UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
INSTITUTO TÉRCIO PACITTI DE APLICAÇÕES E PESQUISAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

LEANDRO JUSTINO PEREIRA VELOSO

IMPLEMENTAÇÃO DO MÉTODO FDTD PARA AS EQUAÇÕES DE MAXWELL EM AMBIENTES DE PROGRAMAÇÃO PARALELA

Rio de Janeiro
2015

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
INSTITUTO TÉRCIO PACITTI DE APLICAÇÕES E PESQUISAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

LEANDRO JUSTINO PEREIRA VELOSO

IMPLEMENTAÇÃO DO MÉTODO FDTD PARA AS EQUAÇÕES DE MAXWELL EM AMBIENTES DE PROGRAMAÇÃO PARALELA

Dissertação de Mestrado submetida ao Corpo Docente do Departamento de Ciência da Computação do Instituto de Matemática, e Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários para obtenção do título de Mestre em Informática.

Orientador: Daniel Gregorio Alfaro Vigo
Coorientadora: Silvana Rossetto

Rio de Janeiro
2015

Veloso, Leandro Justino Pereira

V432i Implementação do método FDTD para as equações de Maxwell em ambientes de programação paralela / Leandro Justino Pereira Veloso. – Rio de Janeiro, 2015.
86 f.: il.

Orientador: Daniel Gregorio Alfaro Vigo.
Coorientadora: Silvana Rossetto.
Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro, Instituto de Matemática, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Programa de Pós-Graduação em Informática, Rio de Janeiro, 2015.

1. FDTD. 2. Programação Paralela. 3. Equações de Maxwell. 4. GPU. 5. Multithreding. – Teses. I. Alfaro Vigo, Daniel Gregorio (Orient.). II. Rossetto, Silvana (Co-orient.). III. Universidade Federal do Rio de Janeiro. IV. Título

CDD:

LEANDRO JUSTINO PEREIRA VELOSO

IMPLEMENTAÇÃO DO MÉTODO FDTD PARA AS EQUAÇÕES DE MAXWELL EM AMBIENTES DE PROGRAMAÇÃO PARALELA

Dissertação de Mestrado submetida ao Corpo Docente do Departamento de Ciência da Computação do Instituto de Matemática, e Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários para obtenção do título de Mestre em Informática.

Aprovado em: Rio de Janeiro, _____ de _____ de _____.

_____

Prof. Dr. Daniel Gregorio Alfaro Vigo (Orientador), D.Sc., UFRJ

_____

Profa. Dra. Silvana Rossetto (Coorientadora)

_____

Profa. Dra. Juliana Vianna Valerio

_____

Prof. Dr. Marcello Goulart Teixeira

_____

Prof. Dr. Esteban Walter Gonzalez Clua

A minha família.

AGRADECIMENTOS

Agradeço à minha namorada, Stella, por todo seu apoio, carinho e paciência durante toda a trajetória do Mestrado . Agradeço ao meu irmão, Leonardo, por todo seu incentivo. Agradeço aos meus orientadores por toda experiência e conhecimento que me passaram durante este período. Também agradeço a todos os colegas de laboratório com quem compartilhei experiências, conhecimento e gargalhadas. Para todos que estiveram presentes em minha vida nesta etapa, tudo que tenho a dizer é obrigado a todos.

RESUMO

VELOSO, Leandro Justino Pereira. **Implementação do método FDTD para as equações de Maxwell em ambientes de programação paralela**. 2015. 86 f. Dissertação (Mestrado em Informática) - PPGI, Instituto de Matemática, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2015.

As equações de Maxwell têm um papel crucial na teoria do eletromagnetismo e suas aplicações. Entretanto, nem sempre é possível resolver essas equações de forma analítica. Por isso, precisamos de métodos numéricos para obter soluções aproximadas das equações de Maxwell. O método FDTD (Finite-Diference Time-Domain), proposto por K. Yee, é amplamente usado devido a sua simplicidade e eficiência. No entanto esse método apresenta um alto custo computacional.

Neste trabalho, propomos uma implementação paralela do método FDTD para execução em GPUs, usando a plataforma CUDA. Nosso objetivo é reduzir o tempo de processamento requerido para viabilizar o uso do método FDTD para a simulação da propagação de ondas eletromagnéticas. Avaliamos o algoritmo proposto considerando condições de contorno de tipo Dirichlet e também condições absorventes. Obtivemos ganhos de desempenho que variam de 7 a 8 vezes, comparando a implementação paralela proposta com uma versão sequencial otimizada.

Palavras-chave: FDTD, Programação Paralela, Equações de Maxwell, GPU, Multithreding.

ABSTRACT

VELOSO, Leandro Justino Pereira. **Implementação do método FDTD para as equações de Maxwell em ambientes de programação paralela**. 2015. 86 f. Dissertação (Mestrado em Informática) - PPGI, Instituto de Matemática, Instituto Tércio Pacitti, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2015.

Maxwell's equations play a crucial role in electromagnetic theory and it's applications. However, it is not always possible to solve these equations analytically. Consequently, we have to use numerical methods in order to get approximate solutions of the Maxwell's equations. The FDTD (Finite-diference Time-Domain), proposed by K. Yee, is widely used to solve Maxwell's equations, due to its efficiency and simplicity. However, this method has a high computational cost.

In this work, we propose a parallel implementation of the FDTD method to run on GPUs using CUDA platform. Our goal is to reduce the processing time required, allowing the use of the FDTD method in the simulation of electromagnetic wave propagation. We evaluate the proposed algorithm considering two different kind of boundary conditions: a Dirichlet type boundary conditions and absorbing boundary conditions. We get a performance gain ranging from 7 to 8 times, when comparing the proposed parallel implementation with an optimized sequential version.

Keywords: FDTD, Parallel Programming, Maxwell's equations, GPU, Multithreding.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A energia gerada por fontes eletromagnéticas e suas interações com o entorno possuem muitas aplicações, entre as quais podemos citar as tecnologias de comunicação sem fio e alguns tratamentos e diagnósticos usados na área médica (EL ZEIN; KHALEGHI, 2007; HAND, 2008; TAFLOVE; HAGNESS, 2000). O desenvolvimento e aprimoramento dessas aplicações requerem um estudo detalhado sobre a interação do campo eletromagnético com o meio e a propagação de ondas eletromagnéticas na região de interesse. Esse estudo toma como base as equações de Maxwell, um sistema de equações em derivadas parciais que descreve a evolução no tempo do campo eletromagnético.

A resolução dessas equações é analiticamente inviável na grande maioria dos casos. Em função disso, em 1966, K. Yee desenvolveu um esquema de diferenças finitas no domínio do tempo e espaço (FDTD) para resolver numericamente, e de forma bastante simples, as equações de Maxwell (YEE, 1966). Trata-se de um método de diferenças finitas explícito sobre uma malha intercalada que possui precisão de ordem 2, tanto no espaço quanto no tempo SULLIVAN (2000); TAFLOVE; HAGNESS (2000).

Apesar de simples, esse método é muito dispendioso computacionalmente devido às restrições na discretização temporal necessárias para garantir a estabilidade numérica. Isso tem levado à busca e desenvolvimento de implementações mais eficientes desse algoritmo que permitam calcular as aproximações em um menor tempo de execução. Por exemplo, algumas técnicas de paralelização desse algoritmo podem ser encontradas em (KASHDAN; GALANTI, 2006; TAFLOVE; HAGNESS, 2000).

Nos últimos anos, a disponibilidade de novas plataformas computacionais, em particular as GPUs, tem motivado o desenvolvimento de novas alternativas de paralelização do algoritmo de Yee com ganhos significativos de desempenho. Diferentemente das CPUs, as GPUs possuem centenas de unidades de processamento com unidades de controle bastante simples. Isso faz com que elas sejam adequadas para execução de aplicações com forte paralelismo de dados, como é o caso dos métodos FDTD.

Os primeiros trabalhos de paralelização do método FDTD para GPUs tiveram como objetivo mostrar a viabilidade dessa proposta, exigindo um significativo esforço para compreender e adaptar o algoritmo para o *pipeline* de computação gráfica (usando as linguagens OpenGL, Brook e Cg) (ADAMS; PAYNE; BOPPANA, 2007; INMAN; ELSHERBENI; SMITH, 2005; KRAKIWSKY; TURNER; OKONIEWSKI, 2004). A partir daí, várias propostas de implementação do método FDTD para GPUs foram avaliadas, explorando ambientes de programação de nível mais alto, como CUDA. Entre essas propostas destacamos (BALEVIC et al., 2008; DEMIR; ELSHERBENI, 2010; DE DONNO et al., 2013; VALCARCE; DE LA ROCHE; ZHANG, 2008).

1.1 Motivação e objetivos

A principal motivação para esse trabalho foi a necessidade de realizar simulações FDTD em tempo razoável para viabilizar o uso desse método na simulação de aplicações reais. Embora existam várias iniciativas de paralelização do método FDTD, nem sempre os algoritmos implementados estão disponíveis para uso. Além disso, dependendo dos objetivos da aplicação real, há várias questões que precisam ser consideradas ou configuradas, como por exemplo as características do meio de propagação e as condições de contorno adotadas.

Neste trabalho, estudamos diferentes ambientes e técnicas de programação paralela com o objetivo de propor e implementar alternativas de paralelização para o método FDTD, visando a redução do tempo de processamento requerido.

Consideramos dois ambientes de programação paralela — CPU multinúcleo (usando *multitreading*) e GPU — dando mais relevância à implementação em GPU por se mostrar mais promissora em termos de capacidade de processamento. Implementamos duas versões do método FDTD, uma usando condições de contorno de Dirichlet, e outra usando condições de contorno absorventes (BERENGER, 1994). Além disso, ambas as implementações permitem que o usuário opte por salvar em arquivo os resultados intermediários da simulação, uma vez que algumas aplicações requerem não apenas o resultado final da simulação mas também os resultados intermediários.

Para avaliar o desempenho das versões paralelas, usamos como referência uma implementação sequencial otimizada, obtendo ganhos da ordem de 8 vezes com a versão paralela em GPU. Além da avaliação de desempenho, apresentamos uma avaliação de corretude dos algoritmos paralelos propostos, tomando como referência um exemplo com solução analítica exata.

1.2 Estrutura do texto

O restante deste texto está organizado da seguinte forma. No Capítulo 2 descrevemos o esquema de diferenças finitas para a resolução das equações de Maxwell proposto por Yee e as condições de contorno adotadas neste trabalho. No Capítulo 3 apresentamos exemplos de resolução das equações de Maxwell para validação dos algoritmos que serão expostos no Capítulo 4. No Capítulo 4, descrevemos os ambientes de programação paralela adotados neste trabalho e os algoritmos paralelos

propostos. No Capítulo 5 detalhamos as avaliações de corretude e desempenho realizadas para os algoritmos implementados. Por fim, no Capítulo 6 apresentamos as conclusões deste trabalho.

2 EQUAÇÕES DE MAXWELL

As equações de Maxwell são vistas como ponto de partida para a obtenção das equações que descrevem os campos das ondas elétricas e magnéticas. Elas foram desenvolvidas pelo físico escocês James Clerk Maxwell (1831-1879), a partir de estudos desenvolvidos por cientistas antecessores.

A resolução destas equações, que se apresentam tanto na forma integral quanto diferencial, é analiticamente inviável para muitos casos. Em vista disso, é necessário a utilização de técnicas computacionais para sua resolução. O esquema de diferenças finitas no tempo e espaço (*Finite-Difference Time-Domain - FDTD*) desenvolvido por Yee (1966) é um método computacional muito popular para resolução numérica das equações do eletromagnetismo, porque apesar de simples oferece uma boa aproximação.

Neste capítulo descreveremos as Equações de Maxwell em sua forma diferencial para os casos tridimensional e bidimensional, assim como o esquema de Yee. As condições de estabilidade de Courant e , por fim, as condições de contorno absorventes.

2.1 Equações de Maxwell - Caso tridimensional

As Equações de Maxwell representam a unificação dos campos elétricos e magnéticos para a predição dos fenômenos eletromagnéticos. Elas são apresentadas tanto na forma de integral quanto de equações diferencias, neste trabalho será tratada a forma diferencial. As equações de Maxwell descrevem a evolução no tempo

do campo eletromagnético em uma região do espaço.

Considerando o caso de um meio sem cargas elétricas ou magnéticas, mas podendo conter materiais que absorvem a energia dos campos, as equações de Maxwell, na forma diferencial, são construídas a partir das seguintes leis do eletromagnetismo (TAFLOVE; HAGNESS, 2000):

Lei de Faraday

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

Lei de Ampere

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

Lei de Gauss para o campo elétrico

$$\nabla . \vec{D} = 0 \tag{2.3}$$

Lei de Gauss para o campo elétrico

$$\nabla . \vec{B} = 0 \tag{2.4}$$

A descrição das variáveis se encontram nas tabelas 2.1 e 2.2.

