if k=y but 0 otherwise // *the value of this element in an Id matrix*
        for m=blockSize-1 to 0 decreasing **do**
            dotprod -= trTopLeft(k,m) × inverseTrTopLeft(m,y)
        end for
        inverseTrTopLeft(k,y) = dotprod / trTopLeft(k,k)
    end for
end if
SynchronizeThreads()
/* *right-multiply block* $A_{i,k}$ *by* $L_{k,k}^{T})^{-1}$ */
for k=0 to blockSize-1 **do**
    prod += sharedBuffx,k × inverseTrTopLeft(k,y)
end for
$A_{bx+x,y+boffset}$ = sharedBuff(x,y) // *copy back current block into matrix*

Algorithm 10 Sub-matrix update (Algo 12 step 3)

Input: $A_{i,j}$ a block from lower-right submatrix of $A_{k,k}$
Input: offset : the coordinate of $A_{k,k}$
Output: $A_{i,j} = A_{i,j} - X_{i,k}X_{j,k}^T$ submatrix update

x = threadId.x *// the thread coordinate x in the CUDA block*
y = ThreadId.y *// the thread coordinate y in the CUDA block*
blockSize = blockDim.y *// the dimension of the block*
bx = (blockIdx.x+1) × blockSize + offset *// the coordinate of the first raw of this block in the whole matrix A*
by = (blockIdx.y+1) × blockSize + offset *// the coordinate of the first column of this block in the whole matrix A*
if (bx < by) **then**
    return *// only compute lower-right triangular part*
end if
sharedX(x,y) = $A_{offset+x,by+y}$ *// $X_{x,k}$ block computed in Algo 9 into shared memory*
sharedTrX(y,x) = $A_{offset+x,y+offset}$ *// copy $X_{y,k}^T$ block into shared memory*
SynchronizeThreads()
/ compute $X_{x,k}X_{y,k}^T$ */*
for k=0 to blockSize-1 **do**
    mul += sharedX(x,k) × sharedTrX(k,y)
end for
SynchronizeThreads()
$A_{bx+x,by+y}$ -= mul *// Update*

6.2 Performance test: optimum block size

Let us first try to determine the most appropriate CUDA block size. As a reminder, this block is a 2D threads block of size between 1 and 22 on each dimension. The Figure 25 show the execution time of the decomposition of the matrices N=2000 and N=3000. For each block size, the step 0 will have to augment the matrix if necessary but we consider it all being result of either case 2000 or 3000 no matter the augmentation.

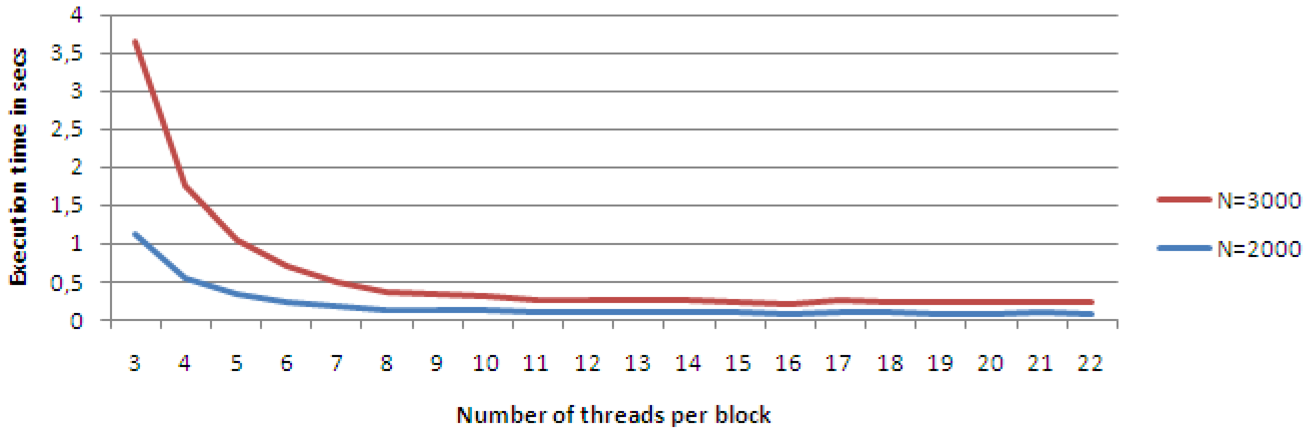

Figure 25: Performance test depending on block dimension
In this algorithm, if the block is chosen big enough, its size have less influence then in previous one. However, the case $N = 16$ still shows a little better performance, consistently with theory of warp size correlation.

According to this study, a block dimension of 16×16 seems to be the most efficient one, equally with 22×22. As 16 is also the size of half a warp (condition for coalescent memory access from [6]) we will now go with it.

6.3 Performance test: results

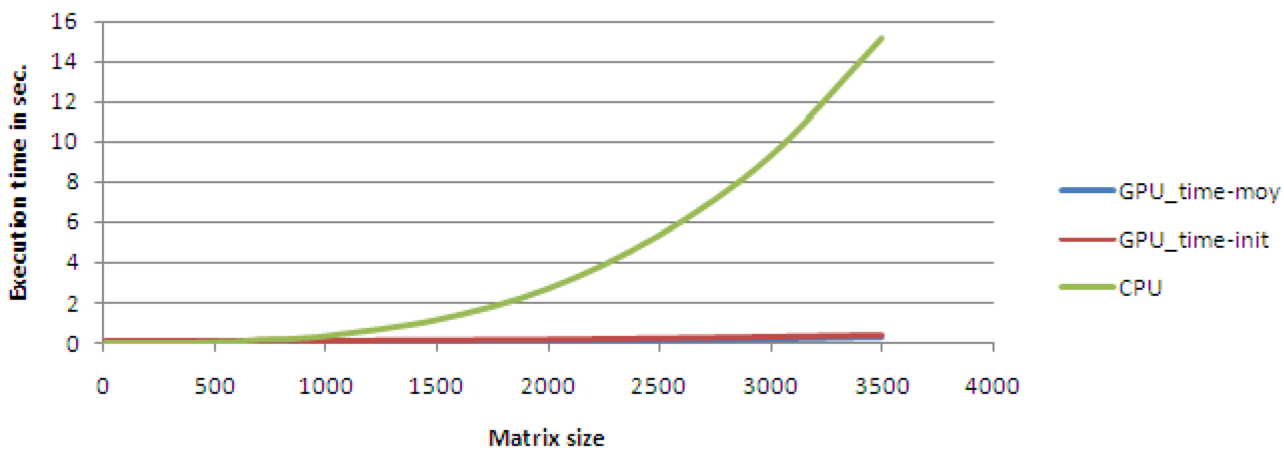

Figure 26: Performance test comparison GPU vs CPU
This algorithm is really faster.

The figure 26 confirm prediction: this algorithm is really faster than previous one. Although it involves more operations than previously, this algorithm take advantage of both shared memory and coalescent memory access. The figure 27, introduces speedup and helps to focus on how much faster this algorithm is, compared with CPU version. The highly benefit part is that this algorithm is faster than a CPU for matrices from N=200. Also, the speed-up increases quickly as the matrices are getting bigger.

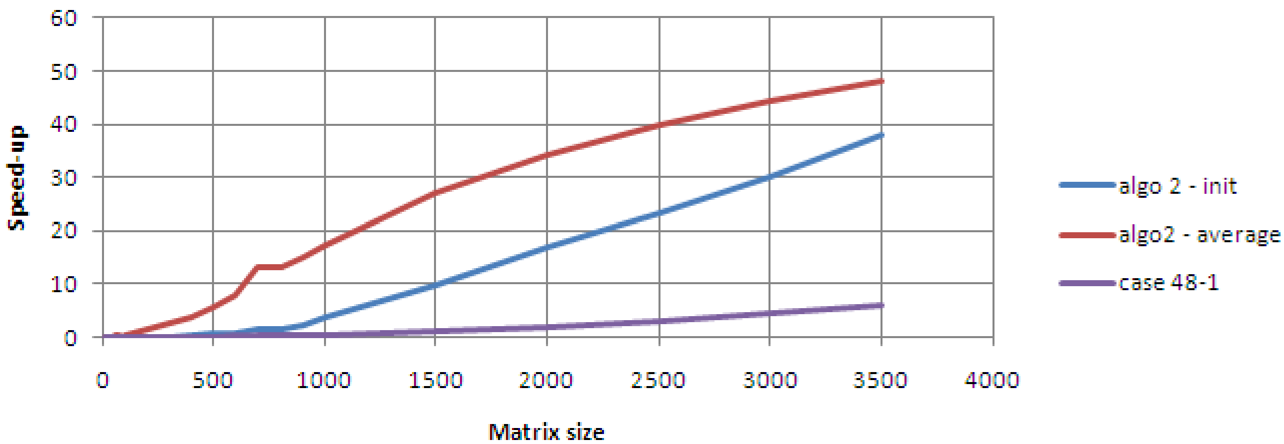

Figure 27: Speed-up comparison with previous algorithm
This figure highlights how the memory storage and memory access optimisation greatly helps the computation.

This speed however has a cost: a little lost in accuracy of the result as shown in Figure 28. The little drop-down of precision is due to the higher number of operations needed and therefore of all the rounding-up it involves.

