THEORETICAL AND NUMERICAL ANALYSIS OF THREE APPROACHES TO THE GPGPU APPLICATION OF THE EXPLICIT FDTD METHOD

A dissertation submitted to the University of Manchester
for the degree of Master of Science
in the Faculty of Engineering and Physical Sciences

2012

By
Aude Giraud
School of Computer Science

Contents

Word Count: 17123

List of Tables

List of Figures

Listings

Abstract

The Finite-Difference Time-Domain method (FDTD) is a modelling technique for electromagnetic waves propagation. There is a great range of domains of application, for example geophysics, defence, microwaves like radar, or biomedicine. However, FDTD is a computationally intensive method, but has potential for parallelisation.

The use of General-Purpose computing on Graphics Processing Units (GPGPU) is examined here to enhance the performance of the FDTD method. GPUs are good candidates thanks to their massively parallel architecture. We expose three principal methods to implement the FDTD method on GPUs. Two of them have been implemented with CUDA and executed on a Tesla architecture.

The results show that one approach take the most of the capability of GPU's parallelism. This approach is at least six times faster than the others in single precision, and at least three times faster in double precision. We searched the reason of such a difference. The best method presents a high capacity of latency hiding, coupled with a good occupancy of the multiprocessors.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

First, I would like to thank Dr Fumie Costen. She supervised me with attention and guided me through this project. Her warm support and encouragements have been of great help, and I enjoyed speaking about technical issues with her. Her enthusiasm was communicative. I am also grateful for her useful advices about academic career and life, she patiently listened to me and I cannot express my gratitude enough for her guidance.

I would like to thank the Kyushu University for access to their machine, which was necessary to run the experiments of this project.

I also want to thank Graham Riley and Professor John Gurd who passed their interest of Parallel Computing on me. Without them, I would not have felt confident to understand and to handle this project.

Finally, I would like to thank my parents who got me here. They permitted me to thrive and follow the studies I liked. I would like to thank my sisters who bring me joy every day. And I would like to thank Mikael for his endless support and kindness.

Chapter 1

Introduction

Some heart diseases and mental diseases (like Alzheimer) receive the benefit from the invasive stimulation of the electromagnetic field. Alternative to the invasive approach, there is a movement to develop non-invasive devices. Non-invasive devices, contrary to drugs, present the advantage to target a special area of the brain or the heart. Non-targeted areas are not affected by the treatment, while drugs can induce side effects. Non-invasive devices also permit to target areas that could not be safely treated with surgery.

Bio-electromagnetics are a medical area where electromagnetic fields stimulate certain human tissues. For example, a common use for bio-electromagnetic therapy is defibrillation. Defibrillation consists in chocking the heart with electromagnetic waves in case of fibrillation (uncoordinated movements of the heart, leading to cardiac arrest). Recent research has shown that electromagnetic stimulation could also help in Alzheimer therapy [2]. However, those experiments cannot be done directly on humans for ethical reasons. In [2] the tests are done on mice. It could be useful to have a human model for further investigation, hence the growing interest for computer sciences in this medical field.

In order to produce safe and efficient medical care devices, we need to understand and model the way they propagate into a human body. The body can be modelled by a grid of points in 3D space. We also need a very fine spatial resolution of the digital human phantom to obtain a highly accurate simulation results.

Beyond the medical field, the results of this project on the speed up of the FDTD calculation could also be applied in geophysics, military defence, and every other fields needing an accurate model for electromagnetic waves propagation.

1.1 Aim and objectives

The Finite Difference Time Domain (FDTD) method was first presented by Yee in 1966 [3]. It is a popular method in the field of electromagnetic studies because of its simplicity and stability. The main idea is to discretize Maxwell's equations on a continuous space into finite-difference equations [3]. The FDTD method is generally used over large amounts of data. The data are organized into matrices, and the same operations are repeated on every cells. This type of computation is commonly called Single-Instruction Multiple-Data (SIMD). Due to data independence between calculations on each cell, the FDTD method is suitable for data parallelism. Its robustness and simple implementation make it a perfect candidate for human modeling. Thus, the FDTD method will be the base of our study.

Graphic Processing Units (GPUs) have recently evolved from a hardware dedicated to computer graphics into general-purpose chips. The advance of General-Purpose computing on GPU (GPGPU) allows programmers to use the power of GPUs without needing to be an expert in computer graphics. Their high throughput capacity is useful for problems that need to deal with a large volume of data, as it is the case for human body, and more generally High Performance Computing (HPC). With the recent enthusiasm for video-games and the pushing need for an inexpensive equipment, the price of GPUs has decreased. Their economic asset coupled with their performance make them of growing interest for parallel programming.

NVIDIA has developed the Compute Unified Device Architecture (CUDA) to ease the work of programmers. CUDA describes both the hardware architecture and the language extension. However in this report we will use the word Tesla for the architecture and CUDA will refer to the language extension. Tesla architecture support GPGPU, and CUDA provides a C extension very close to the original C language. Thus, programmers need a minimal time to learn how to use it. However, CUDA can only be used on NVIDIA architectures. We use a Tesla T10 GPU in our research.

Three main approaches for the FDTD method implementation on GPUs emerge from current researches. They differ in the way they map the data to threads. A first method computes the data grid layer by layer. In each layer, each cell has its own allocated thread. A second approach processes the whole grid at once. One thread operates on every cells of the same column. A third

method also goes through the entire grid, but several threads work on the same column. Our aim is to perform and compare our method and the other approaches that implement FDTD method with GPGPU. It will be applied to choose which method is the best in order to compute the wave propagation.

Depending on the model of the problem, the FDTD method can be more or less complex. We limit our study to a simplified case. We stick to a cube of small dimension, up to $256 \times 256 \times 256$ cells, which may be not sufficient to a fine granularity in brain surgery for example, and restrict the target area to a cubic shape. A hard-source emits electric field at a single-point. The electro-dynamic waves propagate in vacuum while the body has different densities for muscles, bones, or flesh. The cube has a perfectly conducting surface.

Our first objective is to understand the current approach and implement the first and second methods. Their running times are recorded and compared to the existing method. Then the differences between the three implementations are explained. Finally, further improvements are proposed.

1.2 Dissertation overview

Chapter 2 studies the necessary background to our project, supported by literature references. First we introduce the Finite-Difference Time-Domain method and General-Purpose programming on Graphics Processing Units. Then we provide informations about Tesla architecture and we detail CUDA programming. Finally the three approaches we explore in this dissertation are presented.

Chapter 3 describes the implementations of the three approaches and their codes. Further details are given on specific implementation choices, considering their impact on performance.

Chapter 4 exposes the execution times we got from the three methods. First we focus on the two methods implemented this year. Then we compare the three approaches and discuss the sources of the differences in timing results. When we get some unexpected results, we search for explanations.

Chapter 5 concludes on the deliverables and findings of this project. We discuss further possible developments of our work.

Chapter 2

Background and literature survey

2.1 The Finite-Difference Time-Domain method

Maxwell's equations permit to simulate the electromagnetic waves propagation into the body [4] [3]. We focus on Equations (2.1) and (2.2), which are Faraday's law and Ampère's law respectively, in an isotropic and linear medium .

$$\nabla \times \boldsymbol{E} = -\frac{\partial \boldsymbol{B}}{\partial t} \tag{2.1}$$

$$\nabla \times \boldsymbol{H} = \frac{\partial \boldsymbol{D}}{\partial t} + \boldsymbol{J} \tag{2.2}$$

$\boldsymbol{E}$ is the electric field, $\boldsymbol{H}$ is the magnetic field, $\boldsymbol{B}$ is the magnetic flux density, $\boldsymbol{D}$ is the electric flux density and $\boldsymbol{J}$ is the conduction current density.

We consider that we are in a source-free space ($\boldsymbol{J} = 0$), so according to Yee, Equations (2.3) and (2.4) apply [3].

$$\boldsymbol{D} = \epsilon \boldsymbol{E} \tag{2.3}$$

$$\boldsymbol{B} = -\mu \boldsymbol{H} \tag{2.4}$$

ϵ is the permittivity and μ is the permeability of the space.

With Equations (2.3) and (2.4), Equations (2.1) and (2.2) can be rewritten as Equations (2.5) and (2.6).

$$\nabla \times \boldsymbol{E} = -\mu \frac{\partial \boldsymbol{H}}{\partial t} \tag{2.5}$$

$$\nabla \times \boldsymbol{H} = \epsilon \frac{\partial \boldsymbol{E}}{\partial t} \tag{2.6}$$

Equation (2.5) is partially differentiated into Equations (2.7) to (2.9) and Equation (2.6) can be expressed by the three scalar Equations (2.10) to (2.12) [3] [5] [6].

$$\frac{\partial H_x}{\partial t} = \frac{1}{\mu}\left(\frac{\partial E_y}{\partial z} - \frac{\partial E_z}{\partial y}\right) \tag{2.7}$$

$$\frac{\partial H_y}{\partial t} = \frac{1}{\mu}\left(\frac{\partial E_z}{\partial x} - \frac{\partial E_x}{\partial z}\right) \tag{2.8}$$

$$\frac{\partial H_z}{\partial t} = \frac{1}{\mu}\left(\frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x}\right) \tag{2.9}$$

$$\frac{\partial E_x}{\partial t} = \frac{1}{\epsilon}\left(\frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z}\right) \tag{2.10}$$

$$\frac{\partial E_y}{\partial t} = \frac{1}{\epsilon}\left(\frac{\partial H_x}{\partial z} - \frac{\partial H_z}{\partial x}\right) \tag{2.11}$$

$$\frac{\partial E_z}{\partial t} = \frac{1}{\epsilon}\left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y}\right) \tag{2.12}$$

Yee presented the Finite-Difference Time-Domain (FDTD) method in 1966 [3]. The FDTD method is a time domain Maxwell equation solver that computes the frequency response of a transient signal in a medium in one computation [6]. It approximates accurately Maxwell's equations. The principle of the FDTD method is to spatially discretize Equations (2.7) to (2.12) [3].

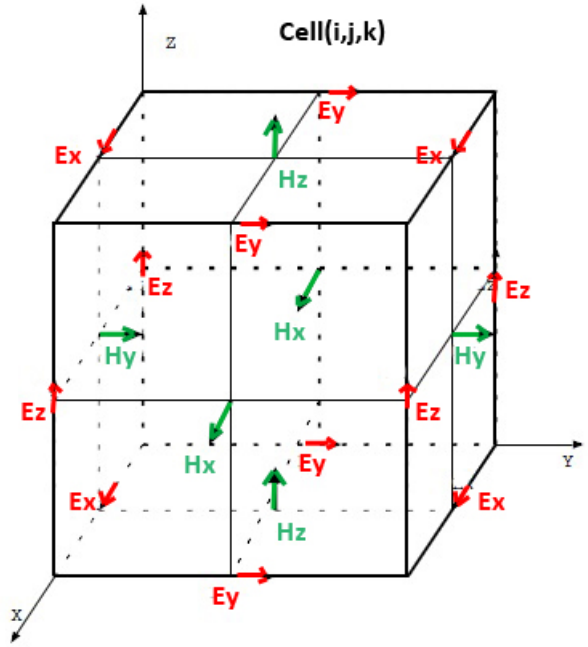

Figure 2.1: Spatial representation of the electromagnetic fields in the FDTD method.

In a discretized model, the space is not continuous anymore. It is represented by a grid of points. Figure 2.1 shows Yee's representation of the electromagnetic fields in a three-dimensional space [3]. Figure 2.1 represents one point in the discretized space. The magnetic field $\boldsymbol{H}$ is perpendicular to the surface of the grid and the electric field $\boldsymbol{E}$ is parallel. They follow one another alternatively, at a step of half a unit in space. Because of this interleaving, $\boldsymbol{H}$ and $\boldsymbol{E}$ are co-dependent [3]. For example, H_z needs E_x and E_y to be computed (this is verified in Equation (2.9)). Yee proposes an algorithm in [3]: first $\boldsymbol{E}$ is calculated with the old values of $\boldsymbol{H}$, and then $\boldsymbol{H}$ is updated with the new values of $\boldsymbol{E}$. This is repeated over several time-steps [4]. A time-step corresponds to the computation of $\boldsymbol{E}$ and $\boldsymbol{H}$ on every points of the three-dimensional grid.

$$H_x^{n+\frac{1}{2}}{}_{(i,j+\frac{1}{2},k+\frac{1}{2})} = H_x^{n-\frac{1}{2}}{}_{(i,j+\frac{1}{2},k+\frac{1}{2})} \tag{2.13}$$
$$+\frac{\Delta t}{\mu(i,j+\frac{1}{2},k+\frac{1}{2})\Delta z}\left[E_y^n{}_{(i,j+\frac{1}{2},k+1)} - E_y^n{}_{(i,j+\frac{1}{2},k)}\right]$$
$$+\frac{\Delta t}{\mu(i,j+\frac{1}{2},k+\frac{1}{2})\Delta y}\left[E_z^n{}_{(i,j,k+\frac{1}{2})} - E_z^n{}_{(i,j+1,k+\frac{1}{2})}\right]$$

$$H_y^{n+\frac{1}{2}}{}_{(i+\frac{1}{2},j,k+\frac{1}{2})} = H_y^{n-\frac{1}{2}}{}_{(i+\frac{1}{2},j,k+\frac{1}{2})} \tag{2.14}$$
$$+\frac{\Delta t}{\mu(i+\frac{1}{2},j,k+\frac{1}{2})\Delta x}\left[E_z^n{}_{(i+1,j,k+\frac{1}{2})} - E_z^n{}_{(i,j,k+\frac{1}{2})}\right]$$
$$+\frac{\Delta t}{\mu(i+\frac{1}{2},j,k+\frac{1}{2})\Delta z}\left[E_x^n{}_{(i+\frac{1}{2},j,k)} - E_x^n{}_{(i+\frac{1}{2},j,k+1)}\right]$$

$$H_z^{n+\frac{1}{2}}{}_{(i+\frac{1}{2},j+\frac{1}{2},k)} = H_z^{n-\frac{1}{2}}{}_{(i+\frac{1}{2},j+\frac{1}{2},k)} \tag{2.15}$$
$$+\frac{\Delta t}{\mu(i+\frac{1}{2},j+\frac{1}{2},k)\Delta y}\left[E_x^n{}_{(i+\frac{1}{2},j+1,k)} - E_x^n{}_{(i+\frac{1}{2},j,k)}\right]$$
$$+\frac{\Delta t}{\mu(i+\frac{1}{2},j+\frac{1}{2},k)\Delta x}\left[E_y^n{}_{(i,j+\frac{1}{2},k)} - E_y^n{}_{(i+1,j+\frac{1}{2},k)}\right]$$

$$E_x^{n+1}{}_{(i+\frac{1}{2},j,k)} = E_x^n{}_{(i+\frac{1}{2},j,k)} \tag{2.16}$$
$$+\frac{\Delta t}{\epsilon(i+\frac{1}{2},j,k)\Delta y}\left[H_z^{n+\frac{1}{2}}{}_{(i+\frac{1}{2},j+\frac{1}{2},k)} - H_z^{n+\frac{1}{2}}{}_{(i+\frac{1}{2},j-\frac{1}{2},k)}\right]$$
$$+\frac{\Delta t}{\epsilon(i+\frac{1}{2},j,k)\Delta z}\left[H_y^{n+\frac{1}{2}}{}_{(i+\frac{1}{2},j,k-\frac{1}{2})} - H_y^{n+\frac{1}{2}}{}_{(i+\frac{1}{2},j,k+\frac{1}{2})}\right]$$

$$E_y^{n+1}{}_{(i,j+\frac{1}{2},k)} = E_y^n{}_{(i,j+\frac{1}{2},k)} \tag{2.17}$$

$$+\frac{\Delta t}{\epsilon(i,j+\frac{1}{2},k)\Delta z}\left[H_x^{n+\frac{1}{2}}{}_{(i,j+\frac{1}{2},k+\frac{1}{2})} - H_x^{n+\frac{1}{2}}{}_{(i,j+\frac{1}{2},k-\frac{1}{2})}\right]$$
$$+\frac{\Delta t}{\epsilon(i,j+\frac{1}{2},k)\Delta x}\left[H_z^{n+\frac{1}{2}}{}_{(i-\frac{1}{2},j+\frac{1}{2},k)} - H_z^{n+\frac{1}{2}}{}_{(i+\frac{1}{2},j+\frac{1}{2},k)}\right]$$

$$E_z^{n+1}{}_{(i,j,k+\frac{1}{2})} = E_z^n{}_{(i,j,k+\frac{1}{2})} \tag{2.18}$$
$$+\frac{\Delta t}{\epsilon(i,j,k+\frac{1}{2})\Delta x}\left[H_y^{n+\frac{1}{2}}{}_{(i+\frac{1}{2},j,k+\frac{1}{2})} - H_y^{n+\frac{1}{2}}{}_{(i-\frac{1}{2},j,k+\frac{1}{2})}\right]$$
$$+\frac{\Delta t}{\epsilon(i,j,k+\frac{1}{2})\Delta y}\left[H_x^{n+\frac{1}{2}}{}_{(i,j-\frac{1}{2},k+\frac{1}{2})} - H_x^{n+\frac{1}{2}}{}_{(i,j+\frac{1}{2},k+\frac{1}{2})}\right]$$

Equations (2.13) to (2.18) present Yee FDTD equations [3]. They are obtained by discretizing (2.7),(2.8),(2.9), (2.10),(2.11) and (2.12), where Δt is temporal discretization and Δx, Δy and Δz are spatial discretization in x, y and z directions [5]. Note that in our case, $\Delta x = \Delta y = \Delta z$.