Considerando meios lineares, isotrópicos e não dispersivos podemos escrever a relação entre a densidade do fluxo elétrico e sua respectiva componente magnética como:

$$\begin{aligned} \vec{D} &= \epsilon \vec{E} = \epsilon_r \epsilon_0 \vec{E} \\ \vec{B} &= \mu \vec{H} = \mu_r \mu_0 \vec{H} \end{aligned} \tag{2.5}$$

Tabela 2.1: Lista de Variáveis 1

Variável	Descrição	Unidade
$\vec{E}$	Campo elétrico	$volts/metro$
$\vec{D}$	Densidade do fluxo elétrico	$Coulombs/metro^2$
$\vec{H}$	Campo magnético	$amperes/metro$
$\vec{B}$	Densidade do fluxo magnético	$webers/metro^2$
$\vec{J}$	Densidade da corrente elétrica	$amperes/metro^2$
$\vec{M}$	Densidade equivalente da corrente magnética	$volts/metro^2$

Tabela 2.2: Lista de Variáveis 2

Variável	Descrição	Unidade
ϵ	Permissividade elétrica	$Faradays/metro$
ϵ_r	Permissividade elétrica relativa	–
ϵ_0	Permissividade elétrica no vácuo	$8,854\,10^{-12}/Faradays/metro$
μ	Permeabilidade magnética	$henrys/metro$
μ_r	Permeabilidade magnética relativa	$amperes/metro^2$
μ_0	Permeabilidade magnética do vácuo	$4\pi\,10^{-7} henrys/metro$

Repare que $\vec{J}$ e $\vec{M}$ atuam como fontes independentes dos campos elétricos e magnéticos. Considerando a possibilidade de perda de intensidade dos campos com sua propagação, podemos reescrever os termos $\vec{J}$ e $\vec{M}$ como:

$$\begin{aligned} \vec{J} &= \vec{J}_{fonte} + \sigma\vec{E} \\ \vec{M} &= \vec{M}_{fonte} + \sigma^*\vec{H} \end{aligned} \tag{2.6}$$

onde σ representa a condutividade elétrica ($siemens/metro$) e σ^* a perda magnética equivalente ($ohms/metro$).

Realizando a substituição das equações (2.6) em (2.1) e (2.2) teremos as equações de Maxwell em sua forma diferencial linear, isotrópica, não dispersiva e

considerando perdas na propagação pelo tipo de material:

$$\begin{aligned}\frac{\partial \vec{H}}{\partial t} &= -\frac{1}{\mu}\nabla \times \vec{E} - \frac{1}{\mu}(\vec{M}_{fonte} + \sigma^* \vec{H}) \\ \frac{\partial \vec{E}}{\partial t} &= \frac{1}{\epsilon}\nabla \times \vec{H} - \frac{1}{\epsilon}(\vec{J}_{fonte} + \sigma \vec{E})\end{aligned} \tag{2.7}$$

Decompondo o operador rotacional, $\nabla\times$, podemos reescrever o sistema acima em componentes vetoriais de coordenadas cartesianas, o que resulta no sistema de equações (2.8) e (2.9).

$$\begin{aligned}\frac{\partial H_x}{\partial t} &= \frac{1}{\mu}\left(\frac{\partial E_y}{\partial z} - \frac{\partial E_z}{\partial y} - (M_x + \sigma^* H_x)\right) \\ \frac{\partial H_y}{\partial t} &= \frac{1}{\mu}\left(\frac{\partial E_z}{\partial x} - \frac{\partial E_x}{\partial z} - (M_y + \sigma^* H_y)\right) \\ \frac{\partial H_z}{\partial t} &= \frac{1}{\mu}\left(\frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x} - (M_z + \sigma^* H_z)\right)\end{aligned} \tag{2.8}$$

e

$$\begin{aligned}\frac{\partial E_x}{\partial t} &= \frac{1}{\varepsilon}\left(\frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z} - (J_x + \sigma E_x)\right) \\ \frac{\partial E_y}{\partial t} &= \frac{1}{\varepsilon}\left(\frac{\partial H_x}{\partial z} - \frac{\partial H_z}{\partial x} - (J_y + \sigma E_y)\right) \\ \frac{\partial E_z}{\partial t} &= \frac{1}{\varepsilon}\left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} - (J_z + \sigma E_z)\right)\end{aligned} \tag{2.9}$$

Este sistema é a base para a utilização do método numérico FDTD para a modelagem da interação e propagação das ondas eletromagnéticas.

2.2 Método FDTD

O método numérico que será adotado para solucionar as equações de Maxwell, neste trabalho, será o esquema de diferenças finitas desenvolvido por Kane Yee em 1966 YEE (1966), o mesmo é conhecido como esquema de Yee ou método FDTD (Finite-Difference Time-Domain). Este esquema é muito utilizado por ser eficaz e

conceitualmente simples, no entanto, sua implementação tem um custo computacional muito elevado, o que fez ele não despertar um grande interesse da comunidade cientifica na época de sua criação. Com a rápida redução do custo e o aumento do poder de processamento dos computadores, o número de trabalhos envolvendo o método vem crescendo muito nas últimas décadas (TAFLOVE; HAGNESS, 2000).

A base do algoritmo se encontra na forma como são discretizados os pontos no domínio. O método de Yee aproxima cada um dos campos em um ponto distinto do domínio, de tal forma que nenhum ponto nunca se sobreponha a outro, intercalando os campos de forma que cada componente elétrica fica no centro de 4 componentes magnéticas que a circulam e as componentes magnéticas também são rodeadas por 4 componentes elétricas. A Figura 2.1 representa esta distribuição espacial conhecida como célula de Yee. O método FDTD também intercala o tempo dos campos elétricos e magnéticos o que permite criar um método explicito.

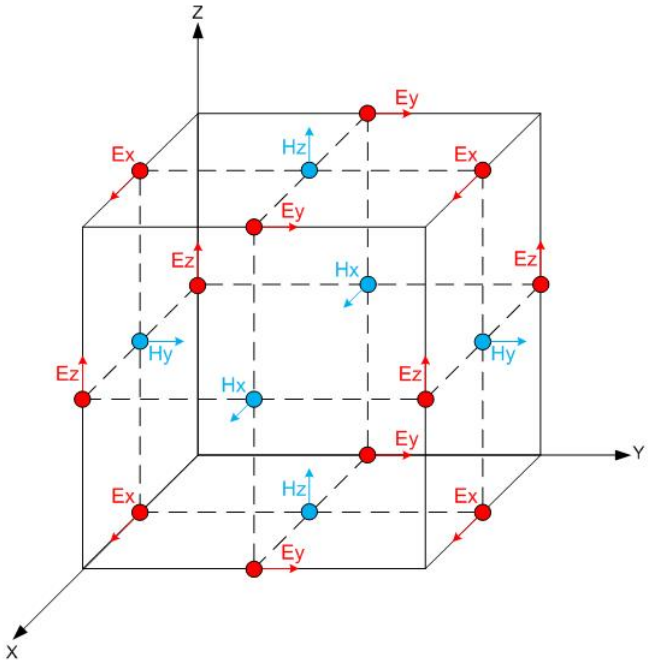

Figura 2.1: Célula de Yee.

O esquema de Yee utiliza aproximações por diferenças finitas centradas no tempo e espaço (esquema de tipo leapfrog) que apresentam erros de ordem 2. A notação utilizada na discretização do domínio será de acordo com

$$u|^n_{i\,j\,k} = u(n\Delta t, i\Delta x, j\Delta y, k\Delta z) \tag{2.10}$$

onde: Δx, Δy e Δz são os espaçamentos entre cada ponto da malha no espaço euclidiano (domínio discretizado) e Δt o espaçamento no tempo.

Aplicando diferenças finitas centradas nas derivadas parciais no tempo e no espaço temos

$$\begin{aligned}
\frac{\partial u|^n_{i\,j\,k}}{\partial x} &= \frac{u|^n_{i+\frac{1}{2}\,j\,k} - u|^n_{i-\frac{1}{2}\,j\,k}}{\Delta x} + O(\Delta x^2) \\
\frac{\partial u|^n_{i\,j\,k}}{\partial y} &= \frac{u|^n_{i\,j+\frac{1}{2}\,k} - u|^n_{i\,j-\frac{1}{2}\,k}}{\Delta y} + O(\Delta y^2) \\
\frac{\partial u|^n_{i\,j\,k}}{\partial z} &= \frac{u|^n_{i\,j\,k+\frac{1}{2}} - u|^n_{i\,j\,k-\frac{1}{2}}}{\Delta z} + O(\Delta z^2) \\
\frac{\partial u|^n_{i\,j\,k}}{\partial t} &= \frac{u|^{n+\frac{1}{2}}_{i\,j\,k} - u|^{n-\frac{1}{2}}_{i\,j\,k}}{\Delta t} + O(\Delta t^2)
\end{aligned} \tag{2.11}$$

Estas derivadas podem ser aproximadas, como mostram as equações abaixo, com erro de ordem 2 por:

$$\begin{aligned}
\frac{\partial u|^n_{i\,j\,k}}{\partial x} &\simeq \frac{u|^n_{i+\frac{1}{2}\,j\,k} - u|^n_{i-\frac{1}{2}\,j\,k}}{\Delta x} \\
\frac{\partial u|^n_{i\,j\,k}}{\partial y} &\simeq \frac{u|^n_{i\,j+\frac{1}{2}\,k} - u|^n_{i\,j-\frac{1}{2}\,k}}{\Delta y} \\
\frac{\partial u|^n_{i\,j\,k}}{\partial z} &\simeq \frac{u|^n_{i\,j\,k+\frac{1}{2}} - u|^n_{i\,j\,k-\frac{1}{2}}}{\Delta z} \\
\frac{\partial u|^n_{i\,j\,k}}{\partial t} &\simeq \frac{u|^{n+\frac{1}{2}}_{i\,j\,k} - u|^{n-\frac{1}{2}}_{i\,j\,k}}{\Delta t}
\end{aligned} \tag{2.12}$$

O esquema de Yee parte destas aproximações, aplicadas às derivadas dos campos elétrico e magnético, intercalando-as no tempo e no espaço, o que resulta num sistema explícito. Cada EDP do sistema é aproximada em um ponto distinto da malha

(discretização do domínio), o que resultará em índices com números inteiros e fracionários da forma $(i, j, k) \pm \frac{1}{2}$.

A equação do campo magnético será aproximada nos pontos $n+\frac{1}{2}, i+\frac{1}{2}, j, k$. Suas derivadas serão aproximadas por 2.13.

$$\begin{aligned}
\frac{\partial H_x|_{i+\frac{1}{2};j;k}^{n+\frac{1}{2}}}{\partial t} &\simeq \frac{H_x|_{i+\frac{1}{2};j;k}^{n+1} - H_x|_{i+\frac{1}{2};j;k}^{n}}{\Delta t} \\
\frac{E_y|_{i+\frac{1}{2};j;k}^{n+\frac{1}{2}}}{\partial z} &\simeq \frac{E_y|_{i+\frac{1}{2};j;k+\frac{1}{2}}^{n+\frac{1}{2}} - E_y|_{i+\frac{1}{2};j;k-\frac{1}{2}}^{n+\frac{1}{2}}}{\Delta z} \\
\frac{E_z|_{i+\frac{1}{2};j;k}^{n+\frac{1}{2}}}{\partial y} &\simeq \frac{E_z|_{i+\frac{1}{2};j+\frac{1}{2};k}^{n+\frac{1}{2}} - E_z|_{i+\frac{1}{2};j-\frac{1}{2};k}^{n+\frac{1}{2}}}{\Delta y}
\end{aligned} \tag{2.13}$$

Substituindo as aproximações no campo H_x teremos:

$$\begin{aligned}
H_x|_{i+\frac{1}{2};j;k}^{n+1} &= H_x|_{i+\frac{1}{2};j;k}^{n} + \\
&\frac{\Delta t}{\mu|_{i+\frac{1}{2};j;k}} \left\{ \begin{array}{l} \frac{E_y|_{i+\frac{1}{2};j;k+\frac{1}{2}}^{n+\frac{1}{2}} - E_y|_{i+\frac{1}{2};j;k-\frac{1}{2}}^{n+\frac{1}{2}}}{\Delta z} - \\ \frac{E_z|_{i+\frac{1}{2};j+\frac{1}{2};k}^{n+\frac{1}{2}} - E_z|_{i+\frac{1}{2};j-\frac{1}{2};k}^{n+\frac{1}{2}}}{\Delta y} - \\ \left(M_{fonte}|_{i+\frac{1}{2};j;k}^{n+\frac{1}{2}} + \sigma^*|_{i+\frac{1}{2};j;k} H_x|_{i+\frac{1}{2};j;k}^{n+\frac{1}{2}} \right) \end{array} \right\}
\end{aligned} \tag{2.14}$$