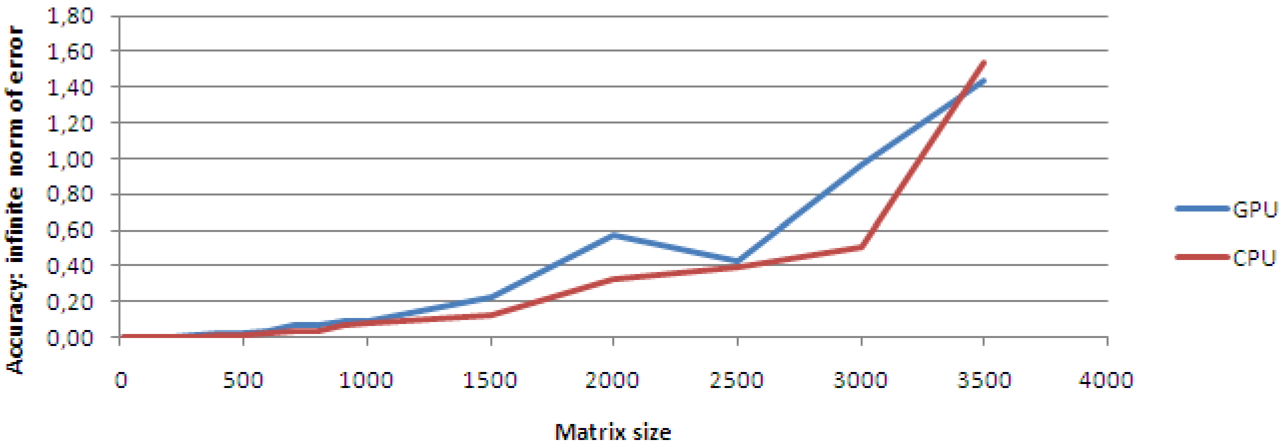

Figure 28: Accuracy of result GPU vs CPU
This algorithm uses more operation to compute each elements. Rounding up issues of those operations affects the accuracy.

6.4 Discussion and Conclusion

The blocked-algorithm introduced in second is more interesting than the previous pointed-algorithm, either in terms of skills needed to implement it and in term of performance. I mean here that its implementation is far more performance then the pointed algorithm (by a factor of 9 for a $N = 3500$ matrix) but requires much programming skills to manage the threads and the block offsets.

Therefore, this algorithm shows two main issues. Firstly, a little decrease in precision due to the larger amount of operations necessary. This decrease is at most 0.5 difference between the infinite matrix norms check between CPU and GPU calculations (see Figure 28). Secondly, this algorithm is harder to implement, debug and maintain. This difficulty is due to the power of a CUDA language. Due to parallelism, for a single line of code, thousands of coefficients are potentially changed. This makes it difficult to follow *on paper*. A deeper understanding of the problem is then required to understand dependency and synchronisation issues between threads.

Final point to underline is that the reference [15] from which comes the idea of this algorithm does not provide test results. Therefore, it is impossible for me to compare my results with ones from the literature. However, an issue has to be highlighted and double checked here: the results previously shown are better than anything I could find during my literature review. Various other works studies different methods. The comparison with their results has to be discussed.

- [24] introduces results using the CUBLAS library (by Nvidia) a linear solver for GPU using iterative methods. The speed-up introduced in the paper is a factor two (2x).

- [42] proposes a result for a sparse Cholesky factorization using "direct solver" without giving the actual algorithm. We suppose it similar to a case 256-16 as their result is a speedup of three (3x). (§3 p.4)
- [44] introduces an iterative solver for extended Cholesky ($A = LDL^T$). Figure 4 shows execution time results. As the graduation of the logarithmic axis of time is not very precise it makes comparison difficult. However, we can extrapolate a speed-up of 20x for N=512 and a speed-up of 2x fro N=2048. Therefore it is interesting to notice that oppositely from the solution in this thesis, their solution is more efficient for small matrices and loses performance as the matrix gets bigger.
- [41] does not detail the algorithm used. However, on a GTX280 card, their speed-up is 4.4x (§6 Table 4)

We have therefore investigated on this issue: why are our results so high compared to what literature offers? To go further in checking, I have tested bigger matrices. The Figure 29 shows the results with matrices up to $N = 10\,000$. The interest of this test is to show that although the graph is high, its shape is similar to the ones in literature: that's a clue of correctness.

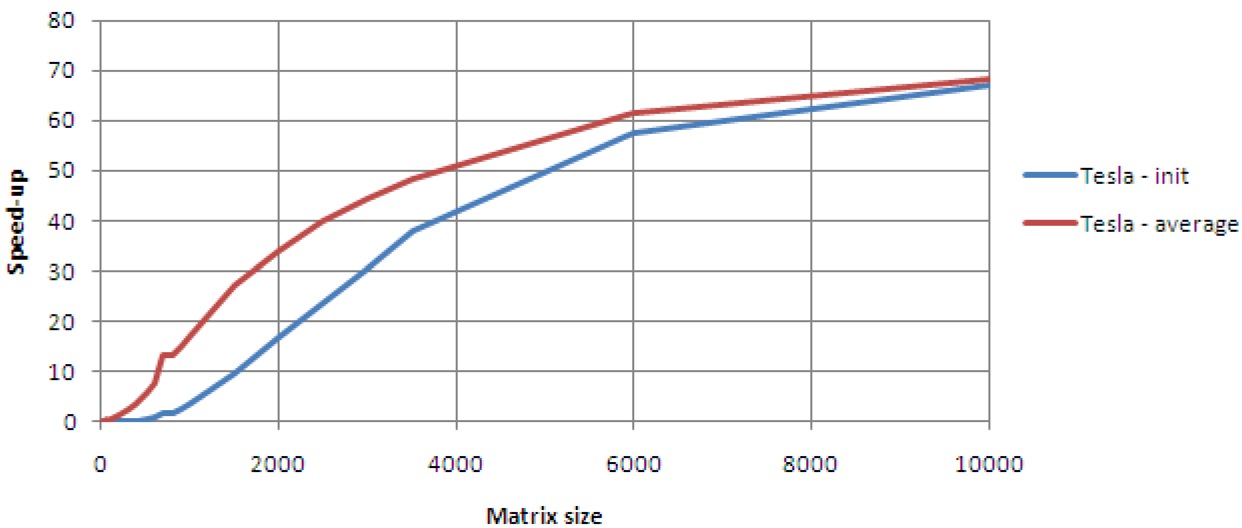

Figure 29: Speed-up using the Tesla environment. Test with matrices up to $N = 10\,000$.
Although the speed-up is higher than what can be found in literature, this graph has the expected shape.

Another clue can be found by crossing the performance results of the developement plateforme with the ones of the testing environment. In other words, testing the speed-up obtained with GeForce 9800GT versus Intel Core 2Duo then with Tesla C2050 versus Intel i7-980X. This is done in Figure 30. As you can see, the speed-up of an earlier graphic card versus a newer CPU is within the order of the ones in literature papers.

This is my arguments for the following explanation of the phenomenon. Only modern and descent GPUs have a CUDA architecture. The papers in litterature review does not often gives details on their hardware specifications. I suspect it tries to compare the computation speed they have with their good CPUs (probably not from individual PCs like I do here) with their good GPUs which definitely were not as good as today ones. The GPU capacities have improved so drastically for the past very few years that comparing results with last generation ones is non sense. In definitive, I propose to explain the correct shape but high value of my speed-up analysis compared to what can be in literature by pointing at improvements of GPU hardware capacities of the past years: we have now better hardware.

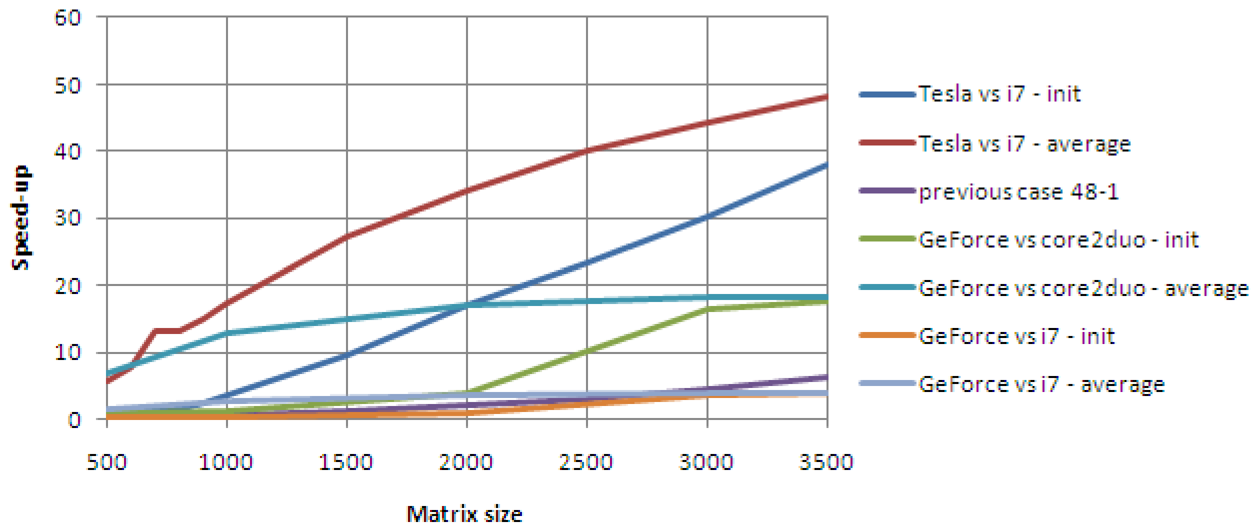

Figure 30: Speed-up comparison of various GPU-CPU combination
A same algorithm tested on earlier and newer GPUs and CPUs have a very different speed-up ratio.

