Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática
2012

Nuno Roberto Pereira Mestre

Comparação do desempenho do FDTD com implementação em CPU e em GPU

Universidade de Aveiro
2012

Departamento de Electrónica,
Telecomunicações e Informática

Nuno Roberto Pereira Mestre

Comparação do desempenho do FDTD com implementação em CPU e em GPU

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Mestrado Integrado em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Rui Alves, Professor Auxiliar Convidado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro e do Doutor Pedro Pinho, Professor Adjunto da Área Departamental de Engenharia Electrónica e Telecomunicações e de Computadores do Instituto Superior de Engenharia de Lisboa

Dedico este trabalho aos meus pais e à minha irmã.

O júri

Presidente

Professor Doutor Joaquim João Estrela Ribeiro Silvestre Madeira
Professor Auxiliar da Universidade de Aveiro

Vogais

Professor Doutor Rui Pedro Amaral Rodrigues
Professor Auxiliar Convidado do Departamento de Engenharia Informática da Faculdade de Engenharia da Universidade do Porto

Professor Doutor Rui Pedro de Oliveira Alves
Professor Auxiliar Convidado da Universidade de Aveiro (orientador)

Professor Doutor Pedro Renato Tavares de Pinho
Professor Adjunto da Área Departamental de Engenharia Electrónica e Telecomunicações e de Computadores do Instituto Superior de Engenharia de Lisboa (co-orientador)

agradecimentos / acknowledgements

Agradecimentos

Agradeço ao Professor Rui Alves e ao Professor Pedro Pinho pela sua orientação. Agradeço também aos meus amigos João Campos, Flávio Fonseca, Vasco Santos e Rui Ferreira pelo apoio prestado.

Palavras Chave

FDTD, CPU, GPU, C++, Paralelização, *Multi-Threading*, CUDA, *Hyper-Threading*, *Speedup*

Resumo

O *Finite-Difference Time-Domain* é um método utilizado em electromagnetismo computacional para simular a propagação de ondas electromagnéticas em meios cujas características podem não ser uniformes. É um método com inúmeras aplicações, e como tal é vantajoso que o seu desempenho possa ser aumentado, de preferência recorrendo a sistemas computacionais de baixo custo.

O propósito desta dissertação é aproveitar duas tecnologias emergentes e de relativo baixo custo para aumentar o desempenho do FDTD em uma e duas dimensões. Essas tecnologias são sistemas com processadores *Multi-Core* e placas gráficas que permitem utilizar as suas características de processamento massivamente paralelo para a execução de código de propósito geral. Para explorar as capacidades de um sistema com processador *Multi-Core*, o algoritmo originalmente sequencial foi alterado de modo a ser executado em múltiplas *thread*s. Por sua vez, para tirar partido da tecnologia CUDA, o algoritmo foi convertido de forma a ser executado num GPU.

Os acréscimos de desempenho obtidos indicam que é vantajoso o uso destas tecnologias comparativamente com implementações puramente sequenciais.

Keywords FDTD, CPU, GPU, C++, Paralelization, Multi-Threading, CUDA, Hyper-Threading, Speedup

Abstract

The Finite-Difference Time-Domain is a method used in computational electromagnetics to simulate the propagation of electromagnetic waves in fields that might not have uniform characteristics. It is a method with countless applications and so it is advantageous to increase its performance, preferably using low cost computer systems.

The purpose of this thesis is to make use of two relatively low-cost emerging technologies to increase the FDTD performance in one and two dimensions. These technologies are Multi-Core systems and graphics cards that allow the use of its massive parallel processing characteristics to run general purpose code. To make use of a Multi-Core system, the original sequential code was changed to be executed by multiple threads. In order to use the CUDA technology, the algorithm was converted so that it could be executed on the GPU.

The performance increase shows that the use of these technologies is advantageous in comparison to pure sequential implementations.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Acrónimos

ABC *Absorbing Boundary Condition.*

API *Application Programming Interface.*

CPU *Central Processing Unit.*

CUDA *Compute Unified Device Architecture.*

DDR3 *Double Data Rate type Three.*

EMC *Electromagnetic Compatibility.*

FDTD *Finite-Difference Time-Domain.*

GDDR3 *Graphics Double Data Rate type Three.*

GPGPU *General Purpose Programming on* GPU.

GPU *Graphics Processing Unit.*

HT *Hyper-Threading Technology.*

NVCC Nvidia® CUDA Compiler.

PML *Perfectly Matched Layer.*

RAM *Random-Access Memory.*

RCS *Radar Cross Section.*

SIMT *Single-Instruction, Multiple-Thread.*

TEM *Transverse Electromagnetic.*

Capítulo 1

Introdução

Desde a sua introdução por Yee em 1966 [1] que o método *Finite-Difference Time-Domain* (FDTD) adquiriu uma grande importância no campo do electromagnetismo computacional, dadas as suas inúmeras aplicações [2], como por exemplo, entre outras, *Radar Cross Section* (RCS), antenas, *Electromagnetic Compatibility* (EMC) [3] ou bioelectromagnetismo [4]. Também apresenta vantagens computacionais, ao ser um método no domínio do tempo, sobre vários outros métodos [1] que recorrem ao domínio da frequência. No entanto, como os cálculos necessários para o algoritmo FDTD completar a sua execução podem exigir muito tempo, torna-se imperativo que possa ser acelerado. Pelas suas características computacionais, o FDTD é susceptível de ser paralelizado, tirando partido de sistemas que disponham de vários processadores. Paralelizar a execução de um algoritmo, porém, requer um sistema que o possa executar em paralelo, como um supercomputador, um *cluster* ou mesmo algum outro tipo de sistema distribuído que divida a carga computacional por vários processadores com algum tipo de conexão entre si. No entanto, estas soluções são dispendiosas ou podem ser de difícil acesso, pelo que se torna desejável encontrar soluções que a custos muito mais baixos obtenham ganhos significativos de desempenho.

Os constantes avanços tecnológicos nos sistemas computacionais estão a possibilitar cada vez mais que tal se proporcione. Temos como exemplos as duas tecnologias exploradas no decurso desta dissertação. A primeira, os sistemas *Multi-Core*, ganha cada vez mais relevância à medida que se começa a atingir os limites [5] a nível de frequência de relógio e número de transístores por circuito integrado, advindo daí que exista cada vez mais uma maior aposta dos fabricantes neste tipo de *Central Processing Unit* (CPU). A segunda, placas gráficas que permitem tirar partido da natureza já de si altamente paralelizável do *Graphics Processing Unit* (GPU) para aplicações com código de propósito geral, designado como *General Purpose Programming on* GPU (GPGPU) [6], são também uma opção muito interessante. Apesar de o uso de GPU para código de propósito geral não estar limitado a placas gráficas da Nvidia [7, 8], foi a tecnologia *Compute Unified Device Architecture* (CUDA) da Nvidia que foi explorada no âmbito desta dissertação.

1.1 Objectivos

O objectivo do trabalho desta dissertação é tirar partido destas duas tecnologias, sistemas *Multi-Core* e GPU com CUDA, para demonstrar que é possível obter ganhos significativos sobre implementações que recorrem a uma abordagem puramente sequencial. Para tal partiu-se

de uma implementação inicial do algoritmo em MATLAB[1], a partir da qual foi desenvolvida em C++ uma implementação sequencial (*Single-Thread*), destinada a ser executada apenas num processador; posteriormente, uma abordagem paralela que divide o problema em partições que são executadas de forma independente (*thread*s), tirando partido de um sistema *Multi-Core*; e por fim uma implementação em CUDA, com o objectivo de ser executada de forma massivamente paralela em GPU. Estas várias implementações foram desenvolvidas para os casos a uma e duas dimensões do FDTD, tendo como objectivo o aumento do desempenho do FDTD, diminuido os seus tempos de computação.

1.2 Organização do relatório

Esta dissertação é composta por seis capítulos. No primeiro capítulo é feita uma introdução ao tema da dissertação e aos seus objectivos e é apresentada a sua estrutura. No segundo capítulo é feita uma descrição do algoritmo FDTD. No terceiro capítulo são abordadas em detalhe as características do CPU e GPU utilizados para a execução das simulações, assim como uma descrição das tecnologias que apresentam e que foram exploradas no decurso do trabalho desta dissertação. No quarto capítulo é descrita e analisada a implementação do algoritmo FDTD a uma dimensão. No quinto capítulo é descrita e analisada a implementação do algoritmo FDTD a duas dimensões. Finalmente, no sexto capítulo são expostas as conclusões e as propostas de trabalho futuro.

[1] A implementação inicial em MATLAB foi fornecida pelo co-orientador desta dissertação, o Professor Doutor Pedro Pinho.

Capítulo 2

Finite-difference time-domain

Introduzido por Yee em 1966 [1], o método FDTD revelou-se ser conceptualmente simples; ainda assim, provou ser um método eficaz e poderoso na obtenção de soluções numéricas para problemas de propagação electromagnética [9, 10].

2.1 Equações e radiação de Maxwell a três dimensões

As equações de Maxwell [11] dependentes do tempo numa região do espaço sem imposição de cargas eléctricas ou magnéticas, mas onde podem existir materiais que absorvem energia do campo magnético ou eléctrico, são dadas na sua forma diferencial pelas equações 2.1 a 2.4.

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

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

$$\nabla \cdot D = \rho \tag{2.3}$$

$$\nabla \cdot B = \rho_m \tag{2.4}$$

sendo:

E Campo eléctrico (V/m);

H Campo magnético (A/m);

D Densidade do fluxo eléctrico (C/m^2);

B Densidade do fluxo magnético (Wb/m^2);

J Densidade da corrente eléctrica (A/m^2);

ρ Densidade da carga eléctrica livre (C/m^3);

J_m Densidade da corrente magnética equivalente (V/m^2);

ρ_m Densidade da carga magnética livre (Wb/m^3);

Num meio linear, não dispersivo (a velocidade de propagação é independente da frequência [12, 13]) e anisotrópico (a magnitude de uma propriedade apenas pode ser definida ao longo de uma determinada direcção [14]), as relações constitutivas são dadas pelas equações 2.5 a 2.8.

$$D = \varepsilon_0 \, \overline{\overline{\varepsilon}}_r \, E \tag{2.5}$$

$$B = \mu_0 \, \overline{\overline{\mu}}_r \, H \tag{2.6}$$

$$J = \overline{\overline{\sigma}} \, E \tag{2.7}$$

$$J_m = \overline{\overline{\sigma^*}} \, H \tag{2.8}$$

sendo:

ε_0 Permitividade eléctrica do espaço livre (F/m);

$\overline{\overline{\varepsilon}}_r$ Tensor da permitividade eléctrica relativa;

μ_0 Permeabilidade magnética do espaço livre (H/m);

$\overline{\overline{\mu}}_r$ Tensor da permeabilidade magnética relativa;

$\overline{\overline{\sigma}}$ Tensor da condutividade eléctrica (S/m);

$\overline{\overline{\sigma^*}}$ Tensor da perda magnética equivalente (Ω/m);

Dado ainda não terem sido observadas na natureza, a densidade da carga magnética e condutividade da carga magnética são quantidades não físicas. No entanto, estas quantidades são importantes para implementar fronteiras do tipo *Perfectly Matched Layer* (PML), que atenuam a existência de reflexões por parte das ondas incidentes nas fronteiras dos campos.

Substituindo as equações 2.5 a 2.8 nas equações 2.1 e 2.2 são obtidas as equações rotacionais de Maxwell, 2.9 e 2.10, em materiais lineares, diagonalmente anisotrópicos, não dispersivos e com perdas:

$$\frac{\partial E}{\partial t} = \frac{\overline{\overline{\varepsilon_r}}^{-1}}{\varepsilon_0}(\nabla \times H - \overline{\overline{\sigma}} \, E) \tag{2.9}$$

$$\frac{\partial H}{\partial t} = -\frac{\overline{\overline{\mu_r}}^{-1}}{\mu_0}(\nabla \times E - \overline{\overline{\sigma^*}} \, H) \tag{2.10}$$

Ao escrever estas equações em coordenadas cartesianas obtêm-se as equações escalares 2.11 a 2.16.

$$\frac{\partial E_x}{\partial t} = \frac{1}{\varepsilon_0 \varepsilon_{xx}} \left[\frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z} - \sigma_{xx} E_x \right] \tag{2.11}$$

$$\frac{\partial E_y}{\partial t} = \frac{1}{\varepsilon_0 \varepsilon_{yy}} \left[\frac{\partial H_x}{\partial z} - \frac{\partial H_z}{\partial x} - \sigma_{yy} E_y \right] \tag{2.12}$$

$$\frac{\partial E_z}{\partial t} = \frac{1}{\varepsilon_0 \varepsilon_{zz}} \left[\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} - \sigma_{zz} E_z \right] \tag{2.13}$$

$$\frac{\partial H_x}{\partial t} = \frac{1}{\mu_0 \mu_{xx}} \left[\frac{\partial E_y}{\partial z} - \frac{\partial E_z}{\partial y} - {\sigma^*}_{xx} H_x \right] \tag{2.14}$$

$$\frac{\partial H_y}{\partial t} = \frac{1}{\mu_0 \mu_{yy}} \left[\frac{\partial E_z}{\partial x} - \frac{\partial E_x}{\partial z} - {\sigma^*}_{yy} H_y \right] \tag{2.15}$$

$$\frac{\partial H_z}{\partial t} = \frac{1}{\mu_0 \mu_{zz}} \left[\frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x} - {\sigma^*}_{zz} H_z \right] \tag{2.16}$$

Estas seis equações formam a base do método FDTD [11]. A sua discretização nas equações 2.17 a 2.22 utilizando o método das diferenças finitas origina um algoritmo numérico que modela fenómenos do campo electromagnético no espaço tridimensional.

$$\begin{aligned} &Ex_t(x,y,z) = cax(x,y,z) \cdot Ex_{t-1}(x,y,z) + \\ &+cbx(x,y,z) \cdot (Hz_{t-1/2}(x,y,z) - Hz_{t-1/2}(x,y-1,z)) + \\ &-ccx(x,y,z) \cdot (Hy_{t-1/2}(x,y,z) - Hy_{t-1/2}(x,y,z-1)) \end{aligned} \tag{2.17}$$

$$\begin{aligned} &Ey_t(x,y,z) = cay(x,y,z) \cdot Ey_{t-1}(x,y,z) + \\ &+cby(x,y,z) \cdot (Hx_{t-1/2}(x,y,z) - Hx_{t-1/2}(x,y,z-1)) + \\ &-ccy(x,y,z) \cdot (Hz_{t-1/2}(x,y,z) - Hz_{t-1/2}(x-1,y,z)) \end{aligned} \tag{2.18}$$

$$\begin{aligned} &Ez_t(x,y,z) = caz(x,y,z) \cdot Ez_{t-1}(x,y,z) + \\ &+cbz(x,y,z) \cdot (Hy_{t-1/2}(x,y,z) - Hy_{t-1/2}(x-1,y,z)) + \\ &-ccz(x,y,z) \cdot (Hx_{t-1/2}(x,y,z) - Hx_{t-1/2}(x,y-1,z)) \end{aligned} \tag{2.19}$$

$$\begin{aligned} &Hx_t(x,y,z) = dax(x,y,z) \cdot Hx_{t-1}(x,y,z) + \\ &+dbx(x,y,z) \cdot (Ey_{t-1/2}(x,y,z+1) - Ey_{t-1/2}(x,y,z)) + \\ &-dcx(x,y,z) \cdot (Ez_{t-1/2}(x,y+1,z) - Ez_{t-1/2}(x,y,z)) \end{aligned} \tag{2.20}$$

$$\begin{aligned} &Hy_t(x,y,z) = day(x,y,z) \cdot Hy_{t-1}(x,y,z) + \\ &+dby(x,y,z) \cdot (Ez_{t-1/2}(x+1,y,z) - Ez_{t-1/2}(x,y,z)) + \\ &-dcy(x,y,z) \cdot (Ex_{t-1/2}(x,y,z+1) - Ex_{t-1/2}(x,y,z)) \end{aligned} \tag{2.21}$$

$$\begin{aligned} &Hz_t(x,y,z) = daz(x,y,z) \cdot Hz_{t-1}(x,y,z) + \\ &+dbz(x,y,z) \cdot (Ex_{t-1/2}(x,y+1,z) - Ex_{t-1/2}(x,y,z)) + \\ &-dcz(x,y,z) \cdot (Ey_{t-1/2}(x+1,y,z) - Ey_{t-1/2}(x,y,z)) \end{aligned} \tag{2.22}$$

Partindo das equações 2.17 a 2.22 podem ser obtidas as versões específicas para os casos bidimensional (equações 5.1 a 5.3) e unidimensional (equações 4.1 e 4.2).

É importante referir que a amostragem dos pontos para as componentes dos campos eléctrico e magnético deve ser posicionada de maneira a que as operações numéricas de derivação no espaço estejam de acordo com as leis de Gauss [11]. Yee propôs um esquema de grelha que satisfaz este requesito e que veio a ser conhecido como célula de Yee. A sua estrutura está representada na figura 2.1.

Na figura 2.1 [11] podem ser observadas as posições das componentes vectoriais dos campos magnético e eléctrico da célula com coordenadas (i, j, k). Apenas as componentes do campo eléctrico a azul e do campo magnético a vermelho pertencem a esta célula; as restantes pertencem a células contíguas. O esquema em grelha de Yee posiciona as componentes

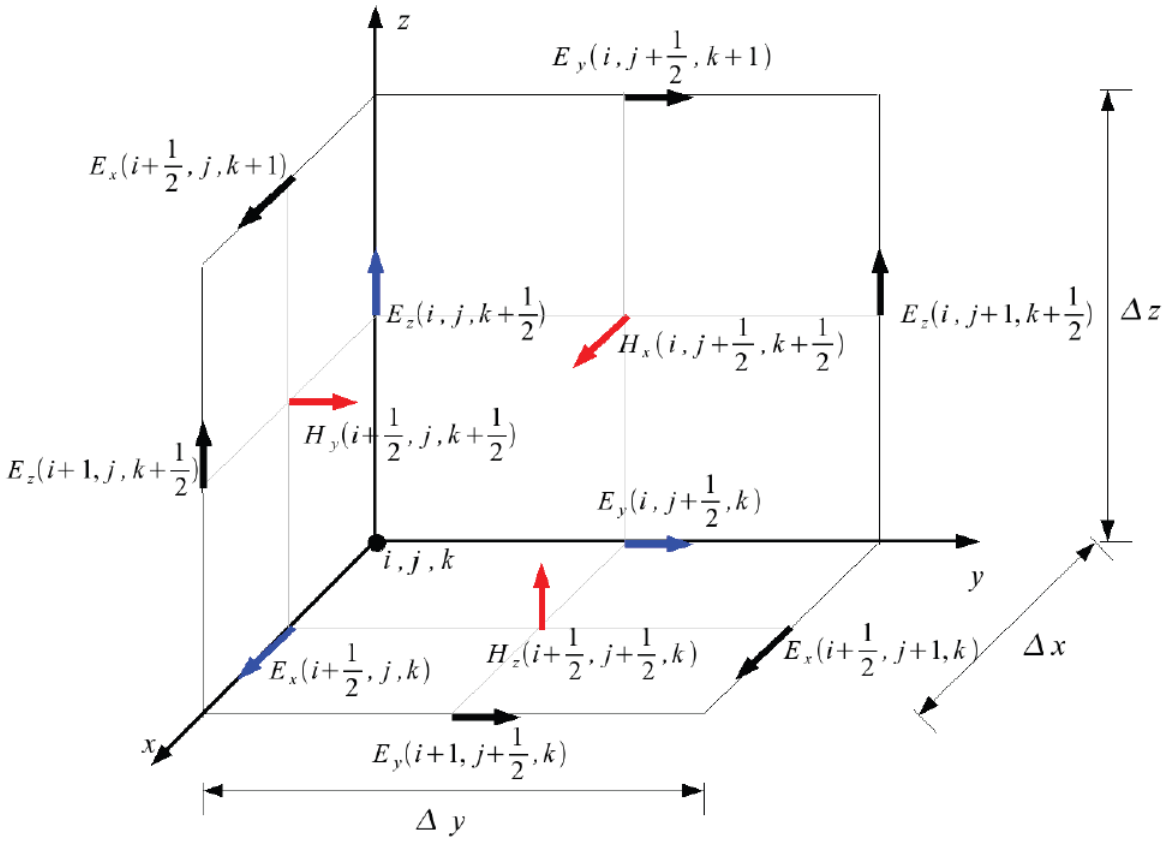

Figura 2.1: Célula de Yee

de campo de forma a que cada componente do campo eléctrico esteja rodeado de quatro componentes do campo magnético e cada componente do campo magnético esteja rodeado de quatro componentes do campo eléctrico, replicando a lei de Ampere e a lei de Faraday ao nível da grelha. No caso particular a duas dimensões, a única componente do campo magnético estará rodeada por duas componentes do campo eléctrico e cada uma das duas componentes do campo eléctrico estará rodeada por apenas essa componente do campo magnético. Para o caso a uma dimensão, apenas existirá uma componente para o campo eléctrico e outra para o campo magnético, pelo que uma componente de um campo apenas dependerá da componente do outro campo.