Note que pela forma escolhida da discretização no tempo, nós não possuímos os instantes $n+\frac{1}{2}$ de H_x. Então, iremos aproximá-lo pela média aritmética em $n+1$ e n:

$$H_x|_{i+\frac{1}{2};j;k}^{n+\frac{1}{2}} \simeq \frac{H_x|_{i+\frac{1}{2};j;k}^{n+1} + H_x|_{i+\frac{1}{2};j;k}^{n}}{2}$$

Assim, H_x é aproximado numericamente por:

$$H_x|^{n+1}_{i+\frac{1}{2};j;k} = \frac{1 - \dfrac{\sigma^*|_{i+\frac{1}{2};j;k}\Delta t}{2\mu|_{i+\frac{1}{2};j;k}}}{1 + \dfrac{\sigma^*|_{i+\frac{1}{2};j;k}\Delta t}{2\mu|_{i+\frac{1}{2};j;k}}} H_x|^{n}_{i+\frac{1}{2};j;k} +$$

$$\frac{\dfrac{\Delta t}{\mu|_{i+\frac{1}{2};j;k}}}{1 + \dfrac{\sigma^*|_{i+\frac{1}{2};j;k}\Delta t}{2\mu|_{i+\frac{1}{2};j;k}}} \left\{ \begin{array}{l} \dfrac{E_y|^{n+\frac{1}{2}}_{i+\frac{1}{2};j;k+\frac{1}{2}} - E_y|^{n+\frac{1}{2}}_{i+\frac{1}{2};j;k-\frac{1}{2}}}{\Delta z} - \quad \dfrac{E_z|^{n+\frac{1}{2}}_{i+\frac{1}{2};j+\frac{1}{2};k} - E_z|^{n+\frac{1}{2}}_{i+\frac{1}{2};j-\frac{1}{2};k}}{\Delta y} - \\ M_{fonte}|^{n+\frac{1}{2}}_{i+\frac{1}{2};j;k} \end{array} \right\}$$

Este processo é repetido analogamente para o restante das Equações de Maxwell sendo:

- H_y nos pontos $n + \frac{1}{2}, i, j + \frac{1}{2}, k$.
- H_z nos pontos $n + \frac{1}{2}, i, j, k + \frac{1}{2}$.
- E_x nos pontos $n + 1, i, j + \frac{1}{2}, k + \frac{1}{2}$.
- E_y nos pontos $n + 1, i + \frac{1}{2}, j, k + \frac{1}{2}$.
- E_z nos pontos $n + 1, i + \frac{1}{2}, j + \frac{1}{2}, k$.

Resultando neste sistema explicito de 6 equações:

Aproximação numérica do Campo Magnético

$$
\begin{aligned}
H_x|_{i+\frac{1}{2};j;k}^{n+1} &= Da|_{i+\frac{1}{2};j;k} H_x|_{i+\frac{1}{2};j;k}^{n} + Db|_{i+\frac{1}{2};j;k} \left\{ \begin{array}{l} \frac{E_y|_{i+\frac{1}{2};j;k+\frac{1}{2}}^{n+\frac{1}{2}} - E_y|_{i+\frac{1}{2};j;k-\frac{1}{2}}^{n+\frac{1}{2}}}{\Delta z} - \\ \frac{E_z|_{i+\frac{1}{2};j+\frac{1}{2};k}^{n+\frac{1}{2}} - E_z|_{i+\frac{1}{2};j-\frac{1}{2};k}^{n+\frac{1}{2}}}{\Delta y} - \\ M_{fonte}|_{i+\frac{1}{2};j;k}^{n+\frac{1}{2}} \end{array} \right\} \\
H_y|_{i;j+\frac{1}{2};k}^{n+1} &= Da|_{i;j+\frac{1}{2};k} H_y|_{i;j+\frac{1}{2};k}^{n} + Db|_{i;j+\frac{1}{2};k} \left\{ \begin{array}{l} \frac{E_z|_{i+\frac{1}{2};j+\frac{1}{2};k}^{n+\frac{1}{2}} - E_z|_{i-\frac{1}{2};j+\frac{1}{2};k}^{n+\frac{1}{2}}}{\Delta x} - \\ \frac{E_x|_{i;j+\frac{1}{2};k+\frac{1}{2}}^{n+\frac{1}{2}} - E_x|_{i;j+\frac{1}{2};k-\frac{1}{2}}^{n+\frac{1}{2}}}{\Delta z} - \\ M_{fonte}|_{i;j+\frac{1}{2};k}^{n+\frac{1}{2}} \end{array} \right\} \\
H_z|_{i;j;k+\frac{1}{2}}^{n+1} &= Da|_{i;j;k+\frac{1}{2}} H_z|_{i;j;k+\frac{1}{2}}^{n} + Db|_{i;j;k+\frac{1}{2}} \left\{ \begin{array}{l} \frac{E_x|_{i;j+\frac{1}{2};k+\frac{1}{2}}^{n+\frac{1}{2}} - E_x|_{i;j-\frac{1}{2};k+\frac{1}{2}}^{n+\frac{1}{2}}}{\Delta y} - \\ \frac{E_y|_{i+\frac{1}{2};j;k+\frac{1}{2}}^{n+\frac{1}{2}} - E_y|_{i-\frac{1}{2};j;k+\frac{1}{2}}^{n+\frac{1}{2}}}{\Delta x} - \\ M_{fonte}|_{i;j;k+\frac{1}{2}}^{n+\frac{1}{2}} \end{array} \right\}
\end{aligned}
\tag{2.15}
$$

Aproximação numérica do Campo Elétrico

$$
\begin{aligned}
E_x|_{i;j+\frac{1}{2};k+\frac{1}{2}}^{n+\frac{3}{2}} &= Ca|_{i;j+\frac{1}{2};k+\frac{1}{2}} E_x|_{i;j+\frac{1}{2};k+\frac{1}{2}}^{n+\frac{1}{2}} + Cb|_{i;j+\frac{1}{2};k+\frac{1}{2}} \left\{ \begin{array}{l} \frac{H_z|_{i;j+1;k+\frac{1}{2}}^{n+1} - H_z|_{i;j;k+\frac{1}{2}}^{n+1}}{\Delta y} - \\ \frac{H_y|_{i;j+\frac{1}{2};k+1}^{n+1} - H_y|_{i;j+\frac{1}{2};k}^{n+1}}{\Delta z} - \\ J_{fonte}|_{i;j+\frac{1}{2};k+\frac{1}{2}}^{n+1} \end{array} \right\} \\
E_y|_{i+\frac{1}{2};j;k+\frac{1}{2}}^{n+\frac{3}{2}} &= Ca|_{i+\frac{1}{2};j;k+\frac{1}{2}} E_y|_{i+\frac{1}{2};j;k+\frac{1}{2}}^{n+\frac{1}{2}} + Cb|_{i+\frac{1}{2};j;k+\frac{1}{2}} \left\{ \begin{array}{l} \frac{H_x|_{i+\frac{1}{2};j;k+1}^{n+1} - H_x|_{i+\frac{1}{2};j;k}^{n+1}}{\Delta z} - \\ \frac{H_z|_{i+1;j;k+\frac{1}{2}}^{n+1} - H_z|_{i;j;k+\frac{1}{2}}^{n+1}}{\Delta x} - \\ J_{fonte}|_{i+\frac{1}{2};j;k+\frac{1}{2}}^{n+1} \end{array} \right\} \\
E_z|_{i+\frac{1}{2};j+\frac{1}{2};k}^{n+\frac{3}{2}} &= Ca|_{i+\frac{1}{2};j+\frac{1}{2};k} E_z|_{i+\frac{1}{2};j+\frac{1}{2};k}^{n+\frac{1}{2}} + Cb|_{i+\frac{1}{2};j+\frac{1}{2};k} \left\{ \begin{array}{l} \frac{H_y|_{i+1;j+\frac{1}{2};k}^{n+1} - H_y|_{i;j+\frac{1}{2};k}^{n+1}}{\Delta x} - \\ \frac{H_x|_{i+\frac{1}{2};j+1;k}^{n+1} - H_x|_{i+\frac{1}{2};j;k}^{n+1}}{\Delta y} - \\ J_{fonte}|_{i+\frac{1}{2};j+\frac{1}{2};k}^{n+1} \end{array} \right\}
\end{aligned}
\tag{2.16}
$$

onde:

$$Da|_{i;j;k} = \frac{1 - \dfrac{\sigma^*|_{i;j;k}\Delta t}{2\mu|_{i;j;k}}}{1 + \dfrac{\sigma^*|_{i;j;k}\Delta t}{2\mu|_{i;j;k}}}; \qquad Db|_{i;j;k} = \frac{\dfrac{\Delta t}{\mu|_{i;j;k}}}{1 + \dfrac{\sigma^*|_{i;j;k}\Delta t}{2\mu|_{i;j;k}}};$$

$$Ca|_{i;j;k} = \frac{1 - \dfrac{\sigma|_{i;j;k}\Delta t}{2\epsilon|_{i;j;k}}}{1 + \dfrac{\sigma|_{i;j;k}\Delta t}{2\epsilon|_{i;j;k}}}; \qquad Cb|_{i;j;k} = \frac{\dfrac{\Delta t}{\epsilon|_{i;j;k}}}{1 + \dfrac{\sigma|_{i;j;k}\Delta t}{2\epsilon|_{i;j;k}}}.$$

O esquema FDTD resulta neste sistemas destas 6 equações acima apresentadas, elas formam um sistema explicito, ou seja, seu cálculo é direto, a partir das devidas condições iniciais do problema, sem a necessidade de inversões matriciais.

Para garantir a estabilidade e a convergência do esquema de Yee é necessário respeitar à condição de estabilidade de Courant-Friedrichs-Lewy (CFL) (TAFLOVE; HAGNESS, 2000) dada por:

$$\Delta t \leq \frac{1}{\upsilon\sqrt{\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}}}, \tag{2.17}$$

onde υ representa a velocidade de propagação da energia dos campos eletromagnéticos no meio dada por: $\upsilon = \frac{1}{\sqrt{\epsilon\mu}}$. No vácuo, a velocidade de propagação é igual a da luz, $3 \cdot 10^8$.

2.3 Equações de Maxwell e FDTD no Caso Bidimensional

Considerando que em relação a um sistema de coordenadas cartesianas fixado o campo eletromagnético e as características do meio não dependem da coordenada z, o sistema (2.7) pode ser reescrito em uma forma mais simplificada. O modo transversal elétrico TE_z (*transverse-electric mode*) é dado pelo seguinte sistema de

equações diferenciais parciais para as componentes E_x, E_y e H_z do campo eletromagnético

$$\begin{aligned}
\frac{\partial H_z}{\partial t} &= \frac{1}{\mu}\left(\frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x} - (M_z + \sigma^* H_z)\right), \\
\frac{\partial E_x}{\partial t} &= \frac{1}{\varepsilon}\left(\frac{\partial H_z}{\partial y} - (J_x + \sigma E_x)\right), \\
\frac{\partial E_y}{\partial t} &= \frac{1}{\varepsilon}\left(-\frac{\partial H_z}{\partial x} - (J_y + \sigma E_y)\right).
\end{aligned} \tag{2.18}$$

As demais componentes do campo eletromagnético satisfazem a um sistema semelhante ao (2.18) que é conhecido como modo transversal magnético TM_z (*transverse-magnetic mode*).

Esses sistemas de equações diferenciais parciais são desacoplados. Por isso devem ser resolvidos considerando uma região do plano cartesiano com coordenadas (x, y), introduzindo as condições iniciais e de contorno apropriadas.

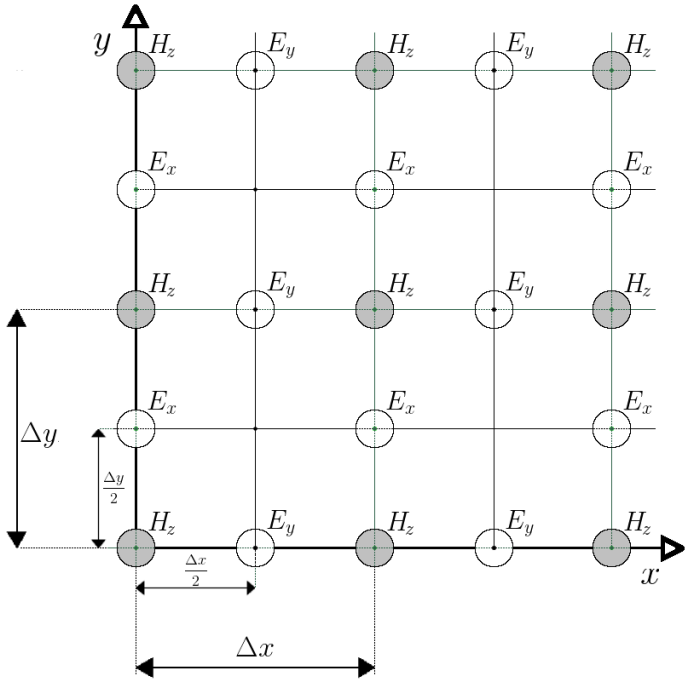

Figura 2.2: Malha de discretização do esquema de Yee para modo TE_z

Na Figura 2.2 apresentamos as malhas usadas na discretização do sistema (2.18). Observamos que cada componente elétrica é rodeada pelas componentes magnéticas. Aplicando aproximações de ordem 2 para as diferentes derivadas chegamos ao sistema explícito (2.19). Esse sistema também pode ser obtido a partir das equações (2.15) e (2.16) levando em conta que as funções não dependem de z.