7 Monte Carlo option pricer

7.1 Quantitative finance overview

Option trading have been officially created in finance in 1973 with the *Chicago Board Options Exchange* (the CBOE). Based on the paper "The Pricing of Options and Corporate Liabilities" from Fischer Black and Myron Scholes published the same year[7], this new way to trade allows to sell and buy options on assets rather than assets themselves.

As I had no previous background on quantitative finance, the acquisition of vocabulary, understanding and knowledge on the subject has not been that easy. Most of it was gained though the book "Paul Wilmott introduces quantitative finance"[43] which will therefore be my main reference in this section.

To make it simple to the reader of this report, I will only detail here the basis of so called European options. Let suppose a company produces an asset A. Today, this asset worth £100. This is called the present value S_0 of the asset. That is the price a customer would pay for it today. This price changes over the time. If it is a produced good, its price is rather stable and vary at the scale of years. If it is a primary resource (petrol, wood, gold, currency, etc...) its value may change every day. The volatility v is a percentage number describing the maximum fluctuation of the value S of the asset from one period δt to another.

Instead of buying the asset, a customer can buy an option on this asset for a price p called *prime* or *premium*. This gives him the right to buy the asset later in time at a certain price decided today when the option is bought. The decided price is called the *strike* price K and the time when the option is issued for is called the *maturity time* T. In our example, the buyer may predict that the asset is going to worth £150 next year. Therefore, he can buy an option for $p = £10$ to have the opportunity to buy the asset in a time $T = 12$ month for £110. This is a right, not a must. In other words, if his prediction is correct, he will be able to buy if for £110 and save £$(S_T - K + p)$. If his prediction was not true and the asset actually only worth £90 at maturity, he may decide not to buy it after all, and therefore only looses his premium p. This example is called a *call*: an option giving the right to buy asset at a certain *strike* price. The same option also have a *put*: the right for the seller to sell the asset at a certain *strike* price. Therefore, the payoff for a call european option is $C = \max(S - K, 0) - p$ and its put is $P = \max(K - S, 0) - p$.

In any case, *call* or *put*, some other variables will interfere. The risk-free interest rate r defines how much money the option buyer would have earned with his money if instead of buying a risky option he had simply placed it at the bank. It also helps to determine the today price of an option according to the expected value of money at maturity. The dividend yield Y is the percentage of money that has to be paid each period on the benefice of the option. Finally δt is the period in which money calculation occurs. The option can therefore be continuous or discrete. In the case of discrete option, the money devaluation, the dividend yield and the interest rates are calculated for every period δt (generally each month) from the strike to the maturity. In the case of continuous, the calculation is made only at maturity and for the whole time at once.

The Figure 31 aims to remind the parameters and notations of option pricing.

Option has a Call and a Put
- p: the premium value of option
- S_t: value of the asset at a certain time
- T : maturity
- K: value of strike
- v: volatility
- r: risk-free interest rate
- Δt : period time on discrete interest rate
- Y_t: dividend yield at period t
- payoff = $\max(S_T - X, 0) - p$ for call
  = $\max(X - S_T, 0) - p$ for put

Figure 31: Option parameters and notations

A *basket* of options is a group of numerous options taken together. The repartition of the N assets A_i in the basket is defined by their weight W_i (percentage) in the basket and therefore $\sum_i W_i = 1$. Those options can be independent or correlated. It the case of correlated options, the value A_i of an asset depends on the value of every other assets in the basket. The correlation matrix C exposes theses coefficients. Therefore, C is a symmetrical matrix with 1 on the diagonal and $0 \leq c_{ij} < 1$ for every non-diagonal coefficients. Plus, for each assets i, its correlation with the other assets is such as $\sum_{j \neq i} c_{ij}$=1. In other words, the sum of every non-diagonal elements in a row is 1: the matrix is definite-positive.

7.2 Monte Carlo basket option pricer

Pricing a basket of option is actually trying to predict the value S_T of any asset in the basket at maturity and therefore determining a good call or put premium to have a positive payoff. The idea behind Monte Carlo pricing is to use random numbers to generate a possible evolution of the asset value over time. This is called a *path* of Monte Carlo. Using the average change over a large number of paths, it is possible to have a good evaluation of the asset's value.

The general idea behind Monte Carlo simulations is to use a random phenomenon to approximate a value that can't be measured otherwise. Let us consider the following easy example of a Monte Carlo usage, illustrated in Figure 32. Let us suppose the army earns a field of well-known surface S_{field}. In this field is a rather strange shaped lake.

The General of this army wants to know its surface, but measuring it is impossible due to its very strange shape. Monte Carlo idea is to use a random phenomenon -the random fire of cannonballs in the field- to approximate the surface S_{lake} of the lake. This surface will be given by the formula $S_{\text{lake}} = (\text{numShots} - \text{numShotsIn})/\text{numShots}) * S_{\text{field}}$ where numShotsIn is the number of cannonballs fired in the field. If the firing position is randomly distributed, the larger the number of cannonballs used is, the better Monte Carlo approximation will be. Such a method could for instead determine the value of PI in such a way.

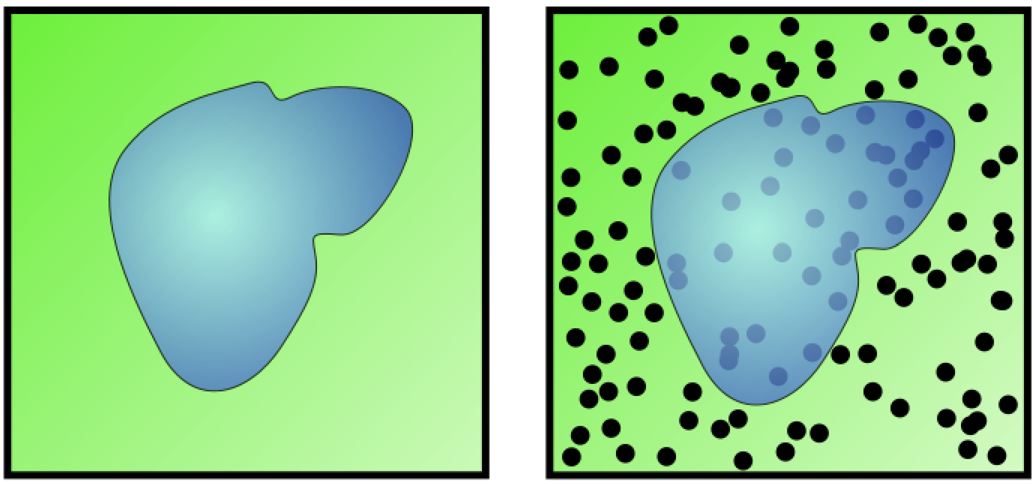

Figure 32: Monte Carlo principles thought surface of lake determination
Monte Carlo aims at evaluated quantities using random phenomenon.

In the case of basked of option pricing, the value $S_{i,\delta t}$ of an option i over the period T with a time step period of δt can be determined using the following formula [43]:

$$S_{i,t+\delta t} = S_{i,t} \times \exp\left((r - Y_i - \frac{1}{2}v_i^2)\delta t + v_i Z_i \sqrt{\delta t}\right) \tag{3}$$

In the Equation (3), Z is a correlated random vector of N elements, N being the number of options in the basket. It is obtained as $Z = L.X$ where L is the Cholesky factor of the correlation matrix C. X is an uncorrelated centred normally distributed random vector $\sigma(0,1)$. Let `rnd(0,1)` be a linearly distributed number picked between 0 and 1. Using the Muller-box principle, a randomly generated vector $\sigma(0,1)$ can be made by choosing the X_i elements such as:

$$X_i = \sqrt{-2\ln\left(\text{rnd}(0{,}1)\right)}\cos\left(2\pi.\text{rnd}(0{,}1)\right)$$

Another approximation, faster to compute but less accurate, can be made using a the central limit theorem approximation:

$$X_i = \sqrt{\frac{12}{N}}\left(\left(\sum_{i=1}^{N} \mathrm{rnd(0,1)}\right) - \frac{N}{2}\right) = \sum_{i=1}^{12} \mathrm{rnd(0,1)} - 6 \text{ with N=12}$$

The confidence in the call and put values calculated with Monte Carlo can be estimated as a 95% trust domain. This interval is given as being:

$$[\mathrm{averageCall} - 1.95 \times \frac{\mathrm{varianceCall}}{\sqrt{\mathrm{NB\_RUN}}}, \mathrm{averageCall} + 1.95 \times \frac{\mathrm{varianceCall}}{\sqrt{\mathrm{NB\_RUN}}}]$$

and similarly for a put option. It means that we can be 95% sure that the values belong to this interval. Therefore, if computed call and put values are not included in their trust intervals, the simulation is obviously wrong. If it belongs to it, then, the smaller the interval is, the more accurate is the computation.

To implement Monte-Carlo pricer, the Formula (3) is embedded in multiple nested loops. The Algorithm 11 shows the main steps in such a simulation. This algorithm as been first programmed sequentially on CPU. The aim was to validate its results.