The electric and magnetic components are evaluated at half a space-step, so Equations (2.13) to (2.12) use non-integer coordinates. This is inconvenient for computation. In 2005, Costen presented a way to implement the finite-difference approximation on (2.7) to (2.12) in the described discretized space, using integer coordinates as seen in Equations (2.19) to (2.24) [6].

$$H_x^{n+1}{}_{(i,j,k)} = H_x^n{}_{(i,j,k)} + \frac{\Delta t}{\mu(i,j,k)\Delta z}\left[E_y^n{}_{(i,j,k)} - E_y^n{}_{(i,j,k-1)}\right] \tag{2.19}$$
$$-\frac{\Delta t}{\mu(i,j,k)\Delta y}\left[E_z^n{}_{(i,j,k)} - E_z^n{}_{(i,j-1,k)}\right]$$

$$H_y^{n+1}{}_{(i,j,k)} = H_y^n{}_{(i,j,k)} + \frac{\Delta t}{\mu(i,j,k)\Delta x}\left[E_z^n{}_{(i,j,k)} - E_z^n{}_{(i-1,j,k)}\right] \tag{2.20}$$
$$-\frac{\Delta t}{\mu(i,j,k)\Delta z}\left[E_x^n{}_{(i,j,k)} - E_x^n{}_{(i,j,k-1)}\right]$$

$$H_z^{n+1}{}_{(i,j,k)} = H_z^n{}_{(i,j,k)} + \frac{\Delta t}{\mu(i,j,k)\Delta y}\left[E_x^n{}_{(i,j,k)} - E_x^n{}_{(i,j-1,k)}\right] - \frac{\Delta t}{\mu(i,j,k)\Delta x}\left[E_y^n{}_{(i,j,k)} - E_y^n{}_{(i-1,j,k)}\right] \tag{2.21}$$

$$E_x^{n+1}{}_{(i,j,k)} = E_x^n{}_{(i,j,k)} + \frac{\Delta t}{\epsilon(i,j,k)\Delta y}\left[H_z^n{}_{(i,j+1,k)} - H_z^n{}_{(i,j,k)}\right] - \frac{\Delta t}{\epsilon(i,j,k)\Delta z}\left[H_y^n{}_{(i,j,k+1)} - H_y^n{}_{(i,j,k)}\right] \tag{2.22}$$

$$E_y^{n+1}{}_{(i,j,k)} = E_y^n{}_{(i,j,k)} + \frac{\Delta t}{\epsilon(i,j,k)\Delta z}\left[H_x^n{}_{(i,j,k+1)} - H_x^n{}_{(i,j,k)}\right] - \frac{\Delta t}{\epsilon(i,j,k)\Delta x}\left[H_z^n{}_{(i+1,j,k)} - H_z^n{}_{(i,j,k)}\right] \tag{2.23}$$

$$E_z^{n+1}{}_{(i,j,k)} = E_z^n{}_{(i,j,k)} + \frac{\Delta t}{\epsilon(i,j,k)\Delta x}\left[H_y^n{}_{(i+1,j,k)} - H_y^n{}_{(i,j,k)}\right] - \frac{\Delta t}{\epsilon(i,j,k)\Delta y}\left[H_x^n{}_{(i,j+1,k)} - H_x^n{}_{(i,j,k)}\right] \tag{2.24}$$

The temporal discretization Δt and the spatial discretization $\Delta x = \Delta y = \Delta z = \Delta s$ are chosen such as the Courant Friedrichs Lewy (CFL) condition is satisfied [6]. The CFL condition ensures the validity and stability of the FDTD method. Equation (2.25) relates the temporal and spatial steps, where v is the highest propagation speed of the signal in the medium [6] [5]. $v\Delta t$ is upper-bounded. This means that if the spatial step (Δs) gets smaller, then the covered distance ($v\Delta t$) has to be even smaller.

$$\Delta t \leq \frac{1}{c}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\right)^{-\frac{1}{2}} \tag{2.25}$$

Figure 2.2 illustrates the CFL condition. The large circles represent the wave-front. The small circles represent the points in the grid. Points are at a distance of Δs from each other, that is a spatial step. The lines of the table represent the wave propagation during one time-step ($(n+1)\Delta t - n\Delta t$) of two cases (a)

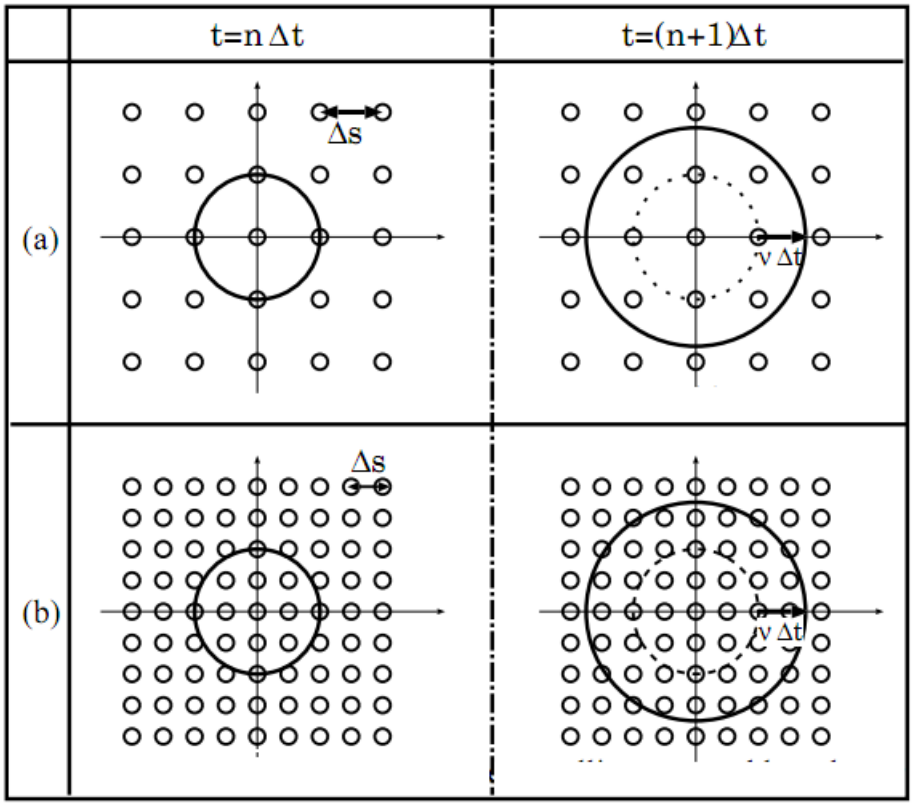

Figure 2.2: Representation of the Courant Friedrichs Lewy condition [6].

and (b). Case (a) of Figure 2.2 is stable because the covered distance during one time-step is smaller than the spatial sampling. Case (b) shows a situation where the CFL condition is not respected, and $v\Delta t$ exceeds the spatial-step Δs. The FDTD calculation has a slower progress than the wave propagation, which leads to instability [6].

The calculation of $\boldsymbol{H}$ needs the values of $\boldsymbol{E}$ in the surrounding cells, and vice-versa. Computers are not actually capable of real infinite calculation, so we have to limit the model to the size of our problem (for instance, a human body). The finite nature of FDTD raises the question of the border calculation. On the cube sides, the cells do not have the necessary neighbours to calculate the equations. Taflove in [7] present the integration of a padding border in the three-dimensional grid model. This is called an "Absorbing Boundary Condition" (ABC). This boundary simulates the infinity space and suppresses the reflection of outgoing waves. It ensures that the FDTD method will give accurate results [4]. Usually, there is a trade-off between the data storage of the boundary, its accuracy

and its stability [6]. Different ABCs exist. The Perfectly Matched Layer (PML) absorbing boundary condition gives a high accuracy, but needs between six and ten layers to wrap the grid [6] [8]. Those layers provide a good reduction of the reflection of outgoing waves, at the cost of memory usage. When the hardware is limited, PML dramatically reduces the useful work in favour of the boundary, and becomes unsuitable [6] [9]. Mur's ABC is less accurate than the PML but requires less memory and is simpler [4]. At the first order, Mur's ABC can adapt to different models but the boundary does not absorb the reflection of outgoing waves very well [6] [8]. At the second order, the absorption is better but the method loses its versatility [6] [10]. Liao's ABC is a non-material ABC, which means that it does not depend on the media parameters as μ and ϵ [6] [5]. Liao's ABC has high accuracy and small data usage, but it is unstable [6].

In our model, we use a Perfect Electric Conductor (PEC) condition. It is like wrapping the grid into metal walls. It means we force the electric and magnetic fields to be equal to zero at the border. It is simple to implement and the boundary has just one layer so the memory usage is limited. On the other hand, it is not realistic (a brain is not in a metal box). Equation (2.26) presents the PEC condition [10] [11].

$$\overrightarrow{n} \times \overrightarrow{E}|_{interface} = 0 \quad ; \quad \overrightarrow{n} \bullet \overrightarrow{H}|_{interface} = 0 \tag{2.26}$$

The difference between an ABC boundary condition and a PEC boundary condition is that the ABC estimates the missing components outside the FDTD grid, to mimic the infinite space [8]. The order of the ABC (first order, second order), determines how far the fields are estimated beyond the border of the grid (one step, several steps) [8]. In the other hand, PEC condition does not simulate the infinite space, it stops the propagation.

Even if the FDTD method is simple to implement, it has several drawbacks. First, memory requirements increase when the problem gets bigger because the values of the electric and magnetic fields need to be stored, as well as the different parameters such as permeability or permittivity [5]. The FDTD method applied on a human body can be computationally intensive according to the granularity of the model. Moreover, in our case we use a cubic model. An approximation of curves can be made by staircasing but it is inaccurate [5].

In practice, different μ and ϵ can be defined at each grid points to simulate

the change in the media (for example the difference between the brain and the cranium). There is a trade-off between the number of points, with different parameters maybe, and the efficiency and memory space.

The FDTD method presents high data parallelism, and the linear equations (2.7) to (2.12) are independent. Those two caracteristics are common with computer graphics [12]. That makes GPUs particularly suitable for the FDTD computation.

2.2 General Purpose programming on Graphics Processing Units

Graphics Processing Units (GPUs) were firstly designed for graphics rendering. The graphic pipeline is characterized by high parallelism, high throughput, and small to nonexistent data reuse [12] [13]. GPUs' hardware targets those specifications. On the other hand, CPU programs need more transistors to control hardware, and thus a substantial space on the chip is dedicated to control functionalities such as branch prediction [12]. Several levels of cache are implemented on CPUs. However, those functionalities are of little use when data is used only once, as it is often the case in computer graphics [12].

CPUs are by nature general-purposed, while the primary use of GPUs is specific to computer graphics, giving a more specialized hardware. According to [12], "CPU memory systems are optimized for minimum latency rather than the maximum throughput targeted by GPU memory systems". CPUs focus on one task to do as fast as possible, in order to make progress, while GPUs favour the number of tasks, to the detriment of latency.

Under the increasing demand of performance from the video-games industry, GPUs have known a fast development. Not only their performance increased, but they also got cheaper. They became interesting for more general-purposed problems. But due to their graphic nature, researchers had to "translate" their problem into a computer graphics one, using pixel colour for example [14]. It hindered the development of programming on GPUs, because they had to learn specific graphics API like OpenGL to code on GPUs, and such APIs were restricting the potential for general-purpose programming [15]. Using graphics API to compute general purpose programs was the beginning of GPGPU development.

NVIDIA introduced the G80 architecture and the Compute Unified Device Architecture (CUDA) in 2006 [15]. This was a turning point in GPGPU evolution. CUDA is an extension of the programming languages C/C++, and has a similar syntax. Thus it is easy to learn for C programmers. G80 was not only supporting C, but it also introduced a shared memory for threads within the same block, and the single-instruction multiple-thread (SIMT) execution model [15]. Thereby GPGPU became more popular because programmers could benefit from a massively parallel processor without the drawbacks inherent in graphics computing.

NVIDIA has improved its architecture by adding more streaming processors, allocating more space for registers, and increasing the number of threads that could execute at the same time [15]. GPU were lacking for double precision support, a major need for scientific calculation, so NVIDIA implemented it on G80 GPUs, even though it was limited [15]. In 2010, NVIDIA has released its new range of GPU architecture, named Fermi. According to [15], Fermi improves the double precision performance and allocates more space for shared memory. What is interesting is that Fermi introduces a cache hierarchy in its GPU, which was a restrictive element on CPUs because of the control functionalities.

The efficiency and inexpensive price of GPUs compared to CPUs architecture make them an excellent choice for high-performance computing. [16] experiments a GPGPU implementation of the FDTD method on different GPU chips. It shows that even a desktop GPU as the NVIDIA Gt130m gets a better time than the CPU Intel Core2 T6500 for a cheaper price. [17] and [18] present the limitations of CPU computation for the FDTD method : it is slow and the need of scientific supercomputer makes it expensive. The use of GPU chips is promising and give a great improvement in execution time. They use OpenGL, and we expect that programming with CUDA on NVIDIA Tesla chips will give even better results because the code is specifically designed for the owner hardware.

2.3 Tesla architecture and CUDA

Figure 2.3 presents NVIDIA Tesla architecture. Table 2.1 gives the specifications of the GPU used for our experiments. The machine used for the project is located in Kyushu University, Japan. It has two Tesla T10 GPUs. In this project, we use only one GPU.

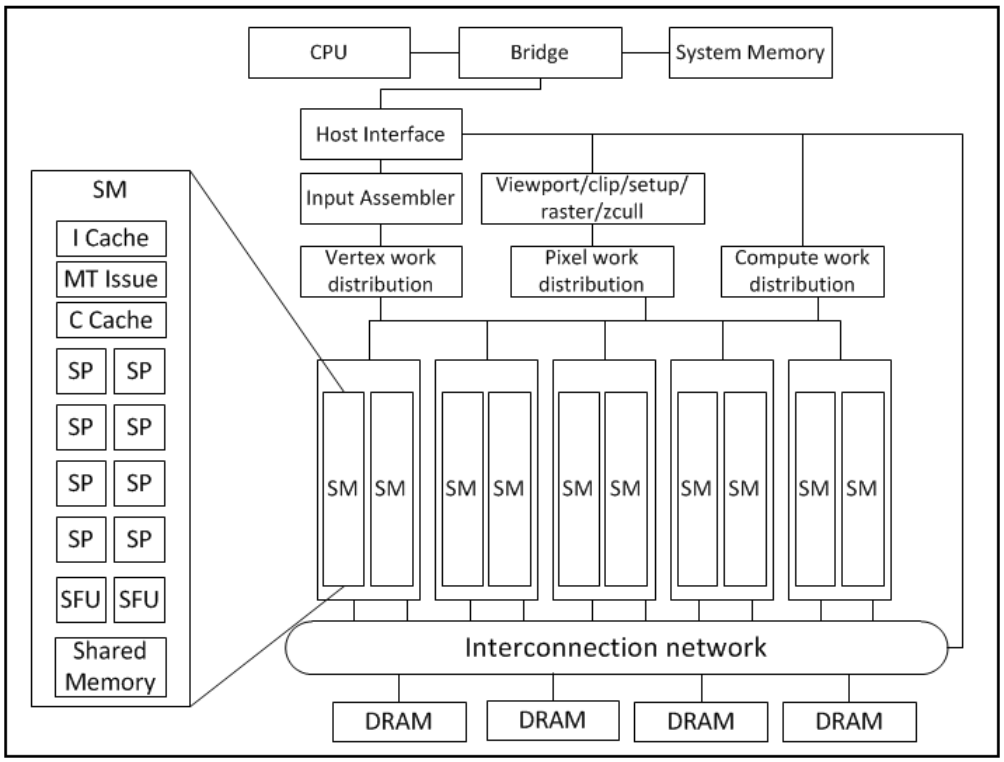

Figure 2.3: The NVIDIA Tesla architecture [13], [1].

Tesla T10 contains 30 streaming multiprocessors (SM). The host passes the instructions on the device. We see in Figure 2.3 that vertex and pixel work distributions, which are of the computer graphics domain, and compute work distribution, have access to the same SMs. This shows that streaming multiprocessor can execute graphic computing programs but also parallel computing programs, which proves that NVIDIA was already thinking about a GPGPU application of its architecture [13]. One streaming multiprocessor contains eight streaming processor (SP) cores, which brings to 240 the total number of SP cores in Tesla T10 GPU. SMs allocate threads to SP cores. The latter can run simple operations such as multiply and add [13]. Each SM has also two special-function units (SFUs). SFUs perform transcendental functions such as exponential function, logarithm, trigonometric functions, and planar attribute interpolation [13]. An interpolation gives the value of a pixel's attribute (color, depth, etc.) based on discrete values of other pixels [13]. The multi-threaded instruction fetch and issue unit (MT Issue) creates the threads, manages the tasks and assigns the SPs that will execute the code [13]. SMs also have an instruction cache (I Cache) and

a read-only constant cache (C Cache) [13]. The shared memory allows collaboration between threads.

As specified in Table 2.1, NVIDIA Tesla T10 can have up to 8 active blocks on each SM, and no more than 1024 active threads. Threads of the same blocks are grouped into *warps*, and executed on SP cores [13]. The partition is made according to threads' IDs [19]. In one dimension, threads are grouped according to their x indices in increasing order. If threads are distributed in two or three dimensions, they are first projected onto a one dimensional array [19]. For example, in two dimensions threads are first ordered according to their y indices, in increasing order: threads with coordinates (x,0) are placed before threads with coordinates (x,1) and so on. Then they are ordered according to x, for instance thread (0,0) is placed before thread (1,0).

Threads in a blocks are grouped into warps to reduce the number of global memory accesses, and thus improve performance [13]. On Tesla T10, warps are made of 32 threads, this means that up to $1024/32 = 32$ warps can reside on a SM at the same time. The instructions within a warp have to be of the same nature, thus they can execute in a Single-Instruction Multiple-Threads (SIMT) manner [20]. Warps are an important mechanism of CUDA programming, they permit to hide latency [19]. When a warp is waiting for results of a high-latency operation (such as accessing global memory), an other warp is executed. If there is enough warps, theoretically there will always be a warp running [19]. Kirk and Hwu refer to it as zero-overhead thread scheduling [19]. The hardware hides the waiting time with the work of other threads' warps.

Occupancy is a metric that measures the capacity of latency hiding of a kernel. Equation 2.27 gives an informal formula of occupancy [20] [21].

$$occupancy = \frac{blocks\ per\ SM \times threads\ per\ block}{maximum\ threads\ per\ SM} \tag{2.27}$$

NVIDIA provides an occupancy calculator to track the occupancy of a program. It shows the trade-off between the number of warps to keep the GPU busy, the number of registers and the use of shared memory. A low occupancy prevents from good performances, and a high occupancy permits a good latency hiding.

SMs have a SIMT execution model [19] [13]. Threads in the same warp execute the same instruction before moving to the next one. SIMT and SIMD are similar

Number of SM	30
Number of SP core	240 (8 per SM)
Clock speed (per core)	1.3 GHz
Single precision performance	933 GFlops
Double precision performance	78 GFlops
Size of a warp	32
Max. number of active blocks per SM	8
Max. number of threads per block	512
Max. number of active warps per SM	32
Max. number of active threads per SM	1024
Registers per SM	16384
Local memory per thread	16KB
Shared memory per SM	16 KB
Constant memory size	64 KB
Global memory size	4 GB
Level 2 cache size	None
Global memory bandwidth	102 GB/s
Compute capacity	1.3

Table 2.1: Tesla T10 specifications.

in the way that they both execute the same instruction on multiple execution units. However, SIMT can be more flexible than SIMD. The main difference is that SIMT executes the same instruction on different threads, whilst SIMD does it on different data lanes [13]. The programmer codes at a thread-level: he writes the behaviour of one thread that will be reproduced by the others [13]. In this way, the programmer does not need to care about hardware capacity such as multiprocessors, or even warps. It will not impact the correctness of the program, but a good design can boost the performance [13]. Lindholm et al. compare it to the role of the cache line: one can neglect the cache line to make a correct code, but a smart use of cache line can greatly improve the performance [13]. Yossi Kreinin shows that SIMD lacks three features that are available in SIMT [22].

1. **Single instruction, multiple registers sets** A kernel launches several threads to execute its instructions. However, it is not technically the same data, as it is stored in different registers. It means that lots of data are duplicated. Each threads have its own version of the variables, even if they have the same values. They could have been stored only once in SIMD [22]. This redundancy leads to a waste of registers [22]. Moreover, SIMD uses a "short vector spelling", which consists in breaking data in short vectors and process them with a *for* loop [22]. On the other hand, SIMT uses a "scalar spelling", no need to use a loop, threads automatically run the same instructions in parallel [22].

2. **Single instruction, multiple addresses** SIMT is capable of data random access. According to [22], the main benefit is that it permits to implement parallelism for some programs that could not be parallelized in the SIMD scheme. Even if SIMD is capable of some random access such as permutations or shuffling, it does so at a register scale [22]. Random access is never efficient when it is done on global memory. It comes from the inner nature of random access: because of this randomness, threads cannot be coalesced which reduces even more the performance when accessing a high-latency memory [22].

3. **Single instruction, multiple flow paths** As detailed later, threads can take different code path, when they encounter an *if* statement. It can be used to suppress updates for example [22]. Based on the threads' indices, the program enters the if statement and updates the value of a variable, or

just waits. The cost of this feature is that only one path is executed at a time, and threads not running are inactive [22].

It is recommended to have a number of threads that is a multiple of 32, the size of a warp. If it is not, the processor utilization is inefficient. Appendix B explores the impact of threads distribution when their number is and is not a multiple of 32. When the number of threads is smaller than 32, the missing threads are added. For instance, if 16 threads are running, 16 other threads are added to the warp, even if they do nothing. Threads inside a warp can take different code paths [13] [19]. If they take the same path, they execute together and there is no penalty [20]. If they do not, we speak of warp divergence [19] [23]. When threads in the same warp take different paths, it reduces the performance. The warp will serially execute each branch, and the threads that do not take a branch are disabled during its execution [13]. For example, in an *if-then-else* statement, threads that take the *then* path are first executed, and the threads that take the *else* path wait. Then the threads of the *else* path are executed, and the *then* threads are inactive. Assuming that both branches take the same amount of clock cycles, the warp takes the double of clock cycles than required. The total cost of a warp execution is the sum of the cost of each branch. When all the branches are executed, all the threads re-converge to the original path [13]. Note that if different warps diverge, there is no penalty. For example one warp can take the *then* path and another warp can take the *else* path without any consequence, as long as all threads within a warp execute together [23].