O sistema de equações discretizadas tem a forma:

$$\begin{aligned}
H_z|_{i;j}^{n+1} &= Da|_{i;j} H_z|_{i;j}^{n} + Db|_{i;j} \left\{ \begin{array}{l} \frac{E_x|_{i;j+\frac{1}{2}}^{n+\frac{1}{2}} - E_x|_{i;j-\frac{1}{2}}^{n+\frac{1}{2}}}{\Delta y} - \\ \frac{E_y|_{i+\frac{1}{2};j}^{n+\frac{1}{2}} - E_y|_{i-\frac{1}{2};j}^{n+\frac{1}{2}}}{\Delta x} - M_z|_{i;j}^{n+\frac{1}{2}} \end{array} \right\} \\
E_x|_{i;j+\frac{1}{2}}^{n+\frac{3}{2}} &= Ca|_{i;j+\frac{1}{2}} E_x|_{i;j+\frac{1}{2}}^{n+\frac{1}{2}} + Cb|_{i;j+\frac{1}{2}} \left\{ \frac{H_z|_{i+;j+1}^{n+1} - H_z|_{i;j}^{n+1}}{\Delta y} - J_x|_{i;j+\frac{1}{2}}^{n} \right\} \\
E_y|_{i+\frac{1}{2};j}^{n+\frac{3}{2}} &= Ca|_{i+\frac{1}{2};j} E_y|_{i+\frac{1}{2};j}^{n+\frac{1}{2}} + Cb|_{i+\frac{1}{2};j} \left\{ -\frac{H_z|_{i+1;j}^{n+1} - H_z|_{i;j}^{n+1}}{\Delta y} - J_y|_{i+\frac{1}{2};j}^{n} \right\}
\end{aligned} \tag{2.19}$$

onde:

$$Da|_{i;j} = \frac{1 - \frac{\sigma^*|_{i;j}\Delta t}{2\mu|_{i;j}}}{1 + \frac{\sigma^*|_{i;j}\Delta t}{2\mu|_{i;j}}}; \qquad Db|_{i;j} = \frac{\frac{\Delta t}{\mu|_{i;j}}}{1 + \frac{\sigma^*|_{i;j}\Delta t}{2\mu|_{i;j}}};$$

$$Ca|_{i;j} = \frac{1 - \frac{\sigma|_{i;j}\Delta t}{2\epsilon|_{i;j}}}{1 + \frac{\sigma|_{i;j}\Delta t}{2\epsilon|_{i;j}}}; \qquad Cb|_{i;j} = \frac{\frac{\Delta t}{\epsilon|_{i;j}}}{1 + \frac{\sigma|_{i;j}\Delta t}{2\epsilon|_{i;j}}}$$

Este sistema também está sujeito à condição CFL (TAFLOVE; HAGNESS, 2000) que é dada por

$$\Delta t \leq \frac{1}{\upsilon\sqrt{\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}}},$$

onde υ representa a velocidade de propagação das ondas eletromagnéticas no meio, dada por $\upsilon = \frac{1}{\sqrt{\epsilon\mu}}$. Esta expressão também pode ser obtida fazendo $\Delta z \to \infty$ em (2.17).

A forma como os campos são atualizados está ilustrada na Figura 2.3. Nela vemos que para realizar o cálculo de um precisamos das informações de E e H do passo anterior.

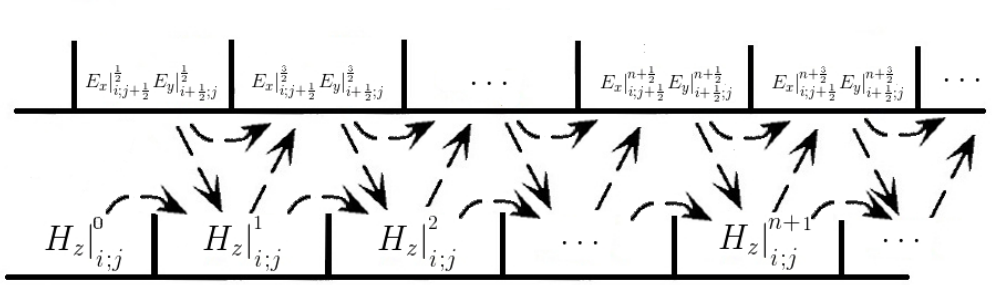

Figura 2.3: Ordem dos Cálculos

No esquema numérico (2.19) o campo elétrico é aproximado apenas nos tempos intermediários $(n+\frac{1}{2})\Delta t$, mas as condições iniciais são conhecidas apenas para $t = 0$, por isso se faz necessário aproximar o campo elétrico no tempo $t = \Delta t/2$. Neste trabalho calculamos essas aproximações usando o método de Euler explícito. Obtemos assim

$$\begin{aligned} E_x|^{\frac{1}{2}}_{i;j+\frac{1}{2}} &= \widetilde{Ca}|_{i;j+\frac{1}{2}} E_x|^0_{i;j+\frac{1}{2}} + \widetilde{Cb}|_{i;j+\frac{1}{2}} \left\{ \frac{H_z|^0_{i;j+1} - H_z|^0_{i;j}}{\Delta y} - J_x|^0_{i;j+\frac{1}{2}} \right\} \\ E_y|^{\frac{1}{2}}_{i+\frac{1}{2};j} &= \widetilde{Ca}|_{i+\frac{1}{2};j} E_x|^0_{i+\frac{1}{2};j} + \widetilde{Cb}|_{i+\frac{1}{2};j} \left\{ -\frac{H_z|^0_{i+1;j} - H_z|^0_{i;j}}{\Delta x} - J_y|^0_{i+\frac{1}{2};j} \right\} \end{aligned} \tag{2.20}$$

em que

$$\widetilde{Ca}|_{i;j} = 1 - \frac{\sigma|_{i;j}\Delta t}{2\epsilon|_{i;j}}; \qquad \widetilde{Cb}|_{i;j} = \frac{\Delta t}{2\epsilon|_{i;j}}.$$

2.4 Condições de Contorno

Na solução de equações diferencias as condições de contorno tem papel crucial para garantir a unicidade da solução do problema. Para domínios limitados, consideraremos condições de contorno em que o campo elétrico ou magnético na direção tangencial ao contorno é especificado. Nesta dissertação consideramos o caso em que o domínio é um retângulo com lados paralelos aos eixos x e y, por isso em cada lado é especificada uma das funções incógnitas. Nos referimos a essas condições de contorno como **condições de contorno de tipo Dirichlet**.

Quando lidamos com um problema da atuação de uma fonte se propagando, uma das primeiras soluções dadas a este problema é a utilização das condições de Dirichlet, isto é, atribuir o valor zero às células referentes a fronteira. No entanto, isto gera uma reflexão das ondas quando elas atingem o limite da malha.

Em muitas aplicações é necessário simular a propagação de ondas eletromagnéticas em um meio não limitado. Entretanto, devido à impossibilidade de se usar uma malha computacional infinita, muitas vezes temos que utilizar condições de contorno absorventes. As condições de contorno absorventes permitem que as ondas que atingem a fronteira da região de interesse abandonem essa região sem provocar reflexões espúrias, introduzindo de forma correta um espalhamento para fora dessa região. Os métodos conhecidos como *Absorbing Boundary Condition* (**ABC**), onde o objetivo é suprimir a reflexão numérica, geralmente segue duas linhas: a simulação de materiais absorventes ou de forma analítica.

O princípio subjacente as 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 (PEREIRA, 2012). Nesta linha, um

dos algoritmos mais populares é o de *Mur* (MUR, 1981) pela implementação relativamente simples e por resultar em um coeficiente de reflexão baixo.

A outra forma de criar algoritmos ABC é pela simulação de camadas extras de células rodeando o domínio de interesse com propriedades de absorção da energia de propagação das ondas eletromagnéticas. Isto é atingido simulando um meio especial projetado para absorção, assim inibindo a reflexão das ondas eletromagnéticas. Nesta linha, uma forma muito eficiente de introduzir condições de contorno absorventes no método FDTD foi feita por Bérenger em 1994 BERENGER (1994), sua proposta é chamada de PML (*Perfectly Matched Layer*). Está técnica se tornou muito popular visto que também é de fácil implementação e, geralmente, incide num erro menor de reflexão do que as **ABC** do tipo analítico.

Nessa abordagem, são introduzidas camadas extras de células (chamadas de PML) rodeando o domínio de interesse em quatro regiões: acima, à esquerda, à direita e abaixo, como ilustrado na Figura 2.4. Em cada célula da PML, a condutividade elétrica é determinada de forma que as ondas eletromagnéticas penetrem sem reflexão na interface entre o meio de estudo e a PML, para qualquer frequência de onda e ângulo de incidência. Isto é feito respeitando a condição da equação (2.21) para garantir que a impedância do meio de análise seja igual a da PML, resultando em reflexão zero (BERENGER, 1994).

$$\frac{\sigma}{\epsilon} = \frac{\sigma^*}{\mu} \tag{2.21}$$

Para a formulação em duas fases da PML de Bérenger, a componente H_z é dividida em H_{zx} e H_{zy}. Assim, as equações de Maxwell modificadas para o caso

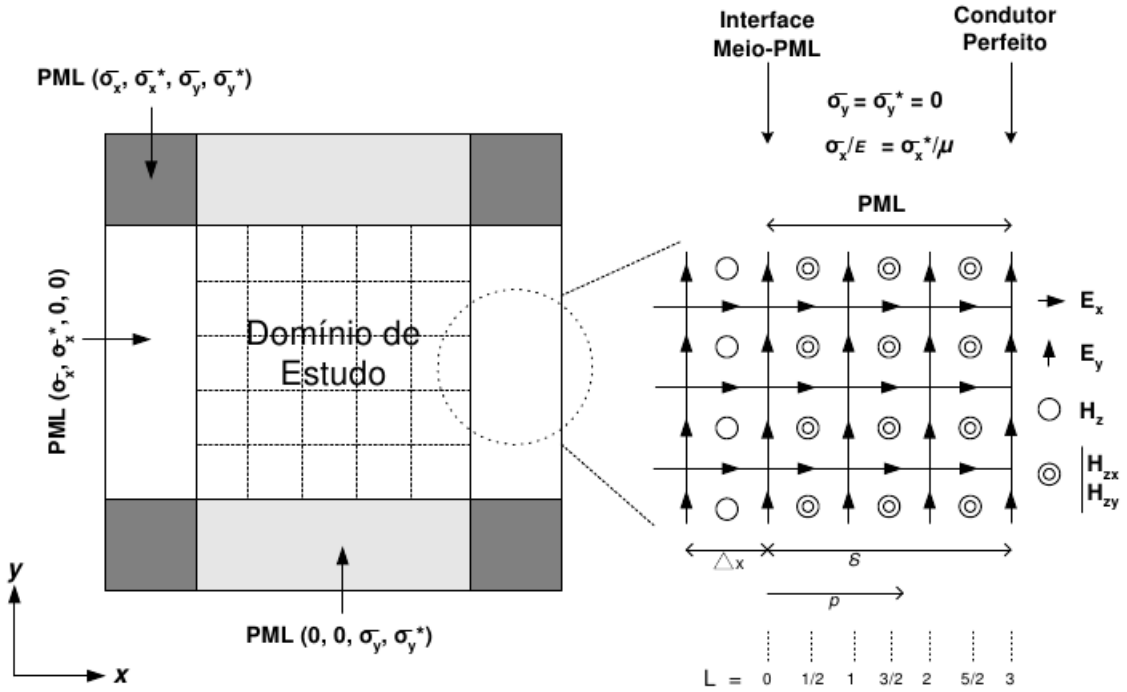

Figura 2.4: TE_z-PML entorno do domínio de análise. fonte: LIMA (2006)

TE_z ficam na forma do sistema de equações em (2.22).

$$\begin{aligned}
\epsilon\frac{\partial E_x}{\partial t} + \sigma_y E_x &= \frac{\partial H_z}{\partial y} \\
\epsilon\frac{\partial E_y}{\partial t} + \sigma_x E_y &= -\frac{\partial H_z}{\partial y} \\
\mu\frac{\partial H_{zx}}{\partial t} + \sigma_x^* H_{zx} &= -\frac{\partial E_y}{\partial x} \\
\mu\frac{\partial H_{zy}}{\partial t} + \sigma_y^* H_{zy} &= \frac{\partial E_x}{\partial y}
\end{aligned} \tag{2.22}$$

onde,

$$H_z = H_{zx} + H_{zy}.$$

Observe que a extensão do esquema de Yee (2.19) para o sistema acima é bastante simples e direta.