The validation has been made using the Black-Scholes pricer offered with the book [43]. Of course, our simulation had to be reduced to a special case solvable using Black-Scholes. The dividend yield is reduced to nil ($Y = 0$), the pricer is made continuous ($\delta t = T$) and the options are uncorrelated (for $i, j \in [\![0, N]\!]\ C_{i,j} = 1$). In those conditions, the Black-Scholes pricer can be applied and gives similar order results. Let us remind that we can not obtain the exact same results as the whole computation is subject to randomness. However, this test validates the algorithm and its nested loops. It also partially validates the formula. This formula will therefore be used as such and we trust its author [43] about its correctness.

Algorithm 11 Main steps in Monte Carlo option pricer

initialise every parameters for each asset A_i of the basket
NB_RUN = number of Monte Carlo run.
for $r = 1$ to NB_RUN **do**
  / reinitialise assets value to value-at-zero */*
  reinitialise S to initial values
  for $t = 1$ to T by step δt **do**
    / create a new correlated CND(0,1) */*
    X = new random vector $\sigma(0, 1)$
    Z = new correlated random vector build with $L_C.X$
    for $i = 1$ to N **do**
      $S_i = S_i \times \exp\left((r - Y_i - \frac{1}{2}v_i^2)\delta t + v_i Z_i \sqrt{\delta t}\right)$
    end for
  end for
  basketPrice = S.P *//P: option weight vector*
  / call and put value for this run */*
  call = MAX(basketPrice-strike, 0)
  put = MAX(strike-basketPrice, 0)
  / sum of every call and put during the runs */*
  sumCall += call;
  sumPut += call
  / sum of every squared call and put during the runs */*
  sumCall2 += call × call
  sumPut2 += put *times* put
end for
/ average put and call during runs */*
averagePut = sumPut/NB_RUN
averageCall = sumCall/NB_RUN
/ variance of call and put during runs */*
varianceCall = sumCall2/NB_RUN - averageCall × averageCall
variancePut = sumPut2/NB_RUN - averagePut × averagePut
/ simulated call and put after money devaluation re-evaluation */*
call = exp(- riskfree × maturity) × averageCall;
put = exp(- × riskfree × maturity) × averagePut;
/ compute 95% trust domain intervals */*
downTrustCall = averageCall - 1.95 × varianceCall / (sqrt(NB_RUN));
upTrustCall = averageCall + 1.95 × varianceCall/ (sqrt(NB_RUN));
downTrustPut = averagePut - 1.95 × variancePut/ (sqrt(NB_RUN));
upTrustPut = averagePut + 1.95 × variancePut/(sqrt(NB_RUN));

To conclude this introducing to Monte Carlo pricer, let us notice that Monte Carlo requires heavy computation as a large number of path are required and the pricing can potentially been made on a large number of options in the basket. To perform the calculation, Quantitative finance simulations are usually made on clusters of 1000 to 50000 nodes which as a high cost (price, electricity, maintenance,..). This is the reason why GPGPU sound as a good alternative: the large number of threads usable in a kernel makes it a low cost but effective built-in solution for massive parallel computation.[5]

7.3 Basic Monte Carlo pricer on GPU

For this first example, we are going to consider a plan of none correlated options to be priced. One possible scenario would be a company seeking to invest by buying multiple independent options. Before doing so, it would therefore seek to evaluate all possible options to choose the most advantageous ones. In this context, a plan of evaluation would be made, considering hundreds of non-correlated options to price.

As explained previously, this whole work on Monte Carlo has been added on the working plan at the last moment. Therefore, we will try to make it brief to end in the deadline. For this reason we will use the implementation of Monte Carlo given in CUDA SDK under the name `MonteCarlo` [28] with only a few changes in data structure and added parameters for a more complete simulation.

7.3.1 Issues

We have seen previously that Monte Carlo is a solution using randomness to evaluate things. In the case of option pricing, it takes a random centred normally distributed vector. The most important issue with a Monte Carlo implementation on GPU is therefore the generation of a random normal distribution. Indeed, their is no such a thing as a `rnd()` function equivalent in CUDA. It is therefore impossible to generate random number unless using the *CURAND library*[29] available only on CUDA framework 3.2 and above, which we do not have access to (graphic cards at university are 1.6 capable only).

To trick this issue, we can generate a cumulative distribution function inverse of PATH_N elements: one for each Monte Carlo path in the simulation to perform. This distribution is obtained through the Moro's inverse method. This will lead to a correct but unique Monte Carlo simulation.

- The simulation will be correct because the fact that elements are ordered does not matter in this case as every option is independent. When the average is done, the order of elements does not change the result.

[5] for the whole paragraph: Andre Nedecoux, Excelian, email correspondence 15/08/2011

- However, this simulation will be unique as the same distribution will be generated again and again. So two different executions of the program will compute the same result, which should not be the case with a proper random Monte Carlo.

The second issue is to validate the accuracy of the computation. Fortunately, as every options are independent, we know the analytical formula to price it using Black-Scholes equation [43]:

$$\text{Call} = S - \text{CND}(d1) - Xe^{RT}\text{CND}(d2) \tag{4}$$

with $d1 = \log \frac{S}{X} + \frac{R-Y+\frac{1}{2}v^2}{v\sqrt{T}}$ and $d2 = d1 - v\sqrt{T}$.

Using this result, we can check the accuracy of the Monte Carlo simulation. The comparison will be made by computing the L1 norms of

$$S_{T,\text{GPU}}^{MonteCarlo} - S_{T,\text{CPU}}^{Black-Scholes} \tag{5}$$

$$S_{T,\text{CPU}}^{MonteCarlo} - S_{T,\text{CPU}}^{Black-Scholes} \tag{6}$$

$$S_{T,\text{GPU}}^{MonteCarlo} - S_{T,\text{CPU}}^{MonteCarlo} \tag{7}$$

The norm Formula (5) will however be the only shown as a graph as it is the most important one. The two other ones aim at verifying that every simulations -GPU and CPU, Monte Carlo and Black-Scholes- have similar results.

7.3.2 Parallelism

The Monte Carlo formula given in Equation (3) will have to be repeated for every OPT_N options and every PATH_N Monte Carlo paths. In term of parallelism, every path for every option is entirely data independent which will help to obtain a very good speed-up. For a small ratio $\frac{PATH\_N}{\text{OPT\_N}} \leq 8192$, one block per option will be used. Each block is composed of THREAD_N threads representing one single Monte Carlo path for the option. This means that we can in theory have up to 65535 options being computed at the same time. As it takes thousands of Monte Carlo paths to simulate one option, for each of those, THREAD_N paths will be computed at the same time by THREAD_N threads. For more clarity, the Figure 33 illustrates the resource allocation.

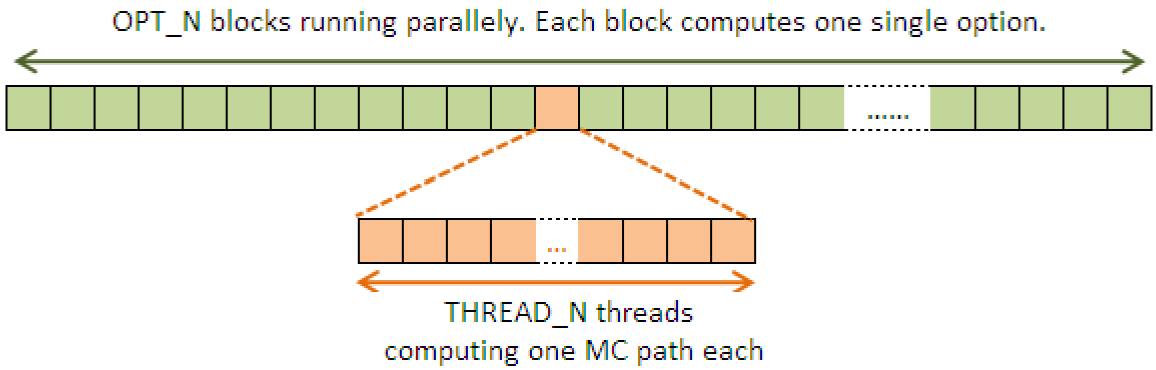

Figure 33: Thread allocation in case of a small path/option ratio
One block per option is used (green). Every block contains THREAD_N threads iterating thought the Monte Carlo paths (orange).

This thread allocation means that THREAD_N Monte Carlo paths are computed simultaneously for every option. On CPU, the complexity was O(PATH_N $\times$ OPT_N). Here the overall complexity will therefore be $O(\frac{\text{PATH\_N}}{\text{THREAD\_N}})$.

When the ratio PATH_N over OPT_N is high ($\frac{PATH\_N}{\text{OPT\_N}} > 8192$) we will use more blocks per thread. The number of blocks per options BLOCK_N will be 64 if OPT_N $<$ 16 and 16 otherwise. These numbers have been determined by Nvidia developer team as being a good ratio to keep the GPU busy [31]. The overall complexity of this situation will therefore be $O(\frac{\text{PATH\_N}}{\text{BLOCK\_N} \times \text{THREAD\_N}})$. The Figure 34 illustrates these choices.