Figure 2.4 shows the different types of memory that are available in the Tesla architecture.

Global memory Each grid owns a global memory that lasts until the end of the application [19]. Both the host and the device can read and write in global memory. At the beginning of the application, the data is transfered from the host memory to the device memory. At the end, results are transfered back from global memory to host memory, and the allocated space is liberated. All threads in all blocks of the same grid have access to global memory [19]. Global memory is an off-chip memory [19], thus extensive accesses can reduce the performance.

Constant memory Like global memory, constant memory is a per-grid memory,

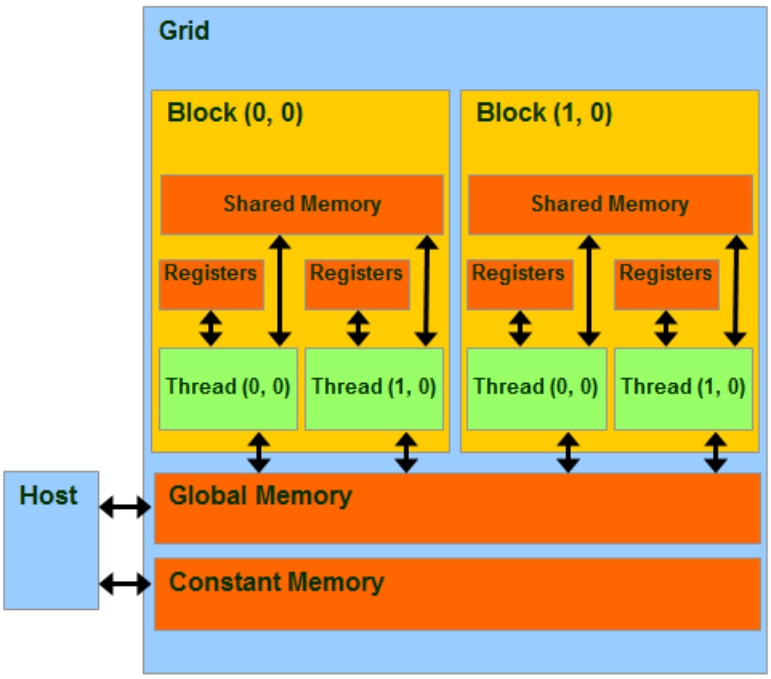

Figure 2.4: CUDA device memory model [19].

and its lifetime is the application [19]. All threads in all blocks see the same version of a constant variable. The host can read and write in constant memory, but this is a read-only memory for the device [19]. With the right access pattern, communication with constant memory can be very fast, but a bad design can lead to congestion [19]. The lifetime of constant memory is the application, so it does not disappear when a kernel finishes, and in fact it is accessible by all the kernels, until the end of the program [19]. Constant variables cannot be declared inside a kernel.

Shared memory The scope of shared memory is a block [19]. All threads within the same block have access to its shared memory, and see the same version of shared variables, but a block cannot read or write in the shared memory of another block. Shared memory is an efficient way to share data between threads, because it is an on-chip memory, much faster to access than global memory [19]. Thus, it is often beneficial to load smaller portions of the data in the shared memory before using it, if data is heavily reused by the threads in the block [19].

For example, De Donno et al. show that the use of the shared memory in a two-dimensional FDTD problem permits to gain more than 2500ms on

the GPU time of the electric kernel, and more than 4000ms on the CPU time, that is more than 6500ms in total [24]. De Donno et al. also observe that using shared memory minimizes the number of uncoalesced accesses [24]. Uncoalesced accesses appear when several threads access unaligned addresses. Multiple transactions are needed, whereas a good alignment permits to get a common bundle, and thus boosts the performance [24].

Shared memory can be a limiting factor if its capacity is not considered. The Tesla T10 GPU allocates 16KB of shared memory per SM, and a SM can host 8 blocks at most. If 8 blocks are assigned to one SM, they cannot use more than $16KB \div 8 = 2KB$ of shared memory. If blocks need more than 2KB of memory each, the number of active blocks in the SM is reduced to respect the capacity [19]. For instance, if a block uses 3KB, a maximum of five blocks can run.

Register Per thread registers are on-chip memory, which allows very fast access [19]. Each thread can write and read on its own registers, and registers are not accessible from the host. When a variable is declared in the kernel body, if it is not an array, then it is stored in a register. Kirk and Hwu call this kind of variable a *scalar variable* [19]. Scalar variables are private, so each thread will create its own versions of the variables. For example, if we have four blocks with 512 threads each, there will be $4 \times 512 = 2048$ copies of one automatic variable (automatic meaning the variable is a scalar or an array). The scope of a scalar variable is the thread, which means that once the thread ends, the variable ceases to exist [19]. As for shared memory, the lifetime of a scalar variable is the kernel [19]. If a kernel is launched multiple times, the values of the variable are not preserved through the iterations [19].

The programmer must pay attention to the number of registers allocated to a thread, at the risk of loosing performance. A SM cannot have more than 16384 registers. The maximum number of threads per SM is 1024. At maximum capacity, each thread has $16384 \div 1024 = 16$ registers. If we have two blocks with 512 threads each, the capacities in term of maximum number of blocks per SM and of threads per blocks are respected. Supposing that each thread needs 16 registers, we stay under the limit of the maximum number of threads per SM. Now, let us presume that, after a modification in the code, each thread needs 17 registers. The program needs $17 \times (512 \times$

2) = 17408 threads, which is beyond the limit. The number of threads has to be reduced. According to Kirk and Hwu, this reduction is done at a "block granularity" [19]. This means in our problem that when the limit is exceeded, we lose 512 threads at a time, because blocks have 512 threads each; if blocks had 256 threads each, we would lose 256 threads at a time, until the limitations were respected. We went from 1024 threads executing in parallel to 512 threads, half of the original capacity has been lost.

Sometimes, using more registers can have a better pay off than using more blocks, because of the fast memory access. But the programmer needs to carefully consider the limited capacities of the device and their impact.

Local memory Local memory is usually used to store private array variables [19]. It is a space dedicated to a thread, but stored in global memory. Local memory is private to one thread, and like registers there are as many copies of a same variable as the number of threads [19]. The scope and the lifetime of local memory are identical to the register ones [19]. The access speed is low because local memory is in global memory, an off-chip memory.

The programmer can indicate the type of memory used in the code. For a constant variable, use the prefix '__constant__'; for a variable in shared memory, use '__shared__' [25]. '__device__' can be added before any variable to put it in global memory, or before '__shared__' and '__constant__' without changing the aforementioned effects. A variable, scalar or array, declared inside the kernel body, is automatically stored in a register or local memory respectively [19].

Global memory access is generally the performance bottleneck, besides the resources' use. The Compute to Global Memory Access (CGMA) ratio compares the number of floating-point calculations (what GPUs are good at) to the number of global memory accesses (what limits GPUs' capacity) [19]. For example, Tesla T10 has a single precision performance capacity of 933 Gigaflops, as seen in Table 2.1. However, the global memory bandwidth is equal to 102 gigabytes per second (GB/s). This limits the throughput to $102 \div 4 = 25.5$ single-precision data per second, which is far from the theoretical 933 GFlops. If a kernel has a CGMA ratio of 1.0, then it will not execute faster than 25.5 GFlops. To improve the performance, the programmer has to increase the CGMA ratio. To do so, he can use registers or shared memory instead of global memory for example, as they

have a faster access. The programmer can also increase the number of arithmetic operations done for one global memory access.

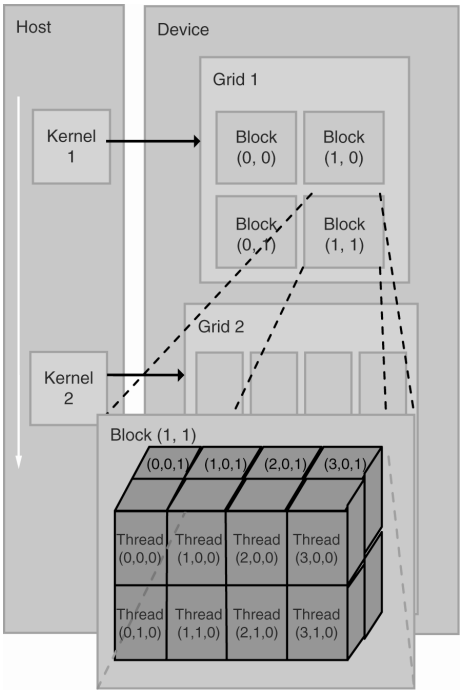

Figure 2.5: CUDA grid organisation [19].

The host executes the sequential code and calls the functions that will run on the device. Those functions are named *kernel* [19] [25]. As presented in Figure 2.5, the kernel is executed by a grid [25]. Different kernels have different grids and cannot be executed in parallel on the same GPU. A grid contain blocks, distributed in one or two dimensions [19]. For example, Grid 1 has 2×2 blocks in Figure 2.5. Note that Grid 2 has a different configuration. Grids do not need to be the same. The dimension of grids is given by *gridDim.x* and *gridDim.y* in CUDA. Blocks themselves contain threads in one, two or three dimensions [19]. *blockDim.x*, *blockDim.y* and *blockDim.z* provide those dimensions. Figure 2.5 illustrates the division of threads. In compute capability 1.3, a block cannot have a thread dimension exceeding 512 on x and y axis, and 64 on z axis. Blocks in the same grid have the same threads distribution. For example, all blocks in Grid 1 of Figure 2.5 have $4 \times 2 \times 2$ threads. Threads' IDs and blocks' IDs are stored in built-in variables that can be accessed with x,y,z, and w fields. The identity of a block is given by *blockIdx.x* and *blockIdx.y* [19]. The identity of a thread within a block is

given by *threadIdx.x*, *threadIdx.y* and *threadIdx.z* [19]. So, the unique identity of a thread in a grid can be found with $x = blockIdx.x \times blockDim.x + threadIdx.x$, and in the same way for y and z.

The grid and block dimensions are specified in the kernel invocation. A kernel call has the form $kernel <<< gridDim, blockDim >>> (arguments)$. It invokes kernel on a grid of gridDim blocks, made of blocks of blockDim threads, and arguments are the usual arguments of a C function. Threads within the same block can share data, be executed in parallel or synchronize. On the other hand, we cannot predict in which order blocks will be executed, so synchronization between blocks is prohibited. Because blocks can be executed in any relative order, this scheduling is very flexible. Kirk and Hwu call it transparent scalability in [19]: the program adapts to the resources, which makes it scalable to various NVIDIA chips. For example in Figure 2.6, the same kernel is executed on two different devices. On the left, the device can run two blocks at the same time, so it takes longer than the device on the right that can run four blocks simultaneously. With transparent scalability, we can run a program on various types of GPU (for instance, lower-end GPUs for casual users, or higher-end GPUs for scientific calculation), without recompiling [13].

[26] shows that the use of CUDA improves the computation time of the FDTD method, with a relative speedup of more than two comparing with the CPU execution. Developers do not need specific GPU knowledge when they use CUDA, which is not the case with more generic language as OpenCL [27]. CUDA is designed for NVIDIA's chip, and being a scalable architecture according to [25], the developer does not need to go through the hardware recognition, and his code is adaptable to a wide range of NVIDIA's chips (as long as they support CUDA). For example Figure 2.6 shows how a multi-thread CUDA program adapts itself to the hardware, even if the execution on the GPU with two SMs will be slower than the execution on the GPU with four SMs. In the other hand, CUDA code cannot be executed on non-NVIDIA chips. OpenCL is an alternative to CUDA. It is an open language, and can be used on a variety of GPUs, including NVIDIA's GPUs. It presents similarities with CUDA. Thread blocks in CUDA are equivalent to work-groups in OpenCL, threads are called work-items and grids are NDRange [28]. OpenCL uses warps and latency hiding. It has transparent scalability to some extent [28]. But with OpenCL, the programmer has to manually recognize the hardware, while CUDA implicitly identifies NVIDIA's chips [19]. In our

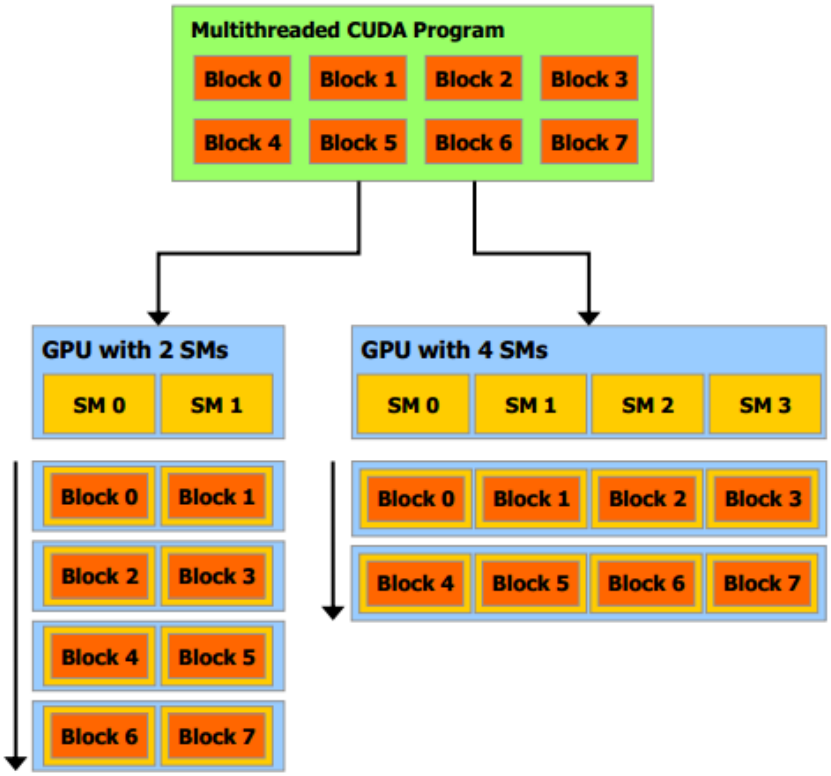

Figure 2.6: Transparent scalability [25].

project, we use CUDA as it simplifies the work on Tesla T10 GPUs, but an OpenCL version could be done.

2.4 Existing work

De Donno et al. published a paper in 2010 about the implementation of a two-dimensional FDTD problem with CUDA [24]. They present the different stages of their optimization. First, they do a simple 2D FDTD kernel that uses only global memory. As seen in Section 2.3, global memory gives a high-latency access. They decide to improve their performance by using shared memory, which has a smaller latency. To do so, they load data tiles into the shared memory [24]. Those tiles contain all the needed informations to compute the fields. Even if it adds data transfers, this modification greatly enhanced the performance of the program [24] (see Section 2.3 for more informations on shared memory). They also profit from specific data structures (called built-in arrays in [24]) to reduce

the number of global memory accesses. Their last improvement is to use the texture memory of their GPU [24].

While De Donno et al. give an interesting overview of the optimization process, they focus on a two-dimensional FDTD. The problem is limited to the size of block and the number of threads per block, and their distribution on the plane. When the FDTD method is extended to three dimensions, some questions are to be considered beforehand. The grid is now a cube, and there are different ways to map the cells to the threads. Three main methods emerge from the current research.

2.4.1 First method

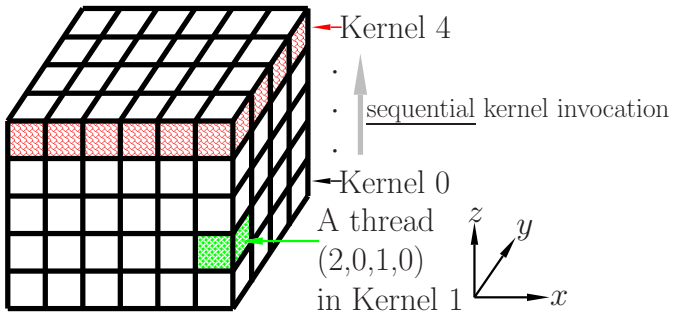

Figure 2.7: The domain decomposition of the 3D FDTD space in a first approach [Image source : F. Costen].

Bo et al. in [16], Demir and Elsherbeni in [29] and Chi et al. in [30] expose a method where the grid is divided into planes. Figure 2.7 illustrates this method. Each plane is divided into blocks of threads, such as each thread maps one yee-cell of the plane. The GPU grid contains a two-dimensional allocation of blocks, and blocks themselves contain a two-dimensional allocation of threads. In [29] this is called a *xyz*-mapping. The size of the blocks is defined by the user. In [16], blocks are of size 32×8 threads, in [30] each block consists of 16×16 threads. In [29], blocks are constructed as one-dimensional arrays of threads. The kernels that update the electric $\boldsymbol{E}$ and magnetic $\boldsymbol{H}$ fields have to be called sequentially for each plane. For example, first the kernel that computes $\boldsymbol{E}$ is executed on all points of plane $z = 1$. Then, it is executed on $z = 2$, and so on until $z = DIM$, DIM is the dimension of the grid in z direction. Once the $\boldsymbol{E}$ kernel has gone through the problem grid, the $\boldsymbol{H}$ kernel is executed in the same manner. Note

that the ***E*** kernel has to finish its execution on all planes before we launch the ***H*** kernel.

The kernels are called sequentially on each plane. The program has to wait for all threads to finish before it goes to the next plane. This means that this methodology exploits the parallelism within a plane, but lacks of parallelisation between planes because of the synchronization.

2.4.2 Second method

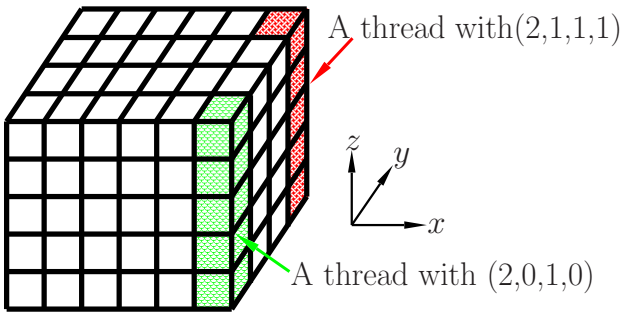

Figure 2.8: The domain decomposition of the 3D FDTD space in a second approach [Image source : F. Costen].

[26] and [9] use an other method, also presented in [29]. As before, the x-y plane of the grid is divided into blocks in two dimensions, but now each thread processes the entire column instead of only one cell. This method is called xy-mapping in [29]. We see in Figure 2.8 that each thread is mapped to a column. Each thread calculates the ***E*** and ***H*** components for $z = 1$ to $z = DIM$, with a *for* loop [29]. Here we work on all the data at once, thus we need to call each kernel only once. First, the ***E*** kernel is called and computes the whole grid. Then, the ***H*** kernel is executed. This differs from the first approach. We do not have threads synchronization between planes, except the last one when the kernel ends. We expect that this approach will be more efficient than the first one because it should have less synchronization overhead.

2.4.3 Third method

The third method was implemented last year in [1] and is also presented in [31]. Figure 2.9 shows the domain decomposition in this method.

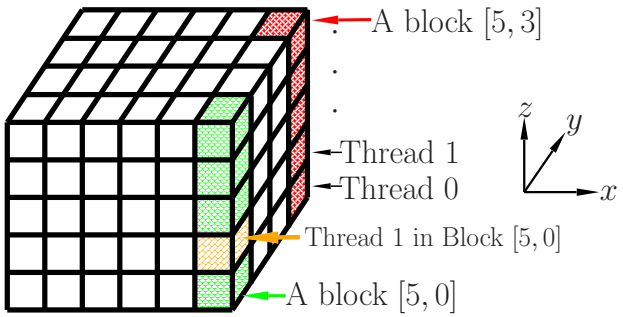

Figure 2.9: The domain decomposition of the 3D FDTD space in a third approach [Image source : F. Costen].