2.2 Condições fronteira

Um problema que se coloca no método FDTD é o que acontece quando uma onda atinge as fronteiras do espaço computacional [11]. Como as células nas regiões fronteira não estão rodeadas por outras células da mesma forma que estaria uma célula no interior da grelha, é necessário tratar estas células de uma forma especial. Tendo em atenção que todas as células do espaço computacional são inicialmente colocadas com o valor zero, se o valor destas não for actualizado ao longo das várias iterações temporais, vão provocar o efeito de reflectir totalmente as ondas incidentes quer do campo eléctrico quer do campo magnético. Existem

contudo muitos casos em que não se pretende que as ondas sejam reflectidas, mas sim simular a propagação das ondas em regiões abertas do espaço com o mínimo de reflexões possível. Como consequência, é necessário um algoritmo adicional que permita simular um domínio computacional de tamanho infinito. Este tipo de algoritmo designa-se por *Absorbing Boundary Condition* (ABC) e apresenta como objectivo teórico a supressão de reflexões numéricas, apesar de na prática apenas se verificar a atenuação de ondas que saem dos limites do espaço computacional até um nível suficientemente baixo para que os campos na vizinhança não sejam afectados por estas reflexões de forma significativa. Existem dois tipos principais de ABC: as que simulam um material absorvente e as analíticas.

O princípio subjacente às ABC analíticas é a utilização de operadores diferenciais, de acordo com a equação de onda, para forçar as ondas a se propagarem fora do domínio computacional ou para apresentar estimativas dos valores que os campos teriam fora do domínio computacional. Um dos tipos mais populares de ABC analíticas foi desenvolvido por Mur [15, 16], e as razões para esta popularidade são a sua implementação relativamente simples e os seus baixos coeficientes de reflexão. Apesar de existirem outras ABC que apresentam menores coeficientes de reflexão, estas não só são mais difíceis de implementar como também são mais complexas do ponto de vista computacional.

Por sua vez, o princípio que rege as ABC que simulam um material absorvente é rodear o domínio computacional com um material que absorva as ondas que se propagam para fora deste, de forma semelhante às paredes de uma câmara anecóica. No entanto, como de início este tipo de ABC apenas absorviam ondas incidentes no plano normal, as suas aplicações em problemas de electromagnetismo eram muito limitadas.

Porém, Bérenger [17] introduziu em 1994 um ABC, denominado de PML, que simula um material absorvente. A PML seria revolucionária, já que tratando-se de uma ABC que simula um material absorvente, conseguia absorver ondas em planos de incidência, polarização e frequência arbitrárias. A maior vantagem da PML reside no facto de, sendo relativamente simples de implementar, originar menos erros de reflexão do que as ABC analíticas, de implementação muito mais difícil. Adicionalmente, nos últimos anos foram desenvolvidas versões de PML para tratar meios não homogéneos, com perdas, dispersivos e não lineares, algo muito difícil de se obter com ABC do tipo analítico.

Neste trabalho, para o caso da implementação a duas dimensões foi utilizada uma ABC do tipo PML para todas a versões desenvolvidas, excepto uma que foi propositadamente simplificada para se obter uma redução máxima de memória ocupada pela implementação. Essa camada PML em torno da região principal do espaço computacional está representada na figura 2.2. Note-se que que as próprias camadas PML também apresentam fronteiras entre si. Para o caso unidimensional vão exisitir duas células fronteira, uma em cada extremo do domínio computacional, estas células fronteira foram tratadas directamente durante a execução do algoritmo FDTD, não tendo sido necessário o recurso a camadas ABC para evitar reflexões.

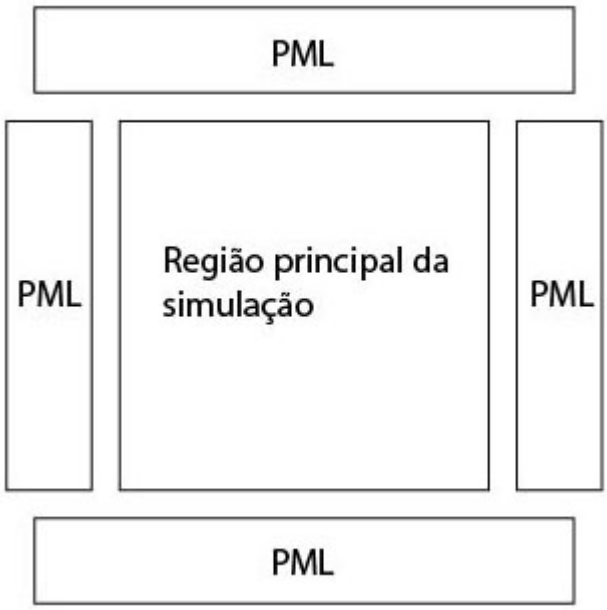

Figura 2.2: Região principal do espaço computacional a duas dimensões rodeada pelas áreas da camada PML (uma para cada fronteira)

2.3 Algoritmo

Uma das características mais importantes do algoritmo de Yee é a forma como as componentes dos campos eléctrico e magnético são calculadas [11, 18]. Esse cálculo acontece para as componentes do campo eléctrico no instante t recorrendo aos valores guardados para as componentes do campo magnético no instante t-$\Delta t/2$. As componentes do campo magnético são então calculadas no instante t+$\Delta t/2$ utilizando os valores guardados para as componentes do campo eléctrico no instante t. Este processo pode ser visualizado na figura 2.3 [11], simplificado para o caso unidimensional.

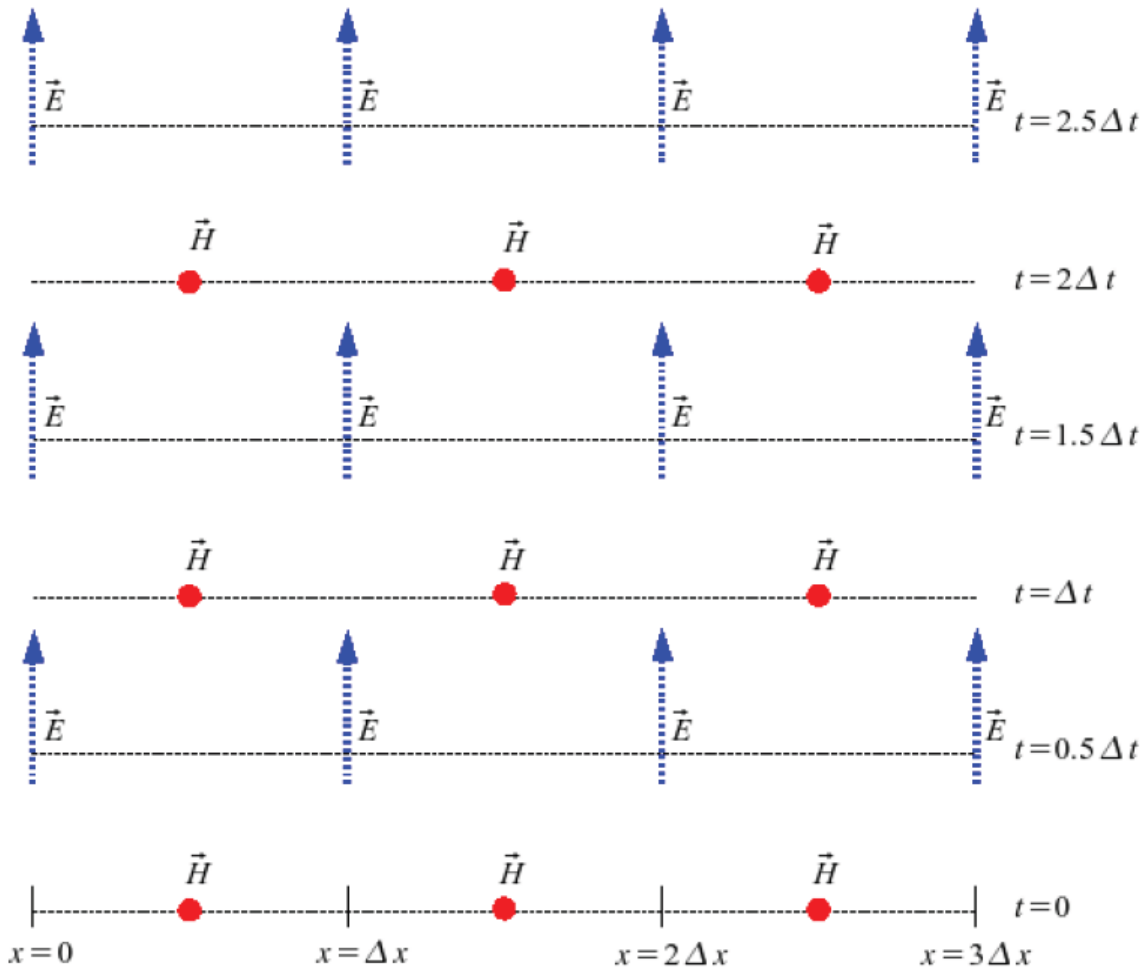

Figura 2.3: Esquema de progressão temporal do algoritmo FDTD a uma dimensão

Este processo de evolução temporal continua até um determinado critério ter sido verificado. Regra geral, são utilizados dois critérios: um deles consiste em terminar a execução do algoritmo ao fim de um determinado número de iterações temporais; o outro consiste em terminar a execução do algoritmo quando a energia do espaço computacional diminui uma dada quantidade abaixo do valor máximo de energia que foi registado no espaço computacional. No trabalho realizado no âmbito desta dissertação apenas foi utilizado o critério do número

máximo de iterações temporais.

Ao nível computacional, os valores das componentes são armazenados em células adjacentes umas às outras, que no seu conjunto constituem uma grelha. Cada célula da grelha vai ter um conjunto de valores numéricos que a definem e lhe atribuem as propriedades de um dado material. No início da execução do algoritmo todos os valores tanto do campo eléctrico como do magnético são inicializados com o valor zero, em todo o espaço computacional. A fonte de excitação está associada a um dos componentes, sendo indiferente se é afectado um valor do campo eléctrico ou um valor do campo magnético. No entanto, e para os casos concretos testados, no caso unidimensional a fonte de excitação estava associada ao campo eléctrico e no caso bidimensional a fonte de excitação estava associada ao campo magnético.

A maior parte do tempo de computação do algoritmo é passado na execução de ciclos para a actualização dos campos eléctrico e magnético onde cada célula é processada individualmente. A actualização do valor de um dado campo em cada célula no instante t utiliza o valor da própria célula no instante temporal anterior assim como os valores das células vizinhas das várias componentes do outro campo cujo valor foi calculado anteriormente no instante t-$\Delta t/2$. A actualização de uma célula em cada um destes ciclos é independente das restantes células pelo que as actualizações podem ser feitas em paralelo, característica importante que é explorada no trabalho efectuado nesta dissertação.

2.4 Vantagens do FDTD

O FDTD apresenta uma série de características que o fazem sobressair [11] em comparação com outros algoritmos que tratam da propagação de ondas electromagnéticas:

- O facto de ser um algoritmo de computação completamente explícita. Isto significa, por exemplo, que o FDTD, ao contrário de outros métodos, não é afectado com inversão de matrizes, por ser um método no domínio do tempo;
- A capacidade de calcular naturalmente a resposta impulsional de um sistema electromagnético;
- A capacidade de modelar a maioria dos materiais, incluindo dieléctricos com perdas, materiais magnéticos, metais com perdas e materiais não convencionais, como plasmas anisotrópicos ou ferrites magnéticas;
- A capacidade de modelar materiais não lineares, dado que comportamentos não lineares podem ser mais facilmente tratados no domínio do tempo do que no domínio da frequência;
- A possibilidade de visualizar um campo tanto no domínio do tempo como no da frequência;
- O facto de permitir a actualização dos campos por iterações temporais permite uma fácil paralelização.

Capítulo 3

Sistema computacional

As aplicações foram desenvolvidas para a plataforma *win32* (32 bit) no sistema operativo Microsoft Windows 7 64 bit Service Pack 1 recorrendo ao ambiente integrado de desenvolvimento Microsoft Visual Studio 2008.

3.1 CPU

O CPU utilizado para executar a aplicação no decurso do trabalho apresenta as seguintes características:

- Intel® Core™ i7 Q720 a 1.6 GHz;
- 4 *core*s com *Hyper-Threading Technology* (HT) (8 *thread*s simultâneos);
- 2.8 GHz modo turbo (*Turbo Boost*);
- 6 MB *cache* nível 3;
- 8 GB *Random-Access Memory* (RAM) *Double Data Rate type Three* (DDR3) a 1066 MHz.

A quantidade de transístores que é possível colocar num processador, assim como as frequências de relógio [5], têm aumentado cada vez mais com o passar do tempo. No entanto, esta é uma tendência que está a atingir o limite de poder de dissipação, sendo este limite proporcional tanto à frequência de relógio como ao número de transístores. Para fazer face a este limite, os fabricantes encontraram na tecnologia *Multi-Core* uma forma de continuar a aumentar o desempenho dos processadores. Graças a esta tecnologia, um só circuito integrado, onde antes apenas um processador podia estar presente, pode agora incluir dois ou mais processadores, aumentando assim o paralelismo disponível para a execução de várias *thread*s em simultâneo. No caso concreto do CPU descrito, esta dispõe de quatro *core*s, os quais partilham entre si 6 MB de *cache*.

Graças a outra tecnologia, a *Hyper-Threading Technology*, o CPU em questão pode executar oito e não apenas quatro *thread*s em simultâneo. A *Hyper-Threading Technology* [19, 20] caracteriza-se por aproveitar recursos no processador que ficavam livres durante a execução de um só *thread*, permitindo que este seja utilizado ao mesmo tempo por várias *thread*s. Assim,

um processador físico com HT [5] é visto pelo sistema operativo como se fossem vários processadores lógicos (um *core* como se fossem vários *core*s). Uma diferença significativa entre um processador que use HT e um sistema que use mais do que um processador, cada um no seu circuito integrado [21], é a quantidade de recursos duplicados. No caso do HT essa quantidade pode ser minimizada, o que permite reduzir o tamanho do circuito integrado.

Apesar de o processador conseguir executar oito *thread*s em simultâneo a 1.6 GHz, o mesmo apresenta uma opção que lhe permite aumentar a sua frequência de relógio até 2.8 GHz, mas com a redução de *core*s activos. Essa opção é disponibilizada pela tecnologia *Turbo Boost*. Com o *Turbo Boost* [22] um processador pode aumentar a sua frequência de relógio com base na temperatura, no número de *core*s activos e nos consumos energéticos estimados. O uso do *Turbo Boost* é possível graças à capacidade de o processador poder desligar *core*s quando estes não estão a ser utilizados, sendo a energia que lhes seria normalmente atribuída redireccionada para os *core*s activos, aumentando a estes a sua voltagem e frequência de relógio sem violar os seus limites no que toca a capacidade térmica, voltagem e energia. A utilização do *Turbo Boost* é benéfica para aplicações *Single-Thread*, mas leva à perda de capacidade de paralelismo, pelo que se revela menos útil para aplicações *Multi-Thread*. No caso do CPU descrito, para que esta tecnologia seja utilizada é necessário activá-la explicitamente através de uma opção de um software específico.

3.2 *Multi-Threading*

O modelo de programação *Multi-Thread* [23] é uma forma especializada de execução multitarefa. Multitarefa é um termo aplicável ao nível do processo e ao nível da *thread*, sendo que no caso concreto deste trabalho apenas é considerado ao nível da *thread*. Uma *thread* corresponde a uma unidade de código que pode ser enviada para execução a um processador. Um processo *Multi-Thread* vai apresentar pelo menos uma *thread* (caso em que passa a ser denominado *Single-Thread*); pode, contudo, apresentar várias, e nesse caso pode executar duas ou mais tarefas de forma concorrente, pois a execução *Multi-Thread* é responsável por diferentes partes de um processo executarem concorrentemente. A presença de vários processadores, cada um no seu circuito integrado ou através de processadores *Multi-Core*, ou ainda uma combinação dos dois, junto com a existência de *thread*s que podem ser executados de forma concorrente, permite ter paralelismo, já que as *thread*s podem ser divididas pelos vários processadores ou *core*s disponíveis. O processo *Multi-Thread* vai ser responsável pela gestão da interacção que as várias *thread*s vão ter entre si. Para o caso concreto do trabalho desta dissertação a sincronização de *thread*s foi conseguida recorrendo à *Application Programming Interface* (API) do *win32* [24, 25], existindo no entanto outras alternativas que não foram exploradas, como sejam as bibliotecas *boost* [26], Intel *Threading Building Blocks* [27] ou *OpenMP* [28].

3.3 GPU

O GPU utilizado para executar a aplicação no decurso do trabalho apresenta as seguintes características:

- Nvidia® GeForce® GTX 260M;
- 1 GB RAM *Graphics Double Data Rate type Three* (GDDR3);

- 112 *core*s;
- 14 multiprocessadores;
- *Warp size* igual a 32;
- *Compute capability* 1.1;
- 8192 registos por bloco;
- 16 KB de *shared memory* por bloco;
- *Constant memory* com total de 64 KB;
- Número máximo de *thread*s por bloco igual a 512.

3.4 CUDA

CUDA [29] é uma tecnologia de computação paralela da Nvidia® que permite a execução de código de propósito geral num GPU (GPGPU). Numa aplicação CUDA típica, os dados são transferidos da memória do CPU para o GPU, a aplicação é executada neste último e o resultado copiado para o CPU.

O código que vai ser executado no GPU é dividido em *kernel*s. Cada *kernel* vai então ser responsável pela execução de parte do código através do lançamento de *thread*s. Estas *thread*s são colocadas em execução em grande número de forma massivamente paralela e executam todas o mesmo conjunto de instruções definidas pelo *kernel*. Somente após todas as *thread*s do *kernel* terminarem a sua execução é que as *thread*s do próximo *kernel* podem ser colocadas em execução. As *thread*s agrupam-se em *warp*s, que correspondem a 32 *thread*s e são a menor unidade de paralelismo que pode ser executada num dispositivo CUDA. As *thread*s também se agrupam em blocos, em que cada bloco corresponde a um grupo de *thread*s que são executadas em simultâneo num dos vários multiprocessadores do GPU. Um multiprocessador corresponde a uma unidade de processamento de um GPU que utiliza uma arquitectura denominada *Single-Instruction, Multiple-Thread* (SIMT). Na arquitectura SIMT uma única instrução é responsável por controlar múltiplas unidades de processamento ao mesmo tempo. Ao conjunto de blocos, por sua vez, dá-se o nome de grelha, sendo que é necessário uma grelha concluir a execução de todos os seus blocos para a próxima fase do programa arrancar. Para efeitos de sincronismo é necessário ter em atenção que as *thread*s só podem sincronizar e partilhar dados entre si ao nível do bloco. Na instrução de invocação do *kernel* são definidos o número de blocos a serem colocados em execução e a quantidade de *thread*s por bloco. Um terceiro parâmetro pode ser necessário nesta instrução de invocação, caso seja utilizada *shared memory* alocada dinamicamente [29].

É necessário ter em atenção o modelo de memória de CUDA, representado na figura 3.1, no desenvolvimento de *kernel*s. Os acessos à memória global do GPU apresentam uma elevada latência [30], o que leva a uma redução significativa do desempenho. É por isso conveniente utilizar os outros tipos de memória que a arquitectura disponibiliza, nomeadamente a *shared memory*. Este tipo de memória é o mais rápido das disponíveis, apesar de ser pequeno [29, 31] (16KB); no entanto, só é acessível ao nível de um bloco, sendo partilhada por todas as *thread*s desse bloco. A *constant memory* também é um tipo de memória pequena (64KB) mas é só de leitura e está definida globalmente, sendo acessível a todas *thread*s sem restrições. A

texture memory resulta da associação de uma região linear da memória global a este tipo de memória, ficando os dados acessíveis de forma mais rápida, pois apesar de esses dados continuarem a residir na memória global passam a ser acedidos em modo apenas de leitura, através de uma *cache*. Exsite ainda a *local memory* [32], que corresponde a um tipo de memória, tal como a *texture memory*, residente na memória global e também acedida através de uma *cache*. A *local memory* é utilizada normalmente para guardar dados de registos e outra informação relacionada com *thread*s quando um multiprocessador fica sem recursos disponíveis, sendo esta memória denominada de *local* porque nela cada *thread* vai ter os seus dados privados, independentes das restantes *thread*s. Para além dos tipos de memória, existe ainda outro factor que condiciona o desempenho de uma aplicação CUDA: os padrões de acesso à memória global [33], onde são obtidos melhores desempenhos se os acessos forem feitos a regiões contíguas da memória.