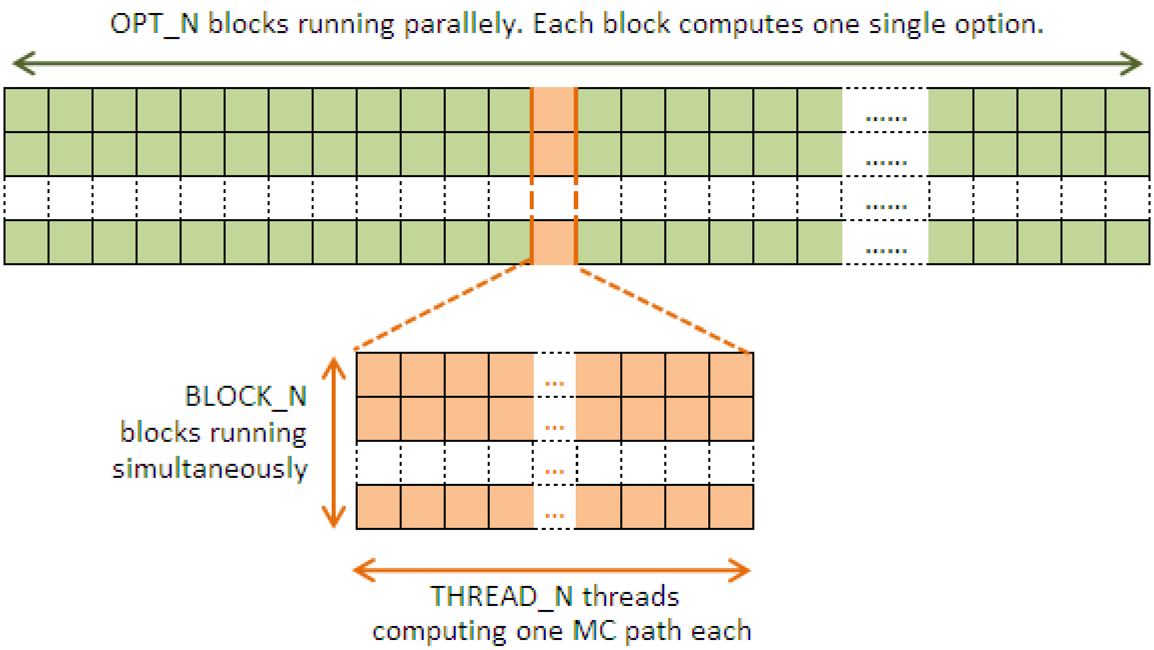

Figure 34: Thread allocation in case of a big path/option ratio
Multiple blocks per option is used (green). Every blocks contains THREAD_N threads iterating thought the Monte Carlo paths (orange).

7.3.3 GPU implementation

First of all, we will define two main data structures: a basket of options and an option. A basket of option has multiple field:

- OPT_N: number of options in the basket
- basket value: the sum of each option value at t, weighted with its weight in the basket
- basket strike: the sum of each option strike, weighted with its weight in the basket
- list of options: the list of every options to price
- PATH_N: number of path for the simulation
- a list of normally distributed values (see issues with random number §7.3.1)

Each option is itself a data structure called `optionData` containing:

- value
- strike
- volatility
- risk-free interest
- maturity

- dividend yield

In order to use shared memory on small structures, the results of pricing will be stored in a third data structure, the `optionValue` containing:

- expected option value: the priced option value
- confidence measure

An array of OPT_N `optionData` will be initialized on the device. And the listof options in the basket is copied into it. Therefore, the device stores the data of each option on global memory. The `MonteCarloKernel` is then called using parallelism according to the thread allocation previously described. Either OPT_N blocks of THREAD_N threads each, in the case of one block per option. Or N_BLOCKS $\times$ OPT_N blocks of THREAD_N threads each, in the case of multiple blocks per option.

Inside the `MonteCarloKernel`, the computation is done simply. For each option, one or more blocks are created. Each block has an array of THREAD_N option values passed as a parameter. Each thread within the block access its own cell in the array to store its path result. In other word, it copies the option data from the block option into local memory and computes the option value: this make one Monte Carlo path. It stores this value to the result array and eventually iterates for another path.

When every path are computed, the array has to be reduced so that every results are summed up. This will help to compute the average of every path results for the particular option of this block. The reduction is an iterative process where half size threads are used at each step. Each threads sums up two data of the array. When its done, the process is started again using half threads as it did at last step. Again, each thread sums two data. This is repeated until cell 0 contains the total data sum. The Figure 35 visually illustrates the process.

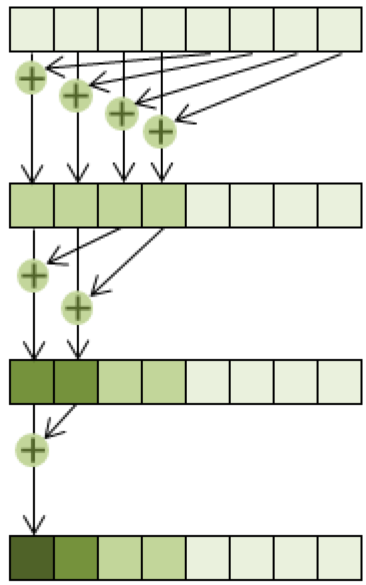

Figure 35: Iterative sum reduction process
At each step, half of the values are summed in a partial calculation.
At the end of the process, the first cell contains the total sum.

This reduction creates the only restriction in term of problem sizing: the number of threads per block has to be a power of two. In order to keep each thread equally busy within a block, they all should iterate the same number of time. Therefore, we extend our condition to PATH_N: from now, the number of paths used for the simulation has also to be a power of two.

Algorithm 12 Monte Carlo Kernel

Input: An array with the data of every options to price.
Output: A priced basket using PATH_N Monte Carlo path.

1: create an array `SumData` of THREAD_N `optionValue` in shared memory
2: copy the data of the option to price
3: **for** (iSum from threadIdx to PATH_N by step of THREAD_N) **do**
4: predict an option value using Formula 3
5: add the result to `SumData` at position iSum
6: **end for**
7: reduce the `SumData` array

7.3.4 Results and Analysis

Optimum number of threads:
Once our algorithm is functional, it takes to determine its optimum settings. Here,

we do not have much to set but the number of threads per block used in each option pricing (ie the number of orange squares in Figures 33 and 34). The following tests were performed to study the behaviour of the computation according to parallelism level:

- **Options/sec**: helps to understand the general behaviour of the parallelism
- **Speed-up** : will help to determine which THREAD_N number leads to optimal performances
- **L1 norm** : the fastest computation choice is maybe not the most accurate one. This will therefore check that the chosen THREAD_N is appropriated in term of accuracy.

Those tests will be achieve with a plan of 256 options using respectively 8192, 524288 and 4194304 paths. The number of paths have been chosen arbitrarily to correspond to a small, medium and rather large simulation.

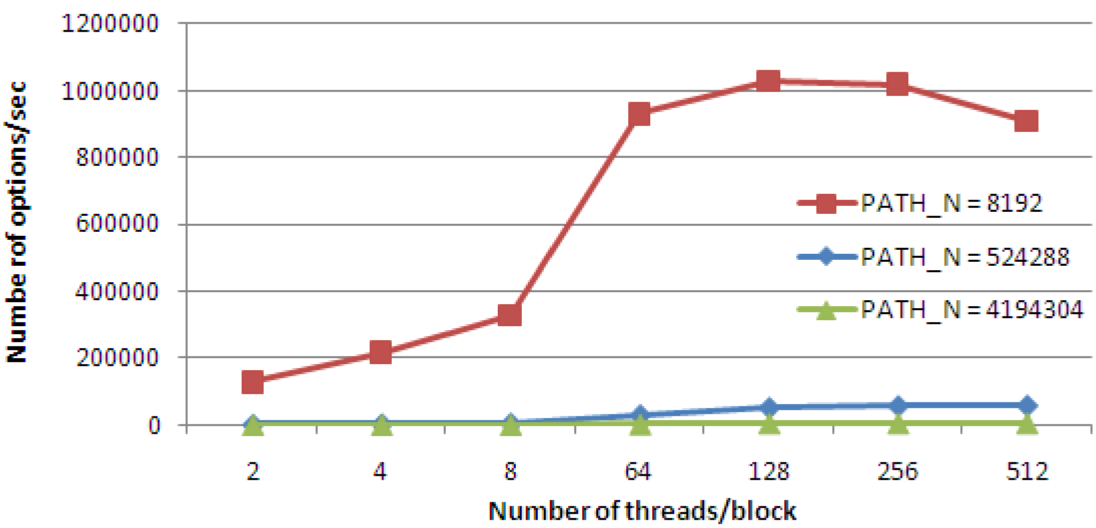

Figure 36: Options per second evolution depending on parallelism
This has been computed with 256 options on three different Monte Carlo number of paths. Notice the decrease of performance after THREAD_N = 256.

The previous Figure 36 emphasis that the larger the basket is in number of options, the more computation will be done and therefore the slower an option is priced. It also shows a decrease of performance for THREAD_N > 256. Except for this, the higher parallelism is, the faster will be computation. This is confirmed in term of speed-up by the Figure 37.