Method 3 looks like Method 2. The x-y plane is also divided into block in two dimensions. But here the blocks have a fixed size: they are all 1×1 blocks. It means that each column of the grid is allocated to a block. If we were in the case of Method 2, each block would contain only one thread that would compute its allocated column. In Method 3, a block has a one-dimensional distribution of threads. Several threads are mapped to the same column. The number of threads is defined by the user, and tuned according to the size of the problem. [31] studies the effect of the number of threads on the program efficiency. Each point will be computed by a thread. According to [1], two different schedules are possible. Each thread can take care of some consecutive points, which is called uncoalesced threads. For example, if we have two threads that have five consecutive points, thread 0 will compute cells (i,j,1) to (i,j,5), then thread 1 will calculate cells (i,j,6) to (i,j,10), again thread 0 will do cells (i,j,11) to (i,j,15), and so on until the height of the grid is reached. A second option is that all threads take care of on point consecutively, before doing the next block. This is named coalesced threads. For instance, with two threads, thread 0 computes cell (i,j,1), thread 1 does (i,j,2), thread 0 cell (i,j,3), thread 1 (i,j,4) and so on. In this case, at each iteration a thread moves by $number\ of\ threads - 1$ cells. The coalesced threads are expected to take more profit out of the memory access, because threads access consecutive space in memory. [1] shows that coalesced threads are more efficient with a little number of threads, but that this improvement compared to uncoalesced memory becomes less significant when we increase the number of threads.

Chapter 2 explored the background of our project. We presented the FDTD method, the hardware we use in our project and the programming API CUDA.

We proposed three different approaches to implement the FDTD method in Section 2.4. Chapter 3 presents the implementation of those three approaches.

Chapter 3

Implementation

The FDTD method is applied on a three-dimensional cubic problem of dimension $DIM \times DIM \times DIM$. In Chapter 2 three methods have been presented. Chapter 3 proposes the implementations of the three methods, and discusses their differences and possible optimizations.

3.1 General algorithm

Listing 3.1: Flowchart code.

```
1  void executeFDTD(void){
2        int t=0;
3
4        fp_type *dev_ex;
5        // Do the same for dev_ey, dev_ez , dev_hx ,
            dev_hy, dev_hz
6
7        /* Allocate space on GPU */
8        cudaMalloc((void **)&dev_ex,VOL*sizeof(fp_type));
9        // Do the same for dev_ey, dev_ez , dev_hx ,
            dev_hy, dev_hz
10
11       /* Copy matrices to CPU */
12       cudaMemcpy(dev_ex,ex,VOL * sizeof(fp_type),
            cudaMemcpyHostToDevice );
```

```
13        // Do the same for dev_ey, dev_ez , dev_hx ,
             dev_hy, dev_hz
14
15        /* Start timer */
16        cudaEventRecord( compstart, 0 );
17
18        /* Create blocks and threads */
19        // Here is Method 1 and Method 2 cases
20        dim3 dimOfGridInBlocks(SQRT_BLOCK_COUNT,
             SQRT_BLOCK_COUNT);
21        dim3 dimOfBlockInThreads(SQRT_THREAD_COUNT,
             SQRT_THREAD_COUNT);
22
23        for(t=1;t<=ITERATIONS;t++)
24        {
25           // Invoke eKernel and hKernel in different
                ways depending on the Method.
26        }
27
28        /* Stop timer */
29        cudaEventRecord( compstop, 0 );
30        cudaEventSynchronize(compstop);
31
32        /* Copy matrixes back to main memory */
33        cudaMemcpy(ex,dev_ex,VOL * sizeof(fp_type),
             cudaMemcpyDeviceToHost );
34        // Do the same for dev_ey, dev_ez , dev_hx ,
             dev_hy, dev_hz
35
36        /* Release device memory */
37        cudaFree(dev_ex);
38        // Do the same for dev_ey, dev_ez , dev_hx ,
             dev_hy, dev_hz
39
40  }
```

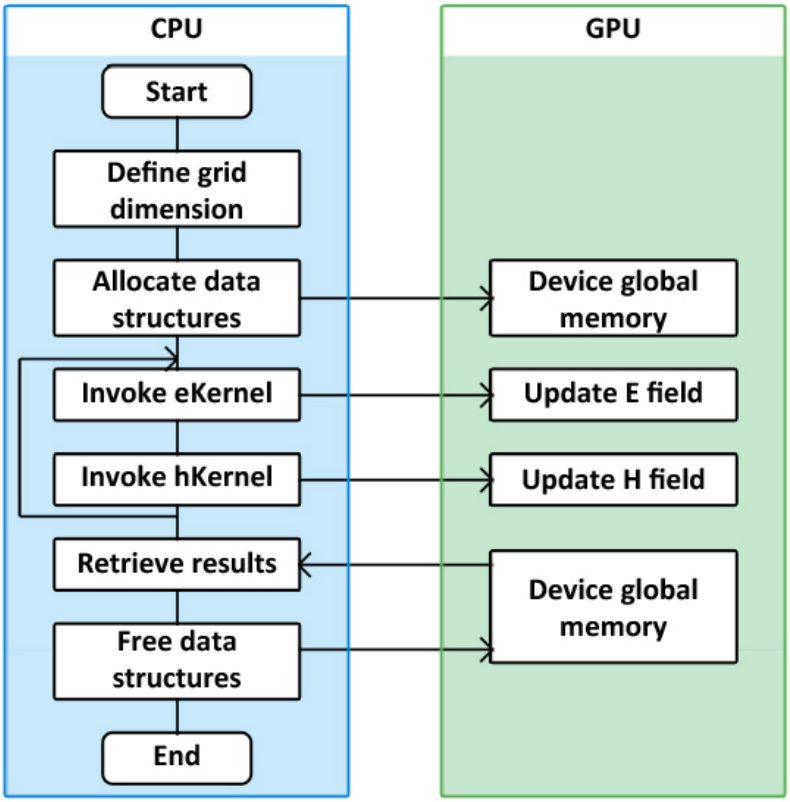

Figure 3.1: A flowchart of the FDTD computation on a GPU.

Figure 3.1 presents the flowchart of the three approaches we studied. Those three methods differ in the way eKernel and hKernel are implemented. This flowchart is implemented in the function executeFDTD in Appendix A. An extract of executeFDTD is presented in Listing 3.1.

The user chooses the dimension of the problem, and the CPU inputs the data structures to the global memory of the GPU. First the space is allocated on the GPU (cudaMalloc Line 8 Listing 3.1), and then the matrices with the original values are transfered (cudaMemcpy Line 12 Listing 3.1). The grid and blocks dimensions are defined Lines 20 and 21 Listing 3.1, but this definition is different depending on the method. Then two kernels are executed on the device to calculate the electric (eKernel) and magnetic (hKernel) fields. The kernels are invoked several iterations Lines 23 to 26 of Listing 3.1. More details on this execution are given in Sections 3.2, 3.4 and 3.5. Finally, the host collects the results (cudaMemcopy Line 33 Listing 3.1) and releases the data structures (cudaFree Line 37 Listing 3.1).

cudaMemcopy is used both to copy data from host to the device, and the other way around. The last argument determines the direction of the data transfer. Line 12 Listing 3.1 we see that the value of the third argument is cudaMemcpyHostToDevice, so the data is copied from the CPU to the GPU. Line 33 Listing 3.1 it is cudaMemcpyDeviceToHost, so the data is retrieved into host memory.

fp_type is defined at the beginning of the program with Listing 3.2. It can take two values: float or double. It makes it easier to change the precision of the code. One has just to change one line in the program instead of replacing every float or double. The reader can see in both Listings 3.1 and 3.2 that CUDA syntax is actually very close to C language, as it was stated in Section 2.3.

Listing 3.2: fp_type definition.

```
1 #define fp_type float
```

The allocation is made respecting the row-major convention used by C language, in opposition to the column-major convention in Fortran. However, there is no benefit in respecting this convention because we do not use the shared memory in blocks, and there is no cache in Tesla architecture. Adjacent cells are not retrieved when a call is made. Section 3.7 further explores the possible use of shared memory.

3.2 Implementation of Method 3

Method 3 has been implemented last year by Livesey [1]. His code was used as a guideline for the implementation of Method 1 and Method 2. In Method 3 the blocks are of size 1×1 which means they map columns of the grid. The threads are allocated to each column. They can be coalesced, which means they take one cell alternately until the height of the grid is reached, or they can be uncoalesced, which means they take several cells consecutively. Figure 3.2 shows the two different configurations. The green cells are computed by Thread 1, and the blue cells by Thread 2. The red line represents a block. The dark grey cells are the border. The cube has been cut for clarity. On the left, we see an uncoalesced schedule for a problem of dimension 4, and on the right is a coalesced schedule.

The number of threads on each column, and the number of cells they take consecutively, depend on the problem size. With a small problem size the coalesced

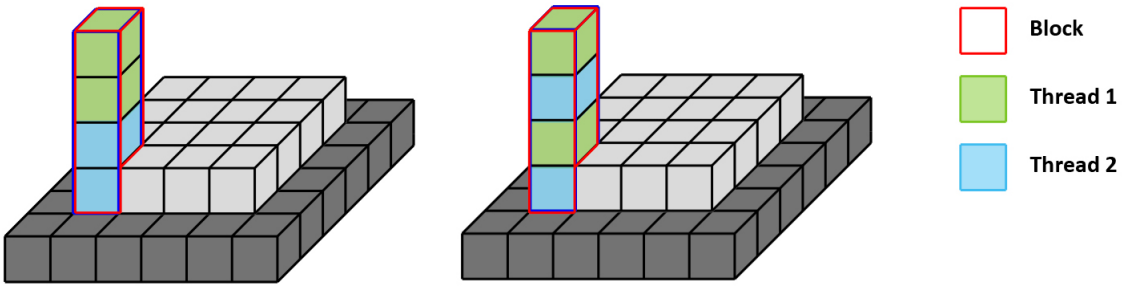

Figure 3.2: Left : Uncoalesced threads ; Right : coalesced threads.

method is more efficient because it takes advantage of locality in the memory access. However, with a big problem size, there is not much difference between coalesced and uncoalesced configurations anymore [1].

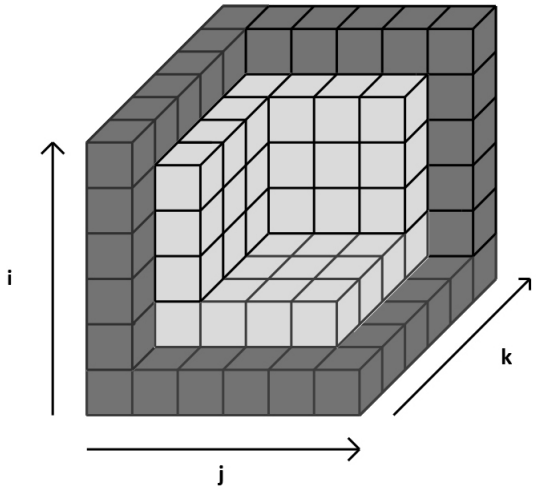

Figure 3.3: Spatial representation of a problem of size 4 with a border of size 1.

As shown in Figure 3.3, a padding wraps up the grid to simulate the external conditions presented in Section 2.1. This border should not be modified during the execution of the program, thus we do not execute the Maxwell's calculation on it. The thickness of the padding is one, and added to the problem's dimension we get a cube's total size of $(DIM + 2) \times (DIM + 2) \times (DIM + 2)$.

In Method 3, every block maps a column of the problem grid. A column has

a size of $1 \times 1 \times (DIM + 2)$. The grid indices begin to 0 and go to $DIM + 1$. We ignore the padding, so the calculations are made on cells with indices i, j and k between 1 and DIM.

Listing 3.3: Calculation of the coordinates of the cells and avoidance of the border in the electric kernel of Method 3 [1].

```
1  int i,j,k;
2  j=blockIdx.x+1;
3  k=blockIdx.y+1;
4  //threadIds start from 0 but because of border
       calculations start from 1:
5  i=threadIdx.x+1;
6
7  while(i<=gridDim.x){
8    /* Calculate Ex(i,j,k), Ey(i,j,k), Ez(i,j,k) */
9    int currentOffset=offset(i,j,k,dim);
10
11   dex[currentOffset] = dex[currentOffset] + dt/pmt * (
       ( (dhz[offset(i,j+1,k,dim)] - dhz[currentOffset] )
       /dy ) - ( (dhy[offset(i,j,k+1,dim)] - dhy[
       currentOffset] )/dz ) ) ;
12
13   dey[currentOffset] = dey[currentOffset] + dt/pmt * (
       ( (dhx[offset(i,j,k+1,dim)] - dhx[currentOffset] )
       /dz ) - ( (dhz[currentOffset+1] - dhz[
       currentOffset] )/dx ) ) ;
14
15   dez[currentOffset] = dez[currentOffset] + dt/pmt * (
       ( (dhy[currentOffset+1] - dhy[currentOffset] )/dx
       ) - ( (dhx[offset(i,j+1,k,dim)] - dhx[
       currentOffset] )/dy ) ) ;
16
17   i+=THREAD_COUNT
18 }
```

Listing 3.3 shows the kernel that computes the electric field in Method 3. The same is done for the magnetic field. Lines 11, 13 and 15 Listing 3.3 compute E_x, E_y and E_z respectively. The grid with the padding is of size $(DIM + 2)^3$. If we divide it into $DIM \times DIM$ blocks of unit length and unit width, we will get j and k going from 0 to $DIM - 1$. We want to execute blocks with j and k going from 1 to DIM to avoid the border, so the blocks' indices are shifted (Lines 3 and 4 of Listing 3.3). It permits to avoid the front, back, left and right borders. After this, the threads' indices are shifted too (Line 7 of Listing 3.3), so that the upper and lower borders are avoided. Listing 3.3 gives an example of a coalesced implementation. We see Line 10 that the step is of $THREAD\_COUNT$. If we were in an uncoalesced scheme, we would have a *for*-loop and a unit step.

Listing 3.4: Kernels launch in Method 3 [1].

```
1  dim3 blocks(DIM,DIM);
2  dim3 threads(THREAD_COUNT);
3
4  for(t=1;t<=ITERATIONS;t++){
5    eKernel<<<blocks,threads>>>(dev_ex,dev_ey,dev_ez,
       dev_hx,dev_hy,dev_hz,dx,dy,dz,pmt,dt);
6
7    *(ez+offset(DIM/2-1,DIM/2-1,DIM/2-1,DIM))=-jz[t];
8
9    cudaMemcpy(&dev_ez[offset(DIM/2-1,DIM/2-1,DIM/2-1,DIM
       )],&ez[offset(DIM/2-1,DIM/2-1,DIM/2-1,DIM)],sizeof
       (fp_type),cudaMemcpyHostToDevice);
10
11   hKernel<<<blocks,threads>>>(dev_ex,dev_ey,dev_ez,
       dev_hx,dev_hy,dev_hz,dx,dy,dz,pma,dt);
12 }
```

In Listing 3.4 the host calls the kernels of the electric field (eKernel Line 5) and magnetic field (hKernel Line 11). It corresponds with Lines 20 to 26 in Listing 3.1. The user gives the parameters DIM, the dimension of the problem, and $ITERATIONS$, the number of iterations the program executes. *dim*3 is a build-in variable type based on uint3, where the default value of non-specified parameters is 1 [25]. For example, Line 1 of Listing 3.4 is the same that

$dim3\ blocks(DIM, DIM, 1)$. The lack of the last parameter indicates that the distribution of blocks in the GPU grid is two-dimensional. Line 2 of Listing 3.4 shows similarly that the threads have a one-dimensional distribution. In Method 3, we have seen that each block has a size of 1×1. The reader can verify it in Line 1 of Listing 3.4: the problem grid is of size $DIM \times DIM \times DIM$, if the base is divided into $DIM \times DIM$ blocks, it means each block is of size 1×1.

CUDA has a specific syntax to launch the kernels. In Line 5 of Listing 3.4 we see that first there is two parameters surrounded by <<< and >>>. The first parameter indicates how the grid is divided into blocks: we saw in Line 1 Listing 3.4 that the grid is divided into $DIM \times DIM$ blocks. The second parameter indicates how the blocks are divided into threads: we read in Line 2 of Listing 3.4 that each block contains $THREAD\_COUNT$ threads. The arguments between round brackets are like the usual C arguments for a function. *dev_ex*, *dev_ey* and *dev_ez* are the components of the electric field on the device. *dev_hx*, *dev_hy* and *dev_hz* are the components of the magnetic field on the device. *dx*, *dy* and *dz* are the spatial steps, and *dt* is the temporal step. *pmt* is the permittivity and *pma* is the permeability. *jz* is the conduction density vector. Refer to Chapter 2 for further information on Maxwell's equations.

3.3 Differences between the three methods

In Method 3, the GPU grid is divided into $DIM \times DIM$ blocks. Each column of the problem grid is allocated to a block of size 1×1 in the GPU. Every block is divided into threads in one dimension. In Method 1 and 2, the GPU grid is divided into $SQRT\_BLOCK\_COUNT \times SQRT\_BLOCK\_COUNT$ blocks. The blocks are of size $(DIM/SQRT\_BLOCK\_COUNT)^2$. They contain $SQRT\_THREAD\_COUNT \times SQRT\_THREAD\_COUNT$ threads where $SQRT\_THREAD\_COUNT = DIM/SQRT\_BLOCK\_COUNT$.

The main difference between Method 1 and Method 2 is that in Method 1, we iteratively compute the layers of the problem grid, so the kernels are launched multiple times, but the threads calculate only one cell each. In Method 2, the kernels are launched only once but each thread has to compute a whole column.

When $SQRT\_BLOCK\_COUNT = DIM$, we are in the case of Method 3 (every block is of size 1×1), except that a column is calculated by one thread,

not a group of threads.

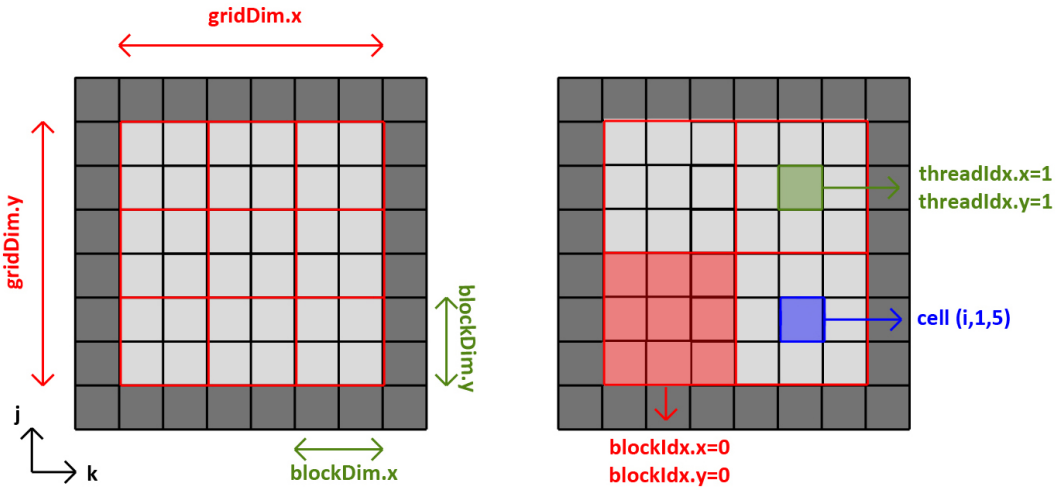

Figure 3.4: Two different assignations for a problem of dimension 6. Left : 3×3 blocks with 2×2 threads each ; Right : 2×2 blocks with 3×3 threads each.

Figure 3.4 presents a problem where $DIM = 6$, so the total size with the border is 8. The blocks are represented by the red squares, the border is of dark grey colour, and a thread takes a unit cell. Figure 3.4 illustrates the different built-in variables gridDim, blockDim, blockIdx and threadIdx. A thread has unique coordinates inside a block, but not necessarily inside the grid. In the right configuration of Figure 3.4, four threads have $threadIdx.x = 1$ and $threadIdx.y = 1$. To identify a thread, we have to use the coordinates of the block containing the thread, and the thread's coordinates. For example, the green thread in Figure 3.4 has the unique identity (4,4), because $blockIdx.x \times blockDim.x + threadIdx.x = 1 \times 3 + 1 = 4$, and in the same manner for y-coordinate.