CUDA é também um modelo de programação que corresponde a uma extensão da linguagem C [34], cujo código, para ser executado no GPU, necessita de ser compilado num compilador específico denominado Nvidia® CUDA Compiler (NVCC), sendo que no final do processo de compilação um *linker* combina o código produzido pelo NVCC com o código produzido pelo compilador de C/C++, originando o executável final. Ocasionalmente, no desenvolvimento das aplicações CUDA, foram utilizadas funções da biblioteca *Thrust* [35].

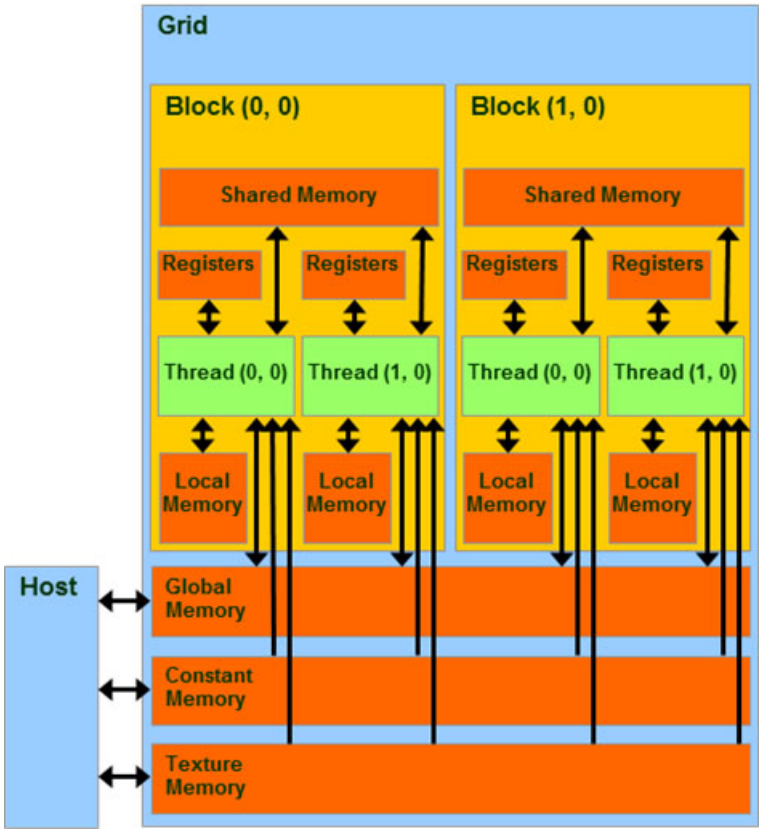

Figura 3.1: Modelo de memória CUDA

O código de um *kernel* convém ser o mais simples possível, pois a existência de expressões condicionais leva a que nem todas as *thread*s sigam o mesmo caminho de execução, o que

provoca uma perda de desempenho. Também é necessário ter em conta o número de registos ocupados por um *kernel*, assim como a *shared memory* ocupada, pois estes factores condicionam a ocupação de um multiprocessador, ou seja, o número máximo de *thread*s de um bloco que podem ser executados em simultâneo num multiprocessador. A taxa de ocupação de um multiprocessador pode ser obtida através de uma ferramenta disponibilizada pela Nvidia denominada *CUDA Occupancy Calculator* [36].

De referir também que o conceito de *thread*s CUDA é diferente do de *thread*s que são executados em CPU. As *thread*s em CUDA correspondem a um só conjunto de instruções definidas por um *kernel* e são colocadas em execução de acordo com as características da arquitectura CUDA de forma massivamente paralela, executando idealmente todas as mesmas instruções, embora tal possa não acontecer devido à existência de expressões condicionais. Por seu lado, as *thread*s em CPU podem não executar todas a mesma sequência de instruções, podendo inclusivamente responder de forma individual a necessidades diferentes do processo, tendo tempos de vida muito variáveis.

O GPU em questão, utilizado para executar a aplicação, apresenta a limitação inerente à *compute capability* 1.1 de não suportar precisão dupla [37] nos seus cálculos, mas apenas precisão simples (*float* na linguagem C/C++). Devido a esta limitação do GPU, foi utilizada precisão simples não só nas variantes desenvolvidas em CUDA mas também nas desenvolvidas para CPU *Single-Thread* e *Multi-Thread*. Acontece ainda que os GPU da Nvidia apresentam diferenças em relação aos CPU no que respeita ao modo como os arredondamentos são efectuados [37], pelo que é necessário ter esta característica em consideração na validação dos valores obtidos entre o CPU e o GPU, que devido a este factor podem não ser exactamente iguais, apesar de o serem quando comparados com uma precisão inferior.

Capítulo 4

FDTD a uma dimensão

O algoritmo FDTD a uma dimensão para uma dada iteração temporal t, para o caso particular do campo eléctrico com uma componente segundo os eixos dos zz e o campo magnético com uma componente segundo o eixo yy, é dado pelas equações 4.1 e 4.2.

$$Ez_t(x) = ca(x) \cdot Ez_{t-1}(x) + cb(x) \cdot (Hy_{t-1/2}(x) - Hy_{t-1/2}(x-1)) \tag{4.1}$$

$$Hy_t(x) = da(x) \cdot Hy_{t-1}(x) + db(x) \cdot (Ez_{t-1/2}(x+1) - Ez_{t-1/2}(x)) \tag{4.2}$$

Ez corresponde ao campo eléctrico e *Hy* corresponde ao campo magnético. As características do meio para o campo eléctrico são definidas por *ca* e *cb* e para o campo magnético por *da* e *db*.

A implementação da versão unidimensional do FDTD foi feita a partir de uma implementação em MATLAB já existente, tendo sido criadas a partir desta variantes em C++ *Single-Thread*, *Multi-Thread* e CUDA[1]. A figura 4.1 mostra um exemplo da grelha computacional utilizada no algoritmo.

A grelha mostra que *Ez* apresenta duas fronteiras enquanto *Hy* não apresenta fronteiras. As setas no interior da grelha mostram, para uma dada componente, quais as componentes de que aquela depende para o cálculo do seu valor, numa iteração temporal. O círculo indica a célula onde foi colocada a fonte de excitação para o caso concreto das simulações realizadas.

O algoritmo consiste no cálculo de todos os valores do campo eléctrico *Ez* (exceptuando as fronteiras), em cada iteração temporal; de seguida, são calculados todos os valores do campo magnético *Hy*. Durante o cálculo dos valores de *Ez* é também aplicada a fonte de excitação na posição previamente definida. Cada um dos campos utiliza dois valores para definir as suas características: *ca* e *cb* no caso de *Ez*, e *da* e *db* no caso de *Hy*. Como nos testes realizados estes valores iriam ser constantes ao longo de uma significativa parte do campo, variando poucas vezes ao longo de todo o espaço computacional, foi efectuada uma optimização nas

[1] A implementação a uma dimensão do algoritmo FDTD descrita neste capítulo originou um paper denominado *Comparing FDTD Computation Efficiencies in Single- and Multi-Thread CPU and GPU Implementations*, elaborado em conjunto com o Doutor Rui Alves, Professor Auxiliar Convidado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro e com o Doutor Pedro Pinho, Professor Adjunto da Área Departamental de Engenharia Electrónica e Telecomunicações e de Computadores do Instituto Superior de Engenharia de Lisboa, orientadores desta dissertação. O paper foi apresentado como poster na conferência EHE2011 - International Conference on Electromagnetic Fields, Health and Environment, realizada em Coimbra entre 26 e 28 de Maio de 2011.

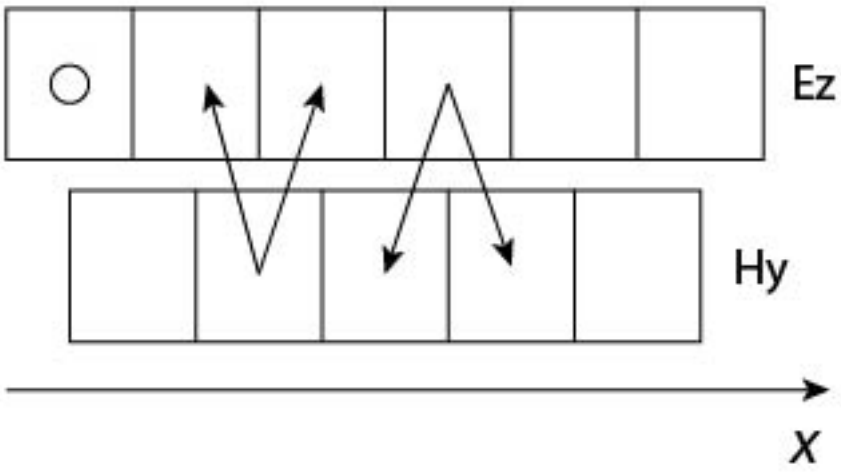

Figura 4.1: Exemplo da grelha FDTD para o caso unidimensional

variantes C++ e CUDA em que não foram utilizados *array*s para guardar os seus valores mas sim uma estrutura de dados que indicava quando, ao longo do eixo dos xx, ocorreram mudanças de características do meio. Esta optimização permitiu reduzir a memória ocupada e o número de acessos à mesma, mas aumentou a complexidade do código. O algoritmo foi executado para um cenário de mil iterações, com a fonte de excitação sinusoidal na fronteira esquerda e a fronteira direita a actuar como espaço aberto. A figura 4.2 mostra o cenário de testes a uma dimensão. A existência de mais do que um tipo de meio ao longo da grelha computacional permite simular o efeito que as ondas electromagnéticas sofrem na transição de materiais com diferentes características.

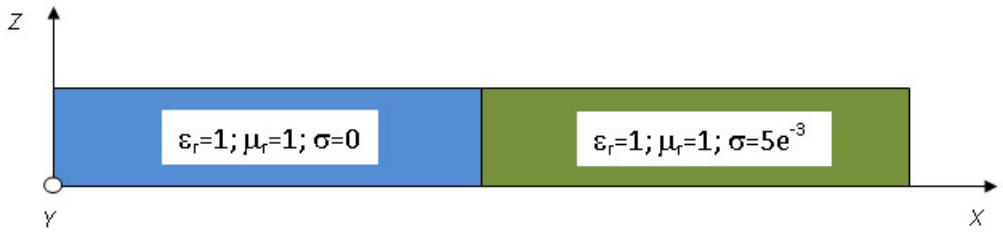

Figura 4.2: Cenário FDTD a 1D

Na figura 4.2 μ_r corresponde à permeabilidade magnética relativa, ε_r corresponde à permitividade eléctrica relativa e σ corresponde à condutividade eléctrica (S/m). A propagação ocorre ao longo de um duplo meio na direcção x em modo *Transverse Electromagnetic* (TEM) [38], ou seja, o campo eléctrico e o campo magnético são ortogonais entre si e ambos transversais à direcção de propagação. A figura 4.3 mostra o estado do espaço computacional para $t = 10$ ns.

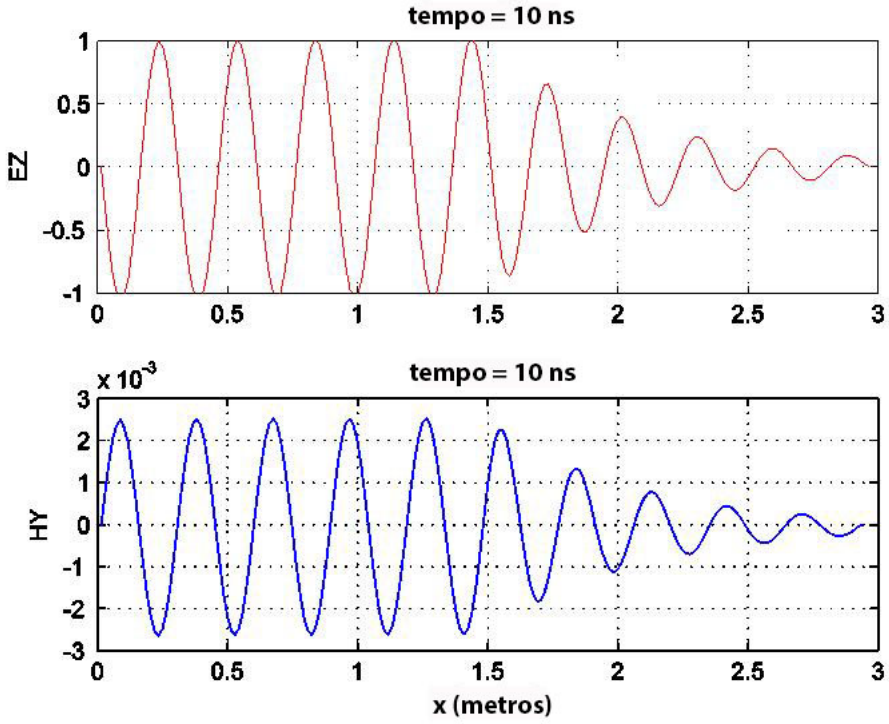

Figura 4.3: Campos eléctrico e magnético com $t = 10$ ns

4.1 MATLAB e CPU *Single-Thread*

O algoritmo FDTD foi implementado em MATLAB e em C++ *Single-Thread* a partir do mesmo algoritmo base, na versão sequencial convencional. Apesar de as implementações em MATLAB e *Single-Thread* serem muito similares, apresentam algumas diferenças entre si. A mais notória advém das características próprias do MATLAB, cuja linguagem matricial permite vectorização [39, 40, 41], enquanto programar em C++ requer indexação explícita. A outra diferença é a optimização na variante *Single-Thread* no que respeita aos valores das constantes que definem o meio. A tabela 4.1 mostra os tempos de computação necessários para se realizarem em MATLAB e CPU *Single-Thread* simulações com diferentes dimensões. No apêndice B está presente o excerto de código MATLAB B.1 e o excerto de código B.2 desenvolvido em C++. Estes excertos correspondem à actualização de *Ez* e *Hy* nessas variantes. O excerto B.2 apresenta uma maior complexidade devido à estrutura de dados utilizada para a definição do meio.

Tabela 4.1: Tempos de computação (em segundos) necessários para FDTD 1D em MATLAB e CPU *Single-Thread*

nº de células/1000	MATLAB (s)	CPU *Single* (s)
10	0,367	0,187
20	0,710	0,390
50	1,640	0,905
100	3,858	2,527
150	14,504	2,956
200	19,825	3,885
350	36,582	5,527
500	53,548	9,922
750	76,825	13,885
1000	105,304	18,892

4.2 CPU *Multi-Thread*

A implementação *Multi-Thread* tira partido de sistemas *Multi-Core* para paralelizar a implementação *Single-Thread* descrita anteriormente, de modo a que o trabalho computacional seja distribuído pelos processadores disponíveis de uma forma eficiente. Esta paralelização foi realizada dividindo as células onde são guardados os valores para os campos eléctrico *Ez* e magnético *Hy* em partições contínuas, em que cada partição vai conter o mesmo número de células, excepto a última partição, que pode ter mais células que as restantes partições se não for possível dividir equitativamente todas as células pelas partições (se o número de células não for múltiplo do número de *thread*s colocadas em execução). Cada *thread* fica responsável por processar a partição que lhe corresponde da mesma forma que acontece no caso *Single-Thread*. No entanto, dado que os campos estão divididos em partições, cada partição vai ter duas fronteiras. Isto exige a sincronização das *thread*s, pois é necessário que as células das fronteiras de cada partição acedam a dados processados por *thread*s vizinhas. Como tal, é necessário garantir que todas as *thread*s estão na mesma iteração temporal. Essa garantia é proporcionada pela inclusão de dois pontos de sincronização durante o ciclo de computação de cada iteração temporal, tal como ilustra o fluxograma representado na figura 4.4.

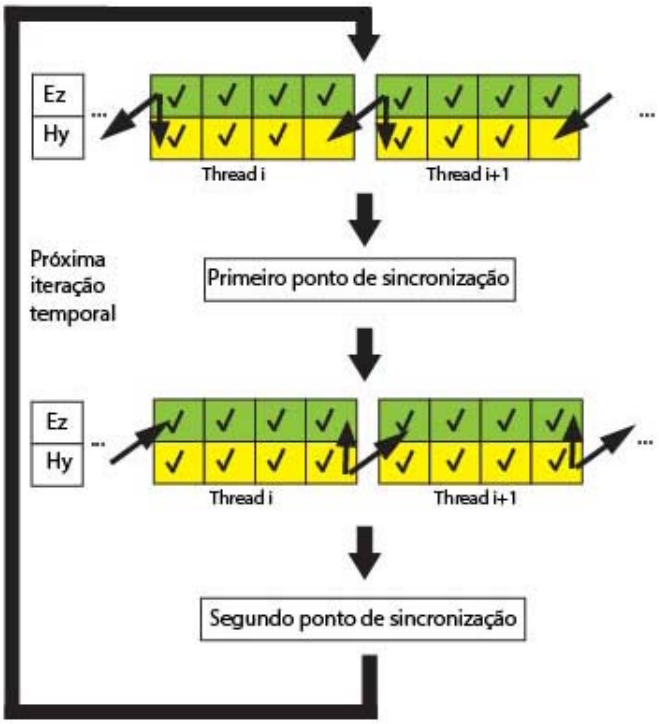

Figura 4.4: Fluxograma da execução do FDTD a 1D *Multi-Thread* onde estão presentes os pontos de sincronização

No início de cada iteração temporal, cada *thread* pode processar livremente todas as células da sua partição excepto o valor de *Hy* na célula mais à direita, pois é necessário garantir que o valor de *Ez* da célula mais à esquerda da partição seguinte utiliza aquele valor de *Hy* ainda não actualizado. Ocorre então o primeiro ponto de sincronização. Após todos as *thread*s terem parado neste ponto de sincronização é então calculado o valor do campo magnético de cada partição que ainda não havia sido modificado, valor este que requer acesso ao valor do campo eléctrico na célula mais à esquerda na partição seguinte. Torna-se assim necessário que todas as *thread*s parem num segundo ponto de sincronização e só após tal ter acontecido podem todas essas *thread*s seguir para o processamento da iteração temporal seguinte. A paragem das *thread*s num ponto de sincronização é controlada através do recurso a uma região de código em que existe aquilo que é denominado como uma zona crítica [42]. Uma zona crítica corresponde a uma secção de código onde são acedidos recursos partilhados por todas ou algumas das *thread*s e apenas uma e uma só *thread* de cada vez pode aceder a esta zona crítica. No caso concreto, cada *thread* que acede a uma das zonas críticas dos dois pontos de sincronização incrementa um contador com o número actual de *thread*s que acederam àquela zona crítica; de seguida, entra num estado de *sleep* e liberta o acesso à zona crítica para que outra *thread* possa entrar. Como as *thread*s podem acordar espontaneamente, é necessário garantir que não prosseguem a sua execução. O modo de o conseguir é colocando as *thread*s num ciclo *while* controlado por uma variável booleana. Assim, sempre que uma das *thread*s acordar espontaneamente, volta imediatamente à acção de *sleep*. Quando a última *thread* entrar na zona crítica o contador vai ser igual ao número de *thread*s que processam partições, pelo que esta *thread* não entra no estado de *sleep* e em vez disso reinicia com o valor zero

o contador com o número de *thread*s que tinham chegado àquela zona crítica. De seguida, coloca com o valor verdadeiro a variável booleana que controla o ciclo de *sleep* da zona crítica do próximo ponto de sincronização, coloca com o valor falso a variável que controla o ciclo de *sleep* da zona crítica do ponto de sincronização actual, sai da zona crítica e executa uma acção que notifica todas as restantes *thread*s a sair do estado de *sleep*. Como a variável booleana que controla o ciclo de *sleep* está agora com o valor falso as *thread*s podem todas prosseguir para a próxima tarefa do algoritmo. O código deste mecanismo de sincronização pode ser consultado no apêndice B no excerto B.3. No segundo ponto de sincronização, a última *thread* que chegar à zona crítica faz ainda a verificação adicional de se tratar da última iteração temporal. Caso o seja, coloca com valor falso uma variável booleana que sinaliza às *thread*s se devem ou não terminar a sua execução, visto que estas executam dentro de um ciclo controlado por esta variável partilhada por todas as *thread*s. Caso não seja, incrementa o contador que regista o número de iterações temporais concluídas. Para evitar que o processo, que corresponde ele próprio a uma *thread*, denominada de principal, continue a sua execução após terem sido colocadas em execução as *thread*s de processamento das partições (pois estas apresentam caminhos de execução independentes da *thread* principal), a *thread* principal bloqueia e espera que estas terminem (*join*). Quando todas as *thread*s de processamento das partições terminarem, a *thread* principal desbloqueia e pode prosseguir para a próxima tarefa. A tabela 4.2 apresenta os tempos de computação necessários para esta variante.