Entretanto, para garantir uma correta implementação dessas condições de contorno absorventes é necessário determinar o número de camadas que irão compor a PML e as condutividades correspondentes, de acordo com o grau de reflexão desejado. Em tese, a taxa de reflexão pode se tornar tão pequena quanto desejável, controlando a espessura da PML. Existem várias formas de determinar a variação espacial da condutividade dentro das camadas da PML, por exemplo, linear, parabólica, polinomial e geométrica BERENGER (1994); TAFLOVE; HAGNESS (2000). As formas polinomial e geométrica têm apresentado os melhores resultados TAFLOVE; HAGNESS (2000). Por isso, neste trabalho, utilizaremos a PML polinomial. Para seu cálculo, seguimos os seguintes passos:

- Cálculo da condutividade da PML na última camada, ou seja, camada mais externa que por sua vez apresenta a maior das condutividades.

$$\sigma_{max} = -\frac{(m+1)\ln(R(0))}{2\eta d}$$

  onde m é o grau do polinômio para a modelagem de σ, geralmente 3 ou 4; $R(0)$ é o fator de reflexão; $\eta = \sqrt{\frac{\mu}{\epsilon}}$ é a impedância intrínseca do meio; e d é a espessura da PML dada por Δx ou Δy multiplicado pelo seu número de células.

  Seja N o número de células para a PML, de acordo com TAFLOVE; HAGNESS (2000) para $N = 10$ usa-se $R(0) = e^{-16}$ e para $N = 5$ usa-se $R(0) = e^{-8}$. Esses valores são considerados ótimos para um grande número de implementações do FDTD.

- As demais condutividades são determinadas utilizando (2.23). Esta etapa está exemplificada na equação (2.23) para uma PML na direção x.

$$\sigma_x(k) = \sigma_{\max}\left(\frac{k}{N}\right)^m, \quad k = 0, 1, 2, .., N \tag{2.23}$$

onde k é a camada da PML, considerando $k = 0$ para a camada de células que compõem a fronteira do domínio de análise e $k = N$ é a última camada da PML.

Após o cálculo de $\sigma_x(k)$ para todas as camadas da PML, devemos utilizar a relação da equação (2.21) para determinar os $\sigma_x^*(k)$ correspondentes.

3 EXEMPLOS DE IMPLEMENTAÇÃO E VALIDAÇÃO

Neste capítulo apresentamos e discutimos os resultados de vários exemplos de aplicação do esquema de Yee, gerados com um código implementado usando o software Matlab. Esses exemplos também foram resolvidos usando três implementações diferentes: sequencial, paralela para CPU e paralela para GPU, e obtivemos nos três casos resultados numéricos compatíveis. Estas três implementações e resultados numéricos serão apresentados no capítulo 4.

Na primeira seção será apresentado a simulação da propagação de uma onda plana estacionária em um domínio limitado. Como nesse caso a solução é dada por uma expressão analítica, isso nos permite avaliar a corretude dos códigos implementados mediante uma análise dos erros obtidos nas aproximações correspondentes.

Na segunda seção, apresentamos a simulação da propagação de uma onda, gerada por um pulso pontual, em um domínio limitado considerando condições de contorno de tipo Dirichlet.

Na terceira seção, em contraposição ao exemplo apresentado na segunda seção, simulamos a propagação desse pulso em um domínio não limitado. Para isso utilizamos os códigos que implementam condições de contorno absorventes usando PML.

3.1 Propagação de uma onda plana estacionária em um domínio limitado

Neste exemplo, considerando que $\epsilon = \mu = 1$, podemos comprovar que para qualquer valor do parâmetro A as funções

$$\begin{aligned} H_z(t,x,y) &= -\frac{2A}{3}\cos(\sqrt{3}t)\cos(x+\sqrt{2}y) \\ E_x(t,x,y) &= A\sin(\sqrt{3}t)\sin(x+\sqrt{2}y) \\ E_y(t,x,y) &= -\frac{A}{\sqrt{2}}\sin(\sqrt{3}t)\sin(x+\sqrt{2}y) \end{aligned} \tag{3.1}$$

dão uma solução das equações de Maxwell (2.18). Esta solução corresponde a uma onda plana estacionária com vetor de onda $\mathbf{k} = (1,\ \sqrt{2})$.

Essa solução foi usada para avaliar a acurácia das aproximações numéricas em nossas implementações do esquema de Yee (2.19). Por se tratar de um método de ordem 2 é esperado encontrar um erro de ordem quadrática.

As soluções numéricas de (2.19) foram obtidas para o domínio retangular $\Omega = [0, 2\pi] \times [0, \sqrt{2}\pi]$ e o intervalo de tempo $[0, 4]$.

Usamos as condições iniciais definidas por (3.1) (com $t = 0$).

$$\begin{aligned} E_x|^0_{i;j+\frac{1}{2}} &= 0 \\ E_y|^0_{i+\frac{1}{2};j} &= 0 \\ H_z|^0_{i;j} &= -\frac{2A}{3}\cos(x_i+\sqrt{2}y_j) \end{aligned} \tag{3.2}$$

As condições de contorno utilizadas foram do tipo Dirichlet para H_z definidas pela terceira equação de (3.1), ou seja:

$$H_z(t,x,y)|_\Gamma = \begin{cases} -\frac{2A}{3}\cos(\sqrt{3}t)\cos(\sqrt{2}y), & \text{se } x = 0 \text{ ou } 2\pi,\ y \in [0, \sqrt{2}\pi]; \\ -\frac{2A}{3}\cos(\sqrt{3}t)\cos(x), & \text{se } x \in [0, 2\pi],\ y = 0 \text{ ou } \sqrt{2}\pi. \end{cases} \tag{3.3}$$

Estamos interessados em validar o algoritmo através de sua proximidade com a solução analítica. Para a avaliação da exatidão das aproximações calculamos o erro na norma do máximo $\|\cdot\|_\infty$ para diferentes escolhas da malha discreta com o intuito de mostrar sua convergência para zero na medida que a malha vai se tornando mais fina.

Teoricamente temos que o erro observado para qualquer ponto da grade de pontos deve seguir a relação da equação

$$\|E\|_\infty \leq C \max\{\Delta t, \Delta x, \Delta y\}^2, \tag{3.4}$$

onde C é uma constante real positiva.

Primeiramente, verificamos o comportamento do erro para uma discretização de 200×200 células ($\Delta x \simeq 3.124\,10^{-2}$ e $\Delta y \simeq 2.210\,10^{-2}$) e $N_t = 222$ ($\Delta t \simeq 1.801\,10^{-2}$), número obtido pelo critério de Courant. A Figura 3.1 apresenta o erro, na norma do máximo, em cada passo no tempo para os 3 campos individualmente.

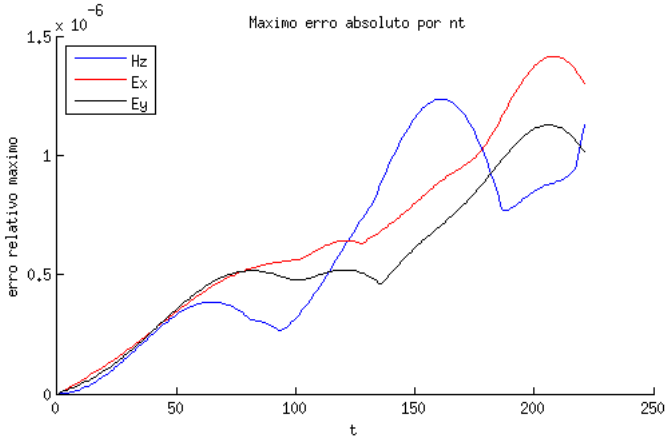

Figura 3.1: Máximo Erro Absoluto para as diferentes componentes do campo eletromagnético.

Note que os erros tem um comportamento cíclico e o erro máximo obtido foi da ordem de 10^{-6}. Como $max(\Delta t, \Delta x, \Delta y)^2$ está na ordem de 10^{-4}, a aproximação

obtida está dentro do esperado. Observe também que o erro obtido na aproximação de H_z possui picos um pouco mais acentuado que os demais campos. Isso se dá por H_z possuir duas aproximações de derivadas espaciais enquanto que E_x e E_x possuem apenas uma deste tipo.

Para continuar as avaliações com diferentes discretizações do método numérico, vamos trabalhar com malhas quadradas, ou seja, o número de pontos tomados na discretização na direção x é igual aos da direção y . O que torna válida a relação $\Delta x > \Delta y$, considerando o domínio $\Omega = [0, 2\pi] \times [0, \sqrt{2}\pi]$. Assim, podemos analisar a ordem de precisão do erro observado nos restringindo apenas a Δt e Δx.

Desta forma, resulta em malhas tais que $\Delta x = \frac{2\Delta y}{\sqrt{2}}$ e $\Delta t \leq \Delta x/\sqrt{3}$ (condição CFL). Dessa forma esperamos que $\|E\|_\infty \simeq C(\Delta x)^q$, ou seja, uma taxa de convergência quadrática dominada pelo espaçamento adotado em x, o que resulta numa taxa de convergência (empírica) observada nas simulações que pode ser determinada como:

$$\frac{\|E_{j+1}\|_\infty}{\|E_j\|_\infty} \simeq \frac{C(\Delta x_{j+1})^q}{C(\Delta x_j)^q}$$

Aplicando logaritmo podemos aproximar q, a ordem de precisão numérica aproximada do erro, pela expressão:

$$q \simeq \frac{\log \frac{\|E_{j+1}\|_\infty}{\|E_j\|_\infty}}{\log \frac{(\Delta x_{j+1})}{(\Delta x_j)}} \tag{3.5}$$

onde $\|E_j\|_\infty$, Δx_j e $\|E_{j+1}\|_\infty$, Δx_{j+1} representam os valores do erro e do espaçamento da malha nas j-ésima e $j+1$-ésima simulações, respectivamente.

Desse modo, a partir dos resultados numéricos é possível identificar a taxa de convergência empírica para avaliar se os resultados numéricos apresentam o comportamento teórico esperado.

A Tabela 3.1 apresenta os resultados de 8 simulações, variando-se o tamanho da discretização. A tabela mostra os erros observados e a taxa de convergência estimada conforme (3.5).

j	Δt_j	Δx_j	$\|E_j\|_\infty$	q (empírico)
1	4×10^{-3}	$6,35 \times 10^{-2}$	$1,34 \times 10^{-3}$	–
2	4×10^{-3}	$3,16 \times 10^{-2}$	$3,22 \times 10^{-4}$	2.053
3	4×10^{-3}	$2,10 \times 10^{-2}$	$1,34 \times 10^{-4}$	2.161
4	4×10^{-3}	$1,57 \times 10^{-2}$	$6,78 \times 10^{-5}$	2.347
5	4×10^{-3}	$1,26 \times 10^{-2}$	$3,74 \times 10^{-5}$	2.652
6	4×10^{-4}	$1,05 \times 10^{-2}$	$2,10 \times 10^{-5}$	3.167
7	4×10^{-4}	$8,99 \times 10^{-3}$	$1,11 \times 10^{-5}$	4.136
8	4×10^{-4}	$7,86 \times 10^{-3}$	$4,46 \times 10^{-6}$	6.483

Tabela 3.1: Valores do erro e da taxa de convergência empírica para diferentes discretizações

Note que os erros obtidos, em geral, são inferiores a $(\Delta x)^2$. Além disso, temos que a ordem de precisão estimada se mostrou maior do que dois e com um comportamento crescente à medida que o tamanho da malha cresce, implicando numa precisão acima do esperado. Assim, dado que os resultados encontrados numericamente estão próximos da solução exata dentro de uma margem de erro inferior ao limite teórico, podemos concluir que a solução numérica está sendo calculada corretamente para esse exemplo.

3.2 Propagação de um pulso em um domínio limitado

Nesse exemplo vamos mostrar a influência de condições de fronteira do tipo Dirichlet considerando todos os $H_z|_\Gamma^0 = 0$. Para isso simulamos numericamente a propagação de uma onda, gerada por um pulso pontual.

Consideramos que a propagação acontece no vácuo, ou seja $\epsilon = 8.854 \times 10^{-12}$

e $\mu = 4\pi \times 10^{-7}$ e portanto $\upsilon = (\epsilon\mu)^{-1/2} \approx 3\,10^8$. O domínio Ω foi discretizado com uma malha formada por 150×150 células quadradas de dimensão $\Delta x = \Delta y = \Delta = 0.01$ e para o tamanho do passo de tempo escolhemos $\Delta t = \frac{\Delta}{2\upsilon} = 6 \times 10^{-11}$.