$6 = 2 \times 3$ so there are two different configurations: 3×3 blocks containing 2×2 threads, or 2×2 blocks with 3×3 threads. On the left in Figure 3.4, $SQRT\_BLOCK\_COUNT = 3$ and $SQRT\_THREAD\_COUNT = 2$. On the right $SQRT\_BLOCK\_COUNT = 2$ and $SQRT\_THREAD\_COUNT = 3$. In Method 3, there was only one possible design, where each block was allocated to a column.

The problem size can be extended to bigger dimensions. In our experiments, DIM is equal to 64, 128, 192 or 256. One has to keep in mind that due to material restrictions, some combinations are not possible. For example, when $DIM = 64$, we cannot implement one block with 64×64 threads because a block

cannot handle more than 512 threads.

3.4 Implementation of Method 1

Listing 3.5: Extract of the electric kernel in Method 1.

```
1 int j,k;
2 int dim=gridDim.x*blockDim.x;
3
4 j=(blockIdx.x) * blockDim.x + threadIdx.x + 1;
5 k=(blockIdx.y) * blockDim.y + threadIdx.y + 1;
6
7 int currentOffset=offset(i,j,k,dim);
8
9 // Calculate Ex(i,j,k), Ey(i,j,k), Ez(i,j,k)
```

Listing 3.5 presents an extract of the electric kernel in Method 1. It is similar for the magnetic kernel. First, the variable i has been deleted since it is passed as a function argument. As in Line 2 the variable dim is introduced. In Method 3, $DIM = gridDim.x$, but in Method 1 and Method 2 DIM is not accessible in the kernels, so dim is created. We can use the built-in functions. Knowing that the dimension is $gridDim.x * blockDim.x$, we declare $dim = gridDim.x * blockDim.x$ (remember that gridDim.x is the number of blocks in the x dimension of the grid, and blockDim.x is the number of threads in the x direction of a block).

j and k are calculated as explained in Section 3.3 There is no loop because a thread executes only one cell. dim is created inside the kernels. This means that each thread will create its own version of dim at the beginning of the kernel, and dim will be deleted once the kernel has ended [19]. We say that the lifetime of dim is the kernel, and its scope is the thread [19]. dim is a constant, so we could have used a global declaration with __constant__ ("__" is two "_"), and have placed this variable into the constant memory of the device. It would have used only one declaration, and it would have been the same variable for every threads and blocks, because the scope of constant variable is the grid [19]. Moreover, the lifetime of a constant variable is the application according to [19] so it would have been accessible by both kernels.

DIM = 256 – 1000 time-steps				
	16^2B containing 16^2T		32^2B containing 8^2T	
	M1	M1 const.	M1	M1 const.
Average (sec.)	308.251	308.617	386.307	386.181
With - without const. (sec.)	0.366		-0.126	

	64^2B containing 4^2T		128^2B containing 2^2T	
	M1	M1 const.	M1	M1 const.
Average (sec.)	554.6936	557.195	426.663	427.907
With - without const. (sec.)	2.503		1.245	

	256^2B containing 1T	
	M1	M1 const.
Average (sec.)	495.551	495.825
With - without const. (sec.)	0.274	

Table 3.1: Use of constant memory in Method 1.

Tables 3.1 and 3.2 present the executing times of Method 1 and Method 2 with and without a constant declaration of *dim*. The values have been rounded up. We took the average of five runs for a problem of dimension 256 in each configuration: M1 (Method 1), M1 const (Method 1 with constant declaration), M2 (Method 2) and M2 const (Method 2 with constant declaration). 16^2B *containing* 16^2T means that we compute the FDTD method on a grid divided into 16×16 blocks containing 16×16 threads. The programs have been run for 1000 time-steps in single precision. We compare the execution times with and without the constant memory. When a cell "With - without const." of Tables 3.1 and 3.2 is negative, it means that the program using the constant declaration takes more time that the one without constant declaration. If it is positive, the use of a constant variable increases performance.

Table 3.1 shows that, overall, the use of constant memory in Method 1 decreases the performance of the program, except for the configuration with 32×32

DIM = 256 – 1000 time-steps				
	16^2B containing 16^2T		32^2B containing 8^2T	
	M2	M2 const.	M2	M2 const.
Average (sec.)	327.278	327.302	411.986	411.806
With - without const. (sec.)	0.024		-0.180	

	64^2B containing 4^2T		128^2B containing 2^2T	
	M2	M2 const.	M2	M2 const.
Average (sec.)	649.073	648.223	672.901	672.829
With - without const. (sec.)	-0.85		-0.0722	

	256^2B containing 1T	
	M2	M2 const.
Average (sec.)	624.302	624.305
With - without const. (sec.)	0.003	

Table 3.2: Use of constant memory in Method 2.

blocks, but the benefit is small (about 0.13 second). When the grid is divided in 128×128 blocks, the program runs 1.24 seconds longer. The bigger gap is for 64×64 blocks where the difference is about 2.50 seconds. This in unexpected. The lifetime of constant memory is the application, so a constant *dim* will persist through the different kernel calls. It will be created only once at the beginning of the program. On the other hand, when *dim* is declared inside the kernel, it is created for every thread and for every iterations and kernel calls. In Method 1, for dimension 256, *dim* is accessed $(256 \times 256 \times 256) \times 2 \times 1000 = 33,554,432,000$ times, and is created as many times when we do not use constant memory.

According to [19], a variable that is declared in a kernel (named a *scalar* variable in the text) will be placed into registers. Registers permit a very fast access to memory, as long as their storage capacity is not exceeded [19]. Constant variables, stored in the constant memory, are slower than registers to access, but they can be accessed at high speed with a proper access pattern [19]. It can explain this unexpected observation. Even if *dim* is created multiple times,

registers are the fastest memory available and the access speed counterbalances the creation cost. Thus, Method 1 gets more benefit from accessing registers instead of the constant memory.

However, this redundancy induces a waste of memory space. The fastest time is obtained with a 16×16 blocks division. The loss of time performance is small enough to consider the use of constant memory, in case of a lack of registers. It depends on what the program needs, a trade-off between speed and memory space. Registers are not a limiting factor in this configuration, so we will keep a scalar variable.

Table 3.2 shows that the use of constant memory leads to a small loss or a small gain in Method 2 timing performance, depending on the division of blocks. The difference is smaller because in Method 2 for dimension 256, *dim* is created $(256 \times 256) \times 2 \times 1000 = 131,072,000$, which is 256 times less than Method 1. Those results confirm that, in both cases, it is better to keep a scalar variable. However they cannot be extended to other programs which may require more registers, or access the constant memory in a different pattern for example.

Listing 3.6: Kernels launch in Method 1.

```
1  dim3 dimOfGridInBlocks(SQRT_BLOCK_COUNT,
       SQRT_BLOCK_COUNT);
2  dim3 dimOfBlockInThreads(SQRT_THREAD_COUNT,
       SQRT_THREAD_COUNT);
3
4  for(t=1;t<=ITERATIONS;t++)
5  {
6    // Calculate E components
7    for(i=1;i<=DIM;i++){
8          eKernel<<<dimOfGridInBlocks,dimOfBlockInThreads
             >>>(i,dev_ex,dev_ey,dev_ez,dev_hx,dev_hy,
             dev_hz,dx,dy,dz,pmt,dt);
9    }
10   // Update source in CPU
11   *(ez+offset(DIM/2-1,DIM/2-1,DIM/2-1,DIM))=-jz[t];
12
13   // Transfer updated source to GPU
```

```
14      cudaMemcpy(&dev_ez[offset(DIM/2-1,DIM/2-1,DIM/2-1,
           DIM)],&ez[offset(DIM/2-1,DIM/2-1,DIM/2-1,DIM)],
           sizeof(fp_type),cudaMemcpyHostToDevice);
15
16    // Calculate H components
17    for(i=1;i<=DIM;i++){
18      hKernel<<<dimOfGridInBlocks,dimOfBlockInThreads>>>(
           i,dev_ex,dev_ey,dev_ez,dev_hx,dev_hy,dev_hz,dx,
           dy,dz,pma,dt);
19    }
20  }
```

In Method 1, the kernels are launched as coded in Listing 3.6. The first difference from Method 3 is Line 1 and Line 2 of Listing 3.6. The GPU grid is now divided into $SQRT\_BLOCK\_COUNT \times SQRT\_BLOCK\_COUNT$ blocks, so blocks are no longer of unit length and width. Threads have a two-dimensional distribution. $SQRT\_BLOCK\_COUNT$ and $SQRT\_THREAD\_COUNT$ depend on the problem size, and can take multiple values, but those values cannot change during the execution of the program.

The second main difference is the appearance of two *for*-loops Line 6 and Line 14 of Listing 3.6. The kernels are executed on layer i, which is passed as a parameter of eKernel and hKernel. We did not use a *for*-loop that includes both eKernel and hKernel because hKernel uses the values computed by eKernel. With only one loop, the fields are executed in a different order, and the final results are different. For example, cells on a layer will use the new values of $\boldsymbol{E}$ on one side, and the old values on the other side to compute $\boldsymbol{H}$. Remember in Chapter 2 that to compute the magnetic components of a cell, the values of the electric field in the neighbouring cells are used. With one *for*-loop, the layer $i+1$ is not updated when we calculate the i layer. eKernel has to be applied to all the layers before we can launch hKernel, that is why there is two loops.

3.5 Implementation of Method 2

Listing 3.7: Extract of the electric kernel in Method 2.

```
1  int i,j,k;
2  int dim=gridDim.x*blockDim.x;
3
4  j=(blockIdx.x) * blockDim.x + threadIdx.x + 1;
5  k=(blockIdx.y) * blockDim.y + threadIdx.y + 1;
6
7  for(i=1; i<=dim; i++){
8
9    // Calculate Ex(i,j,k), Ey(i,j,k), Ez(i,j,k)
10
11 }
```

Listing 3.7 presents an extract of the electric kernel in Method 2. The magnetic kernel is identical, but it computes H_x, H_y and H_z instead of E_x, E_y and E_z. This is almost the same as Method 1, except that the i variable is kept (Line 1 Listing 3.7). i is the index in z direction. In Method 1, i was the layer where the thread was. In Method 2, threads execute a column so we do a *for*-loop on i (Line 7 Listing 3.3), so that they execute on every layer of the grid. It is like we have moved the *for*-loops from the CPU to the kernels. So now, each thread executes itself the *for*-loop. It means that they are no synchronization between threads for each i, as in Method 1. In Method 1, the sequential call of the kernels forced all threads to stop and start together for each layer. In Method 2, they start together at layer 1, and stop together at layer dim.

Listing 3.8: Kernels launch in Method 2.

```
1  dim3 dimOfGridInBlocks(SQRT_BLOCK_COUNT,
     SQRT_BLOCK_COUNT);
2  dim3 dimOfBlockInThreads(SQRT_THREAD_COUNT,
     SQRT_THREAD_COUNT);
3
4  for(t=1;t<=ITERATIONS;t++)
5  {
6    // Calculate E components
```

```
7    eKernel<<<dimOfGridInBlocks,dimOfBlockInThreads>>>(
        dev_ex,dev_ey,dev_ez,dev_hx,dev_hy,dev_hz,dx,dy,dz
        ,pmt,dt);
8
9    // Update source in CPU
10   *(ez+offset(DIM/2-1,DIM/2-1,DIM/2-1,DIM))=-jz[t];
11
12   // Transfer updated source to GPU
13   cudaMemcpy(&dev_ez[offset(DIM/2-1,DIM/2-1,DIM/2-1,DIM
        )],&ez[offset(DIM/2-1,DIM/2-1,DIM/2-1,DIM)],sizeof
        (fp_type),cudaMemcpyHostToDevice);
14
15   // Calculate H components
16   hKernel<<<dimOfGridInBlocks,dimOfBlockInThreads>>>(
        dev_ex,dev_ey,dev_ez,dev_hx,dev_hy,dev_hz,dx,dy,dz
        ,pma,dt);
17
18 }
```

Listing 3.8 shows the setting up of kernels in Method 2. We do not need the two *for*-loops on the i parameter because one iteration of eKernel computes the electric field on each cell of the problem grid, not on just one layer. The *for*-loop is included inside the kernel as seen in Listing 3.7. The kernel does not have i as an argument. This launch is very similar to Listing 3.4. The only difference is Lines 1 and 2 Listing 3.8 where the disposition of blocks and threads changes. The GPU grid is divided into $SQRT\_BLOCK\_COUNT \times SQRT\_BLOCK\_COUNT$ blocks in Method 2, and it was divided into $DIM \times DIM$ blocks in Method 3. Blocks have $SQRT\_THREAD\_COUNT \times SQRT\_THREAD\_COUNT$ in Method 2, a two-dimensional distribution, while in Method 3 they have $THREAD\_COUNT$ threads, in one dimension.

3.6 Accuracy

We compare the values of $E_x(189, 189, 189)$ calculated by the three approaches in a 256 dimension. The source is located at the point (127,127,127). The measures are done for single and double precisions, every 500 time-steps. Method 1 and

Method 2 are executed with a division of 16×16 blocks. We took the code given by Livesey in [1] for Method 3 in single and double precisions, with 256 coalesced threads.

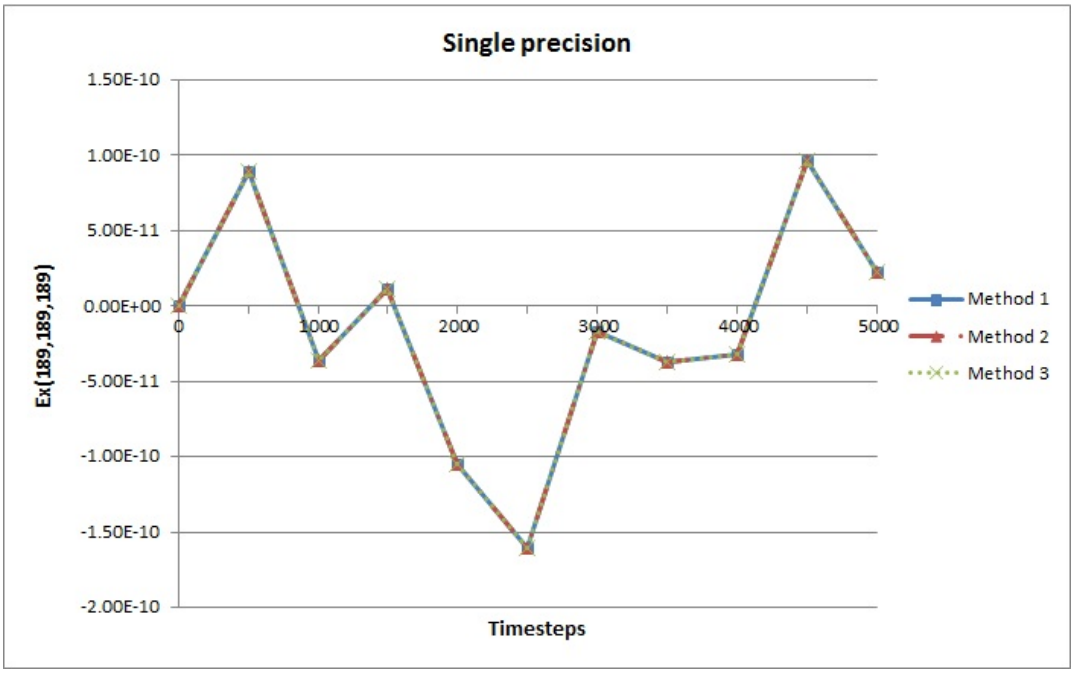

Figure 3.5: Accuracy in dimension 256, in single precision. The excitation is at (127,127,127).

Figure 3.5 presents the single precision results. The curves are identical. In fact, the results for every implementations are equal to each other. Figure 3.6 shows the double precision results. Here again, the results for every implementations are the same.

Figures 3.5 and 3.6 looks very similar. However, there is some differences between single and double precisions results. Sometimes, the second significant figure of the double precision results is different from the single precision results, and all the numbers are different after the third significant figure. Livesey in [1] states that "the single-precision results match the double-precision results to 3 significant figures". Apparently this is not the case here. One possible explanation is that Livesey recorded the value of $E_z(10, 0, 0)$. We looked at $E_z(10, 0, 0)$ to verify our hypothesis and found the same results as [1]. The cell (10,0,0) is on a border, and its value stabilizes after about 500 iterations, thanks to the border cells that keep the same values. The cell (10,0,0) is placed side by side with cells that do not vary, so the change between single and double precisions has

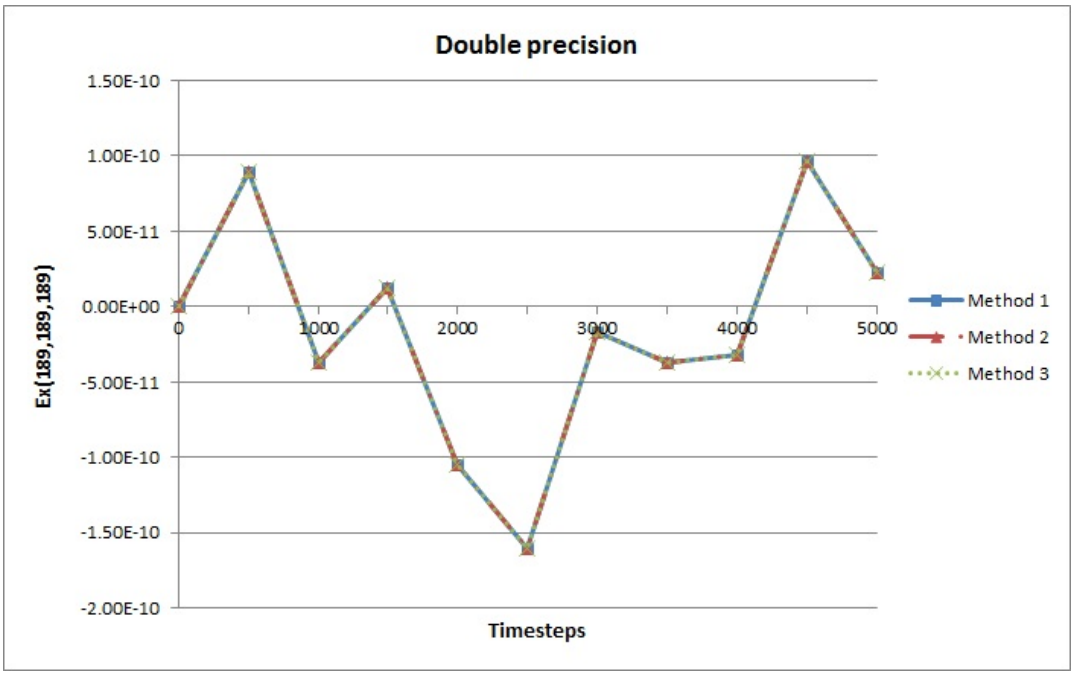

Figure 3.6: Accuracy in dimension 256, in double precision. The excitation is at (127,127,127).

little impact. The cell (189,189,189) is surrounded by cells that are varying over time. It can explain that single and double precisions implementations are almost identical in [1] while our results indicate that there is a slight but still sensible difference. For sensitive applications, we cannot content ourselves with single precision, even if a single-precision program runs faster than a double-precision one. We should favour the safety of the patient, and use double precision.

3.7 CGMA analyze

In the three approaches, we did not use shared memory. We saw in Section 2.3 that the Compute to Global Memory Access (CGMA) ratio is a good indicator of performance, along with the occupancy. According to [19], one can increase the CGMA ratio with constant and shared memories. In Section 3.4, we proved that the use of constant memory does not improve the performances. Here we will see the impact of shared memory on the CGMA ratio.