Tabela 4.2: Tempos de computação (em segundos) necessários para FDTD 1D em CPU *Multi-Thread*

nº de células/1000	CPU *Multi* (s)
10	0,109
20	0,156
50	0,343
100	0,640
150	1,123
200	1,435
350	2,640
500	3,542
750	5,435
1000	7,317

Visto que o teste foi realizado num processador de quatro *core*s com HT, à partida seria de esperar um ganho que se traduziria em *speedup*s oito vezes superiores ao obtido para a variante *Single-Thread*. Contudo, os desempenhos obtidos revelaram-se muito aquém desse limite máximo. Isto deve-se a dois factores: os *thread*s não correrem livremente, pois param frequentemente nos pontos de sincronização à espera que os restantes terminem, e ao facto de a *Hyper-Threading Technology* não disponibilizar uma verdadeira paralelização a nível de hardware tal como dois processadores físicos disponibilizariam, tirando antes partido das características do processador para simular dois *thread*s a executar em simultâneo, estando os ganhos de desempenho dependentes das características inerentes à aplicação [20].

4.3 GPU/CUDA

A implementação CUDA utiliza uma estratégia diferente da utilizada nas variantes em CPU. Para esta implementação foram utilizados dois *kernel*s. Um *kernel* efectua o processamento dos valores para o campo eléctrico *Ez* e outro *kernel* efectua o processamento dos valores para o campo magnético *Hy*. O processamento ocorre em sequência, sendo primeiro colocadas em execução todas as *thread*s para *Ez*; quando estas terminarem são então colocadas em execução todas as *thread*s para o campo magnético. Como nem todas as *thread*s podem ser colocadas em execução em simultâneo, são divididas em blocos de 256 *thread*s cada. O número máximo de *thread*s por bloco suportado pelo GPU usado é 512; porém, após alguns testes, o valor de 256 *thread*s por bloco revelou obter melhores desempenhos. Tal deve-se aos *kernel*s condicionarem a ocupação dos multiprocessadores do GPU. Devido às alterações de optimização para os valores que definem o meio, em que os diferentes meios na grelha computacional são representados por uma estrutura ao invés de definidos para cada célula da grelha num vector, as *thread*s, tanto para *Ez* como para *Hy*, não podem ser todas colocadas em execução de uma só vez, dada a estratégia de desenvolvimento seguida. São antes colocadas em execução para cada *kernel* apenas as *thread*s para um tipo de meio; quando as características do meio electromagnético mudam, são colocadas em execução as *thread*s para esse meio, e assim sucessivamente para os meios restantes.

4.4 Optimização GPU/CUDA

A implementação CUDA descrita anteriormente não tira partido do modelo de memória disponibilizado pela tecnologia CUDA. Como tal, os desempenhos obtidos foram de modo geral inferiores aos obtidos para a variante *Single-Thread*. Isto deveu-se ao acesso ser exclusivamente à memória global, que apresenta a maior latência de todos tipos de memória disponibilizados pela tecnologia CUDA. Para superar esta situação foi necessário ter em atenção o modelo de memória CUDA de modo a que este fosse aproveitado para aumentar o desempenho do algoritmo FDTD 1D. A cada iteração temporal, a actualização de cada valor de uma componente (excepto as fronteira de *Ez*) necessita desse valor dessa componente na iteração temporal anterior e dois valores da outra componente. Desse modo, cada valor de uma componente é usado para cálculos de dois valores consecutivos no outro campo. Para aumentar o desempenho do algoritmo esta reutilização pode ser aproveitada guardando esses valores vizinhos em *shared memory*. O uso de *shared memory* revelou um aumento dos *speedup*s na ordem dos 30%. Para além desta foi ainda utilizada a *constant memory* para valores constantes. A figura 4.5 mostra o fluxograma para a versão optimizada e não optimizada da variante CUDA. No apêndice B pode ser consultado o excerto de código B.4 correspondente ao *kernel* responsável pela actualização de *Hy* na versão optimizada. O excerto B.5 corresponde à instrução de invocação do *kernel* B.4.

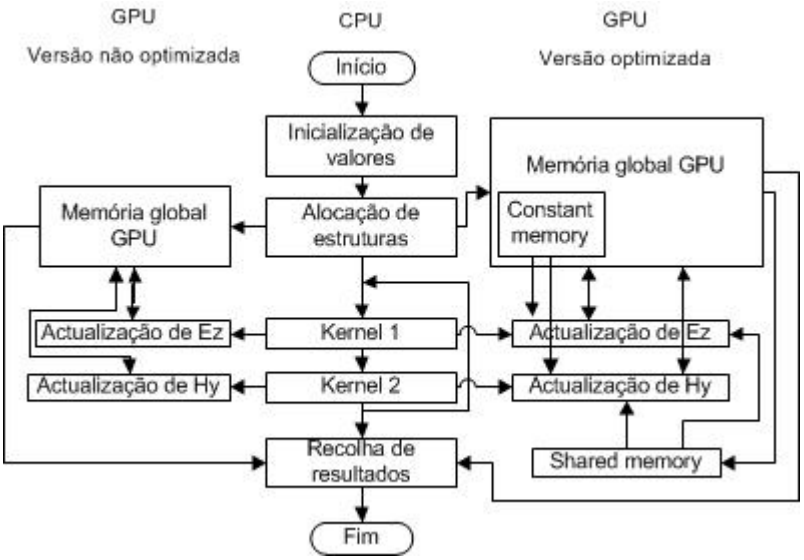

Figura 4.5: Fluxograma das implementações FDTD 1D em CUDA, a mais simples à esquerda e a optimizada à direita

Após terem sido aplicadas as modificações referidas, os acessos à memória global foram reduzidos ao mínimo e o ganho de desempenho foi considerável, podendo os tempos de computação registados para a versão optimizada ser consultados na tabela 4.3.

Tabela 4.3: Tempos de computação (em segundos) para FDTD 1D obtidos em GPU

nº de células/1000	GPU (s)
10	0,390
20	0,405
50	0,624
100	0,609
150	0,780
200	0,827
350	1,123
500	1,404
750	1,856
1000	2,320

4.5 Análise de resultados

Os tempos de computação foram medidos recorrendo ao registo do momento em que o algoritmo iniciava a sua execução, após a alocação das estruturas de dados em CPU, e registando o momento imediatamente após a conclusão da execução do algoritmo, antes de qualquer outra acção. O tempo de computação corresponde à diferença entre esses dois momentos.

Todas as implementações do algoritmo FDTD 1D obtiveram tempos de computação mais rápidos que em MATLAB, incluindo a variante *Single-Thread*. Uma das razões para tal reside no facto de os comandos MATLAB serem interpretados [43] enquanto o código C++ e CUDA é compilado. Por sua vez a implementação *Multi-Thread* apresentou melhor desempenho que a *Single-Thread* embora não oito vezes superior como se poderia esperar. Isto acontece devido aos pontos de sincronização que a variante *Multi-Thread* apresenta e às características da *Hyper-Threading Technology*, em que são obtidas melhorias de desempenho variáveis de acordo com as especificidades das aplicações que beneficiam desta tecnologia [20]. Finalmente, a implementação CUDA foi a que obteve melhor desempenho. A tabela 4.4 contém os valores de *speedup* para MATLAB em comparação com GPU e CPU *Multi-Thread* respectivamente, onde se pode verificar que houve ganhos de velocidade de computação 45 vezes superiores para GPU e para a variante *Multi-Thread* esses ganhos são próximos de 15 vezes.

Tabela 4.4: *Speedup*s de MATLAB versus GPU e MATLAB versus CPU *Multi-Thread*

nº de células / 1000	MATLAB vs GPU	MATLAB vs CPU *Multi*
10	0,942	3,370
20	1,753	4,551
50	2,627	4,780
100	6,335	6,028
150	18,594	12,915
200	23,972	13,815
350	32,575	13,857
500	38,140	15,118
750	41,393	14,135
1000	45,390	14,392

Na tabela 4.5 podem ser visualizados os *speedup* de GPU em relação às variantes CPU, onde se pode observar valores de *speedup* de oito para *Single-Thread* e três para *Multi-Thread*.

Tabela 4.5: *Speedup*s de CPU *Single-Thread* versus GPU e CPU *Multi-Thread* versus GPU

nº de células / 1000	CPU *Single* vs GPU	CPU *Multi* vs GPU
10	0,479	0,279
20	0,963	0,385
50	1,450	0,550
100	4,149	1,051
150	3,790	1,440
200	4,698	1,735
350	4,922	2,351
500	7,067	2,523
750	7,481	2,928
1000	8,143	3,154

A figura 4.6 permite visualizar a forma como os tempos de computação variam com a dimensão do problema nas diversas variantes implementadas. Por sua vez a figura 4.7 mostra os *speedup*s entre algumas dessas variantes. Um aspecto importante da figura 4.7 é as curvas de

speedup para os casos MATLAB/GPU e CPU *Single-Thread*/GPU terem declives positivos, ou seja, à medida que a dimensão do problema aumenta os *speedup*s também aumentam. Como consequência, à medida que o tamanho aumenta, o desempenho também aumenta, o que leva a crer que a implementação CUDA obtenha desempenhos ainda mais elevados para dimensões do espaço computacional superiores.

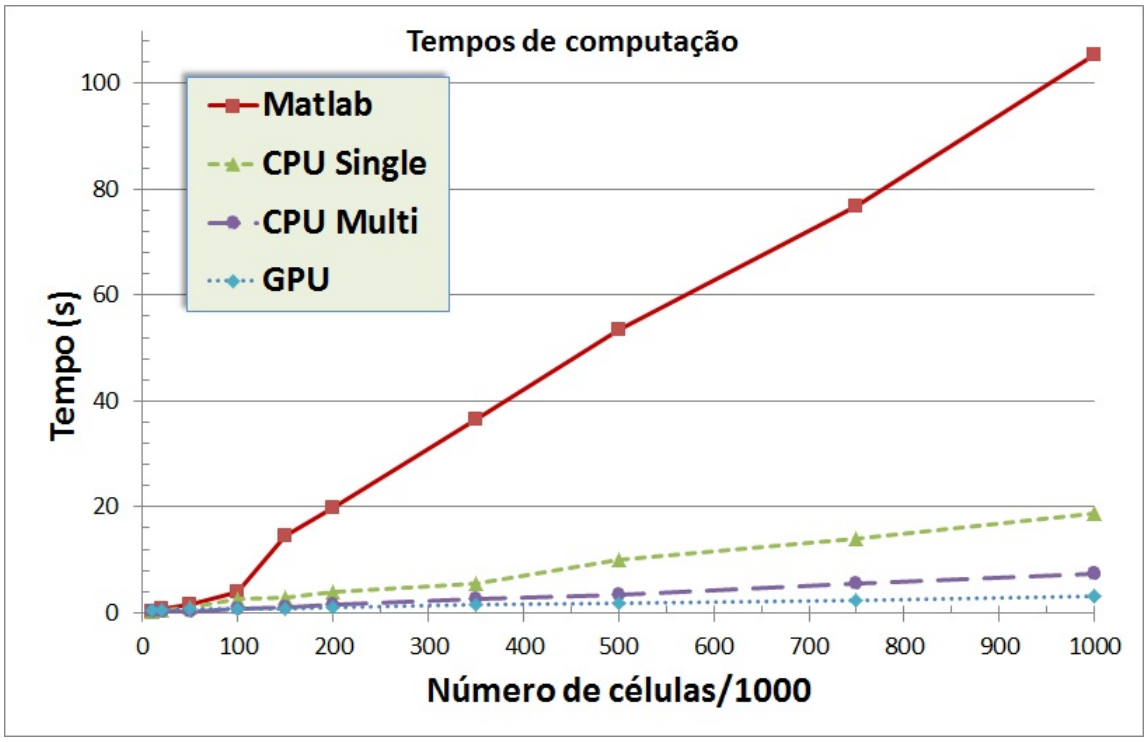

Figura 4.6: Tempos de computação (em segundos) nas diferentes plataformas

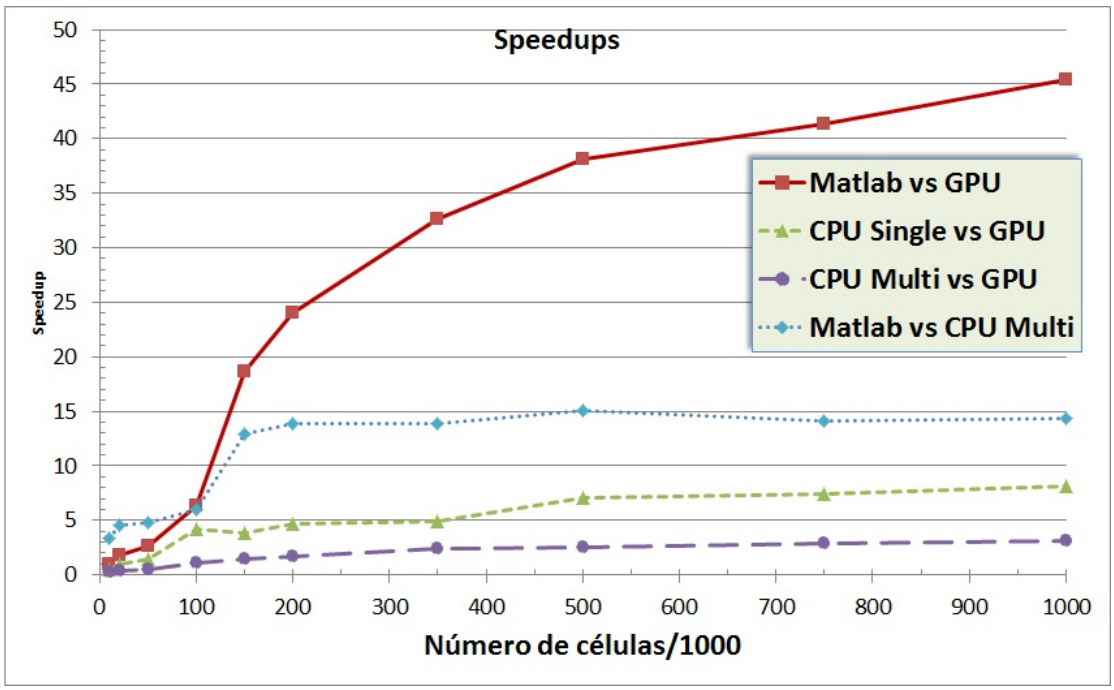

Figura 4.7: Valores de *speedup* entre algumas das plataformas

Capítulo 5

FDTD a duas dimensões

O algoritmo FDTD a duas dimensões para uma dada iteração temporal t, para o caso particular do campo eléctrico com componentes segundo os eixos do xx e do yy e o campo magnético com uma componente segundo o eixo dos zz, é dado pelas equações 5.1, 5.2 e 5.3.

$$Ex_t(x,y) = caex(x,y) \cdot Ex_{t-1}(x,y) + cbex(x,y) \cdot (Hz_{t-1/2}(x,y) - Hz_{t-1/2}(x,y-1)) \quad (5.1)$$

$$Ey_t(x,y) = caey(x,y) \cdot Ey_{t-1}(x,y) + cbey(x,y) \cdot (Hz_{t-1/2}(x-1,y) - Hz_{t-1/2}(x,y)) \quad (5.2)$$

$$\begin{aligned} &Hz_t(x,y) = dahz(x,y) \cdot Hz_{t-1}(x,y) + dbhz(x,y) \cdot \\ &(Ex_{t-1/2}(x,y+1) - Ex_{t-1/2}(x,y) + Ey_{t-1/2}(x,y) - Ey_{t-1/2}(x+1,y) \end{aligned} \quad (5.3)$$

Ex e *Ey* correspondem às componentes do campo eléctrico e *Hz* à componente do campo magnético. As características do meio para *Ex* são dadas por *caex* e *cbex*, para *Ey* são dadas por *caey* e *cbey* e para *Hz* são dadas por *dahz* e *dbhz*.

As equações 5.4 a 5.7 correspondem à forma geral do algoritmo da região esquerda da camada PML para uma dada iteração temporal t:

$$\begin{aligned} &Exbcl_t(x,y) = caexbcl(x,y) \cdot Exbcl_{t-1}(x,y) - cbexbcl(x,y) \cdot \\ &(Hzxbcl_{t-1/2}(x,y-1) + Hzybcl_{t-1/2}(x,y-1) - Hzxbcl_{t-1/2}(x,y) - Hzybc_{t-1/2}(x,y)) \end{aligned} \quad (5.4)$$

$$\begin{aligned} &Eybcl_t(x,y) = caeybcl(x,y) \cdot Eybcl_{t-1}(x,y) - cbeybcl(x,y) \cdot \\ &(Hzxbcl_{t-1/2}(x,y) + Hzybcl_{t-1/2}(x,y) - Hzxbcl_{t-1/2}(x-1,y) - Hzybcl_{t-1/2}(x-1,y)) \end{aligned} \quad (5.5)$$

$$\begin{aligned} &Hzxbcl_t(x,y) = dahzxbcl(x,y) \cdot Hzxbcl_{t-1}(x,y) + \\ &-dbhzxbcl(x,y) \cdot (Eybc_{t-1/2}(x+1,y) - Eybcl_{t-1/2}(x,y)) \end{aligned} \quad (5.6)$$

$$\begin{aligned} &Hzybcl_t(x,y) = dahzybcl(x,y) \cdot Hzybcl_{t-1}(x,y) + \\ &-dbhzybcl(x,y) \cdot (Exbcl_{t-1/2}(x,y) - Exbcl_{t-1/2}(x,y+1)) \end{aligned} \quad (5.7)$$

Exbcl e *Eybcl* correspondem às componentes do campo eléctrico nesta camada para esta região enquanto *Hzxbcl* e *Hzybcl* correspondem às componentes do campo magnético. As características do meio para *Exbcl* são dadas por *caexbcl* e *cbexbcl*, para *Eybcl* são dadas por *caeybcl* e *cbeybcl*, para *Hzxbcl* são dadas por *dahzxbcl* e *dbhzxbcl* e para *Hzybcl* são dadas por *dahzybcl* e *dbhzybcl*. As restantes regiões da camada PML têm equações semelhantes para o cálculo das suas componentes dos campos eléctrico e magnético. Um esquema da camada PML a envolver a região principal do espaço computacional 2D pode ser consultada na figura 2.2.

A implementação da versão bidimensional do FDTD foi feita a partir de uma implementação em MATLAB já existente, tendo sido criadas a partir desta variantes em C++ *Single-Thread*, *Multi-Thread* e CUDA. A figura 5.1 mostra um exemplo da grelha computacional utilizada no algoritmo. Na grelha representada, os segmentos de recta verticais representam *Ey*, os segmentos de recta horizontais representam *Ex* e os pontos representam *Hz*. As setas no interior da grelha mostram para uma componente de um campo, que componentes do outro campo aquela depende para o cálculo do seu valor, numa iteração temporal.