No ponto central de Ω atua uma fonte pontual com intensidade:

$$H_z|^n = \begin{cases} \frac{1}{32}\left\{10 - 15\cos(n\pi/20) + 6\cos(2n\pi/20) - \cos(3n\pi/20)\right\}, & n \leq 40, \\ 0, & n > 40. \end{cases} \tag{3.6}$$

Os gráficos da figura 3.2 mostram a propagação do pulso para algumas variações de passos no tempo. Assim, podemos observar o comportamento quando energia do campo alcança as fronteiras.

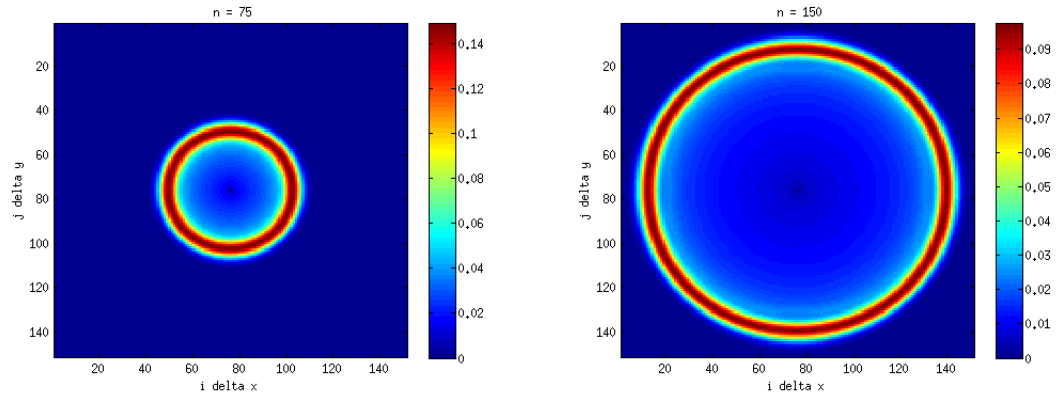

Figura 3.2: Propagação do pulso pontual dado por (3.6) no vácuo - (a) após 75 passos de tempo e (b) após 150 passos de tempo.

Podemos observar que a propagação da onda inicialmente é de forma rádial, no entanto, quando a onda atinge a fronteira ocorre uma reflexão total da onda, devido ao uso das condições de contorno de Dirichlet (Figura 3.3).

Devido a esta reflexão, quando desejamos simular um meio não limitado por esta abordagem devemos realizar um aumento da malha computacional de forma,

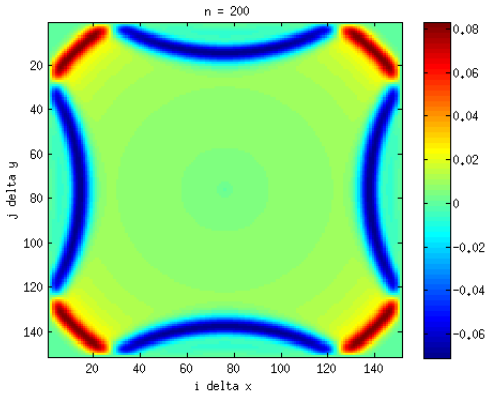

Figura 3.3: Propagação do pulso pontual dado por (3.6) no vácuo, após 200 passos de tempo.

que a a onda não atinja a fronteira do domínio de interesse dentro do tempo de análise. Isso seria muito custoso computacionalmente e por isso o enfoque mais apropriado é usar condições de contorno absorventes.

3.3 Propagação de um pulso em um domínio não limitado

Conforme visto na seção anterior, as simulações usando condições de contorno de Dirichlet geram reflexão total das ondas eletromagnéticas quando estas atingem a fronteira da malha computacional. Esta reflexão cria uma limitação quando temos interesse em simular ambientes abertos. No seguinte exemplo mostramos o enfoque alternativo baseado em condições de contorno absorventes usando PML.

Para isto, vamos simular numericamente a propagação de uma onda cilíndrica gerada pelo pulso pontual (3.6) no vácuo, usando PML. Para avaliar a corretude da implementação vamos determinar a quantidade de energia espúria que foi refletida devido à presença da PML. A energia refletida é calculada usando uma outra solução

aproximada mas em um domínio suficientemente extenso para que as condições de fronteira não tenham nenhuma influência (TAFLOVE; HAGNESS, 2000).

A primeira malha possui 50×50 células que representam uma discretização da região Ω_0 e inclui uma PML. Os parâmetros da discretização são $\Delta x = \Delta y = \Delta = 0.01$ e $\Delta t = \frac{\Delta}{2v}$. Nas simulações usamos uma PML com $N = 5$ e 10 camadas e uma variação da condutividade de tipo polinomial de grau $m = 4$. As aproximações obtidas nesse caso são representadas por $H_z^{\mathrm{PML}}|_{i,j}^n$. Na segunda consideramos um domínio truncado com as condições de Dirichlet, i. e., sem o uso de PML. Este domínio contém o Ω_0 e o chamaremos de Ω_D. Sua discretização será de 200×200 e os nós da interseção dos domínios são sobrepostos. As aproximações correspondentes são a Ω_D são representadas por $H_z^{\infty}|_{i,j}^n$. Na Figura 3.4 apresentamos à ilustração dos domínios Ω_0 e Ω_D.

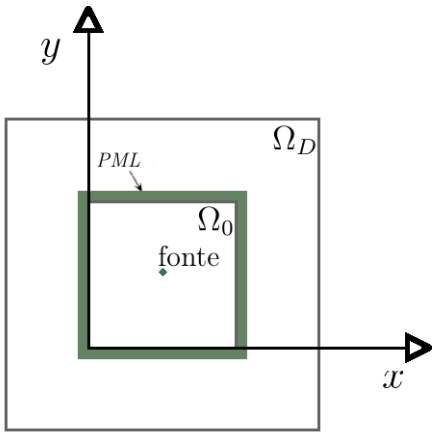

Figura 3.4: Região referente a Ω_0 e Ω_D.

Nas simulações calculamos a propagação do pulso (3.6), durante 200 passos de tempo, considerando condições iniciais nulas. A energia média refletida para o interior da região Ω_0 ao longo da sua fronteira $\partial\Omega_0$ no passo de tempo n, devido à

presença da PML, pode ser calculada por

$$\mathcal{E}^n_{\text{ref}} = 1/N_f \sum_{(x_i,y_j)\in\partial\Omega_0} \left[H_z^{PML}|^n_{i,j} - H_z^{\infty}|^n_{i,j}\right]^2,$$

em que N_f é a quantidade de pontos em $\partial\Omega_0$. Dessa forma a presença da PML produz um erro adicional na região Ω_0 que pode ser calculado como

$$E^n_{\text{PML}} = 1/N_d \sum_{(x_i,y_j)\in\Omega_0} \left[H_z^{PML}|^n_{i,j} - H_z^{\infty}|^n_{i,j}\right]^2,$$

em que N_d é a quantidade de pontos em Ω_0.

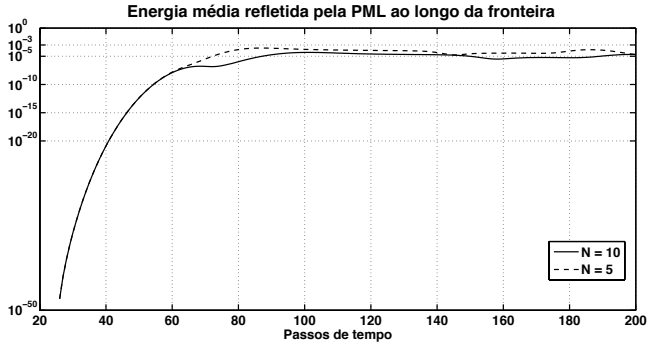

Figura 3.5: Energia média refletida ao longo da PML para dentro da região Ω_0 quando a PML contem 5 e 10 camadas

Na Figura 3.5 apresentamos gráficos da energia média refletida ao longo da fronteira da PML quando ela contem 5 e 10 camadas. Observamos que após a onda atingir a fronteira começam a aparecer as reflexões espúrias que depois decrescem lentamente se estabilizando em torno do valor 10^{-5}. Esse valor parece compatível com os erros numéricos detectados no primeiro exemplo.

Na Figura 3.6, os gráficos correspondem ao erro médio quadrático na região Ω_0. Podemos observar como as reflexões espúrias causadas pela PML produzem um aumento dos erros, mas estes também se estabilizam em torno do valor 10^{-5}. Ambas as figuras indicam que o resultado para o caso de 10 camadas foi um pouco melhor.

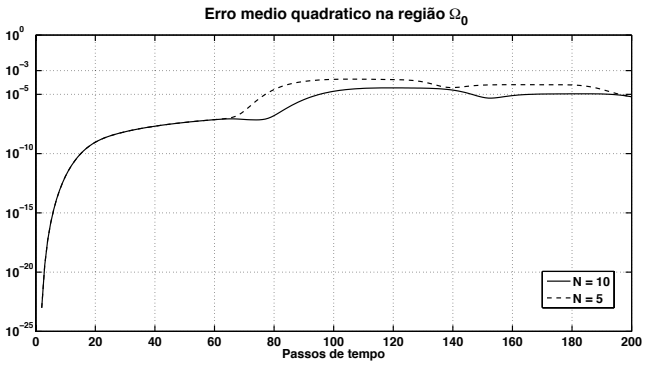

Figura 3.6: Erro médio quadrático na região Ω_0 devido à presença de PMLs contendo 5 e 10 camadas

Para observarmos o efeito da onda quando ela atinge a fronteira, vamos refazer a simulação considerando a dimensão utilizada na seção anterior, 150×150. Na Figura 3.7 temos uma visualização do encontro da onda, no passo de tempo $N_t = 200$, com a fronteira do domínio de análise que possui uma *PML* com $N = 10$ camadas em cada lado. Como pode ser observado, diferentemente do efeito reflexivo visto na Figura 3.3, a onda abandona a região sem provocar uma reflexão perceptível.

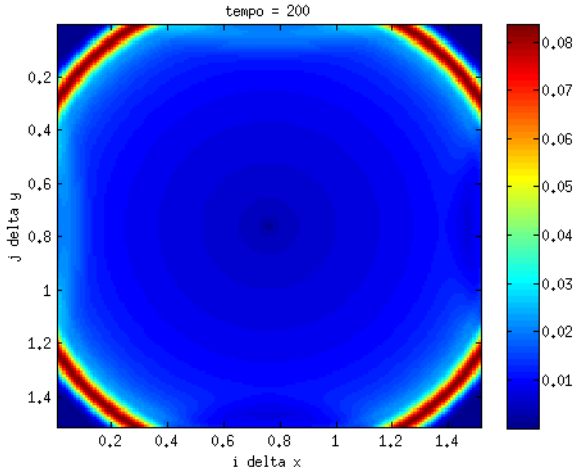

Figura 3.7: Propagação do pulso pontual dado por (3.6) no vácuo em encontro a *PML*, após 200 passos de tempo.

4 IMPLEMENTAÇÕES PARALELAS DO MÉTODO FDTD

Existem diversos ambientes de programação paralela e cada um tem apresentado resultados muito significativos em diversas aplicações. Muitos destes ambientes podem ser vistos de forma aprofundada em (BRYANT; O'HALLARON, 2011). A contar de 2004, os processadores estão alcançando seu limite teórico de clocks/segundo pela tecnologia atual. Dada esta limitação, o aumento do poder de processamento vem sendo realizado por meio da utilização de múltiplos núcleos de processamento em apenas uma CPU. As unidades de processamento gráfico (GPUs – *General Processing Units*) possuem um número elevado de núcleos de processamento que nos permite dispor de computação de alto desempenho por um custo relativo muito baixo (KIRK; HWU, 2010; PATTERSON; HENNESSY, 2013).

Neste capítulo vamos abordar os ambientes e técnicas de programação paralela que serão utilizadas para reduzir o tempo de processamento do método FDTD. Para todas as implementações adotaremos a linguagem C, vista como uma boa escolha para implementações de aplicações de alto desempenho (CHELLAPPA; FRANCHETTI; PÜSCHEL, 2008). Foram implementadas três versões para o caso bidimensional do FDTD, sendo a primeira um programa sequencial, a segunda utilizando programação paralela voltada para processadores com vários núcleos de processamento e a terceira utilizando programação paralela aplicada a GPU por meio de CUDA. Isto nos permitirá avaliar em qual ambiente de programação o método numérico terá melhor desempenho em relação ao tempo de processamento.