Listing 3.9: eKernel of Method 1.

```
1  __global__ void eKernel(int i, fp_type* dex, fp_type*
       dey, fp_type* dez, fp_type* dhx, fp_type* dhy,
       fp_type* dhz, fp_type dx, fp_type dy, fp_type dz,
       fp_type pmt, fp_type dt) {
2
3      int j,k;
4      int dim=gridDim.x*blockDim.x;
5
6    j=(blockIdx.x) * blockDim.x + threadIdx.x + 1;
7    k=(blockIdx.y) * blockDim.y + threadIdx.y + 1;
8
9    int currentOffset=offset(i,j,k,dim);
10
11     dex[currentOffset] = dex[currentOffset] + dt/pmt *
         ( ( ( dhz[offset(i,j+1,k,dim)] - dhz[
         currentOffset] )/dy ) - ( ( dhy[offset(i,j,k+1,
         dim)] - dhy[currentOffset] )/dz ) ) ;
12
13     dey[currentOffset] = dey[currentOffset] + dt/pmt *
         ( ( ( dhx[offset(i,j,k+1,dim)] - dhx[
         currentOffset] )/dz ) - ( ( dhz[currentOffset+1]
          - dhz[currentOffset] )/dx ) ) ;
14
15     dez[currentOffset] = dez[currentOffset] + dt/pmt *
         ( ( ( dhy[currentOffset+1] - dhy[currentOffset]
         )/dx ) - ( ( dhx[offset(i,j+1,k,dim)] - dhx[
         currentOffset] )/dy ) ) ;
16 }
```

Listing 3.9 is the code of the electric kernel in Method 1. The magnetic kernel is the same with $\boldsymbol{H}$ elements, and Method 2 and Method 3 have similar kernels. Our CGMA analyse will be based on this kernel. In Line 11 Listing 3.9, $E_x(currentOffset)$ is written. In order to execute Line 11, we need 1 global memory write and 5 global memory reads. Thus in total 6 global memory accesses are required.

For the compute part of CGMA ratio, Line 11 Listing 3.9 has 3 additions, 3 subtractions, 1 multiplication and 3 divisions, so 10 instructions in total. There are also two calls to the offset function presented in Listing 3.10. Offset has 2 multiplications and 5 additions. The total number of arithmetic operations for Line 11 Listing 3.9 is $10 + 2 \times 7 = 24$ operations.

Listing 3.10: Offset function.

```
1 __host__ __device__ int offset(int i,int j,int k,int
     dimIn)
2 {
3       return i+ ( j * (dimIn+2) ) + ( k * (dimIn+2) *(
           dimIn+2) );
4 }
```

Lines 13 and 15 have only one call to offset instead of two, but have an extra addition, so they have $11 + 7 = 18$ instructions each. Finally, there are $24 + 2 \times 18 = 60$ operations for every $3 \times 6 = 18$ global memory accesses, so the CGMA ratio is $60 \div 18 \approx 3.33$ for a kernel in Method 1.

Shared memory can be used when several threads access the same memory space. It is the case in Lines 13 and 15 of Listing 3.9. Figure 3.7 illustrates the possible use of shared memory. The red arrows are global memory accesses, the green arrows are shared memory accesses. The transparent arrows represent the global memory accesses which we do not have to carry when we use shared memory. Consecutive threads share one value: as represented in Figure 3.7, currentOffset + 1 for the thread T0 is the same data as currentOffset for the following thread T1. So T0 can load currentOffset + 1 in shared memory and T1 can read it from shared memory instead of global memory. However, T1 will have to read currentOffsetT1 + 1 from global memory and put it in shared memory for T2. Thus, we reduce by half the number of accesses to global memory. And we do it for two calculations over three (in eKernel in Listing 3.9, only E_y Line 13 and E_z Line 15 have this kind of access pattern, and the same applies for $\boldsymbol{H}$). Finally, it reduces the global memory accesses to $6+2\times(6\div2) = 12$. The CGMA ratio increases to $60 \div 12 = 5$.

The CGMA ratio is increased from 3.33 to 5, but it is still small. Other elements have to be considered. The use of shared memory will require to redesign

the algorithm, which is likely to be more complex. Moreover, additional synchronization mechanisms have to be implemented to ensure that T1 will not access shared memory before T0 has stored the data in it. We guess that some of the benefit will be lost into synchronization overhead. Kirk and Hwu's experiments support this idea: they do a matrix multiplication kernel, and when they load tiles into shared memory they add barrier synchronization between loading the data and executing the multiplication [19]. Another element to consider is that by using shared memory in our algorithm, we will use shared memory space. Shared memory space is finite and becomes a possible limiting factor. We assume that each thread will store two integers (4 bytes): one for Line 13 and one for Line 15 Listing 3.9. The program needs *the number of threads* $\times$ 4 bytes twice. That is, if a block has 256 threads for example, a block will need 2048 bytes of shared memory. As specified in Table 2.1, one multiprocessor (SM) has 16KB of shared memory, divided between the blocks. Up to 8 blocks can run on a SM, so shared memory is not a limiting factor as long as it does not exceed 2KB per block, that is 2048 bytes per block. So finally, shared memory will not limit the performance, in single precision, if blocks do not have more than $2048 \div (2 \times 4) = 256$ threads.

The use of shared memory increases the CGMA ratio from 3.33 approximately to 5. However, it is far from the optimum. Based on the T10 specifications in Table 2.1, the GPU can perform at 933 GFlops in single precision, that is $933 \cdot 10^9$ floating point operations per second. The global memory bandwidth is 102 GB/s, and in single precision we use 4 bytes, so the maximum throughput is $102 \cdot 10^9 \div 4 = 25.5 \cdot 10^9$ bytes per second. Therefore, the optimum CGMA ratio is $(933 \cdot 10^9) \div (25.5 \cdot 10^9) \approx 36.59$ in single precision, much greater than 5. In double precision, the GPU can perform at 78 GFlops and the maximum throughput is $102 \cdot 10^9 \div 8 = 12.75 \cdot 10^9$ bytes per second. The maximum CGMA ratio is $78 \div 12.75 \approx 6.11$, closer to our results.

In Chapter 3, we presented the major deliverable of our project. It consists of the code for the different methods, in particular Method 1 and Method 2. We observed the impact of constant and shared memory on the performance. In Chapter 4, we will run this code and report the timing results. We will compare the three approaches to determine which is the best.

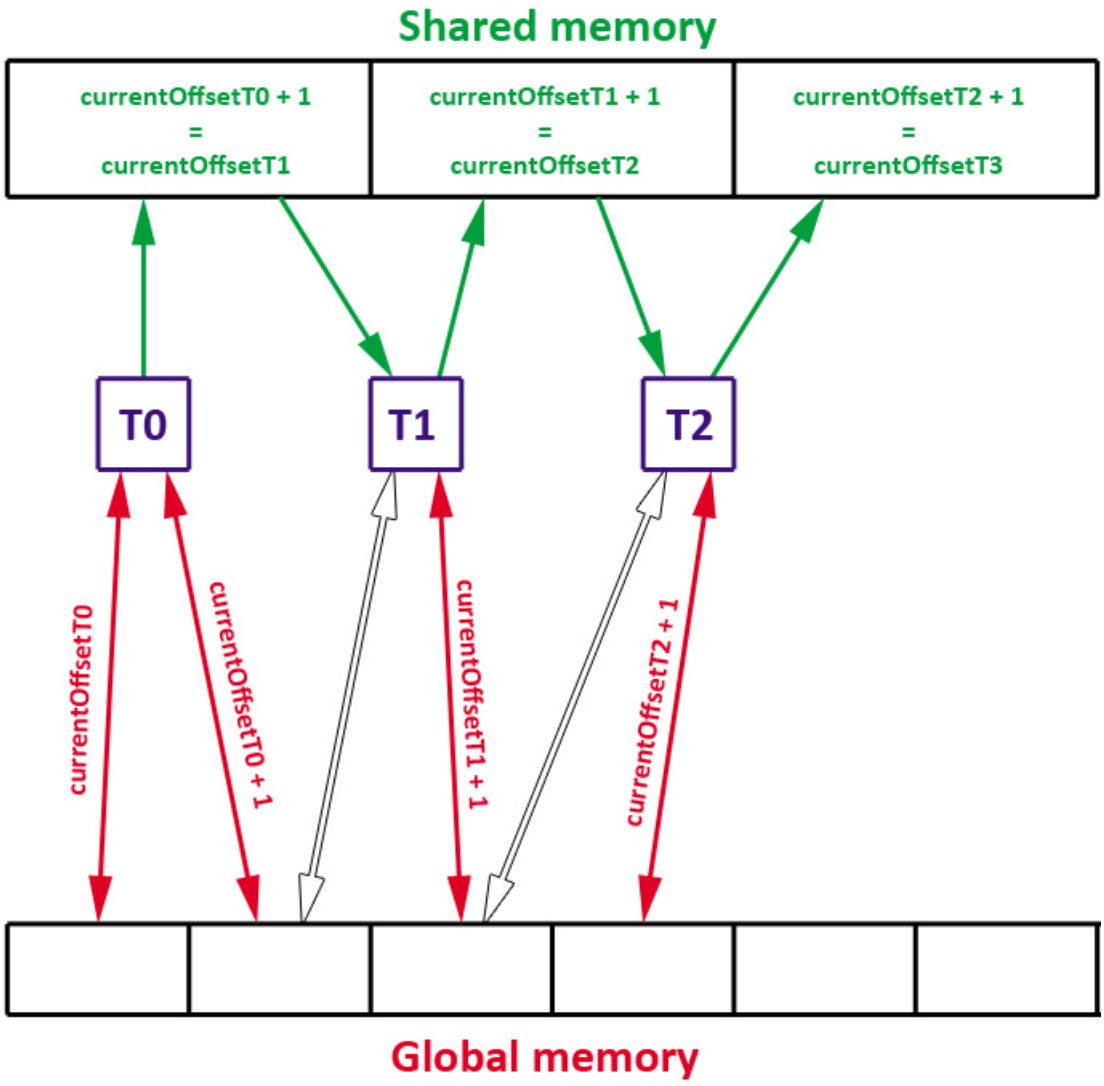

Figure 3.7: Use of shared memory in a kernel.

Chapter 4

Experiments and results

In Chapter 4, we tune the code presented in Chapter 3 to see which number of blocks per grid is the best. The number of blocks per grid times the number of threads per block is the dimension of the grid. The number of threads directly depends on the number of blocks, which is the only parameter.

We ran the code 10 times for every factorisations of each implementation, with simple and double precision, and for 100 time-steps. We took the average of the ten measures.

In the following figures, "X^2B Y^2T" means that the grid is made of $X \times X$ blocks and that each block is made of $Y \times Y$ thread. We saw that $SQRT\_BLOCK\_COUNT \times SQRT\_THREAD\_COUNT = PROBLEM\_SIZE$, so we should explore all the possible factorisations of PROBLEM_SIZE. However, in the Tesla architecture the maximum number of threads per block is 512. We have to check that $THREAD\_COUNT < 512$. That is why all the factorisations are not executed. For example with a problem size of 64, we cannot execute the problem with only one block because it means a block contains $64 \times 64 = 4096$ threads.

4.1 Single precision

Figure 4.1 presents the results for a grid of $64 \times 64 \times 64$ cells. Method 1 and Method 2 are compared. We did not execute the program for the case with 2×2 blocks and the one with 1 block as they contain more than 512 threads. Method 1 has the best timing, 0.590 seconds approximately, when the grid is divided into

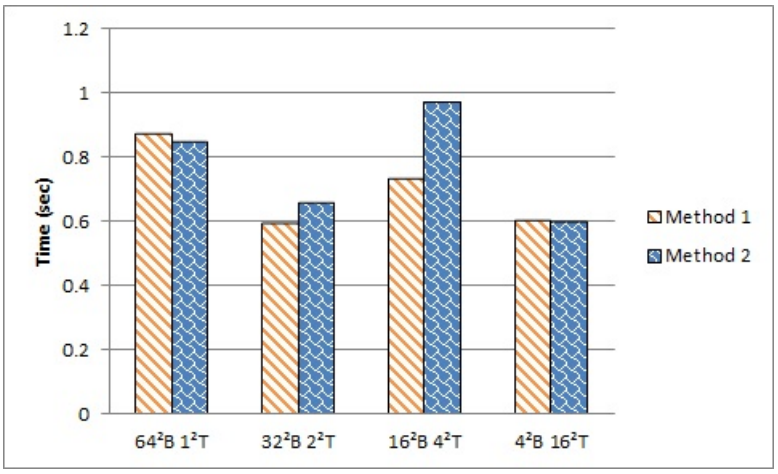

Figure 4.1: Results for a problem of dimension 64 with single precision.

32×32 blocks. The division with 4×4 blocks is very close, it has about 0.011 seconds more. Method 2 performs the best for a 16×16 blocks division, with 0.599 seconds.

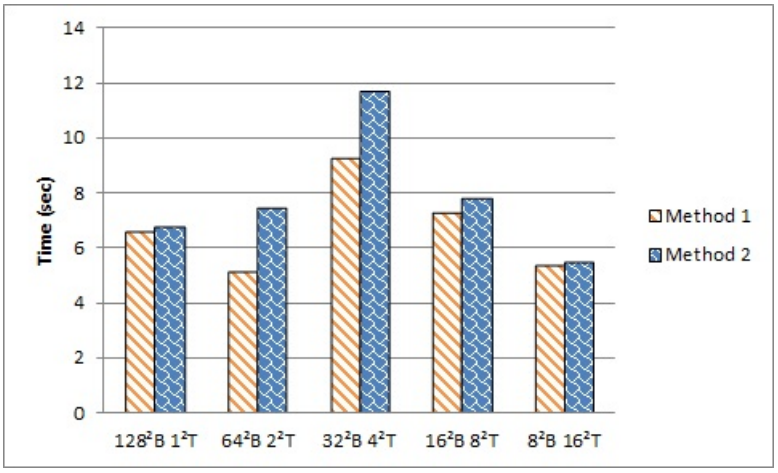

Figure 4.2: Results for a problem of dimension 128 with single precision.

Figure 4.2 shows the results when the dimension is 128. The smallest time of Method 1 is 5.09 seconds, when the program is executed on 64×64 blocks. Method 2 gets the best performance when the grid is divided into 8×8 blocks. It runs in 5.459 seconds.

In Figure 4.3, we see the execution times for different methods of the grid of dimension 192. The best time for Method 1 is 20.764 seconds, when the problem grid is shared between 96×96 blocks with 2×2 threads each. Method 2 shows a peak in performance for 192×192 blocks, with 22.45 seconds.

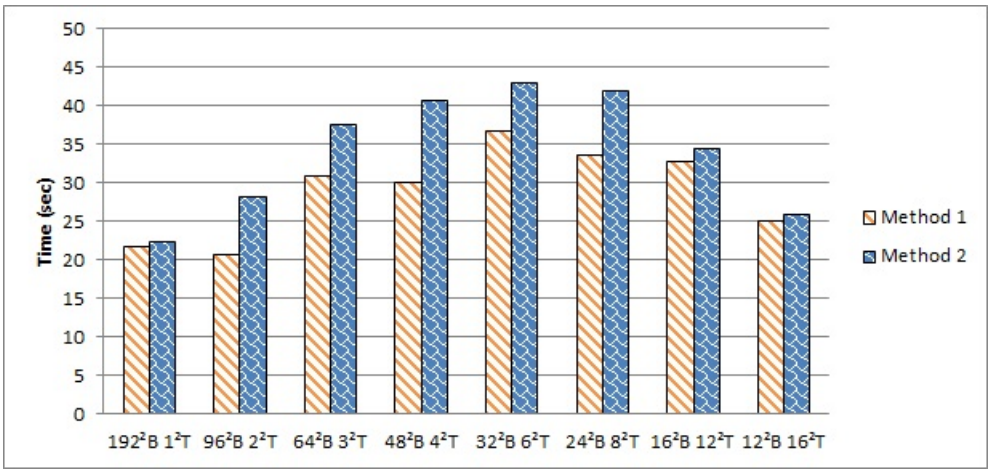

Figure 4.3: Results for a problem of dimension 192 with single precision.

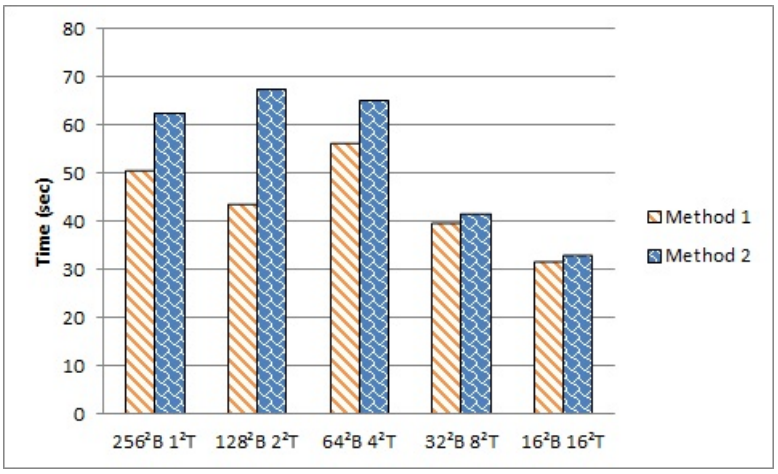

Figure 4.4: Results for a problem of dimension 256 with single precision.

Figure 4.4 presents the results for a cubic grid of $256 \times 256 \times 256$ cells. This is the biggest tested dimension. Both Method 1 and Method 2 perform the best for 16×16 blocks, recording 31.563 seconds and 32.929 seconds respectively.

The curves for different dimensions do not seem to have a behaviour in common.

4.2 Double precision

Figure 4.5 presents the results for a grid of $64 \times 64 \times 64$ cells. Method 1 and Method 2 are compared. Method 1 has the best timing, 0.782 seconds approximately, when the grid is divided into 4×4 blocks. Method 2 performs the best for a 4×4

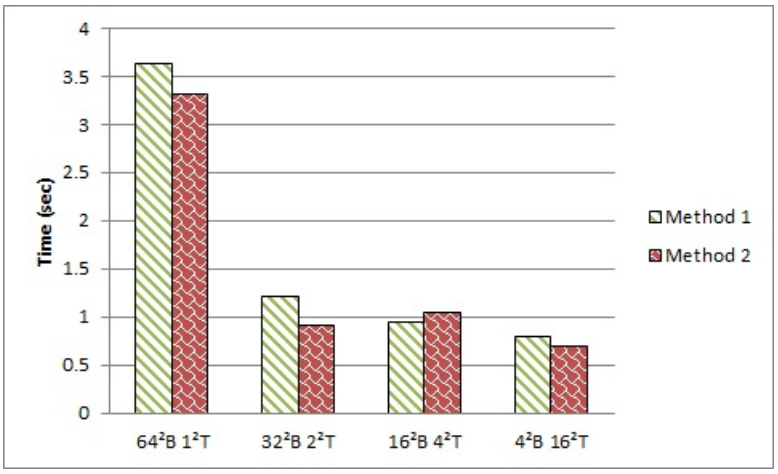

Figure 4.5: Results for a problem of dimension 64 with double precision.

blocks division, with 0.684 seconds.

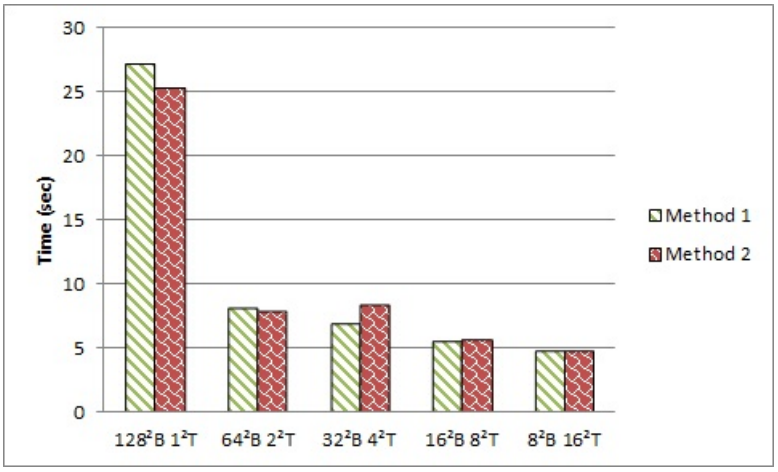

Figure 4.6: Results for a problem of dimension 128 with double precision.

Figure 4.6 shows the results when the dimension is 128. The smallest time of Method 1 is 4.693 seconds, when the program is executed on 8×8 blocks. Method 2 gets the best performance when the grid is divided into 8×8 blocks. It runs in 4.652 seconds.