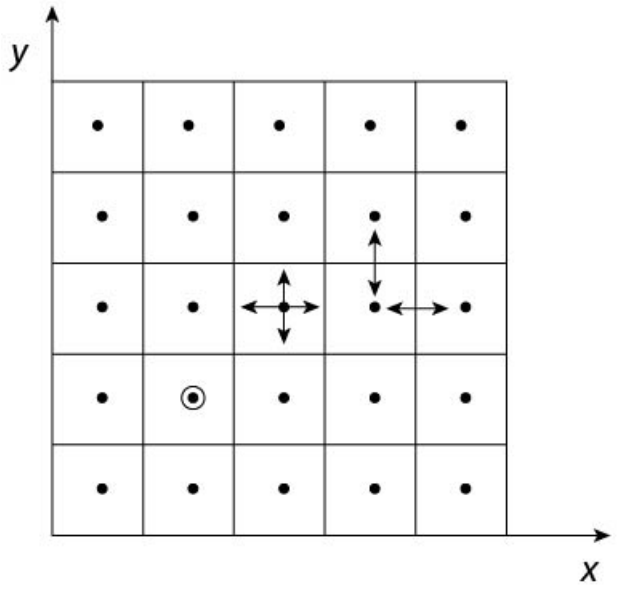

Figura 5.1: Exemplo da grelha computacional FDTD para o caso bidimensional

Para o caso particular do algoritmo FDTD 2D, o campo eléctrico apresenta duas componentes, *Ex* e *Ey*, e o campo magnético uma componente, *Hz*. Para designar o espaço computacional associado a estas componentes vai ser utilizado o termo de região principal. *Hz* vai depender dos valores de *Ex* e *Ey* que o rodeiam, *Ex* depende dos valores de *Hz* na sua célula e na célula abaixo e *Ey* depende dos valores de *Hz* na sua célula e na célula à sua esquerda. O círculo em redor da componente do campo magnético indica uma possível localização para a fonte de excitação. O campo eléctrico vai apresentar fronteiras, que correspondem a linhas no espaço computacional, com *Ex* e *Ey* a apresentarem cada um duas fronteiras; já o campo magnético *Hz* não apresenta fronteiras. De modo a impedir que as ondas sejam reflectidas ao chegarem às fronteiras, foi adicionado a toda à volta da região principal uma camada denominada PML. Esta camada PML tem como objectivo atenuar as ondas que chegam até esta e é dividida em quatro áreas: uma para a fronteira esquerda, uma

para a fronteira da direita, uma para fronteira da frente e outra para a fronteira de trás. A camada PML contém, tal como a região principal, duas componentes para o campo eléctrico *Ex* e *Ey*; difere porém no campo magnético da região principal com duas componentes para o campo magnético, *Hzx* e *Hzy*. A tabela 5.1 contem as dimensões (número de células no eixo dos xx (ie); número de células no eixo dos yy (je)) consideradas na região principal do espaço computacional e o total de células correspondente. A figura 5.2 mostra o fluxograma do algoritmo FDTD a duas dimensões, incluindo onde ocorre paralelização para os casos *Multi-Thread* e CUDA.

Tabela 5.1: Dimensões do espaço computacional e o total de células correspondente

ie	je	Total de células/1000
300	300	90
1500	1500	2250
2100	2100	4410
2700	2700	7290
3300	3300	10890
3900	3900	15210
4500	4500	20250

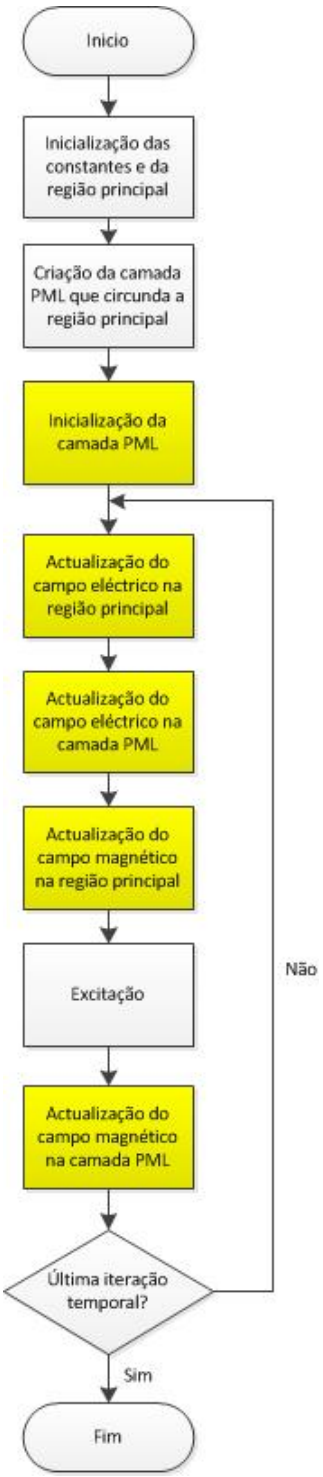

Figura 5.2: Fluxograma do algoritmo FDTD 2D. Os blocos coloridos indicam onde é possível existir paralelização em CPU *Multi-Thread* e GPU

Para além das dimensões presentes na tabela 5.1, de início estava prevista a obtenção de tempos de computação inclusivamente para dimensões de 5100 por 5100 células, contabilizando um total de 26010×10^3 células. Porém, para essas dimensões a variante CUDA não executava, não sendo possível obter valores para os seus tempos de computação. Tal impossibilidade motivou a obtenção dos valores da memória ocupada pelo algoritmo. Na tabela 5.2 estão os valores de memória ocupada pelo algoritmo, registados para as várias dimensões utilizadas no espaço computacional, estes valores foram obtidos na variante CPU *Single-Thread* através do gestor de tarefas do Microsoft Windows.

Tabela 5.2: Memória ocupada (em megabytes) para diferentes tamanhos do espaço computacional

Total de células/1000	Memória (MB)
90	7,797
2250	87,375
4410	166,047
7290	274,352
10890	391,203
15210	553,723
20250	712,156
26010	911,559

Os valores obtidos na tabela 5.2 mostram que o espaço computacional necessita muito rapidamente de grandes quantidades de memória disponível, à medida que o total de células aumenta. Para essa necessidade contribui, para além das necessidades de memória da região principal e da camada PML, todas as componentes quer da região principal quer da camada PML terem ainda associados a cada uma das suas células um par de valores que definem as características do meio. Estas necessidades de memória do algoritmo FDTD a duas dimensões revelam-se problemáticas principalmente para o caso do GPU (que possui apenas 1 GB de memória disponível). Apesar de para o caso de 5100 por 5100 células o total de memória requerida não atingir 1 GB, o valor é suficientemente elevado para não ser possível a sua execução pelo GPU, já que o GPU em questão não é exclusivamente dedicado à execução de aplicações CUDA e necessita de alguma memória para processamento gráfico.

Tendo em conta os enormes requisitos de memória de que o espaço computacional necessita, foram desenvolvidas quatro implementações diferentes em CPU e CUDA, de modo a estudar o desempenho do algoritmo de acordo com a diminuição dos requisitos de memória necessária. Essas implementações correspondem a quatro versões: a versão V1, que é a mais completa e que corresponde à transposição do código original MATLAB para as variantes CPU e CUDA, (os valores apresentados na tabela 5.2 referem-se a esta versão); a versão V2, idêntica a V1 mas tendo sido retirado um dos valores do par de valores que definem o meio na região principal (por exemplo, *Ex* utiliza *caex* e *cbex* para definir o meio e passa a utilizar apenas *cbex*); a versão V3, por sua vez, deixa de utilizar matrizes de constantes na região principal para definir o meio, em que cada célula pode guardar um valor diferente das que a rodeiam, para passar a ter um par de valores constantes que definem todo o meio, isto é, todo o meio da região principal é constante; e a versão V4, idêntica à V3 na região principal, mas à qual foi retirada a camada PML em volta. Na tabela 5.3 podem ser consultados os valores de memória obtidos para as versões V2, V3 e V4 (os valores de V1 encontram-se disponíveis

na tabela 5.2). A figura 5.3 apresenta os valores de memória ocupada obtidos para as quatro versões. Os valores de tempos de computação foram obtidos executando o algoritmo em cenários de mil iterações.

Tabela 5.3: Memória ocupada (em megabytes) para V2, V3 e V4

Total de células/1000	V2 Memória (MB)	V3 Memória (MB)	V4 Memória (MB)
90	6,398	5,516	4,465
2250	60,320	33,640	30,273
4410	113,336	60,930	56,047
7290	185,918	97,750	91,691
10890	264,078	137,570	130,445
15210	373,340	192,711	183,859
20250	449,855	246,777	235,902
26010	612,566	313,910	301,887

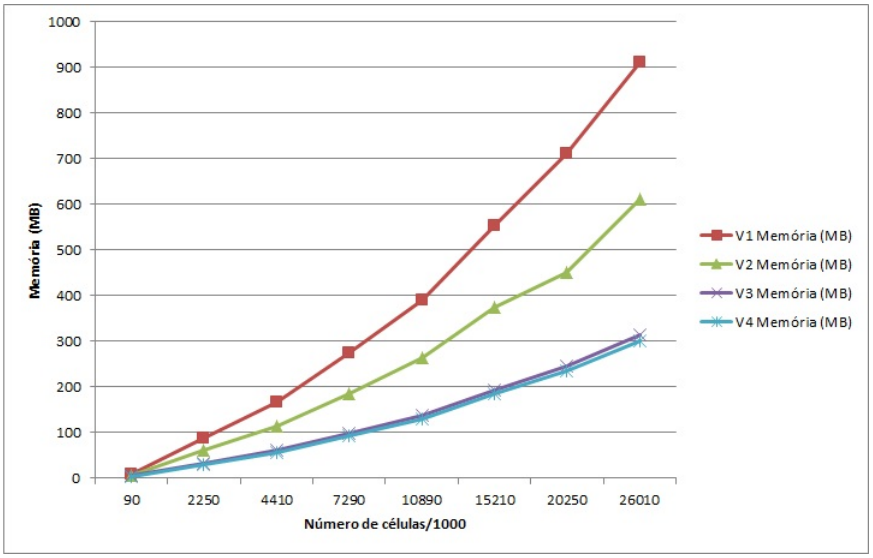

Figura 5.3: Memória ocupada (em megabytes) para as várias dimensões do espaço computacional nas diferentes versões

No que respeita à obtenção dos tempos de computação, a versão V2 foi testada, para além do caso de um meio constante, ainda para mais dois casos específicos: V2 (metades distintas) em que metade do meio da região principal apresenta características diferentes da outra metade e V2 (rectângulo no interior) em que todo o meio é constante com excepção de um rectângulo colocado dentro da região principal, no qual as características do meio são diferentes. As tabelas e gráficos apresentados ao longo deste capítulo serão apenas para a versão V1, sendo que as tabelas e figuras para as restantes versões encontram-se disponíveis no apêndice A.

5.1 MATLAB e CPU *Single-Thread*

A implementação da variante *Single-Thread* é muito semelhante à MATLAB de onde se partiu. A diferença mais notória advém das características próprias do MATLAB, cuja linguagem matricial permite vectorização [39, 40, 41], enquanto programar em C++ requer indexação explícita. No apêndice C está presente o excerto de código MATLAB C.1 e o excerto de código C.2 desenvolvido em C++. Este excertos correspondem à actualização de *Ex* e *Ey* nessas variantes. A tabela 5.4 e a figura 5.4 apresentam os valores dos tempos de computação para estas duas variantes.

Tabela 5.4: Tempos de computação (em segundos) para MATLAB e CPU *Single-Thread* (V1)

Total de células/1000	CPU *Single* (s)	MATLAB (s)
90	2,995	2,000
2250	48,734	85,949
4410	86,050	170,719
7290	137,327	290,169
10890	202,878	421,847
15210	279,000	599,747
20250	390,000	811,110

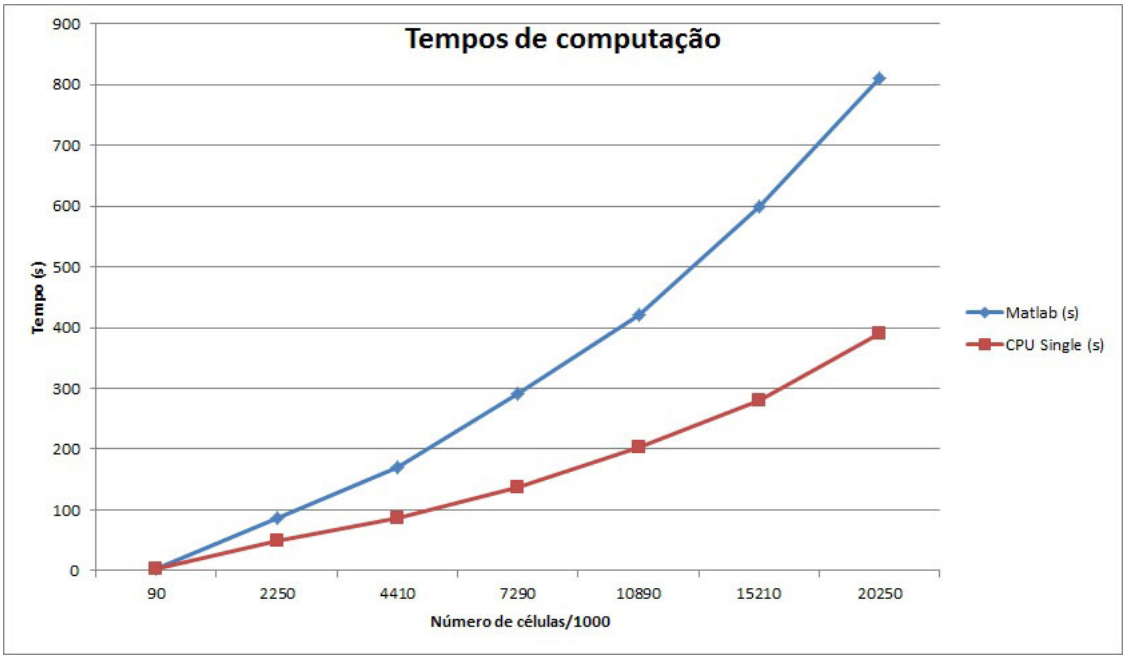

Figura 5.4: Tempos de computação (em segundos) para CPU *Single-Thread* e MATLAB (V1)

Para a variante *Single-Thread* foi ainda efectuado um teste associado à tecnologia *Turbo Boost*. No caso do CPU em concreto a frequência de relógio podia aumentar 1.2 GHz acima da sua base de 1.6 GHz. A tabela 5.5 e a figura 5.5 mostram os tempos computação obtidos e a tabela 5.6 e a figura 5.6 os *speedup*s respectivos.

Tabela 5.5: Tempo de computação (em segundos) para CPU *Single-Thread* em modo Turbo e modo Normal

Total de células/1000	Normal (s)	Turbo (s)
90	2,995	2,059
2250	48,734	48,485
4410	86,050	85,676
7290	137,327	135,939
10890	202,878	197,980
15210	279,000	293,000
20250	390,000	360,626

Tabela 5.6: *Speedup*s para CPU *Single-Thread* em modo Turbo e modo Normal

Total de células/1000	Normal/Turbo
90	1,456
2250	1,005
4410	1,004
7290	1,010
10890	1,025
15210	0,951
20250	1,081

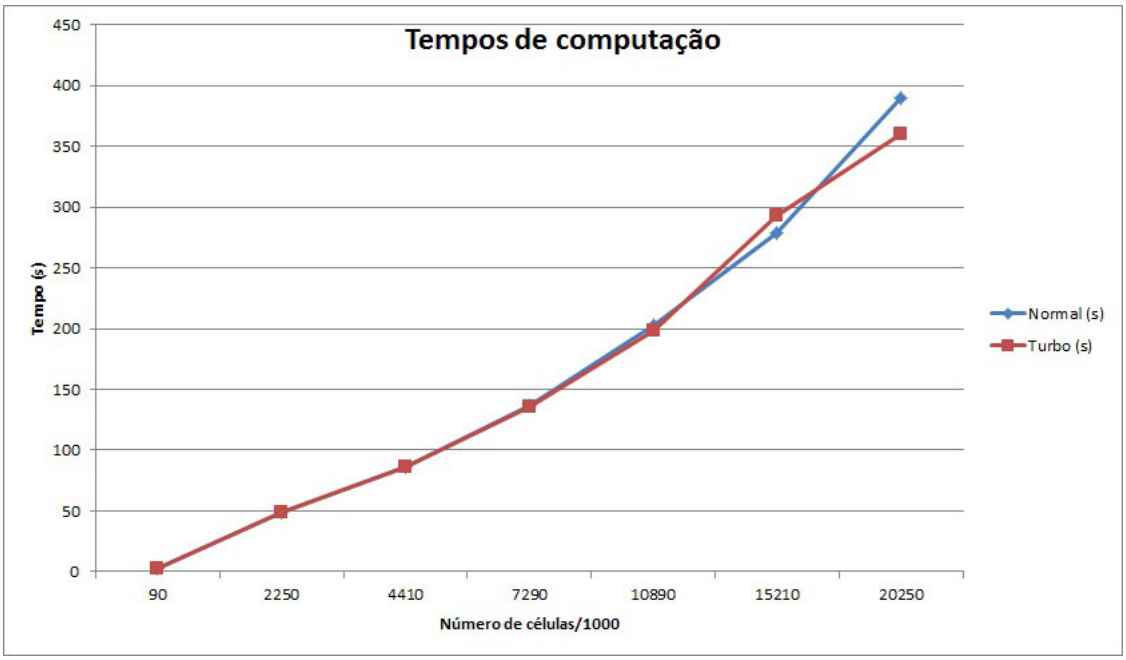

Figura 5.5: Tempos de computação (em segundos) para CPU *Single-Thread* em modo Normal e Turbo (V1)

Figura 5.6: *Speedup*s de CPU *Single-Thread* para o modo Normal em relação ao modo Turbo (V1)

5.2 CPU *Multi-Thread* e GPU/CUDA

Dada a complexidade do problema a duas dimensões ser maior que a uma dimensão, foi seguida uma abordagem de desenvolvimento diferente da usada para o caso unidimensional no que respeita à variante *Multi-Thread*. Essa abordagem passa por não existirem pontos de sincronização explícitos, tal como existia para o caso unidimensional. Continua a existir um desdobramento do problema em partições, sendo uma *thread* responsável pelo processamento de uma partição específica. No entanto, deixa de existir o problema com as fronteiras das partições porque as *thread*s passam agora a executar uma tarefa muito específica, existindo *thread*s diferentes para as várias tarefas necessárias. Estas *thread*s são colocadas em execução quando a tarefa específica que tratam é necessária. Quando esta acaba as *thread*s também terminam. Para garantir que o processo não avança enquanto determinado grupo de *thread*s ainda não terminou o seu trabalho, o processo bloqueia logo após o lançamento das *thread*s numa operação de *join*. Quando todas as *thread*s de um grupo chegarem ao *join* correspondente, o processo então avança e o próximo grupo de *thread*s é colocado em execução. No apêndice C pode ser consultado o excerto de código C.3 em que está presente a função responsável pelo processamento de uma partição de *Hz*. No excerto C.4 está presente a função responsável por colocar em execução as *thread*s para o processamento de *Hz* e que recorre à função presente no excerto C.3.

A implementação GPU para o caso bidimensional foi desenvolvida desde o início com o modelo de memória CUDA presente, pelo que não foram necessárias optimizações posteriores para tirar partido deste modelo. Sempre que possível foi utilizada *shared memory* nos *kernel*s, e sempre que existiam acessos só de leitura à memória global, nos quais não era possível utilizar *shared memory*, foi utilizada *texture memory*, garantido assim que os acessos directos à memória global eram reduzidos ao mínimo. A abordagem seguida na variante CPU *Multi-Thread* é análoga ao que acontece com CUDA, no que respeita a cada *kernel* tratar de uma parte específica do problema, sendo só colocadas em execução as *thread*s do próximo *kernel* quando todas as *thread*s do *kernel* actual tiverem terminado a sua execução.

As estruturas de dados bidimensionais utilizadas nas variantes CPU foram, para a variante CUDA, convertidas em vectores, passando desse modo o acesso a ser efectuado de forma linear. No apêndice C pode ser consultado o excerto de código C.5 correspondente ao *kernel* responsável pela actualização de *Ex*. O excerto C.6 corresponde à instrução de invocação do *kernel* C.5.

5.3 Análise de resultados

Os tempos de computação foram medidos recorrendo ao registo do momento em que o algoritmo iniciava a sua execução, após a alocação das estruturas de dados em CPU, e registando o momento imediatamente após a conclusão da execução do algoritmo, antes de qualquer outra acção. O tempo de computação corresponde à diferença entre esses dois momentos.

Os valores de desempenho obtidos mostram que, tal como para o caso unidimensional, a variante MATLAB foi a que exibiu o desempenho mais baixo, tal como mostra a tabela 5.7 e a figura 5.7 no que respeita aos *speedup*s desta variante em relação às restantes. Uma das razões para tal reside no facto de os comandos MATLAB serem interpretados [43] enquanto o código C++ e CUDA ser compilado.