Na seção 4.1 serão abordadas técnicas de otimização de código voltadas para programação sequencial, tais como as que utilizam hierarquia de memória, *pipeline* e

as otimizações oferecidas pelo compilador *gcc*. Na seção 4.2 discutiremos estratégias de programação paralela voltadas para processadores multinúcleos com auxílio da biblioteca *pthread*. Por fim, na seção 4.3, apresentaremos a tecnologia CUDA, que permite a programação com uso da GPU. Para cada um desses ambientes, descreveremos os algoritmos propostos para implementar o método FDTD.

4.1 Implementação sequencial

Um código executado por apenas uma *thread* — unidade independente de execução inerente a um processo — é conhecido como código sequencial (ou *single-thread*). Esta é a estrutura mais simples e direta de implementação de um código. No entanto, ela apresenta algumas formas de otimização que podem ser complexas e, muitas vezes, nada intuitivas, mas resultam num ganho de desempenho muito significativo e que, em alguns casos, supera o ganho de desempenho adquirido na programação paralela.

Para a construção de um código eficiente devemos, primeiramente, ter conhecimento sobre a estrutura de acesso à memória. O acesso à memória é um dos fatores mais relevantes na construção de um algoritmo quando buscamos sua otimização. Existem diferentes tipos de memória e cada uma possui características distintas em termos de capacidade e velocidade de acesso, formando assim uma hierarquia de memória (Figura 4.1). Esta hierarquia forma uma pirâmide com a seguinte estrutura: em seu topo temos os registradores, que são a memória de maior velocidade, seguidos pelas memórias cache L1, L2 e L3; logo depois vem a memória RAM, as memórias de disco e a memória de rede.

A quantidade de ciclos para acessar os dados depende da parte da memória em que eles estão. Para exemplificar, se um programa precisa de uma informação

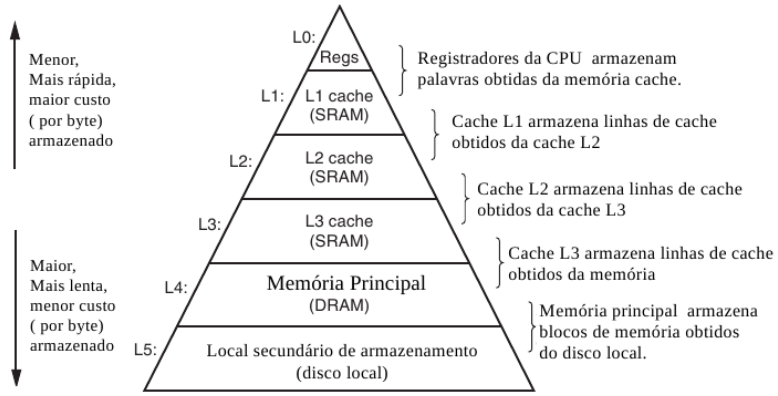

Figura 4.1: Hierarquia de memoria. Fonte: BRYANT; O'HALLARON (2011)

que ainda está no registrador, ela pode ser acessada com zero ciclo de processamento. Caso ela se encontre na memória cache, este exercício pode levar de 1 a 30 ciclos. Se a informação estiver na memória RAM, o tempo aumenta aproximadamente 10 vezes, podendo chegar a 300 ciclos. Quando ela está em disco, podem ser necessários até 10 milhões de ciclos (10 Mhz) para acessá-la BRYANT; O'HALLARON (2011).

Como pode ser visto em (CHELLAPPA; FRANCHETTI; PÜSCHEL, 2008), construir um código que possa tirar vantagem da hierarquia de memória é uma das mais importantes abordagens para produzir um código rápido. Para um melhor uso da memória duas "regras" principais devem ser seguidas:

- Reuso da memória temporária. Uma vez que um dado é levado para memória cache, ou mesmo para um registrador, é importante o programa estar estruturado para reutilizá-lo quantas vezes forem possíveis para evitar um futuro acesso lento.

- Uso da vizinhança. Os dados são trazidos em blocos para memória cache, o que nos permite tirar proveito da localidade física dos dados. Podemos exemplificar

com cálculo das posições de uma matriz onde calcular suas entradas fixando a linha e varrendo as colunas é mais eficiente quando trabalhamos com a linguagem C, pois nesta uma matriz de dados é armazenada na memória na forma um vetor linha.

O algoritmo sequencial que desenvolvemos para o método FDTD visa explorar as otimizações apresentadas acima. A estrutura geral do código sequencial está expressa nos fluxogramas apresentados na Figura 4.2. O fluxograma à esquerda usa condições de contorno específicas de Dirichlet e o fluxograma à direita usa condições de contorno absorventes (PML).

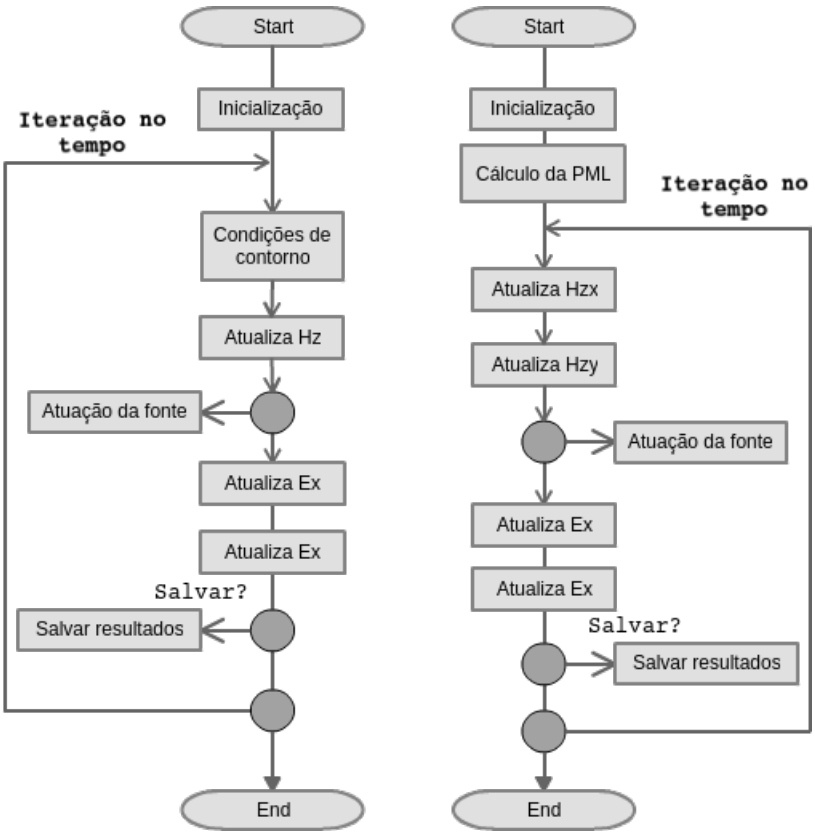

Figura 4.2: Algoritmos sequenciais do método FDTD: (a) Dirichlet (b) PML

Para o primeiro código, com condição de contorno de Dirichlet, temos uma etapa de inicialização das variáveis onde são definidas as propriedades do meio e a discretização do domínio em malhas. Os campos elétricos e magnéticos são declarados de forma vetorial, utilizando indexação explícita para referenciar as posições da matriz. A atualização dos campos é feita conforme o código 4.1, onde temos a execução de dois *loops*. Os *loops* percorrem os elementos da matriz em linha, o que permite tirar proveito dos dados trazidos em blocos para a memória. Além disso, note no código 4.1 que os índices das matrizes são armazenados em variáveis temporárias para facilitar o seu reuso. Os campos E_x e E_y são atualizados da mesma forma. A alternativa de salvar os resultados em disco é opcional e pode ser feita de forma cíclica para economia de espaço e redução do alto custo de escrita em disco. Por exemplo, podemos salvar as soluções intermediárias a cada 100 passos no tempo.

```
for (i=1;i<nx-1;i++) {
        int I=ny*i; // indice i+1 ou i+1/2
        int I_1=ny*(i-1);// indice i ou i-1/2
        for (j=1;j<ny-1;j++) {
                Hz[I+j]=Da_Hz[I+j]*Hz[I+j]+Db_Hz[I+j]*(dx*( Ex
                   [I+j] -Ex[I+j-1]) - dy*(Ey[I+j] -Ey[I_1+j])
                    );
    }
}
```

Listing 4.1: Cálculo do campo H_z

O código PML difere em sua inicialização pois temos uma etapa de cálculo da PML e definição da camada extra de células. Neste código não precisamos especificar condições de contorno e temos o cálculo de um campo extra, dada a divisão do H_z em duas componentes. O cálculo de uma componente extra demanda cerca de 33% a mais de memória, além de um acréscimo pequeno devido às novas camadas que envolvem o domínio.

Para melhorar o desempenho dos códigos apresentados, utilizamos também as opções de otimização oferecidas pelo próprio compilador *gcc*. Estas opções per-

mitem um ganho de desempenho bastante considerável STALLMAN (2010). Elas conseguem reestruturar o código no momento da compilação para uso de diversas técnicas de otimização automaticamente, como as apresentadas e outras. Para efeito de comparação, usaremos as opções "$-O0$"(sem otimização), "$-O1$", "$-O2$"e "$-O3$".

Existem algumas outras técnicas como *paralelismo a nível de instrução* e outras que podem ser vistas em (CHELLAPPA; FRANCHETTI; PÜSCHEL, 2008). No entanto, não obtivemos resultados favoráveis destas técnicas quando usadas conjuntamente com as otimizações oferecidas pelo compilador *gcc*, que implicitamente já as utiliza. Assim, optamos por não as utilizar.

Para avaliar a corretude dos códigos sequenciais apresentados, comparamos os resultados obtidos pela execução do código sequencial com os resultados obtidos por uma versão em `Matlab` (usada no capítulo 3). A comparação foi realizada usando máximo erro relativo entre os 3 campos, calculado conforme a Equação (4.1).

$$\begin{aligned} E_{rel}(H_z) &= \max_{i,j,t} \left[\frac{\left| {H_z}_{seq}|_{i,j}^{t} - H_z|_{i,j}^{t} \right|}{\left| H_z|_{i,j}^{t} \right| + 2^{-40}} \right] \\ \|E_{rel}\|_\infty &= \max \left[E_{rel}(H_z), E_{rel}(E_x), E_{rel}(E_y) \right] \end{aligned} \tag{4.1}$$

onde ${H_z}_{seq}|_{i,j}^{t}$ representa o cálculo do campo magnético através do código sequencial; $H_z|_{i,j}^{t}$ o código executado em `Matlab`; e o fator 2^{-40} é para evitar possíveis divisões por zero. A Tabela 4.1 apresenta os resultados dos erros obtidos nas implementações do exemplo com solução analítica (2ª coluna) e para a implementação PML com tamanho de 10 células em cada direção (3ª coluna) para alguns tamanhos de discretização (1ª coluna).

Como a máxima diferença relativa se manteve numa ordem baixa entre as duas implementações, podemos validar o algoritmo sequencial exposto.

Tabela 4.1: Tabela $\|E_{rel}\|_\infty$

Dimensão	C.C. tipo Dirichlet	PML (N=10)
200×200	$3,869\,10^{-7}$	$6,069\,10^{-9}$
400×400	$1,253\,10^{-8}$	$2,927\,10^{-8}$
600×600	$1,809\,10^{-8}$	$6,253\,10^{-7}$
800×800	$2,031\,10^{-8}$	$1,859\,10^{-6}$
1000×1000	$1,825\,10^{-8}$	$9,209\,10^{-7}$

4.2 Implementação paralela para CPU

Embora, normalmente, executamos processos com apenas um fluxo de controle, é possível a execução de processos constituídos por múltiplas unidades de execução, as quais chamamos de *threads*. A implementação de várias *threads* dentro de uma mesma aplicação nos permite explorar as diversas unidades de processamento de uma máquina, nos inserindo no contexto da programação paralela (BRYANT; O'HALLARON, 2011), ou *multithreading*.

O modelo de programação paralela utilizada nos computadores multiprocessadores é baseado no uso de memória compartilhada. Neste modelo, as diferentes linhas de execução (*threads*) de uma aplicação podem acessar o mesmo espaço de endereçamento da memória, permitindo troca de informação entre elas.

A estrutura de um programa *multithreading* funciona com várias *threads* que trabalham de forma assíncrona, isto é, as *threads* são processadas de forma não ordenada podendo ler e escrever nas variáveis compartilhadas a qualquer momento. Dado isto, a comunicação entre threads via memória compartilhada gera a necessidade de sincronização para garantir resultados adequados. Os trechos de código onde ocorrem instruções de escrita em uma mesma posição de memória, acessada por mais de uma *thread*, são denominados de **seções críticas**. Identificar os trechos

de seção crítica e coordenar a sua execução é um dos problemas fundamentais da programação paralela com memória compartilhada.