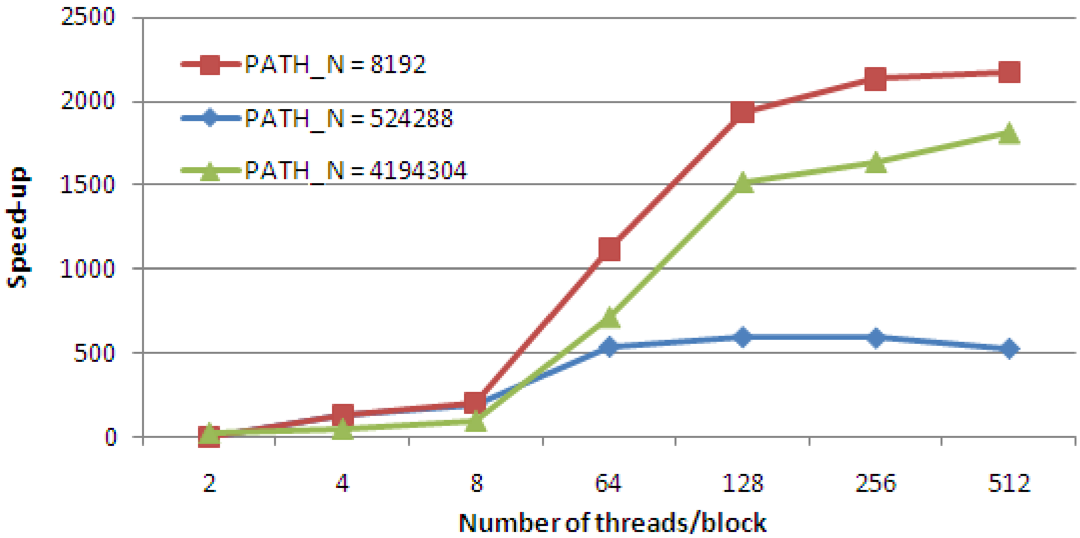

Figure 37: Speed-up evolution depending on parallelism
This has been computed with 256 options on three different Monte Carlo number of paths. Notice the decrease when THREAD_N > 256.

The Figure 37 indicates a very high speed-up: a factor above 2000 when used with more then 128 threads/block. Such a speed can be achieve for various reasons. First, every computation are independent from each other. Therefore, threads have no data dependencies and can be computed in massive parallelism, without synchronisation or delay. Secondly, the usage of shared memory within a block and GPU reduction to compute the averages over blocks also greatly improves the efficiency.

One more time, we have this little decreasing in performance when THREAD_N > 256. For this reason, it seems fair to choose 256 as an appropriate setting for THREAD_N. Moreover, the Figure 38 shows that precision for THREAD_N = 256 is as good as for any other large number of threads per block. Notice also that the accuracy of the solution is made better as a large number of Monte Carlo paths is used (the lower the L1 norm is, the better is the accuracy).

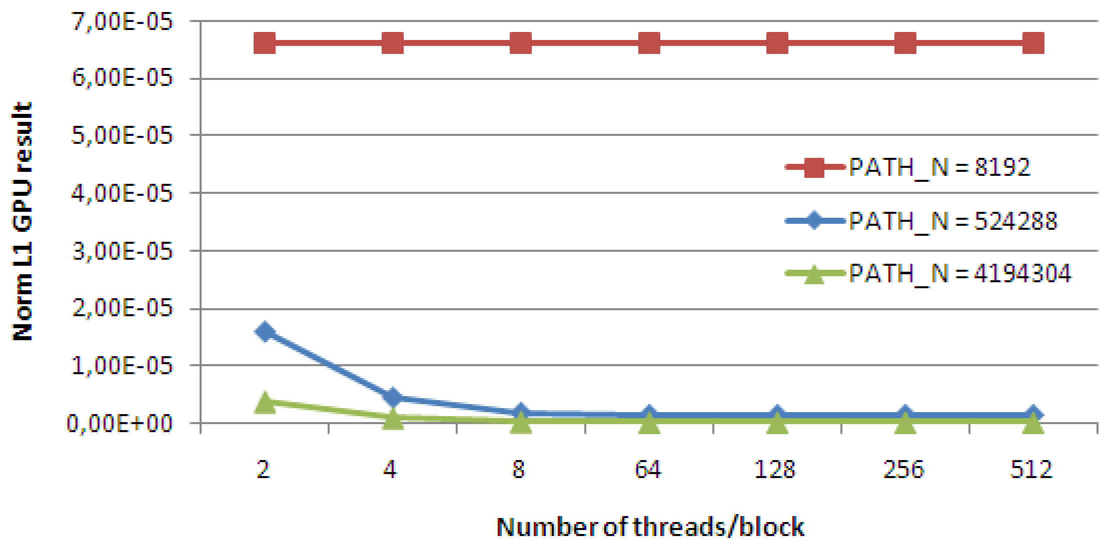

Figure 38: Accuracy of the solution depending on parallelism
This has been computed with 256 options on three different Monte Carlo number of paths. The lower is the better. Notice therefore that precision rises as the number of MC paths rises.

To conclude with theses preliminary tests, we have determined here that THREAD_N = 256 seems to be the optimal degree of parallelism within a same block. That is the setting we will continue with for performance test.

Performance tests:
Providing the preliminary results of THREAD_N determination, we can expect a very interesting accuracy and speed-up for this Monte-Carlo pricer.

Let us start with the accuracy of the simulation. Indeed, this is the condition for the simulation to worth it. The Figure 39 shows the evolution of the precision for various number of options depending on the number of Monte Carlo paths. In this graph, the lower is the better. As expected, three main points can be noticed.

- Precision improves as the number of Monte Carlo paths augments. this is due to a higher number of simulation which converges in average to the solution, reducing the confidence domain.
- Precision does not depend on the number of options to price. This is true in this scenario as options are independent. However, with correlated options this would not be observed.
- After a number of simulation, the precision gain is negligible compared to the resources involved in the computation.

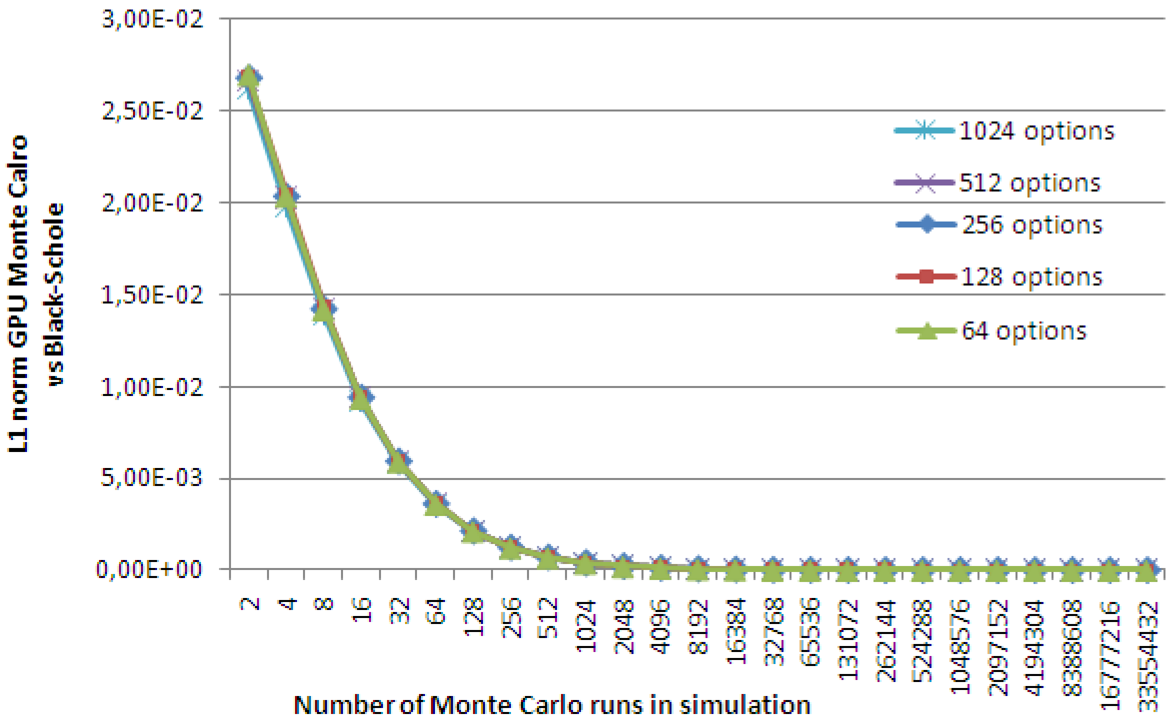

Figure 39: Average accuracy evolution
- precision rises as the number of paths augments
- precision does not depend on the numbe rof options to price
- after a number of simulation, the benefit for accuracy to enlarge the simulation is not.

If the precision obtained with a GPU Monte Carlo pricer is good, let us investigate its benefice in term of performances. This is what shows the Figure 40. In this graph, we can easily see the highly benefit of GPU computation over CPU as soon as the number of paths rises. Because every path and every option is independent, because we use shared memory and high parallelism, for those reasons the speed-up obtained is in the order of a few thousands. The rising of speed-up according to the number of paths comes from the single block per option parallelism. THREAD_N threads are allocated to compute all paths of each options. Therefore, when less then THREAD_N paths are computed, each thread computes its path without having to iterate. The time recorded is therefore: start kernel → one iteration → stop kernel. When many paths has to be computed, each thread iterates many times. The measured time is therefore :
start kernel → one iteration → another iteration → ... → stop kernel. This is less time then if the kernel had to be started and stopped each time: the number of path/sec processed gets better and so does the speed-up. Of course, this converges to a limit where starting and stopping kernel time is negligible compared to computation. This explains the flatten happening in the graphs and particularly visible for large number

of option in Figure 40. This happens when the ratio $\frac{PATH\_N}{\text{OPT\_N}}$ approaches 8192.