In Figure 4.7, we see the execution times for different methods of the grid of dimension 192. The best time for Method 1 is 23.190 seconds, when the problem grid is shared between 12×12 blocks with 16×16 threads each. Method 2 shows a peak in performance for 12×12 blocks, with 24.165 seconds.

Figure 4.8 presents the results for a cubic grid of $256 \times 256 \times 256$ cells. Both Method 1 and Method 2 perform the best for 16×16 blocks, recording 34.670

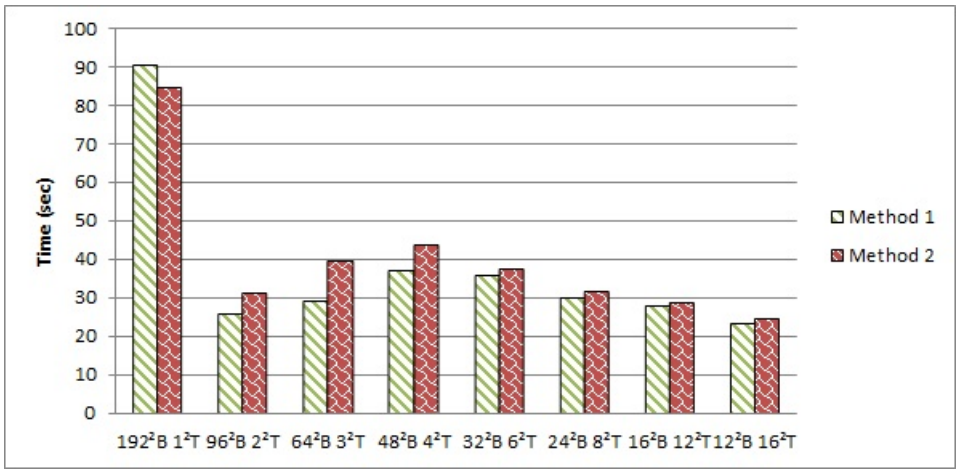

Figure 4.7: Results for a problem of dimension 192 with double precision.

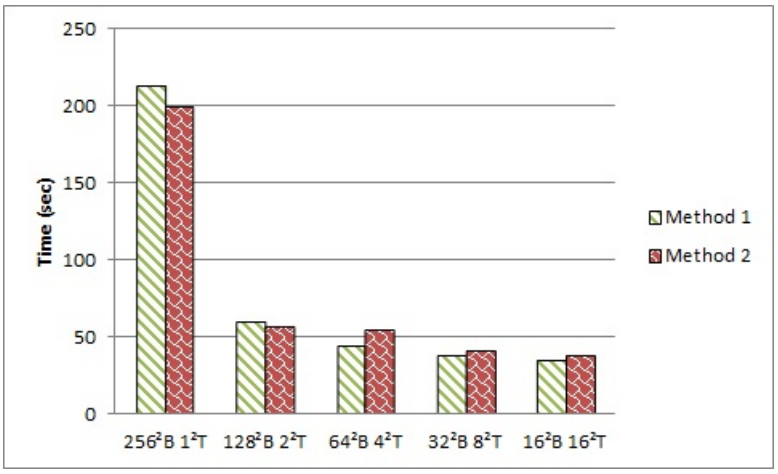

Figure 4.8: Results for a problem of dimension 256 with double precision.

seconds and 38.291 seconds respectively.

Here again, we do not detect a tendency in common between the different curves. However, in double precision, Method 1 and Method 2 get their best results for the same divisions. It seems like the bigger blocks are, the better the timings are.

4.3 Comparison of the three approaches

Figure 4.9 compares the execution times of the three approaches, for dimensions 64, 128, 192 and 256, in single precision. For each dimension, we select the best division for Method 1 and Method 2. In dimension 64, we take the 32×32 blocks

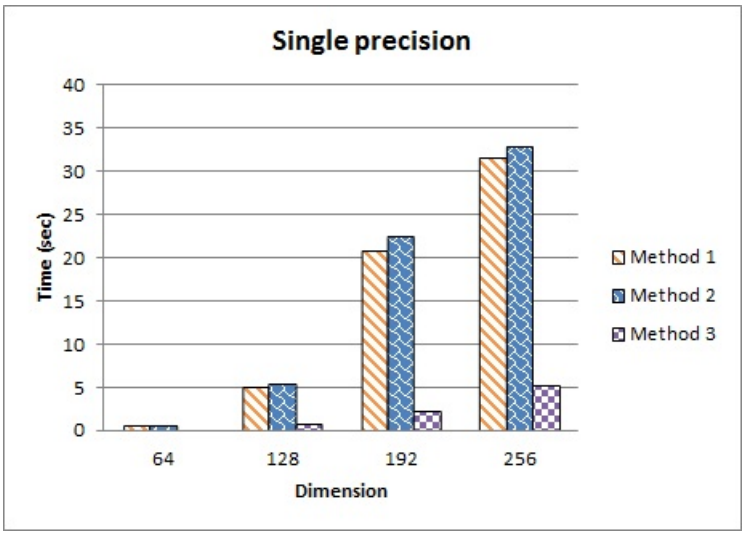

Figure 4.9: Execution time (seconds) for problems of dimension 64, 128, 192, and 256, with single precision, with Approaches 1,2 and 3.

division for Method 1, and 4×4 blocks division for Method 2. For dimension 128, Method 1 is executed on a grid of 64×64 blocks, and Method 2 on 8×8 blocks. Method 1 works on 96×96 blocks in dimension 192, and Method 2 on 192×192 blocks. Finally, in dimension 256 Method 1 and Method 2 have 16×16 blocks. For dimension 64, the timing of Method 3 is so small that it does not appear on the graph: Method 3 spends 0.087 seconds while Method 1 has 0.590 seconds approximately, and Method 2 has about 0.599 seconds. Method 3 outperforms the two other approaches. For a dimension of 256, Method 3 runs about 6 times faster than Method 1, and 6.3 times faster than Method 2.

Overall, as seen in Sections 4.1 and 4.2, Method 1 has better time results than Method 2. This is unexpected, because Method 1 launches the kernels multiple times. The kernels are executed on one layer at the time, it means that the threads have to synchronize at the end of each layer. In Method 2, the threads synchronize only once, when they reach the top of the grid. We measured the execution time of a thread in Method 1 and Method 2. When we add the times for each layer, we get a smaller time than the total execution time, which confirms that there is some synchronization overhead. However, the total time is still smaller than the execution time of Method 2. Load imbalance occurs when some threads have to execute more tasks than others. Theoretically, every thread in Method 2 has the same amount of work to do, so we discard the hypothesis of load imbalance.

Another hypothesis is memory access overhead. We did not use shared memory to "cache" the data. Nevertheless, when threads access consecutive data in memory, they can do it in a warp fashion that increases the performance. In Method 1, threads begin together at each layer. Thus they access memory efficiently, because their warps are coordinated. One hypothesis is that in Method 2, threads have a common behaviour at the beginning. But through time, some begin to be late and the gap between the more advanced thread and the least one gets bigger. Warp accesses lose their asset. To test our hypothesis, we added the __syncthreads() function at the beginning of both kernels in Method 2. __syncthreads() is the equivalent of a barrier mechanism. When a thread comes to __syncthreads(), it has to wait that all the other threads in its blocks arrive to __syncthreads() before pursuing. We ran Method 1, Method 2 and Method 2 with __syncthreads() once for 500 time-steps in dimension 256. The results infirmed this hypothesis. Not only the synchronised version of Method 2 did not perform better than Method 1, but it had worse results than the unsynchronised version of Method 2.

Kirk and Hwu highlight that memory access is often the limiting factor for parallelism [19]. We check the number of registers allocated to each thread with the compilation option `--ptxas-options=-v`. The compiler automatically chooses the number of registers. Method 1 has 12 registers per thread in single precision, and Method 2 has 17 registers per thread. We use the CUDA GPU Occupancy calculator to track the occupancy of our program. Figure 4.10 shows the NVIDIA's CUDA GPU Occupancy calculator. This calculator is just an excel spreadsheet, where the user specifies the compute capability (green line of Figure 4.10), and the resource usage (orange lines). We entered the values for Method 1 where the grid is divided into 16×16 blocks. If there are 16×16 blocks, then each block contains $16 \times 16 = 256$ threads. The compiler allocates 12 registers per thread, and we did not use shared memory. The blue lines and yellow lines give the results of the occupancy analysis. In this configuration, the occupancy is maximum. Each multiprocessor has 1024 active threads, that is 32 warps. The limitation comes from the maximum number of warps per multiprocessor. A block contains 8 warps, and on Tesla T10 chip a multiprocessor cannot handle more than 32 warps at once, so the number of block is limited to 4, as highlighted in red. If we look to the registers, there are 12 registers per thread, that is $12 \times 1024 = 3,072$ allocated registers per multiprocessor. A multiprocessor has

a maximum of 16,384 registers, or $16384 \div 3072 = 5.33$ blocks, rounded to 5. The shared memory is not a limiting factor because we do not use it, so the result is 8 blocks as it is the maximum number of active blocks on a multiprocessor. So, the first limiting factor here is the number of warps.

We also did the test with Method 2, with a division into 16×16 blocks to be in the same configuration. The compiler changed the number of registers to 17 registers per threads. The occupancy drops to 75%. We get a maximum of 768 active threads per multiprocessor, instead of 1024, which is 3 active blocks. There is still a maximum of 4 blocks per multiprocessor according to the possible number of warps, and 8 according to the use of shared memory, but now the registers limit the number of blocks to 3 instead of 5 like in Method 1. 3 blocks $\times$256 threads $\times$17 registers = 13026 registers, under the limit of 16,384, but with one block more it needs 17,378 registers, which exceeds the limit. We saw in Section 2.3 that the adjustment is done at a block granularity, so we lose an entire block to fit the hardware specifications. Method 2 needs more registers because threads store the value i inside their registers, while in Method 1 i is passed as an argument of the kernel.

Method 3 uses 17 registers per thread too, but runs much faster than Method 2, whereas it uses 75% of the occupancy when we allocate 256 threads per block. A plausible hypothesis is that Method 3 has more warps than Method 2 and thus it is easier to hide latency. For example, in dimension 256, Method 2 has $256 \times 256 \div 32 = 2048$ warps in total. In Method 3 with 256 threads per column, there is $256 \times 256 \times 256 \div 32 = 524,288$ warps, 256 times more than in Method 2. To compare the impact of the number of threads, we execute on a problem of dimension 256: Method 3 with 256 threads per block, Method 3 with one thread per block and Method 2 on 256×256 blocks. The two last configurations are equivalent. Method 3 with 256 threads runs in 25 seconds approximately. Method 3 with one thread per blocks runs in 312.42 seconds, very close to the execution time of 312.22 seconds for Method 2. These experiments confirm that warps hide latency, contributing to the improvement of the computational efficiency. However, note that for this test the occupancy dropped to 25% for the last two configurations.

CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	**1.3**	(Help)
2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	12	
Shared Memory Per Block (bytes)	0	
(Don't edit anything below this line)		
3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	**1024**	(Help)
Active Warps per Multiprocessor	**32**	
Active Thread Blocks per Multiprocessor	**4**	
Occupancy of each Multiprocessor	**100%**	

Physical Limits for GPU Compute Capability:	**1.3**
Threads per Warp	32
Warps per Multiprocessor	32
Threads per Multiprocessor	1024
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	16384
Register allocation unit size	512
Register allocation granularity	block
Shared Memory per Multiprocessor (bytes)	16384
Shared Memory Allocation unit size	512
Warp allocation granularity (for register allocation)	2

Allocation Per Thread Block	
Warps	8
Registers	3072
Shared Memory	0

These data are used in computing the occupancy data in blue

Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Blocks per Multiprocessor	**4**
Limited by Registers per Multiprocessor	5
Limited by Shared Memory per Multiprocessor	8

Thread Block Limit Per Multiprocessor highlighted **RED**

Figure 4.10: NVIDIA's CUDA GPU Occupancy calculator for Method 1.

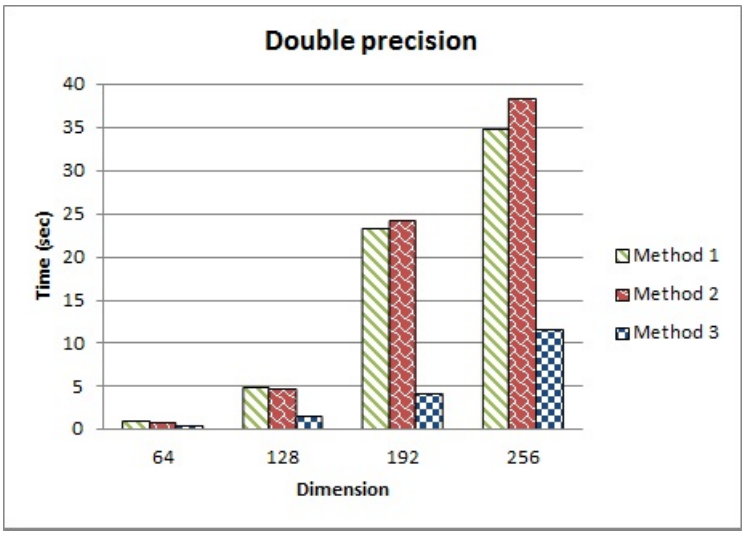

Figure 4.11: Execution time (seconds) for problems of dimension 64, 128, 192, and 256, with double precision, with Methods 1,2 and 3.

Figure 4.11 compares the timings in double precision. Here again, Method 3 is the most efficient for every dimensions. For each dimension, we selected the best division for Method 1 and Method 2. In dimension 64, we take the 4×4 blocks division for Method 1, and 4×4 blocks division for Method 2. For dimension 128, Method 1 is executed on a grid of 8×8 blocks, and Method 2 on 8×8 blocks. Method 1 works on 12×12 blocks in dimension 192, and Method 2 on 12×12 blocks. Finally, in dimension 256 Method 1 and Method 2 have 16×16 blocks. We remark that the best performance for Method 1 and Method 2 happens when the blocks have the biggest number of threads. The best results are always when blocks have a size of 16×16 threads. However, even their best timings cannot surpass Method 3. At dimension 256, the speedup from Method 1 to Method 3 is 3, and from Method 2 to Method 3 it is 3.14.

In double precision, Method 1 uses 33 registers and Methods 2 and 3 use 43 registers. The occupancy for the three approaches drops to 25%. Only one block can execute on a multiprocessor. However, Method 3 still performs much better than Methods 1 and 2. We think that again this is due to its better latency hiding. The program of Method 3 has 256 times more warps than those of Method 1 and Method 2, thus global memory accesses can be hidden.

Table 4.1 presents the summary of the best timings, and Table 4.2 records the configurations that give the best results.

	Single precision		
Dimension	**Method 1**	**Method 2**	**Method 3**
64	0.5897158	0.5986422	0.0868365
128	5.08712925	5.4587824	0.6738065
192	20.764266	22.4529935	2.2033141
256	31.5632431	32.9286777	5.2363695

	Double precision		
Dimension	**Method 1**	**Method 2**	**Method 3**
64	0.7819706	0.6839819	0.2117402
128	4.6926335	4.6524887	1.3504614
192	23.189912	24.165326	4.0018935
256	34.670661	38.291403	11.5300783

Table 4.1: Summary of the best results for each method in each dimension.

	Single precision		
Dimension	**Method 1**	**Method 2**	**Method 3**
64	32x32B 2x2T	4x4B 16x16T	64x64B 64T
128	64x64B 2x2T	8x8B 16x16T	128x128B 128T
192	96x96B 2x2T	192x192B 1x1T	192x192B 128T
256	16x16B 16x16T	16x16B 16x16T	256x256B 256T

	Double precision		
Dimension	**Method 1**	**Method 2**	**Method 3**
64	4x4B 16x16T	4x4B 16x16T	64x64B 64T
128	8x8B 16x16T	8x8B 16x16T	128x128B 64T
192	12x12B 16x16T	12x12B 16x16T	192x192B 64T
256	16x16B 16x16T	16x16B 16x16T	256x256B 256T

Table 4.2: Summary of the best configurations for each method in each dimension.

We presented the single and double precisions results. Method 3 is by far the best of the three approaches. We see here that it is important to pay attention to hardware specifications such as registers limitation, and CUDA's behaviour such as warps and latency hiding. They have a huge impact on the performance. Chapter 5 will conclude on our work and observations, and we will propose different open questions and further work.

Chapter 5

Conclusion and prospective work

Parallel computing is well suitable to execute the Finite-Difference Time-Domain method. The FDTD method discretizes the space into a grid and approximates Maxwell's equations. It is computationally intensive so numerous attempts have been made to reduce its execution time. Recently, programming on GPU has been generalized and a new kind of programming emerged: General Purpose computing on GPU. Now, one can use the power of GPU without being a computer graphics expert. GPU have a high bandwidth and can process a big amount of data in parallel. Thus, they have been of great interest to accelerate the FDTD method.

In [1], Livesey worked on the difference between Streaming SIMD Extensions, Tesla implementation and Fermi implementation of the FDTD method. In our project, we continued that research. In Chapter 2 we presented the necessary background that supports our project. We presented the mathematical functions used in our programs. We detailed the architecture of Tesla GPUs and the different types of memory. We spoke about the Compute Unified Device Architecture. And finally, we reviewed the state of the art for computing the FDTD method on GPUs.

In Chapters 3 and 4, we explored three different approaches for implementing the FDTD method on a Tesla GPU. The codes for every methods are usable and consist in a major deliverable of this project. We did not work on a Fermi GPU due to financial resources. The first two methods focus on block-parallelism and the third method uses parallelism at a block level and a thread level. As expected, the third method performs the best, it is at least six times faster than the others in single precision, and has a speed-up of at least three in double precision. It was found that this efficiency was explained by the better parallelism at both

block and thread levels, but also by a bigger number of threads. Numerous warps make latency hiding easier. We observed that the first method was faster than the second one, which was unexpected. We proved with an occupancy calculator that the first method used less registers and therefore could execute more threads at once than the second method. Moreover, we stated that global memory accesses are costly and that there is little data reuse. In order to reduce the number of global memory accesses, we studied the impact of shared and constant memories. We showed that constant memory was inefficient because variables were already stored in registers, the fastest available memory. With the help of the compute to global memory access ratio, we proved that shared memory would improve a little the performance, at the cost of code complexity and additional synchronization mechanisms.

We examined different combinations for blocks and threads, but blocks had to be distributed in one or two dimensions. For compute capability 2.0 and after, GPU grid can have a three-dimensional block distribution. This, together with better specifications, open new opportunities to the FDTD mapping and its performance. For example, we can imagine that we cut the problem grid into block cubes, filled with threads that compute one cell each. Moreover, we have worked on a machine that has two T10 GPUs, but we have used only one of them. The use of two GPUs may increase the performance because more threads can be executed in parallel. We have seen that the FDTD method requires a lot of memory space. The problem grid must be entirely stored in the global memory. The use of two or more devices allows as one chooses: to divide the data requirements between them, or to multiply the size of the grid according to their global memory capacity. However, if the grid is divided between several GPUs, the programmer will have to handle data transfer between them. Indeed, the boundaries will be updated at each time-step so it is important to keep the data consistent abroad the different devices. Thus, a natural continuation of our work would be to use multiple devices and study their performance. Finally, we could increase the complexity of the problem with different electromagnetic parameters depending on the position in space, in order to simulate the different types of muscles, bones or flesh.

Bibliography

[1] M.J. Livesey. Accelerating the FDTD method using SSE and graphics processing units. Master's thesis, University of Manchester, 2011.

[2] G. W. Arendash, J. Sanchez-Ramos, T. Mori, M. Mamcarz, M. Runfeldt X.Lin, L. Wang, G.Zhang, V. Sava, J. Tan, and C. Cao. Electromagnetic field treatment protects against and reverses cognitive impairment in alzheimer's disease mice. *Journal of Alzheimer's Disease*, 19:191–210, January 2010.

[3] K. Yee. Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. *Antennas and Propagation, IEEE Transactions*, May 1966.