Tabela 5.7: *Speedup*s das várias implementações em relação à MATLAB (V1)

Total de células/1000	MATLAB/*Single*	MATLAB/*Multi*	MATLAB/GPU
90	0,707	0,147	1,141
2250	1,764	2,374	6,183
4410	1,984	3,712	6,275
7290	2,113	3,844	3,667
10890	2,079	3,536	2,815
15210	2,149	3,513	2,788
20250	2,080	3,865	2,882

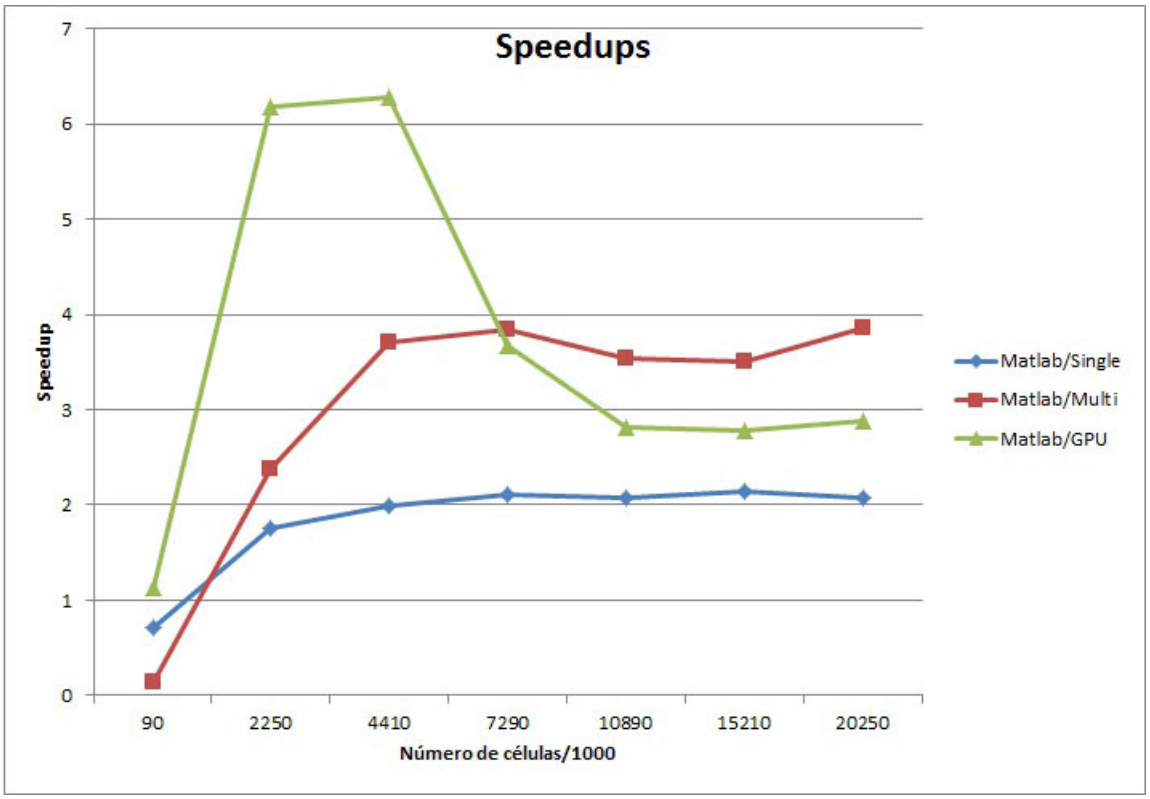

Figura 5.7: *Speedup*s das implementações em CPU e GPU em relação à implementação MATLAB (V1)

No que respeita às variantes CPU *Single-Thread*, *Multi-Thread* e CUDA, os dados das tabelas 5.8 e 5.9 e das respectivas figuras 5.8 e 5.9 mostram que a variante *Single-Thread* apresenta o desempenho mais baixo, e que a variante CUDA apresentou bom desempenho ao início mas que veio a ser suplantada pela variante *Multi-Thread* a partir da dada altura. O motivo está relacionado com o facto de os acessos à memória do GPU não serem sempre, para este caso, a regiões sequenciais de memória, factor que condiciona o desempenho de uma aplicação CUDA. No entanto, quer a variante *Multi-Thread* quer a variante CUDA apresentaram sempre melhor desempenho que a *Single-Thread* à medida que o problema cresce em tamanho.

Tabela 5.8: Tempos de computação (em segundos) para V1

Total de células/1000	CPU *Single* (s)	CPU *Multi* (s)	GPU (s)
90	2,995	14,000	1,857
2250	48,734	36,207	13,900
4410	86,050	45,989	27,207
7290	137,327	75,488	79,124
10890	202,878	119,309	149,869
15210	279,000	171,000	215,093
20250	390,000	210,000	281,393

Tabela 5.9: *Speedup*s para V1

Total de células/1000	*Single*/*Multi*	*Single*/GPU	*Multi*/GPU
90	0,208	1,613	7,754
2250	1,346	3,506	2,605
4410	1,871	3,163	1,690
7290	1,819	1,736	0,954
10890	1,700	1,354	0,796
15210	1,635	1,298	0,794
20250	1,858	1,386	0,746

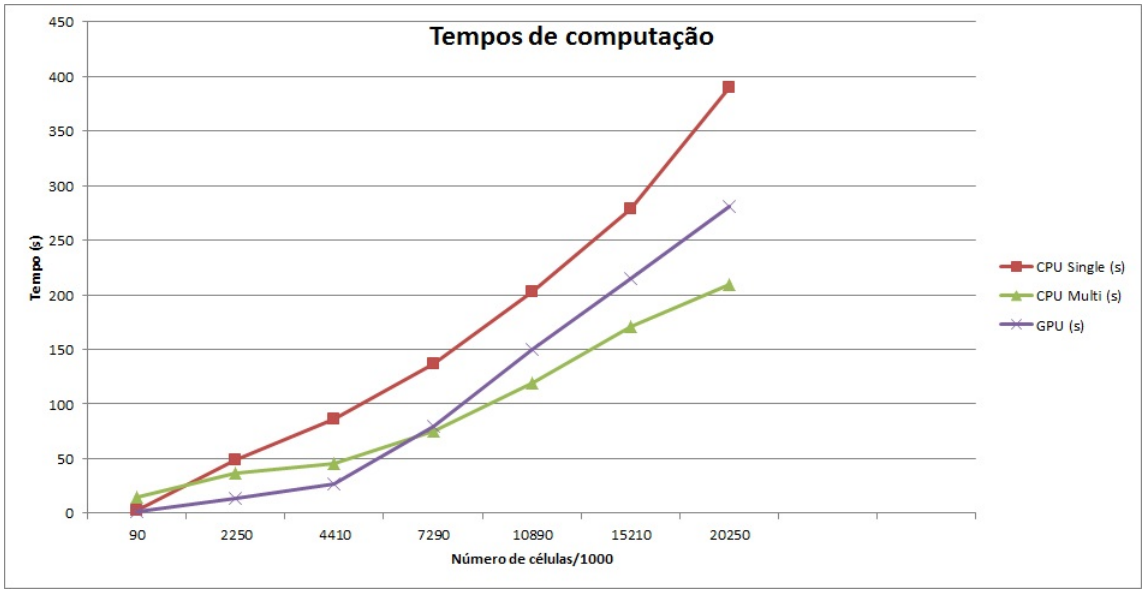

Figura 5.8: Tempos de computação (em segundos) para V1

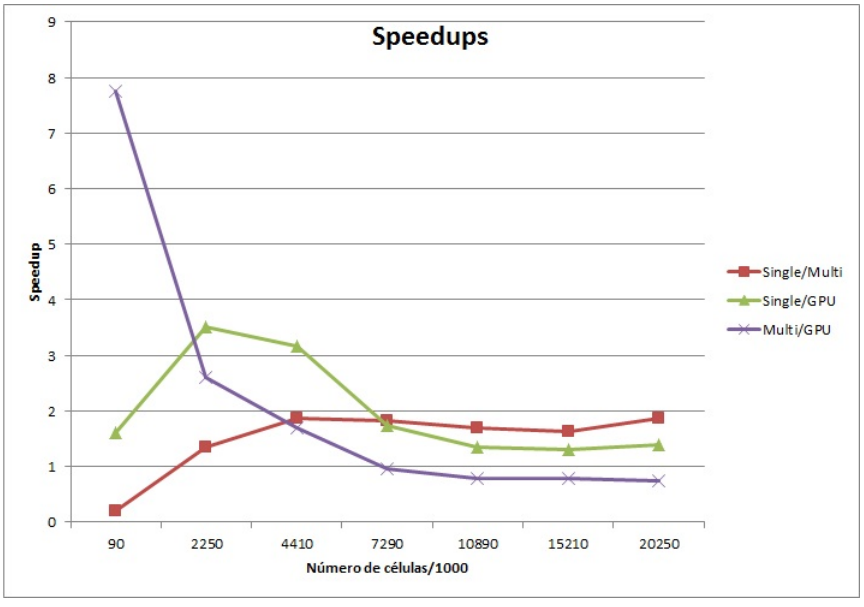

Figura 5.9: *Speedup*s para V1

O teste com o modo *Turbo Boost* obteve desempenhos inesperados. Apesar do aumento significativo da frequência a que o processador estava sujeito, não existiu um aumento significativo de desempenho. Este resultado sugere que a velocidade a que a memória opera é um factor muito significativo no desempenho do algoritmo, o que faz sentido dado o elevado número de acessos à memória que o algoritmo efectua no decurso da sua execução.

Para as restantes versões, os valores que podem ser consultados no apêndice A mostram que, tal como esperado, uma diminuição do número de acessos à memória leva a um aumento do desempenho para todas as variantes. Assim, V2 apresentou melhores desempenhos que V1. V2 (metades distintas) e V2 (rectângulo no interior) apresentaram desempenhos similares a V2. V3 apresentou melhores desempenhos que V2 e V4 apresentou os melhores desempenhos de todas as versões testadas.

Capítulo 6

Conclusões e trabalho futuro

Apesar de existirem soluções como *clusters* de computadores ou supercomputadores para aumentar o desempenho de uma aplicação, estas são caras ou normalmente apresentam um acesso limitado. No entanto, existem soluções que oferecem bom desempenho a preços muito mais reduzidos, nomeadamente sistemas *Multi-Core* e GPU. As melhorias de desempenho obtidas nesta dissertação demonstram que é vantajoso recorrer a estas tecnologias de paralelização para se conseguir aumentar o desempenho do algoritmo FDTD.

Apesar de se terem obtido ganhos tanto para o caso a uma dimensão como a duas dimensões, o caso a uma dimensão obteve em GPU ganhos de desempenho muito superiores. Para tais desempenhos contribuiu a maior simplicidade do algoritmo FDTD 1D devido a este estar limitado a uma só dimensão. A presença deste factor permitiu padrões de acesso à memória global [33] do GPU mais eficientes, um factor importante no desempenho de aplicações CUDA. Para o caso a duas dimensões, estes padrões já eram mais variáveis, o que originou alguma perda de desempenho. A partição do espaço computacional em múltiplas *thread*s também se revelou uma opção interessante. Apesar de a uma dimensão ficar abaixo do desempenho obtido com CUDA, a duas dimensões esta abordagem demonstrou melhor desempenho, acabando inclusivamente por se sobrepor, a dada altura, ao desempenho obtido com CUDA. Outro factor importante é a memória ocupada pela aplicação, que é muito significativa no caso a duas dimensões, em que existe a camada PML em volta da região principal do espaço computacional. Uma ocupação tão elevada de memória limita significativamente o tamanho máximo de células que o espaço computacional pode ter (principalmente no GPU que no caso concreto apresentava 1 GB de memória global disponível). Outro resultado obtido mostrou que o FDTD a ser executado em modo *Single-Thread* apresenta melhorias de desempenho relativamente pouco significativas utilizando o CPU no modo turbo, em que o CPU utilizado pode atingir uma frequência de relógio 1.2 GHz acima do modo normal, ainda que apenas para um *core*, o que beneficia principalmente o modo *Single-Thread* em relação ao modo *Multi-Thread*. Tal resultado indica que, dado o elevado número de acessos que o algoritmo faz à memória, a velocidade a que a memória opera é um factor determinante no desempenho obtido. Em relação ao MATLAB, de onde se partiu para as restantes implementações, os desempenhos obtidos tanto para o caso uni como bi-dimensional mostraram ser muito melhores para as várias implementações em CPU e GPU, inclusivamente nos casos *Single-Thread*. O motivo para tal prende-se com o relativo baixo desempenho da execução de código interpretado em MATLAB [43], comparativamente com linguagens em que o código é compilado, tal como é o caso do C++.

Em relação à aplicação desenvolvida, um aspecto a ter em consideração para desenvolvimentos futuros passa por utilizar para a variante GPU a duas dimensões, estruturas de dados bidimensionais que CUDA disponibiliza ao programador. Futuramente também será interessante estender a implementação do algoritmo para o caso a três dimensões. A própria evolução dos GPU e da tecnologia CUDA abre um leque de novas possibilidades a explorar, nomeadamente sistemas GPU com *compute capability* acima de 1.1, já que existem actualmente GPU com *compute capability* na versão 3.0 [44, 45], que apresentam uma série de avanços tecnológicos em relação à GPU utilizada no decurso desta dissertação ou ainda sistemas *Multi*-GPU [46, 47, 48] em que múltiplos GPU são utilizados em simultâneo nos cálculos necessários ao algoritmo. Outras possibilidades poderão envolver técnicas de sistemas distribuídos em conjunto com CUDA.

Apêndice A

Tabelas e figuras da implementação FDTD 2D

Tabela A.1: Tempos de computação (em segundos) para V2

Total de células/1000	CPU *Single* (s)	CPU *Multi* (s)	GPU (s)
90	2,590	15,000	1,747
2250	40,029	33,056	12,917
4410	70,216	42,183	24,320
7290	113,506	54,615	49,280
10890	161,928	85,207	103,974
15210	223,000	117,000	174,205
20250	303,000	155,000	231,270

Tabela A.2: *Speedup*s para V2

Total de células/1000	*Single*/*Multi*	*Single*/GPU	*Multi*/GPU
90	0,174	1,483	8,500
2250	1,210	3,099	2,559
4410	1,665	2,887	1,734
7290	2,078	2,303	1,108
10890	1,900	1,557	0,820
15210	1,905	1,282	0,673
20250	1,957	1,310	0,669

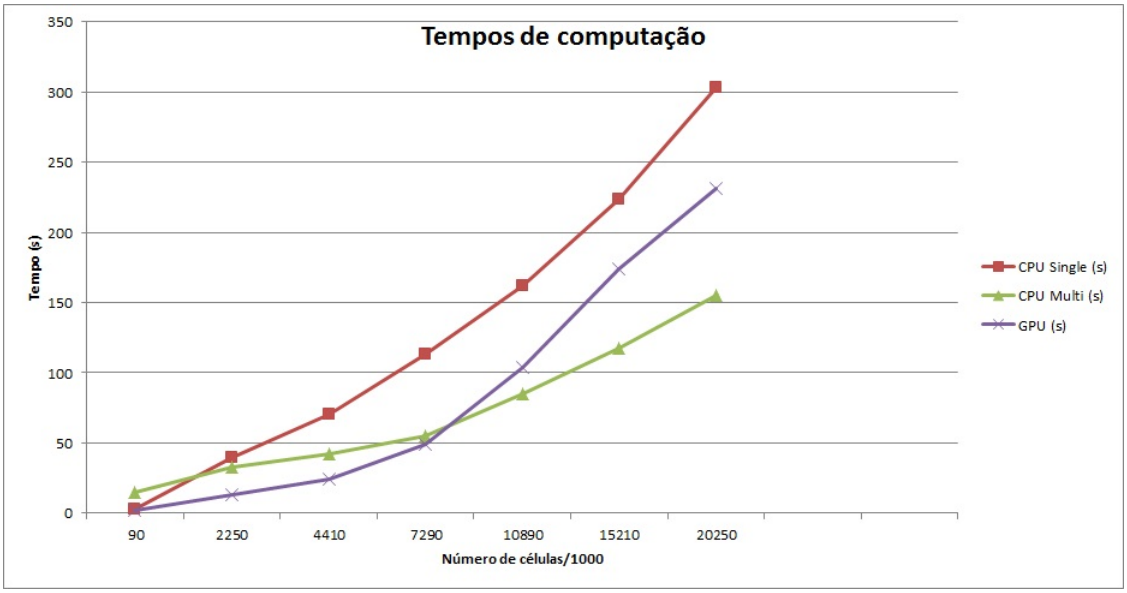

Figura A.1: Tempos de computação (em segundos) para V2

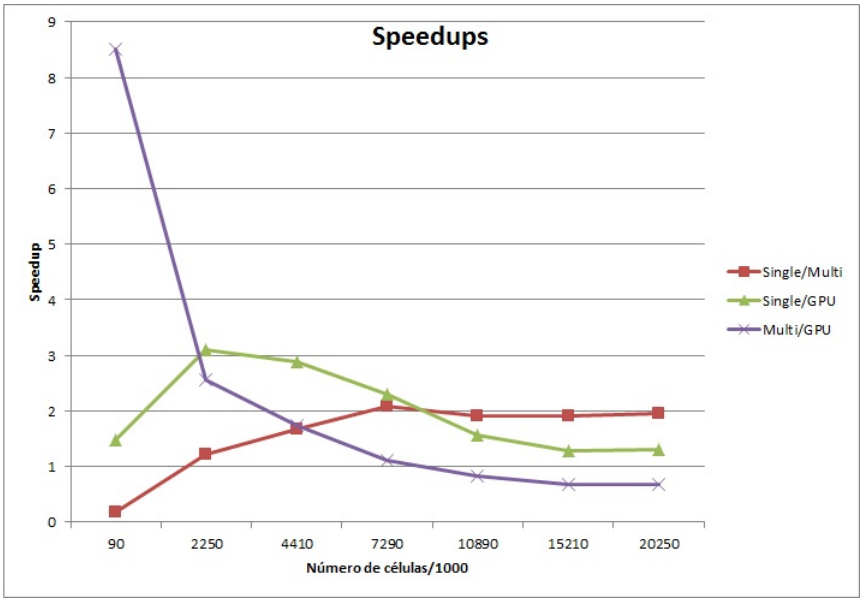

Figura A.2: *Speedup*s para V2

Tabela A.3: Tempos de computação (em segundos) para V2 (metades distintas)

Total de células/1000	CPU *Single* (s)	CPU *Multi* (s)	GPU (s)
90	2,000	14,000	1,763
2250	42,245	33,041	12,746
4410	75,192	42,510	25,990
7290	115,472	57,471	56,566
10890	157,311	88,893	119,497
15210	230,000	123,000	198,838
20250	290,000	172,000	247,276

Tabela A.4: *Speedup*s para V2 (metades distintas)

Total de células/1000	*Single*/*Multi*	*Single*/GPU	*Multi*/GPU
90	0,142	1,150	8,123
2250	1,279	3,314	2,592
4410	1,769	2,893	1,636
7290	2,009	2,041	1,016
10890	1,770	1,316	0,744
15210	1,875	1,159	0,618
20250	1,682	1,172	0,696

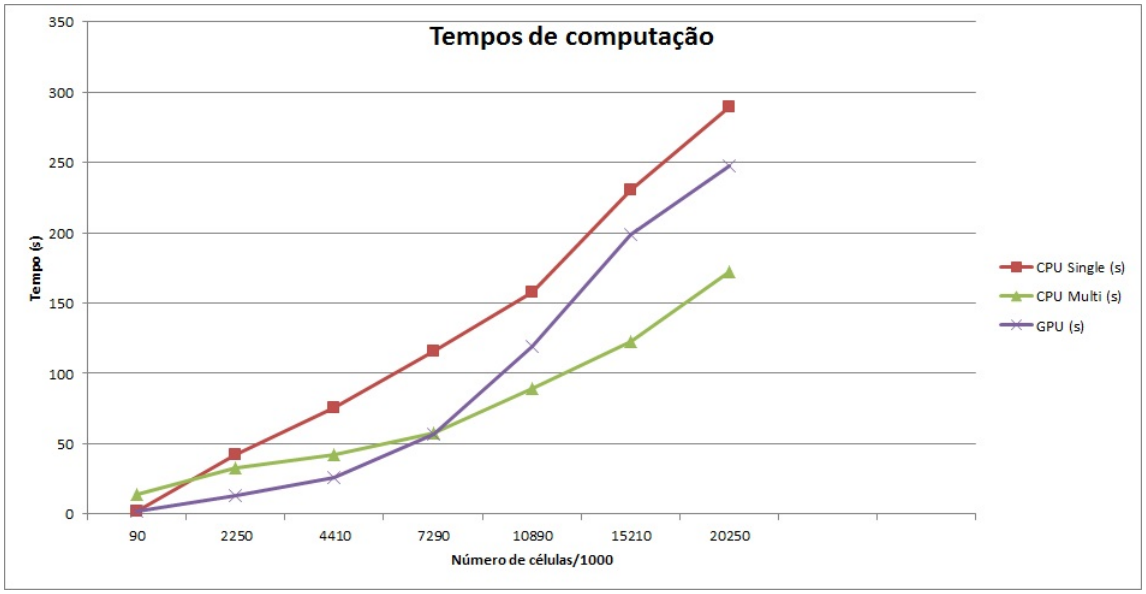

Figura A.3: Tempos de computação (em segundos) para V2 (metades distintas)