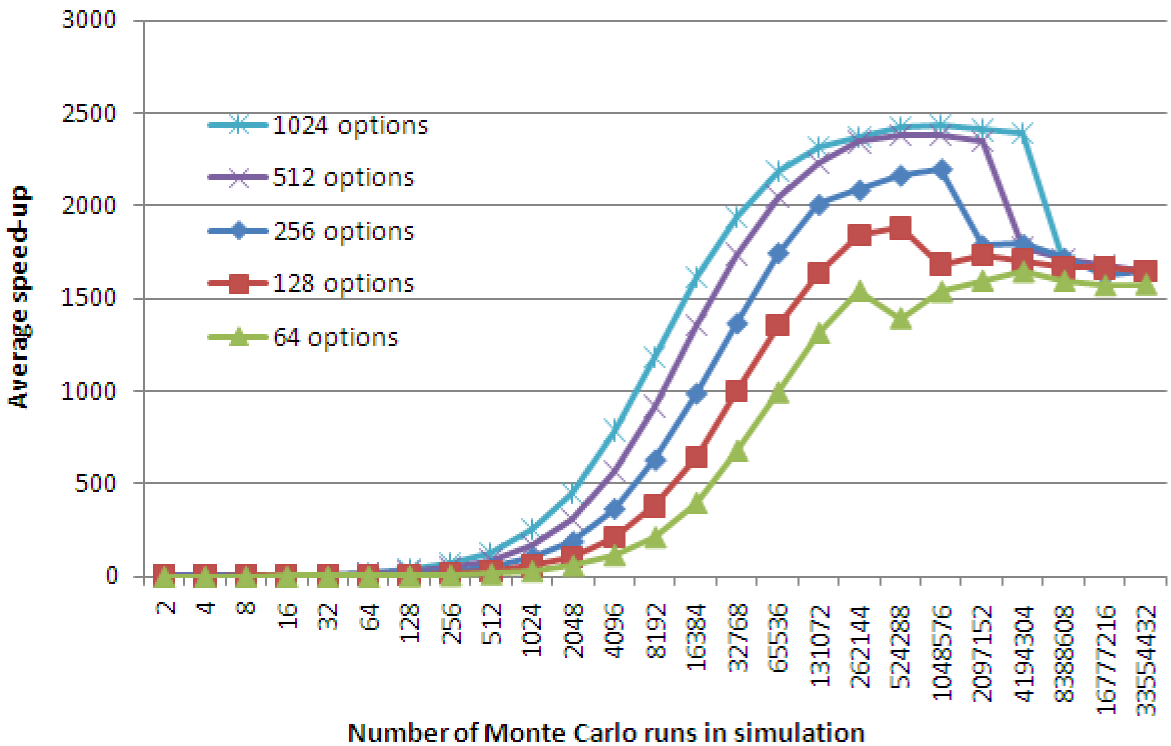

Figure 40: Average speed-up evolution
The ratio $\frac{PATH\_N}{OPT\_N}$ *has an important role in speed-up evolution with a noticeable point at* 8192.

Finally, notice that when the ratio get over 8192, the simulation starts using multiple blocks per option. As the number of blocks used per option is quite large, this offer a constant speed-up no matter the number of options and paths in the simulation.

To conclude this analysis, we would advice to simulate Monte Carlo using several hundreds thousands of paths and up to a million or so. Indeed, the gain of accuracy for larger simulation is very low. Also we would like to point at the very high speed-up obtained. This helps to reduce a 38 seconds CPU simulation (for 512 options priced with 1048576 paths and 256 threads/block) to 15 ms on GPU.

7.4 Advanced Monte Carlo option pricer on GPU

An advanced Monte Carlo pricer on GPU would be able to price correlated options such as explained in §7.1 and described by Algorithm 11. Theses options can be correlated in value (example: if the value of asset A depends at 40% on the value of asset B and 60% on asset C) and/or in term of weight in the basket (example: if the basket is composed of 30% of asset A, 60% of asset B and 10% of asset C).

Contrarily to §7.3, in this scenario, there is no more data independence between options in the pricing of each path. Indeed, the overall basket value will need to gather the value of every priced options to be computed. We will deal with this issue by inverting the green and orange squares for Figure 34. In other words, we will have multiple blocks per path. Each of theses blocks will compute THREAD_N options simultaneously. This leads to the situation shown on Figure 41.

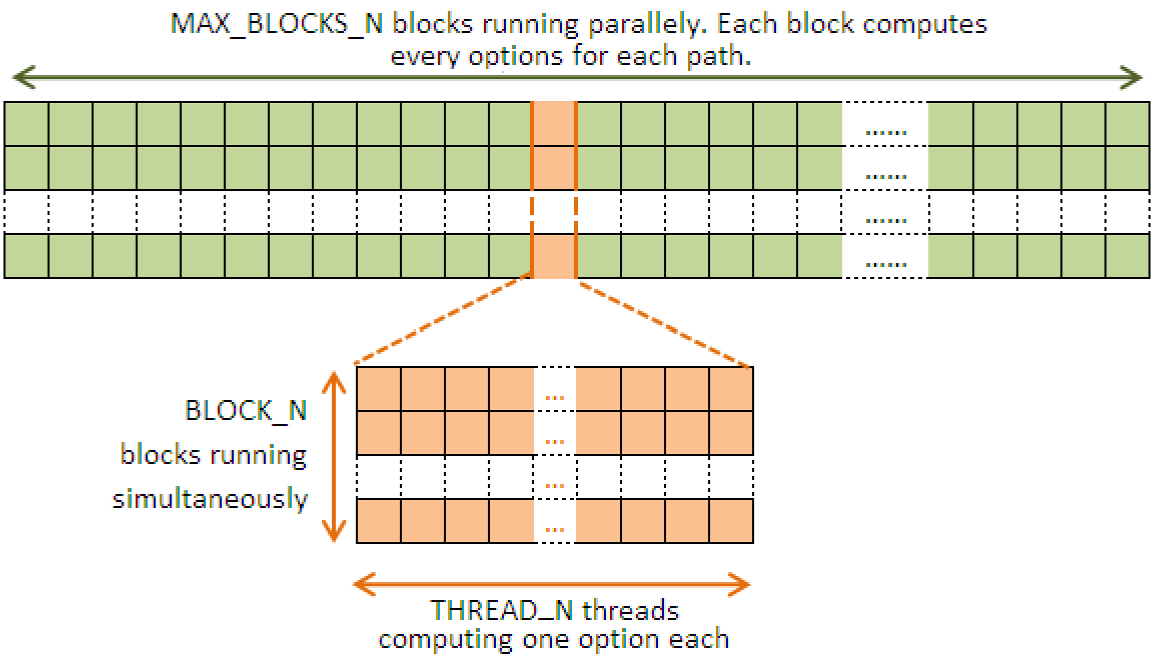

Figure 41: Thread allocation in case of a big path/option ratio
Multiple blocks per path is used. Every blocks contains THREAD_N threads iterating thought the basket options.

To fit with the requirements of this simulation, the `optionData` data structure has to be extended to include option weight and option correlation vector. The `optionValue` will also be extended to store intermediate values necessary to compute basket call and put. Finally, the `basket` itself should be extended to include a price, a strike, a call, a put and a domain confidence for both call and put.

To create the correlated random vector, we will use the Cholesky factor of the correlation matrix. This correlation matrix is a N=OPT_N square matrix. As a result of our study of Cholesky decomposition, we now know what is the gain of GPU computation for this operation. Therefore, we will use the CPU implementation for OPT_N $\leq$ 28 and the GPU blocked algorithm for bigger basket.

7.4.1 Issues and troubles

As explained in §7.3.1, there is no such a thing as random generation in earlier versions to CUDA 3.2. Therefore, we could not generate the necessary random correlated centred normal distribution vector. In the basic Monte Carlo, this could be solved by generating a cumulative distribution. The numbers were ordered and not random, but it was enough to ensure a proper Monte Carlo. In this situation however, this is not enough. We could correlate ordered number, but it would have no sense in term of Monte Carlo simulation. Another idea to solve the issue could be to generate the random numbers on the CPU and to transfer it to GPU. However, this requires PATH_N $\times$ OPT_N numbers and we have seen that PATH_N can be large. Such an array would be long and inefficient to generate on CPU and would not even fit in GPU memory. Even if it did, it would then require as many memory access to global memory as there is coefficients to use. This is an issue we could not solve in the short time available after the implementation was asked.

Another issue is the data transfer and usage (in the device) of CPU declared data structure. For instance, the computation of `optionValue` is complex to follow "by hand" due to the large parallelism scale and the great number of variables involved in the computation. I have made a functional implementation that has trouble in data transfer: some values are not returned correctly. This is frustrating as the implementation is therefore very close to be functional but without debugger, finding the problem is like searching for a needle in a stack. This is the reason why I could not finish the implementation early enough to introduce performance testing in this report.

8 Conclusion and future work

This thesis work has answered every questions from the original Excelian requirements.

Through the implementations and performance tests, we have been able to demonstrate the benefits and difficulties of CUDA parallel implementations. We also have been able to determine the difficulties, the challenges and the parameters involved to obtain a performance code. We have achieve a Cholesky decomposition implementation using a blocked-algorithm offering a speed-up of a factor 50 compared to an i7 CPU computation for square matrices of 2500 elements (see Figure 29). We also have determined that a speed-up of a factor over 2000 (for 128 options and more) can be achieved with a GPU Monte Carlo European option pricer using over a million of Monte Calro paths (see Figure 40).