[4] A. Taflove and S.C. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House Publishers, New York, 3 edition, 2005.

[5] H.K. Rouf. *Unconditionally stable finite difference time domain methods for frequency independent media*. PhD thesis, University of Manchester, 2010.

[6] F. Costen. *High Speed Computational Modeling in the Applicaton of UWB Signals*. PhD thesis, University of Manchester, 2005.

[7] A. Taflove. *Advances in Computational Electrodynamics: The Finite-Difference Time-Domain Method (Artech House Antenna Library)*. Artech House Inc, 1998.

[8] U.S. Inan and R.A. Marshall. *Numerical Electromagnetics: The FDTD Method*. Cambridge University Press, 2011.

[9] T. Nagaoka and S. Watanabe. A GPU-based calculation using the three-dimensional FDTD method for electromagnetic field analysis. In *Engineering in Medicine and Biology Society (EMBC), 2010 Annual International Conference of the IEEE*, pages 327 –330, 31 2010-sept. 4 2010.

[10] T.B.A. Senior, J.L. Volakis, and Institution of Electrical Engineers. *Approximate Boundary Conditions in Electromagnetics*. IEE Electromagnetic Waves Series. Institution of Electrical Engineers, 1995.

[11] S.V. Yuferev and N. Ida. *Surface Impedance Boundary Conditions: A Comprehensive Approach*. CRC Press/Taylor & Francis, 2009.

[12] M. Pharr and R. Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. 2005.

[13] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39 –55, March-April 2008.

[14] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[15] NVIDIA. NVIDIA's Next Generation CUDA Compute Archiecture: Fermi. 2009.

[16] Z. Bo, X. , Zheng-hui, R. Wu, L. Wei-ming, and S. Xin-qing. Accelerating FDTD algorithm using GPU computing. In *Microwave Technology & Computational Electromagnetics (ICMTCE), 2011 IEEE International Conference on*, pages 410–413, may 2011.

[17] S. Adams, J. Payne, and R. Boppana. Finite difference time domain (FDTD) simulations using graphics processors. In *Proceedings of the 2007 DoD High Performance Computing Modernization Program Users Group Conference*, HPCMP-UGC '07, pages 334–338, Washington, DC, USA, 2007. IEEE Computer Society.

[18] S.E. Krakiwsky, L.E. Turner, and M.M. Okoniewski. Acceleration of finite-difference time-domain (FDTD) using graphics processor units (GPU). In *Microwave Symposium Digest, 2004 IEEE MTT-S International*, volume 2, pages 1033 – 1036 Vol.2, june 2004.

[19] D.B. Kirk and W.W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series)*. Morgan Kaufmann, 2010.

[20] T. Lanfear. HPC Computing with CUDA and Tesla hardware. Technical report, 2009. `http://www.slideshare.net/maheshkha/cuda-tutorial`.

[21] J. Luitjens and S. Rennich. CUDA Warps and Occupancy. Technical report, 2011. `http://developer.download.nvidia.com/CUDA/training/cuda_webinars_WarpsAndOccupancy.pdf`.

[22] Y. Kreinin. SIMD < SIMT < SMT: parallelism in NVIDIA GPUs. Technical report, November 2011. `http://www.yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html`.

[23] M. Giles. Lecture 3: control flow and synchronisation, July 2012. `http://people.maths.ox.ac.uk/gilesm/cuda/lecs/lec3-2x2.pdf`.

[24] D. De Donno, A. Esposito, L. Tarricone, and L. Catarinucci. Introduction to GPU Computing and CUDA Programming: A Case Study on FDTD [EM Programmer's Notebook]. *Antennas and Propagation Magazine, IEEE*, 52(3):116 –122, june 2010.

[25] NVIDIA. NVIDIA CUDA C Programming Guide 4.2. Technical report, March 2011.

[26] J.F. Stack. Accelerating the finite difference time domain (FDTD) method with CUDA. *Appl. Comput. Electromagn. Soc. Conf*, 2011.

[27] R. Amorim, G. Haase, M. Liebmann, and R. Weber dos Santos. Comparing CUDA and OpenGL implementations for a Jacobi iteration. *High Performance Computing & Simulation, 2009. HPCS '09. International Conference*, pages 22–32, June 2009. 10.1109/HPCSIM.2009.5192847.

[28] G. Leaver. Introduction to OpenCL, June/July 2012. `http://wiki.rcs.manchester.ac.uk/community/OpenCL`.

[29] V. Demir and A. Elsherbeni. Compute Unified Device Architecture (CUDA) Based Finite- Difference Time-Domain (FDTD) Implementation. *Journal of the Applied Computational Electromagnetics Society (ACES)*, 25(4), March 2010.

[30] J. Chi, F. Liu, E. Weber, Y. Li, and S. Crozier. GPU-accelerated FDTD modeling of radio-frequency field tissue interactions in high-field MRI. *IEEE TRANS. Biomed. Eng.*, 58(6):1789–1796, june 2011.

[31] P. Sypek, A. Dziekonski, and M. Mrozowski. How to render FDTD computations more effective using a graphics accelerator. *Magnetics, IEEE Transactions on*, 45(3):1324 –1327, march 2009.

Appendix A

Compact disc

A compact disc is provided with this dissertation. It includes the following material.

- The source code for the new and old versions of Method 1 and Method 2.
- The compilation script and batch execution scripts used to implement and test the work presented in this project.
- The raw data used for our experiments in text format.
- The charts and data in Windows Excel format(.xlsx).
- An electronic copy of this document.

Appendix B

Importance of mapping

At some point during the write up of this dissertation, it appeared that a better implementation of Method 1 and Method 2 was possible. We coded the new implementations. All the presentation is based on the new implementation. The implementations in Chapter 3 and the results in Chapter 4 are based on the new versions. However, several experiments had already been done on the old versions, so we expose them in this appendix. Appendix B presents the difference between the two versions of Methods 1 and 2.

B.1 Difference between old and new implementations

Figure B.1 exposes the difference between the old and new implementations on a problem of dimension 4×4. The light grey squares are the cells of the problem grid. The dark grey squares are the border cells. The big red squares show how blocks are mapped to the problem. Both versions presents an implementation where the GPU grid has a 2×2 block distribution. In the old version, the blocks contain the border. Blocks are of size 6×6 threads. In the new version, blocks do not map the padding cells. They have a 2×2 threads distribution.

The first consequence is that for the same dimension, the new implementation needs less threads than the old one. Thus, more blocks can run simultaneously. The performance increases with the new version because of its better parallelism. Secondly, in Methods 1 and 2 one thread maps each cell or column. This means that in the old implementation, some threads are allocated to the padding cells.

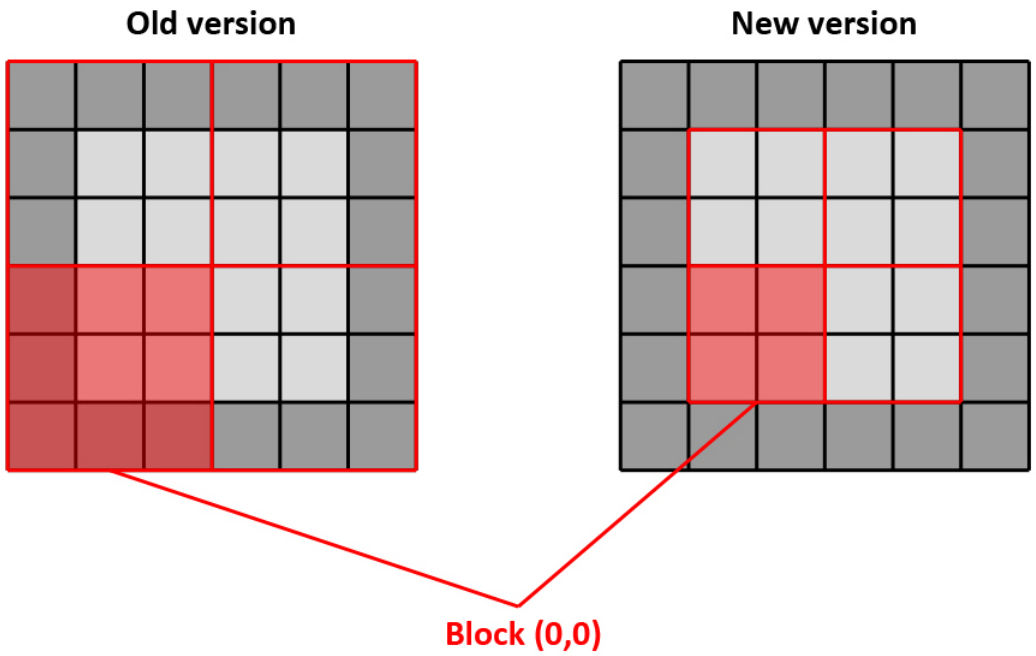

Figure B.1: First implementation (to the left) and new implementation (to the right).

It has two fallouts. First, the border cells must not be computed. We have to introduce an *if* statement to verify if a thread is on the border. Listings B.1 and B.2 show the differences between the old and new codes. This *if* statement affects the performance, increasing with the dimension. Not only an *if* statement inherently adds some execution time, it also exhibits the phenomenon of warp divergence explained in Section 2.3. For example in Figure B.1, in the red block, threads (0,0), (0,1), (0,2), (1,0) and (2,0) will take the *else* path, that is "do nothing", while threads (1,1),(1,2),(2,1) and (2,2) will take the *then* path and execute the FDTD. The threads on the border are left idle, which is a waste of resources. An other disadvantage of the old implementation is that the number of thread is never a multiple of 32. Threads cannot be divided evenly into warps and some useless threads are added to the incomplete warps, wasting even more resources.

Listing B.1: Old version of the electric kernel in Method 2.

```
1 int i,j,k;
2 int dim=gridDim.x*blockDim.x - 2;
3
```

```
4  j=(blockIdx.x) * blockDim.x + threadIdx.x;
5  k=(blockIdx.y) * blockDim.y + threadIdx.y;
6
7  for(i=1; i<=dim; i++){
8    if(k != 0 && j != 0 && k != dim+1 && j != dim+1){
9      // Calculate Ex(i,j,k), Ey(i,j,k), Ez(i,j,k)
10   }
11 }
```

Listing B.2: New version of the electric kernel in Method 2.

```
1  int i,j,k;
2  int dim=gridDim.x*blockDim.x;
3
4  j=(blockIdx.x) * blockDim.x + threadIdx.x + 1;
5  k=(blockIdx.y) * blockDim.y + threadIdx.y + 1;
6
7  for(i=1; i<=dim; i++){
8    // Calculate Ex(i,j,k), Ey(i,j,k), Ez(i,j,k)
9  }
```

Lines 2 of Listings B.1 and B.2 are different. In the example of figure B.1, at the left $gridDim.x = 2$ and $blockDim.x = 3$, at the right $gridDim.x = 2$ and $blockDim.x = 2$. The dimension of the problem is 4, so we want $dim = 4$ and we have to subtract 2 from the old version. Lines 4 and 5 of Listing B.1 also differ from Listing B.2. In the new method we add 1 to shift the values and avoid the border. In the old version, the *if* statement is checking that a thread is or is not on the border so we do not need to shift the indices.

The border must not be computed. In Method 3, we look at the block coordinates to know if the program passes by the border or not. This is possible because blocks are of size 1×1. In Method 1 and Method 2, the blocks can have a bigger size. In the old implementations, blocks at the border contain some border cells and some cells that need to be computed. If we shift the blocks, we ignore some of the cells. For example, in Method 3 the block (0,0) is ignored. In Figure B.1 at the left, the block (0,0) has four cells that has to be calculated, and five that should not. So the block coordinates are not sufficient to decide whether the block should be ignored or not. We have to base our approach on

thread identities. For instance, the thread (0,0) will not be computed.

In CUDA, all blocks have the same threads distribution. In old Method 1 and Method 2, this means that inside a block some threads will be idle. Indeed, depending on the blocks division, some blocks can contain idle threads and active threads, and other blocks can be full of active threads. In Figure B.1, all the blocks in the old versions have fives padding cells, but at a bigger dimension it is easier to get some blocks with full activity. Thus, not only there is load imbalance between threads, but it leads to load imbalance between blocks.

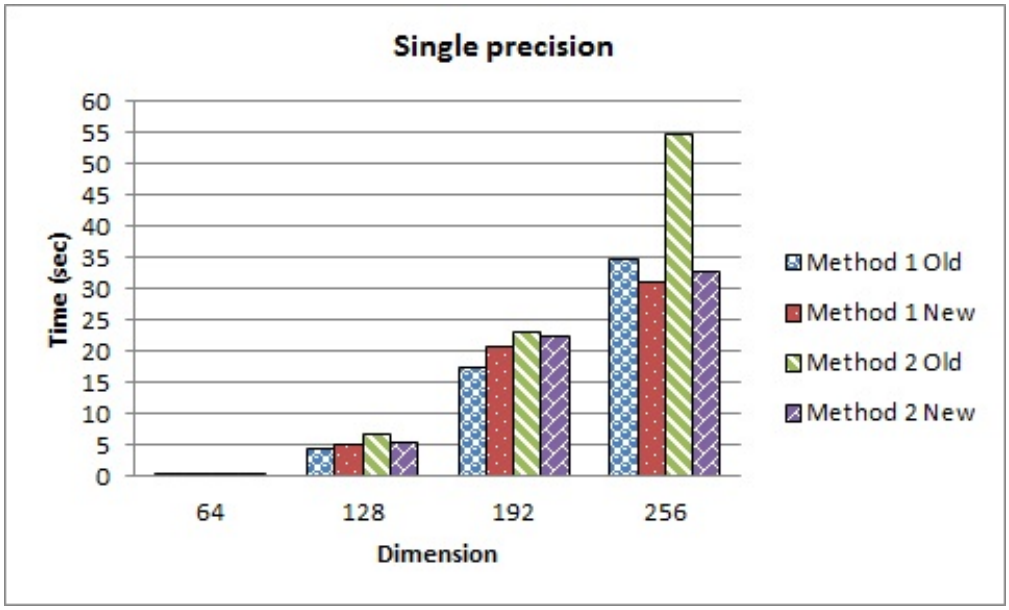

Figure B.2: Executing times for old and new implementations of Method 1 and Method 2, in single precision.

Figures B.2 and B.3 present the execution times in single and double precisions of Method 1 and Method 2 in the two versions. We focus on dimension 256 where the difference is more apparent. In single precision, old Method 1 runs on 129×129 blocks with 2×2 threads each. It is normal that $129 \times 2 = 258$ and not 256 because we count the padding cells too, so $SQRT\_BLOCKS \times SQRT\_THREADS = DIM + 2$. The new versions of Methods 1 and 2 use 16×16 blocks with 16×16 threads (here $SQRT\_BLOCKS \times SQRT\_THREADS = DIM$). The grid in the old version of Method 2 is divided into 43×43 blocks with 6×6 threads. Method 1 improves by 3.69 seconds and Method 2 by 22 seconds. In double precision, the new version of Methods 1 and 2 is executed on 16×16 blocks. The old version of

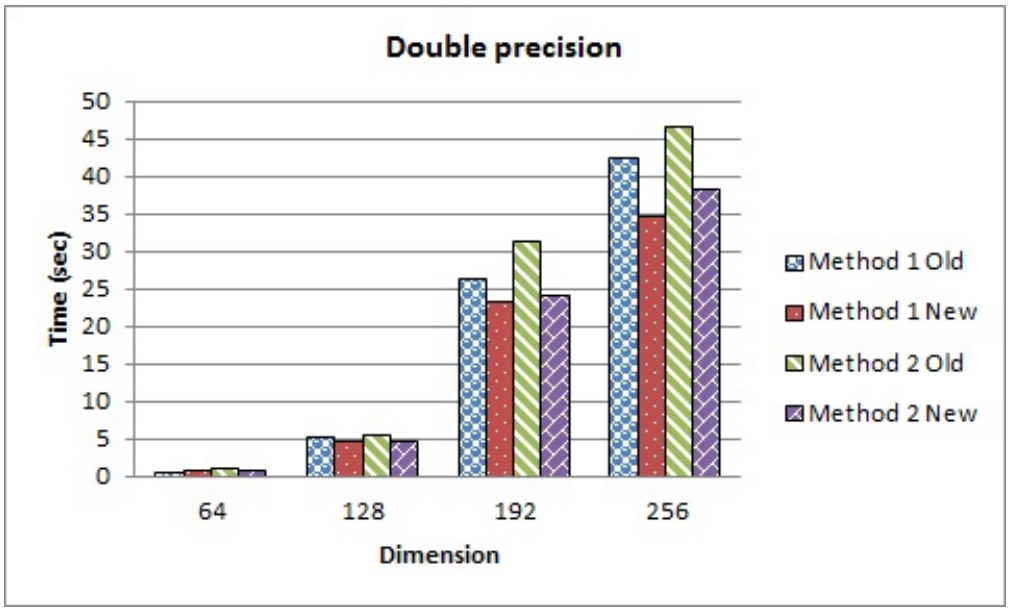

Figure B.3: Executing times for old and new implementations of Method 1 and Method 2, in double precision.

Methods 1 and 2 run on 86×86 blocks with 3×3 threads. The timing of Method 1 is shorter by 7.7 seconds and Method 2 by 8.23 seconds.

Old versions always take more time that new versions of Methods 1 and 2. There is a load imbalance overhead in the old versions. Threads on the border do not do anything, while threads on the grid have 256 cells to calculate. New Method 1 has an occupancy of 100%, and the new Method 2 has an occupancy of 75% as stated in Section 4.3, whereas the old versions perform at 25%. Old Method 1 has 2×2 threads per blocks: blocks at the corners have 3 idle threads and 1 active thread, at the border they have 2 active and 2 idle threads, and otherwise they have 4 active threads. In total, 1028 threads do nothing, and delay the execution. Because blocks are small (old Method 1 has 4 threads per block and new Method 1 has 256), the number of threads and registers are not a limiting factor. However, a multiprocessor cannot run more than 8 blocks, so even if there is a lot of space in term of threads and registers capacity, the multiprocessor runs 8 blocks, that is $4 \times 8 = 32$ threads, far away from the 1024 maximum capacity. So here the blocks' number is the limiting factor.

DIM = 256 – 1000 time-steps				
	43^2B containing 6^2T		86^2B containing 3^2T	
	M1	M1 const.	M1	M1 const.
Average (sec.)	441.420	444.230	420.807	424.162
With - without const. (sec.)	2.81		3.355	

	129^2B containing 2^2T		258^2B containing 1T	
	M1	M1 const.	M1	M1 const.
Average (sec.)	635.712	637.688	2308.206	2302.684
With - without const. (sec.)	1.976		-5.522	

Table B.1: Method 1.

B.2 Constant memory in the old implementation

We had analysed the impact of constant memory and thought the results may be of interest for the reader. We ran the previous Methods 1 and 2 five times for every distributions on 1000 time-steps, in dimension 256, in double precision. Tables B.1 and B.2 present the impact of constant memory on the execution time of the old version of Method 1 and Method 2. In Section 3.4 the constant memory did not improve the performance, because registers are the fastest memory available on Tesla T10 GPU. Here, the results are ambivalent.

In Tables B.1 and B.2 we see the difference between Method 1 and Method 2 execution times with and without a constant variable *dim*. In Section 3.4 it was always better to keep the variable into registers. Here, it is the case except for the division into 258×258 threads. In old Method 2 the constant variable is also beneficial for the 129×129 blocks division. This is unexpected because registers have fast memory access and are not a limiting factor in this configuration. However, there is a difference of 5.5 and 6.8 seconds.

DIM = 256 – 1000 time-steps				
	43^2B containing 6^2T		86^2B containing 3^2T	
	M2	M2 const.	M2	M2 const.
Average (sec.)	475.049	475.376	458.396	462.644
With - without const. (sec.)	0.327		4.248	

	129^2B containing 2^2T		258^2B containing 1T	
	M2	M2 const.	M2	M2 const.
Average (sec.)	668.096	667.510	2453.212	2446.406
With - without const. (sec.)	-0.586		-6.806	

Table B.2: Method 2.