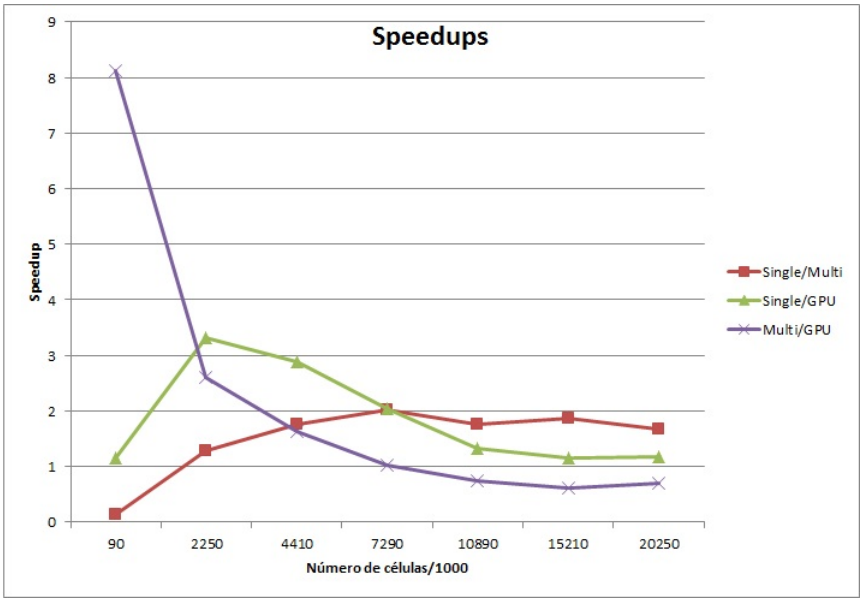

Figura A.4: *Speedup*s para V2 (metades distintas)

Tabela A.5: Tempos de computação (em segundos) para V2 (rectângulo no interior)

Total de células/1000	CPU *Single* (s)	CPU *Multi* (s)	GPU (s)
90	1,888	15,000	1,794
2250	43,602	33,470	12,730
4410	74,318	42,464	28,314
7290	108,904	65,131	77,610
10890	157,700	97,936	142,990
15210	221,000	136,000	209,414
20250	275,000	141,000	146,952

Tabela A.6: *Speedup*s para V2 (rectângulo no interior)

Total de células/1000	*Single*/*Multi*	*Single*/GPU	*Multi*/GPU
90	0,129	1,052	8,156
2250	1,302	3,425	2,629
4410	1,750	2,625	1,500
7290	1,672	1,403	0,839
10890	1,610	1,103	0,685
15210	1,623	1,057	0,651
20250	1,957	1,873	0,957

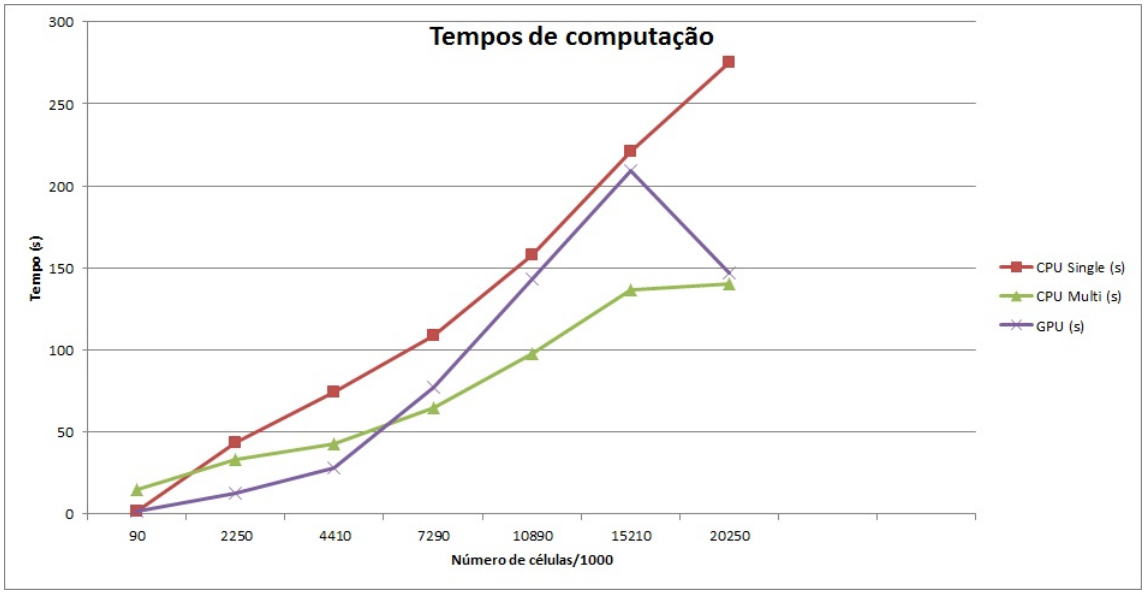

Figura A.5: Tempos de computação (em segundos) para V2 (rectângulo no interior)

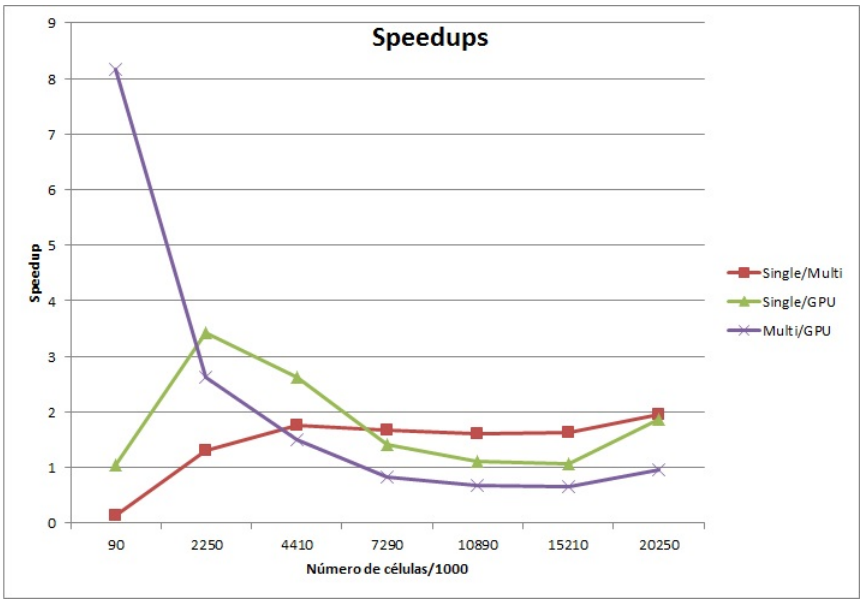

Figura A.6: *Speedup*s para V2 (rectângulo no interior)

Tabela A.7: Tempos de computação (em segundos) para V3

Total de células/1000	CPU *Single* (s)	CPU *Multi* (s)	GPU (s)
90	2,152	15,000	1,638
2250	34,460	30,295	11,637
4410	58,625	37,362	21,606
7290	92,524	45,693	37,815
10890	132,632	59,639	71,947
15210	187,000	85,000	132,132
20250	242,000	115,000	201,786

Tabela A.8: *Speedup*s para V3

Total de células/1000	*Single*/*Multi*	*Single*/GPU	*Multi*/GPU
90	0,145	1,314	9,067
2250	1,137	2,961	2,603
4410	1,569	2,713	1,729
7290	2,025	2,447	1,208
10890	2,224	1,843	0,829
15210	2,196	1,418	0,646
20250	2,097	1,198	0,571

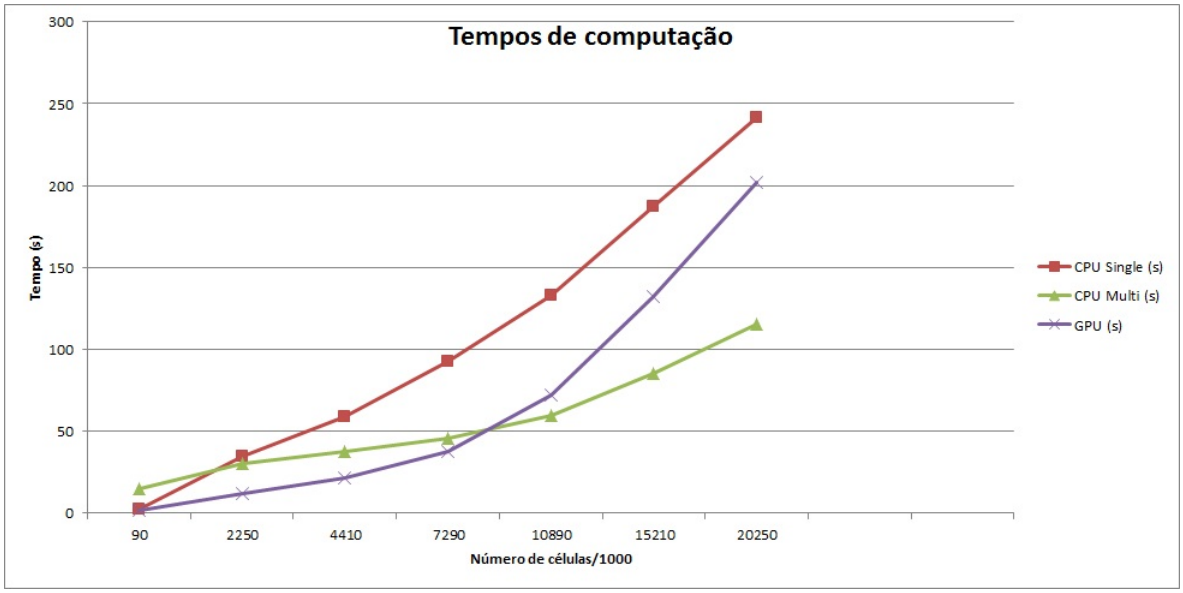

Figura A.7: Tempos de computação (em segundos) para V3

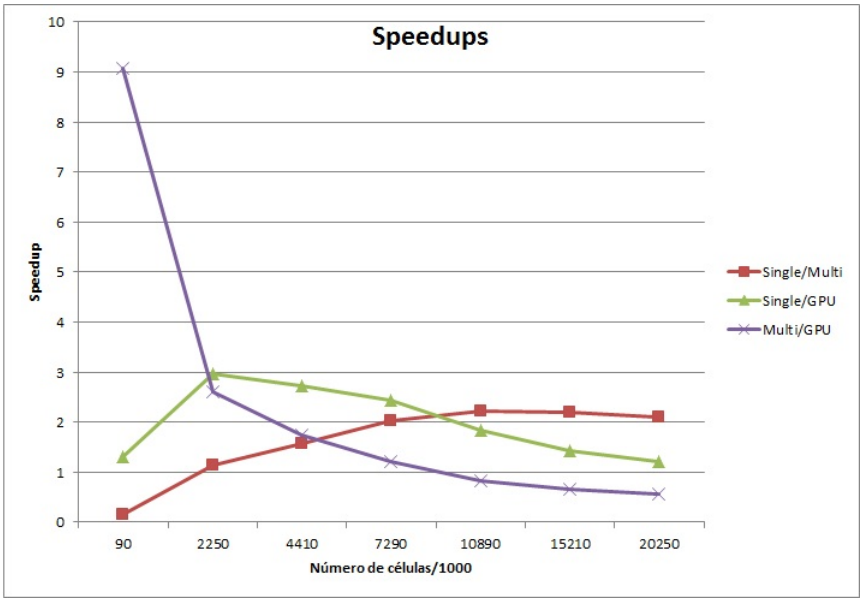

Figura A.8: *Speedup*s para V3

Tabela A.9: Tempos de computação (em segundos) para V4

Total de células/1000	CPU *Single* (s)	CPU *Multi* (s)	GPU (s)
90	1,000	3,000	0,717
2250	30,233	10,982	11,107
4410	50,341	19,827	21,886
7290	77,095	32,089	38,813
10890	112,180	51,464	78,827
15210	169,000	71,000	127,125
20250	206,000	78,000	167,310

Tabela A.10: *Speedup*s para V4

Total de células/1000	*Single*/*Multi*	*Single*/GPU	*Multi*/GPU
90	0,396	1,609	4,068
2250	2,753	2,722	0,989
4410	2,539	2,300	0,906
7290	2,403	1,986	0,827
10890	2,180	1,423	0,653
15210	2,376	1,326	0,558
20250	2,651	1,229	0,464

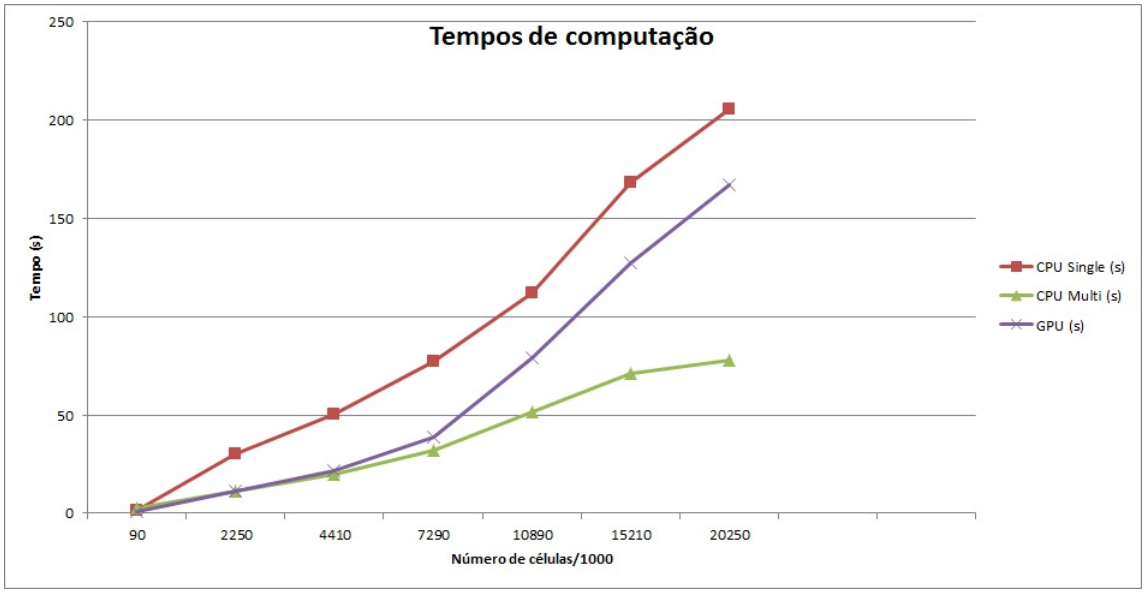

Figura A.9: Tempos de computação (em segundos) para V4

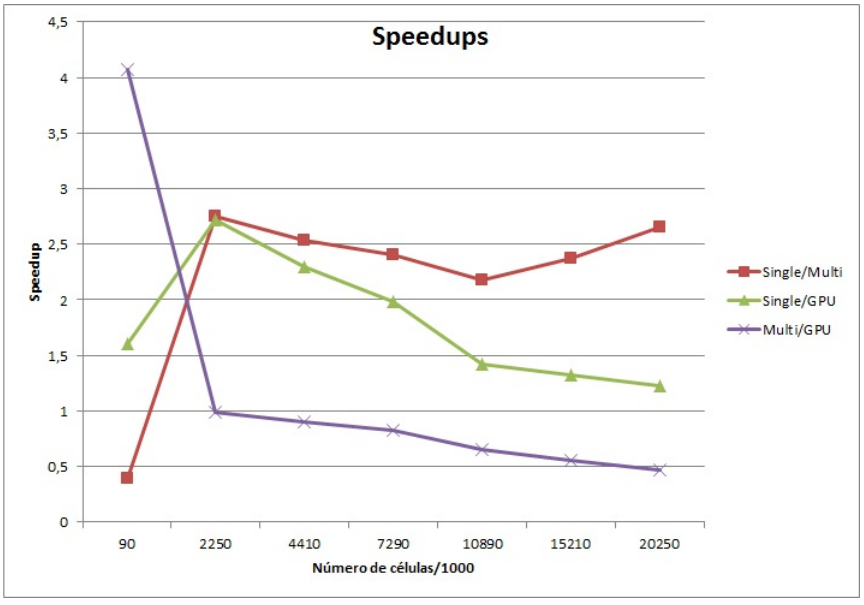

Figura A.10: *Speedup*s para V4

Apêndice B

Excertos de código da implementação FDTD 1D

Excerto de código B.1: Actualização de *Ez* e *Hy* em MATLAB

```
1  %****************************************************************
2  %      BEGIN TIME-STEPPING LOOP
3  %****************************************************************
4
5  for n=1:nmax
6
7  %****************************************************************
8  %      Update electric fields
9  %****************************************************************
10
11 ez(1)=scfact*sin(omegadt*n);
12
13 rbc=ez(ie);
14 ez(2:ie)=ca*ez(2:ie)+cb*(hy(2:ie)-hy(1:ie-1));
15 ez(ib)=rbc;
16
17 %****************************************************************
18 %      Update magnetic fields
19 %****************************************************************
20
21 hy(1:ie)=hy(1:ie)+ez(2:ib)-ez(1:ie);
22
23 %****************************************************************
24 %      END TIME-STEPPING LOOP
25 %****************************************************************
26
27 end
```

Excerto de código B.2: Actualização de *Ez* e *Hy* em C++ na variante CPU *Single-Thread*

```
1  /* TIME-STEPPING LOOP */
2  for(unsigned long n = 1; n <= nmax; ++n) {
3          /* Update electric fields */
4
5          if (lftLim == 'T')
6                  ez[0] = ez[1];
7
8          if (excPos == 0)
9                  ez[0] = scfact*sin(omegadt*(float)n);
10
11         if (rhtLim == 'T')
12                 ez[ie] = ez[ie-1];
13
14         fieldC = fieldD = 0;
15
16         lengthD = lengthC = fields->getOffsetByField(0)+
17         fields->getSizeByField(0);
18
19         caT = fields->getCaByField(0);
20         cbT = fields->getCbByField(0);
21         daT = fields->getDaByField(0);
22         dbT = fields->getDbByField(0);
23
24         for(unsigned long k = 1; k<ie; ++k) {
25
26                 if(k>lengthC) {
27                         lengthC =
28                         fields->getOffsetByField(++fieldC)+
29                         fields->getSizeByField(fieldC);
30
31                         caT = fields->getCaByField(fieldC);
32                         cbT = fields->getCbByField(fieldC);
33                 }
34
35                 if((k-1)>lengthD) {
36                         lengthD =
37                         fields->getOffsetByField(++fieldD)+
38                         fields->getSizeByField(fieldD);
39
40                         daT = fields->getDaByField(fieldD);
41                         dbT = fields->getDbByField(fieldD);
42                 }
43
44                 if (k!=excPos)
45                         ez[k] = caT*ez[k]+cbT*(hy[k]-hy[k-1]);
46                 else
47                         ez[k] = scfact*sin(omegadt*(float)n);
```

```
48
49                      hy[k-1] = daT*hy[k-1]+dbT*(ez[k]-ez[k-1]);
50              }
51
52              if((ie-1)>lengthD) {
53                      lengthD =
54                      fields->getOffsetByField(++fieldD)+
55                      fields->getSizeByField(fieldD);
56
57                      daT =
58                      fields->getDaByField(fieldD);
59                      dbT = fields->getDbByField(fieldD);
60              }
61
62              hy[ie-1] = daT*hy[ie-1]+dbT*(ez[ie]-ez[ie-1]);
63
64      } /* TIME-STEPPING LOOP */
```

Excerto de código B.3: Zona crítica, em C++, correspondente ao primeiro ponto de sincronização da variante CPU *Multi-Thread*

```
1  EnterCriticalSection (&lock);
2  ++actualConcludedThreadsRE;
3
4  if  (actualConcludedThreadsRE < numOfThreads) {
5          while (blockRE) {
6                  SleepConditionVariableCS (&cvrE, &lock, INFINITE);
7          }
8          LeaveCriticalSection (&lock);
9  }
10
11 else {
12         actualConcludedThreadsRE = 0;
13         blockRM = true;
14         blockRE = false;
15         LeaveCriticalSection (&lock);
16         WakeAllConditionVariable (&cvrE);
17 }
```