To sum it up, the main difficulties in programming using CUDA are the restriction due to the technology itself. First comes from the CUDA framework. Before version 3.2, many restrictions constrains the programming. The random number generation as been one of the issue discovered with the Monte Carlo option pricer. Second difficulty comes from debugging: inside debugger needs two graphic cards to operate and I could not use it due to IT restrictions in the university. This has considerably slow down the implementation.

The main parameters I have found to be involved in performance are:

- The parallel algorithm itself: data dependencies and synchronism should be avoided as much as possible.
- Memory storage: the algorithm should be made to have the strongest usage of local and shared memory between threads. This often means gathering dependent calculation in a same block, even if it requires more iteration of every threads (see §7.3.4).
- Memory access: accessing data stored as an array in global memory should be made coalescent. The speed-up obtain with coalescent versus non-coalescent memory read is absolutely non-negligible as shown with Cholesky decomposition (see §6).
- Parallelism level: a wise choice in the number of blocks and threads per block to allocate to a kernel helps its performance. The maximum resources is not necessarily the better (see case 512-22 and 42-1 §5).

Although our work was made in an academic environment, it has shown that GPGPU can offer high speed-up for quantitative finance problems and it would worth to test it with real-world scenarios to clearly demonstrate feasibility and interest. The main question has however found an answer: GPGPU worth investigating as a solution for quantitative finance simulations.

As a future work, we would like to recommend extending Monte Carlo simulation to correlated options using a proper random generation. This would be solved either by moving to a more recent CUDA framework (equal or subsequent to 3.2) or by spending time on quasi random generator from scratch. Indeed, at the scale of the number of paths required for Monte Carlo times the number of options to price, it would be impossible to create and transfer to the device such a large array of random value generated from the CPU. We would also recommend Excelian to gather study scenarios from real use case to help future students to get closer to business reality.

References

[1] Jqantlib, quantlib port to java. http://www.jquantlib.org/ (accessed 01-07-2011).

[2] Matrixmarket. http://math.nist.gov/MatrixMarket/ (accessed 01-07-2011).

[3] Quantlib library. http://quantlib.org/index.shtml (accessed 01-07-2011).

[4] AMD. Amd accelerated parallel processing (app) sdk. http://developer.amd.com/gpu/AMDAPPSDK/Pages/default.aspx (accessed 01-07-2011).

[5] C. Ashcraft, S. C. Eisenstat, and J. W. Liu. A fan-in algorithm for distributed sparse numerical factorization. 1989. http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA211569&Location=U2&doc=GetTRDoc.pdf (accessed 01-07-2011).

[6] S. Barnes. Gpgpu programming using cuda. *Lecture Notes*, Academic year 2010/11.

[7] F. Black and M. Scholes. The pricing of options and corporate liabilities. *The journal of political economy*, pages 637–654, 1973. ThePricingofOptionsandCorporateLiabilities (accessed 05-08-2011).

[8] Université de Besançon. Ufr sciences et techniques, université de besançon, programmation gpgpu. http://raphaello.univ-fcomte.fr/Ig/GPGPU/GPGPU.htm (accessed 01-07-2011).

[9] J. Ecole Polytechnique de Montreal Guerin. Notes d'analyses numeriques, maths2210a, lecture notes, 2009. http://www.mgi.polymtl.ca/JeanGuerin/Mth2210A/jg_mth2210a.htm (accessed 01-07-2011).

[10] J. Erhel, M. Hahad, T. Priol, and Institut national de recherche en informatique et en automatique (France). *Factorisation parallèle de Cholesky pour matrices creuses sur une mémoire virtuelle partagée*. Citeseer, 1993. http://hal.inria.fr/docs/00/07/46/84/PDF/RR-1988.pdf (accessed 01-07-2011).

[11] P. C. Fischer and R. L. Probert. Storage reorganization techniques for matrix computation in a paging environment. *Communications of the ACM*, 22(7):405–415, 1979.

[12] D. Göddeke. Fast and accurate finite-element multigrid solvers for pde simulations on gpu clusters, 2010. http://www.mathematik.tu-dortmund.de/lsiii/cms/papers/Goeddeke2010.pdf (accessed 01-07-2011).

[13] Khronos Group. Opengl. http://www.opengl.org (accessed 01-07-2011).

[14] M. T. Heath, E. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33(3):420–460, 1991. http://www.docjava.com/courses/cr346/data/papers/2031442.pdf (accessed 01-07-2011).

[15] S. Henry. Parallelizing cholesky's decomposition algorithm. Technical report, INRIA Bordeaux, 2009. http://runtime.bordeaux.inria.fr/shenry/papers/HS_Cholesky.pdf (accessed 01-07-2011).

[16] Intel. Intel 6 core i7-980x processor. http://ark.intel.com/Product.aspx?id=47932 (accessed 01-07-2011).

[17] J. H. Jung and D. P. O'Leary. Cholesky decomposition and linear programming on a gpu. *Scholarly Paper, University of Maryland*, 2006. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.84.8623&rep=rep1&type=pdf (accessed 01-07-2011).

[18] Matrix Market. I/o matrix storage file. http://math.nist.gov/MatrixMarket/mmio-c.html (accessed 01-07-2011).

[19] AC McKellar and E. G. Coffman Jr. Organizing matrices and matrix operations for paged memory systems. *Communications of the ACM*, 12(3):153–165, 1969.

[20] J. Michalakes and M. Vachharajani. Gpu acceleration of numerical weather prediction. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–7. IEEE, 2008. http://www.mmm.ucar.edu/wrf/WG2/michalakes_lspp.pdf (accessed 01-07-2011).

[21] Microsoft. Microsoft directx11. http://www.gamesforwindows.com/en-US/directx/ (accessed 01-07-2011).

[22] Microsoft MSDNA. Pipeline stages (direct3d 10). http://msdn.microsoft.com/en-us/library/bb205123.aspx (accessed 01-07-2011).

[23] A. Munshi. Opencl 1.0 specification, 2009. http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf (accessed 01-07-2011).

[24] M. Naumov. Incomplete-lu and cholesky preconditioned iterative methods using cusparse and cublas. 2011. http://developer.download.nvidia.com/assets/cuda/files/psts_white_paper_final.pdf (accessed 25-07-2011).

[25] J. Von Neumann and M. D. Godfrey. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993. http://www.virtualtravelog.net/wp/wp-content/media/2003-08-TheFirstDraft.pdf (accessed 01-07-2011).

[26] Nvidia. Cuda c programming guide. Technical Report Paragraphe 3.2 page 18. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf (accessed 01-07-2011).

[27] Nvidia. Cuda c programming guide. Technical Report Paragraphe 3.2 page 54-55. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf (accessed 01-07-2011).

[28] Nvidia. Cuda sdk code sample: Montecarlo option pricer. http://developer.nvidia.com/cuda-cc-sdk-code-samples (accessed 21/08/2011).

[29] Nvidia. Curand library for nvidia cuda. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CURAND_Library.pdf (accessed 09/08/2011).

[30] Nvidia. High performance fermi-based gpgpu telsa c2050. Technical report. http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf (accessed 01-07-2011).

[31] Nvidia. Monte carlo option pricer. http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/MonteCarlo/doc/MonteCarlo.pdf (accessed 12/08/2011).

[32] Nvidia. Cuda toolkit 4.0, April 2011. http://developer.nvidia.com/cuda-toolkit-40 (accessed 01-07-2011).

[33] University of Michigan. Research topic 3: Compilation for gpus., Nov 29, 2010. www.eecs.umich.edu/~mahlke/583f10/lectures/583L22a.ppt (accessed 01-07-2011).

[34] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007. www.idav.ucdavis.edu/func/return_pdf?pub_id=844 (accessed 01-07-2011).

[35] A. Pothen and C. Sun. A mapping algorithm for parallel sparse cholesky factorization. *SIAM Journal on Scientific Computing*, 14:1253–1253, 1993. http://www.cs.cornell.edu/Info/People/csun/papers/mapping.ps (accessed 01-07-2011).

[36] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. Numerical recipes in c. the art of scientific computing. *Cambridge University Press, Cambridge*, page 994, 1992. http://astronu.jinr.ru/wiki/upload/d/d6/NumericalRecipesinC.pdf (accessed 01-07-2011).

[37] J. Sanders and E. Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley, 2010. http://developer.download.nvidia.com/books/cuda-by-example/cuda-by-example-sample.pdf (accessed 01-07-2011).

[38] Medical Systems TechniScan. Techniscan svara™ warm bath ultrasound (wbu™) imaging system. http://www.techniscanmedicalsystems.com (accessed 01-07-2011).

[39] L. N. Trefethen and D. Bau. *Numerical linear algebra*. Number 50. Society for Industrial Mathematics, 1997.

[40] Lecture Notes Cranfield University. Computational methods in linear algebra. *Applied Mathematics and Computing Group*, Academic year 2010/11.

[41] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11. IEEE, 2008. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.160.148&rep=rep1&type=pdf (accessed 01-03-2011).

[42] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure. On the limits of gpu acceleration. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, pages 13–13. USENIX Association, 2010. http://www.usenix.org/event/hotpar10/tech/full_papers/main.pdf (accessed 07-07-2011).

[43] P. Wilmott. *Paul Wilmott introduces quantitative finance*. Wiley, 2007.

[44] D. Yang, G. Liang, D. D. Jenkins, G. D. Peterson, and H. Li. High performance relevance vector machine on gpus. In *Symposium on Application Accelerators in High Performance Computing*, 2010. http://saahpc.ncsa.illinois.edu/10/papers/paper_46.pdf (accessed 07-07-2011).