Excerto de código B.4: *Kernel* CUDA responsável pela actualização do campo *Hz*

```
1  __global__ void HyUpdate(float *ez, float *hy, float da, float db,
2  unsigned long offset, unsigned long length)
3  {
4          extern __shared__ float sharedEz[];
5
6          const unsigned long tid = threadIdx.x;
7          const unsigned long idx =
```

```
 8           blockIdx.x*blockDim.x + tid + offset;
 9
10           if (tid == 0) {
11                   sharedEz[0] = ez[idx];
12           }
13
14           if (idx <= length) {
15                   sharedEz[tid+1] = ez[idx+1];
16
17                   __syncthreads();
18
19
20                   hy[idx] =
21                   da*hy[idx]+db*(sharedEz[tid+1]-sharedEz[tid]);
22           }
23
24   }
```

Excerto de código B.5: Invocação do *kernel* CUDA B.4

```
1  HyUpdate<<< nBlocks, blockSize, sizeof(float) * (blockSize + 1) >>>
2  (ezD, hyD, da, db, offset, length);
```

Apêndice C

Excertos de código da implementação FDTD 2D

Excerto de código C.1: Actualização de *Ex* e *Ey* em MATLAB

```
ex(:,2:je)=caex(:,2:je).*ex(:,2:je)+...
           cbex(:,2:je).*(hz(:,2:je)-hz(:,1:je-1));

ey(2:ie,:)=caey(2:ie,:).*ey(2:ie,:)+...
           cbey(2:ie,:).*(hz(1:ie-1,:)-hz(2:ie,:));
```

Excerto de código C.2: Actualização de *Ex* e *Ey* em C++ na variante CPU *Single-Thread*

```
for (unsigned j = 1; j<je; ++j)
{
        ex[0][j] = caex[0][j]*ex[0][j]+
        cbex[0][j]*(hz[0][j]-hz[0][j-1]);
}

for (unsigned i = 1; i<ie; ++i)
{
        ey[i][0] = caey[i][0]*ey[i][0]+
        cbey[i][0]*(hz[i-1][0]-hz[i][0]);
}

for (unsigned i = 1; i<ie; ++i)
{
        for (unsigned j = 1; j<je; ++j)
        {
                ex[i][j] =
                caex[i][j]*ex[i][j]+
                cbex[i][j]*(hz[i][j]-hz[i][j-1]);
                ey[i][j] =
                caey[i][j]*ey[i][j]+
                cbey[i][j]*(hz[i-1][j]-hz[i][j]);
```

```
23          }
24  }
```

Excerto de código C.3: Função em C++, na variante CPU *Multi-Thread*, responsável por numa *thread* processar uma partição de *Hz*

```
1  DWORD WINAPI upHzT (PVOID p)
2  {
3           ThrArgsHz &args = *(ThrArgsHz *)p;
4           unsigned startI = args.startI;
5           unsigned finishI = args.finishI;
6           unsigned startJ = args.startJ;
7           unsigned finishJ = args.finishJ;
8           float **hz = args.hz;
9           float **dahz = args.dahz;
10          float **dbhz = args.dbhz;
11          float **ex = args.ex;
12          float **ey = args.ey;
13
14          for (unsigned i = startI; i<finishI; ++i)
15          {
16                  for (unsigned j = startJ; j<finishJ; ++j)
17                  {
18                          hz[i][j] = dahz[i][j]*hz[i][j]+dbhz[i][j]*
19                          (ex[i][j+1]-ex[i][j]+ey[i][j]-ey[i+1][j]);
20                  }
21          }
22
23          return 0;
24
25  }
```

Excerto de código C.4: Actualização de *Hz* em C++ na variante CPU *Multi-Thread* recorrendo à função C.3

```
1  void Par::UpHz( ThrArgsHz *thrArgs, unsigned startPointI,
2  unsigned sizeI, unsigned startPointJ, unsigned sizeJ, float **hz,
3  float **dahz, float **dbhz, float **ex, float **ey )
4  {
5           unsigned start = args.start;
6           unsigned finish = args.finish;
7           unsigned inc = args.inc;
8           unsigned last = args.last;
9           unsigned cpuThrs = args.cpuThrs;
10          unsigned rem = args.rem;
11          unsigned length = args.length;
12
13
```

```
14
15          WinThread::ResetID();
16          for (unsigned k = 0; k<cpuThrs; ++k)
17          {
18                  start = finish;
19                  finish += inc;
20                  if (k == last)
21                  {
22                          finish += rem;
23                  }
24
25                  thrArgs[k].startI = start;
26                  thrArgs[k].finishI = finish;
27                  thrArgs[k].startJ = startPointJ;
28                  thrArgs[k].finishJ = sizeJ;
29                  thrArgs[k].hz = hz;
30                  thrArgs[k].dahz = dahz;
31                  thrArgs[k].dbhz = dbhz;
32                  thrArgs[k].ex = ex;
33                  thrArgs[k].ey = ey;
34
35                  thrs[k].Run(upHzT, (PVOID)&thrArgs[k]);
36          }
37
38          Join(cpuThrs);
39  }
```

Excerto de código C.5: *kernel* CUDA responsável por actualizar *Ex*

```
1   __global__ void
2   kEx(float *ex, float *hz, unsigned length, unsigned jb)
3   {
4           extern __shared__ float s_hz[];
5
6           unsigned tIdx = threadIdx.x;
7
8           // determine where in the thread grid we are
9           unsigned idx = blockIdx.x * blockDim.x + tIdx;
10
11          if (idx>=length)
12          {
13                  return;
14          }
15
16          unsigned row = idx/jb;
17
18          if (idx%jb==0)
19          {
```

```
20                  s_hz[tIdx+1] = hz[idx-row];
21          }
22
23           //Block until all threads in the block
24           //have written their data to shared mem
25          __syncthreads();
26
27          // set output
28          if (((idx+1)%jb!=0) && (idx%jb!=0) && (idx<length))
29          {
30                  if (tIdx == 0)
31                  {
32                          s_hz[0] = hz[idx-row-1];
33                  }
34
35                  s_hz[tIdx+1] = hz[idx-row];
36
37                   //Block until all threads in the block
38                   //have written their data to shared mem
39                  __syncthreads();
40
41                  ex[idx] = tex1Dfetch(caex, idx)*ex[idx]+
42                  tex1Dfetch(cbex, idx)*(s_hz[tIdx+1]-s_hz[tIdx]);
43          }
44  }
```

Excerto de código C.6: Invocação do *kernel* CUDA C.5

```
1  kEx<<< sizeGrid(ie*jb, cudaThreads), dimBlock, sharedMemSize >>>
2  (auxPar.ex, auxPar.hz, ie*jb, jb);
```

Glossário

Array Estrutura de dados em que a posição dos seus elementos pode ser acedida através de um índice.

Cache Memória de acesso muito rápido, normalmente pequena e que está próxima ou no próprio circuito integrado do CPU ou GPU.

Central Processing Unit Unidade de processamento central, elemento de um sistema computacional onde são executadas as instruções de um programa. Para os efeitos deste trabalho o termo é mais abrangente e utiliza-se indistintamente para se referir a todo o sistema computacional, incluindo a memória RAM e os vários *core*s, excepto o GPU.

Compute capability CUDA: características de hardware e software do modelo do GPU.

Constant memory CUDA: memória só de leitura, definida globalmente.

Core Um CPU de um sistema *Multi-Core*.

Graphics Processing Unit Unidade especializada no processamento gráfico. Para os efeitos deste trabalho o termo é mais abrangente e utiliza-se indistintamente para se referir a todo o dispositivo de processamento gráfico onde reside o GPU e os seus outros diversos componentes.

Hyper-Threading Technology Tecnologia desenvolvida pela Intel® que permite a um único *core* físico executar mais do que uma *thread* em simultâneo, criando *core*s virtuais.

Join Após um conjunto de *thread*s ser colocado em execução por uma *thread* base, essa *thread* base prossegue a sua execução. A acção de *join* bloqueia a execução dessa *thread* base enquanto espera que um conjunto de *thread*s colocadas em execução por ela termine.

Kernel CUDA: conjunto de instruções a serem processadas em paralelo pelo GPU.

Linker Programa que liga objectos criados por um compilador, originando o ficheiro executável final.

Local memory CUDA: tipo de memória residente na memória global e acedida através de uma *cache*, utilizada normalmente para guardar dados de registos e outra informação relacionada com *thread*s quando um multiprocessador fica sem recursos disponíveis, denominada de *local* porque nesta memória cada *thread* vai ter os seus dados privados, independentes das restantes *thread*s.

Multi-Core Sistema computacional que consiste em colocar mais do que um CPU num único circuito integrado, o que torna possível paralelismo.

Multi-Thread Processo com duas ou mais *thread*s, o que leva ao aparecimento de concorrência.

Shared memory CUDA: a mais rápida das memórias disponíveis no GPU, partilhada por todas as *thread*s de um bloco.

Single-Instruction, Multiple-Thread Arquitectura utilizada em multiprocessadores de GPUs em que uma única instrução controla múltiplos elementos de processamento.

Single-Thread Processo executado de forma sequencial, com apenas uma *thread*.

Sleep Acção que coloca uma *thread* em espera durante um determinado período de tempo ou até que outra *thread* a notifique para que saia desse estado.

Speedup Razão entre duas velocidades de computação. Para A/B, se o valor obtido for superior ao valor um significa que B é mais rápido que A; caso a razão seja inferior ao valor um, significa que B é mais lento que A.

Texture memory CUDA: tipo de memória só de leitura que se associa a regiões lineares da memória global do GPU, continuando a informação a residir na memória global mas passando a ser acedida de forma mais rápida através de uma *cache*.

Thread CUDA: instância de um *kernel* em execução, em que as instruções do mesmo são executadas pelo GPU.

Thread Unidade independente de execução inerente a um processo. Também é utilizada a designação de fio de execução.

Transverse Electromagnetic Modo de propagação em que o campo eléctrico e o campo magnético são ortogonais entre si e ambos transversais à direcção de propagação.

Turbo Boost Tecnologia desenvolvida pela Intel® que permite que um CPU atinja frequências de relógio superiores à sua base, desligando um determinado número de *core*s de acordo com o incremento de frequência.

Warp CUDA: unidade mais pequena de paralelismo em dispositivos CUDA, a que corresponde 32 *thread*s.

Bloco CUDA: bloco ou *block*, grupo de *thread*s que são executados simultaneamente num dos vários multiprocessadores do GPU, podendo as *thread*s de um bloco sincronizar e partilhar dados entre si.

Grelha CUDA: grelha ou *grid*, conjunto de blocos.

Matriz *Array* bi-dimensional.

Multiprocessador Tipo de sistema com um conjunto de processadores em que os recursos, como a memória, são partilhados por todos ou por alguns dos múltiplos processadores que constituem esse conjunto.

Processo Instância de um programa em execução.

Registo Pequena unidade de memória intrínseca ao processador a que este pode aceder de forma extremamente rápida.

Vector *Array* uni-dimensional.

Zona crítica Zona crítica, secção crítica ou região crítica. No paradigma *Multi-Thread* corresponde a uma parte de código que apresenta acesso mutuamente exclusivo a recursos partilhados. Como tal, apenas uma *thread* de cada vez lhe pode aceder.

Bibliografia

[1] K. S. Yee. Numerical solution of initial boundary value problems involving Maxwell's equation in isotropic media. *IEEE Transactions on Antennas and Propagation*, AP-14, 4:302–307, May 1966.

[2] A. Taflove. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House, MA, 1996.

[3] T. Martin. Advanced FDTD applications. In *Computational Electromagnetics in Time-Domain, 2007. CEM-TD 2007. Workshop on*, pages 1 –4, oct. 2007.

[4] M.A. Eleiwa and A.Z. Elsherbeni. Accurate FDTD simulation of biological tissues for bioelectromagnetic applications. In *SoutheastCon 2001. Proceedings. IEEE*, pages 174 –178, 2001.

[5] Cheong Ghil Kim. Accelerating multimedia applications using Intel threading building blocks on multi-core processors. In *Information Science and Applications (ICISA), 2011 International Conference on*, pages 1 –7, april 2011.

[6] G.S. Murthy, M. Ravishankar, M.M. Baskaran, and P. Sadayappan. Optimal loop unrolling for gpgpu programs. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –11, april 2010.

[7] R.O. Topaloglu and B. Gaster. GPU programming for EDA with OpenCL. In *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, pages 63 –66, nov. 2011.

[8] Gang Chen, Guobo Li, Songwen Pei, and Baifeng Wu. GPGPU supported cooperative acceleration in molecular dynamics. In *Computer Supported Cooperative Work in Design, 2009. CSCWD 2009. 13th International Conference on*, pages 113 –118, april 2009.

[9] L.J. Nickisch and P.M. Franke. Finite-difference time-domain solution of Maxwell's equations for the dispersive ionosphere. *Antennas and Propagation Magazine, IEEE*, 34(5):33 –39, oct. 1992.

[10] Yong-Dan Kong and Qing-Xin Chu. A novel three-dimensional unconditionally-stable FDTD method. In *Microwave Symposium Digest, 2009. MTT '09. IEEE MTT-S International*, pages 317 –320, june 2009.

[11] Henrique Manuel Lindgrén Amaral Fernandes. Development of software for antenna analysis and design using FDTD. Master's thesis, Instituto Superior Técnico, Setembro 2007.

[12] Departamento de Física do Instituto Superior de Engenharia do Porto DEFI ISEP. Laboratórios de física, propagação de ondas em líquidos. `https://www.dfi.isep.ipp.pt/uploads/ficheiros/3008.pdf`. *Online*, acedido em 26/12/2012.

[13] Centro de Fusão Nuclear do Instituto Superior Técnico CFN IST. `http://www.cfn.ist.utl.pt/pt/consultorio/listD.html`. *Online*, acedido em 26/12/2012.

[14] Luís Filipe da Silva Pragosa. Uma nova perspectiva geométrica sobre a propagação de ondas electromagnéticas em meios anisotrópicos. Master's thesis, Instituto Superior Técnico, Abril 2010.

[15] G. Mur. Absorbing boundary conditions for the finite-difference approximation of the time-domain electromagnetic-field equations. *Electromagnetic Compatibility, IEEE Transactions on*, EMC-23(4):377 –382, nov. 1981.

[16] G. Mur. Total-field absorbing boundary conditions for the time-domain electromagnetic field equations. *Electromagnetic Compatibility, IEEE Transactions on*, 40(2):100 –102, may 1998.

[17] Jean-Pierre Berenger. A perfectly matched layer for the absorption of electromagnetic waves. *Journal of Computational Physics*, 114(2):185 – 200, 1994.

[18] S. Adams, J. Payne, and R. Boppana. Finite difference time domain (fdtd) simulations using graphics processors. In *DoD High Performance Computing Modernization Program Users Group Conference, 2007*, pages 334 –338, june 2007.

[19] INTEL. Dual-core processor / hyper-threading technology. `http://www.intel.com/reseller/products/demo/dual_core/demo.htm`. *Online*, acedido em 26/12/2012.

[20] Antonio C. Valles Garrett Drysdale and Matt Gillespie. Performance insights to Intel® hyper-threading technology. `http://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology`. *Online*, acedido em 26/12/2012.

[21] Yen-Kuang Chen, E. Debes, R. Lienhart, M. Holliman, and M. Yeung. Evaluating and improving performance of multimedia applications on simultaneous multi-threading. In *Parallel and Distributed Systems, 2002. Proceedings. Ninth International Conference on*, pages 529 – 534, dec. 2002.

[22] J. Charles, P. Jassi, N.S. Ananth, A. Sadat, and A. Fedorova. Evaluation of the Intel® Core™ i7 Turbo Boost feature. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 188 –197, oct. 2009.

[23] U.S. Kanniah and A. Samsudin. Multi-threading elliptic curve cryptosystems. In *Telecommunications and Malaysia International Conference on Communications, 2007. ICT-MICC 2007. IEEE International Conference on*, pages 134 –139, may 2007.

[24] R.selvam. Thread synchronization for beginners. `http://www.codeproject.com/Articles/7953/Thread-Synchronization-for-Beginners`. *Online*, acedido em 27/12/2012.

[25] Thomas Lee. Synchronizing execution of multiple threads (Windows). `http://msdn.microsoft.com/en-us/library/windows/desktop/ms686689%28v=vs.85%29.aspx`. *Online*, acedido em 27/12/2012.

[26] Anthony Williams. Synchronization. `http://www.boost.org/doc/libs/1_41_0/doc/html/thread/synchronization.html`. *Online*, acedido em 27/12/2012.

[27] Intel. Intel threading building blocks (Intel TBB). `http://threadingbuildingblocks.org/`. *Online*, acedido em 27/12/2012.

[28] OpenMP. OpenMP. `http://openmp.org/wp/`. *Online*, acedido em 27/12/2012.

[29] NVIDIA Corporation Technical Staff. *NVIDIA CUDA Programming Guide 2.2*. NVIDIA Corporation, 2009.

[30] Paulius Micikevicius. CUDA optimization. `http://gpgpu.org/wp/wp-content/uploads/2009/11/SC09_Optimization_Micikevicius.pdf`. *Online*, acedido em 26/12/2012.

[31] David Kirk and Wen-mei Hwu. *Programming Massively Parallel Processors – A Hands-on Approach*. Elsevier Inc., 2010.

[32] Paulius Micikevicius. Local memory and register spilling. `http://developer.download.nvidia.com/CUDA/training/register_spilling.pdf`. *Online*, acedido em 27/12/2012.

[33] Cyril Zeller. Tutorial CUDA. `http://people.maths.ox.ac.uk/~gilesm/hpc/NVIDIA/NVIDIA_CUDA_Tutorial_No_NDA_Apr08.pdf`. *Online*, acedido em 26/12/2012.

[34] Richard Membarth. CUDA parallel programming tutorial. `http://pdsgroup.hpclab.ceid.upatras.gr/files/CUDA-Parallel-Programming-Tutorial.pdf`. *Online*, acedido em 26/12/2012.

[35] Jared Hoberock and Nathan Bell. Thrust. `http://code.google.com/p/thrust/`. *Online*, acedido em 27/12/2012.

[36] NVIDIA. CUDA occupancy calculator. `http://news.developer.nvidia.com/2007/03/cuda_occupancy_.html`. *Online*, acedido em 26/12/2012.

[37] Nathan Whitehead and Alex Fit-Florea. Precision & performance: Floating point and IEEE 754 compliance for nvidia gpus. `https://developer.nvidia.com/sites/default/files/akamai/cuda/files/NVIDIA-CUDA-Floating-Point.pdf`. *Online*, acedido em 26/12/2012.

[38] Carlos Alberto Barreiro Mendes e Henrique José da Silva. Teoria das linhas de transmissão. http://www.deetc.isel.ipl.pt/sistemastele/Pr1/Arquivo/Sebenta/Linhas/II_Teoria.pdf, 2005. *Online*, acedido em 26/12/2012.

[39] The MathWorks, Inc. *MATLAB Language Reference Manual*.

[40] Steve Eddins and Loren Shure. Matrix Indexing in MATLAB. http://www.mathworks.com/company/newsletters/articles/Matrix-Indexing-in-MATLAB/matrix.html. *Online*, acedido em 27/12/2012.

[41] José Manuel Neto Vieira. Matlab num Instante. http://www.ieeta.pt/~vieira/MyDocs/MatlabNumInstante.pdf. *Online*, acedido em 27/12/2012.

[42] M. Aater Suleman, O. Mutlu, M.K. Qureshi, and Y.N. Patt. Accelerating critical section execution with asymmetric multicore architectures. *Micro, IEEE*, 30(1):60 –70, jan.-feb. 2010.

[43] Chun-Yu Shei, A. Yoga, M. Ramesh, and A. Chauhan. Matlab parallelization through scalarization. In *Interaction between Compilers and Computer Architectures (INTERACT), 2011 15th Workshop on*, pages 44 –53, feb. 2011.

[44] Geeks3d. (gpu computing) nvidia cuda compute capability comparative table. http://www.geeks3d.com/20100606/gpu-computing-nvidia-cuda-compute-capability-comparative-table/. *Online*, acedido em 26/12/2012.

[45] Nvidia. CUDA GPUs. https://developer.nvidia.com/cuda-gpus. *Online*, acedido em 26/12/2012.

[46] D. Schaa and D. Kaeli. Exploring the multiple-gpu design space. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1 –12, may 2009.

[47] S. Potluri, H. Wang, D. Bureddy, A.K. Singh, C. Rosales, and D.K. Panda. Optimizing mpi communication on multi-GPU systems using cuda inter-process communication. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1848 –1857, may 2012.

[48] Long Chen, O. Villa, and G.R. Gao. Exploring fine-grained task-based execution on multi-GPU systems. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 386 –394, sept. 2011.