Para garantir uma execução correta dentro de uma seção crítica, utilizamos mecanismos de **exclusão mútua** (por exemplo, *locks*), isto é, quando uma thread entra numa seção crítica, as demais threads não podem entrar. A exclusão mútua garante que apenas uma thread esteja na seção crítica por vez.

Além das sincronizações necessárias para proteger o acesso às seções críticas do código, no contexto *multithreading* devemos tratar também as sincronizações lógicas que surgem do algoritmo. Um exemplo é a dependência da conclusão de uma tarefa antes de iniciar a próxima. No esquema de Yee, possuímos dependência lógica entre os passos no tempo pois precisamos da informação calculada no passo anterior para calcular a informação do passo corrente. Além desta, também existe uma dependência natural no espaço por precisarmos da informação de Hz para atualização de E_x e E_y. Esta dependência pode ser observada na Figura 4.3. Para tratarmos a dependência lógica, devemos garantir a finalização de um bloco de tarefas de cada *thread* antes de prosseguir com a execução, o que pode ser atingido utilizando-se um mecanismo de **barreira**. Entretanto, a implementação de barreiras pode resultar em um custo muito elevado para o programa, muitas vezes limitando o ganho de desempenho pelo uso de vários núcleos de processamento.

Nossa implementação *multithreading* tem como base o código sequencial. Para adaptá-lo ao modelo de programação paralela com memória compartilhada, fazemos uso da biblioteca *pthread* que implementa o modelo *POSIX* (*Portable Operating System Interface for UNIX*) CARVER; TAI (2005). A divisão das tarefas entre as *threads* é realizada a partir da malha computacional dos campos elétricos e magnético, agrupando-se as células em submatrizes retangulares, conforme mostra a Figura 4.3. A Figura 4.3 ilustra a divisão da malha computacional em 4 subma-

trizes, onde cada *thread* será responsável pela atualização de uma submatriz. Nessa figura é possível visualizar a dependência lógica espacial entre os campos destacada por uma sombra mais escura. Observe que a dependência ocorre apenas no campo E_y que se encontra no limite de duas submatrizes.

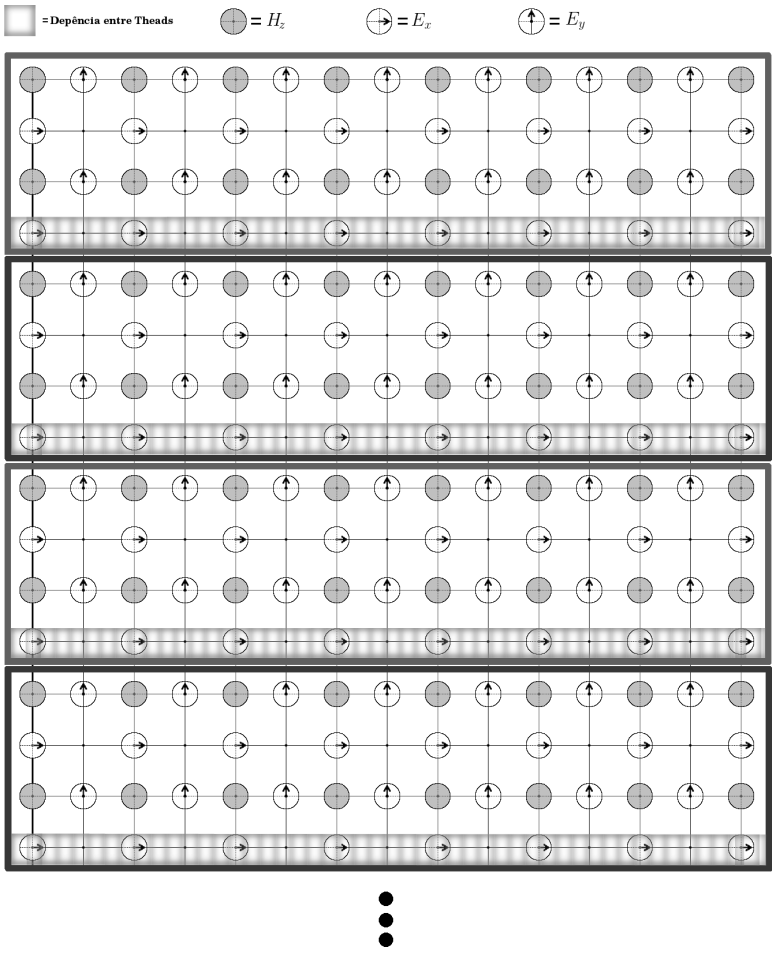

Figura 4.3: Divisão da malha computacional e dependência espacial entre as threads

A estrutura do nosso algoritmo *multithreading* é apresentada na Figura 4.4, considerando as condições de contorno de Dirichlet e PML. O algoritmo é inicializado de forma sequencial, diferindo-se da estrutura sequencial na etapa de atualização dos campos. Cada *thread* é responsável pelo cálculo de uma submatriz retangular de

cada campo (magnético e elétrico). Devido à dependência espacial, é necessário usar uma barreira após o cálculo do campo magnético para garantir o cálculo correto do campo elétrico. Antes de avançar na iteração do tempo, caracterizada pelo *loop* mais externo do fluxograma, também é necessário uma barreira. Por fim, caso a opção de salvar os resultados seja selecionada, faz-se necessário uma terceira barreira para garantir que os dados do campo magnético não sejam sobrescritos durante o processo de gravação.

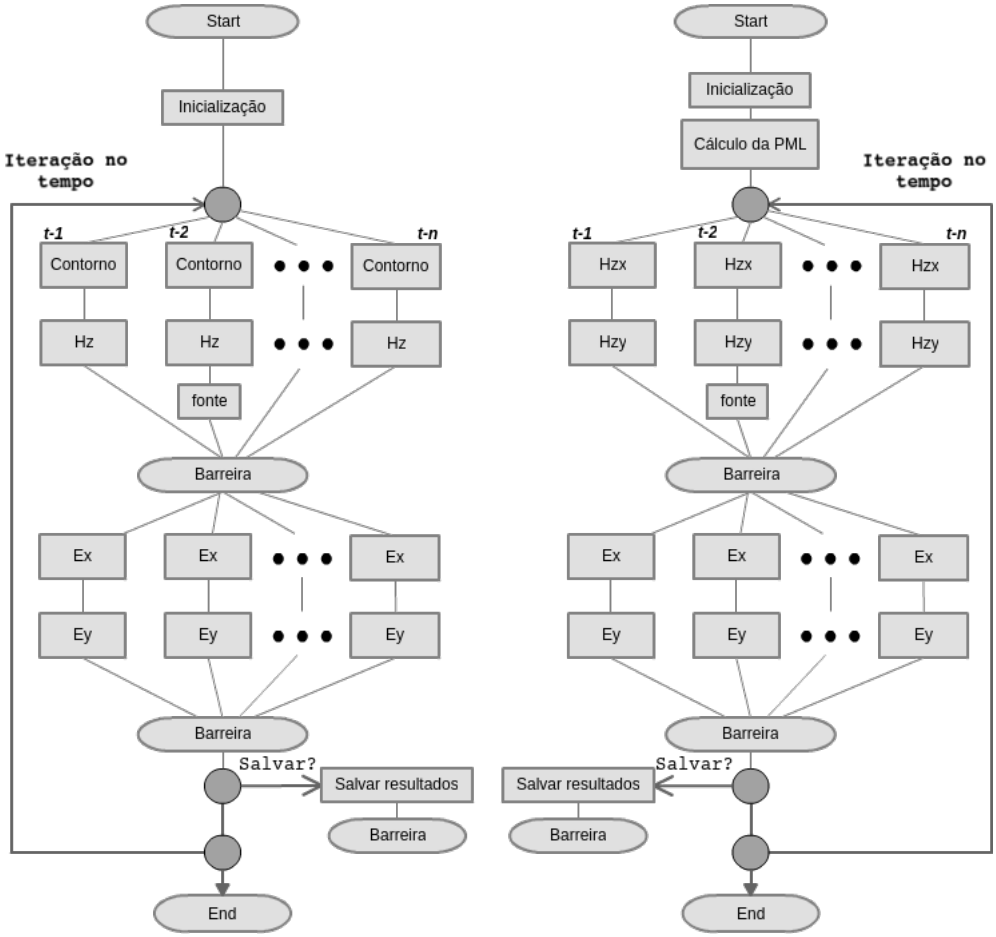

Figura 4.4: Algoritmos *multithreading* do método FDTD: (a) Dirichlet (b) PML

A barreira utilizada no algoritmo foi implementada conforme o código abaixo.

```
void barreira(){
```

```
        pthread_mutex_lock(&x_mutex);
        cont_thread++;
        if (cont_thread<NTHREADS)
                pthread_cond_wait(&x_cond, &x_mutex);
        else {
                cont_thread=0;
                pthread_cond_broadcast(&x_cond);
        }
        pthread_mutex_unlock(&x_mutex);
}
```

A função **barreira** é iniciada com a primitiva **pthread_mutex_lock** que funciona como uma fechadura para garantir a exclusão mútua entre as threads na execução dessa função. **cont_thread** é um contador (variável compartilhada) para indicar o número de *threads* que já chamaram a função `barreira`. Se esta não for a última thread, a primitiva **pthread_cond_wait** é invocada, fazendo a *thread* esperar na variável de condição **x_cond** e liberar o *lock*. A última *thread* a invocar a função `barreira` será responsável por zerar o contador e liberar as demais threads, chamando a primitiva **pthread_cond_broadcast**.

Usamos o mesmo critério apresentado na seção 4.1 para validar o algoritmo *multithreading*. Os resultados são apresentados na Tabela 4.2, onde a 1ª coluna é referente a implementação do exemplo com solução analítica e a coluna para a implementação PML com tamanho de 10 células em cada direção. A diferença observada apresentou os mesmos valores do ambiente anterior. Assim, validamos o algoritmo *multithreading*.

Tabela 4.2: Tabela $\|E_{rel}\|_\infty$

Dimensão	C.C. tipo Dirichlet	PML (N=10)
200×200	$3,869\,10^{-7}$	$6,069\,10^{-9}$
400×400	$1,253\,10^{-8}$	$2,927\,10^{-8}$
600×600	$1,809\,10^{-8}$	$6,253\,10^{-7}$
800×800	$2,031\,10^{-8}$	$1,859\,10^{-6}$
1000×1000	$1,825\,10^{-8}$	$9,209\,10^{-7}$

4.3 Implementação paralela para GPU

As unidades gráficas de processamento (GPUs – *Graphic Processor Units*) são dispositivos aceleradores que trabalham em conjunto com as *CPUs*. Inicialmente, as *GPUs* apresentavam apenas funções gráficas, mas seu potencial para processamento foi percebido e então as GPUs passaram a ser usadas também para processamento geral (STRINGHINI; GONÇALVES; GOLDMAN, 2012).

As *GPUs* possuem um número elevado de núcleos de processamento que nos permite dispor de computação de alto desempenho por um custo relativo muito baixo KIRK; HWU (2010); PATTERSON; HENNESSY (2013). Aplicações com forte paralelismo de dados — como é o caso dos métodos FDTD — são boas candidatas a usufruir dessa capacidade de processamento paralelo das GPUs.

Desde 2006, a NVIDIA (um dos principais fabricantes de placas gráficas) disponibiliza uma plataforma de computação paralela, juntamente com um modelo de programação, para o desenvolvimento de aplicações paralelas que executam nas GPUs, chamado CUDA (*Compute Unified Device Architecture*) (O QUE é CUDA?, 2015). CUDA permite o uso das linguagens C e Fortran, oferecendo um conjunto de funções para acessar e gerenciar o processamento de dados na GPU.

Um programa em CUDA é dividido em duas partes: uma parte que executa na CPU (ou *host*) e outra parte que executa na GPU (ou *device*). A parte que executa na CPU é responsável pela entrada e saída de dados e pela gerência de acesso à GPU. O acesso à GPU envolve: a reserva e liberação de espaço de memória; a transferência de dados da memória principal do computador para a memória da GPU e vice-versa; e a chamada de funções que executam na GPU, denominadas *kernels*. A parte que executa na GPU é tipicamente responsável pelo processamento principal da aplicação e pode ser implementada em um ou vários *kernels*.

No modelo de programação disponibilizado por CUDA, quando uma função *kernel* é disparada para execução na GPU, essa função é executada por várias unidades de processamento ao mesmo tempo, criando fluxos de execução distintos denominados *threads*. Internamente, cada *thread* usa seu identificar único para selecionar quais dados ela irá processar.

O modelo de programação de CUDA é definido como SIMT (*Simple Instruction Multiple Thread*) (PATTERSON; HENNESSY, 2013), o qual expressa o paralelismo da execução de forma explícita. Todas as unidades de processamento executam o mesmo programa (*kernel*), mas cada uma delas sobre uma parte distinta dos dados.

Por exemplo, para incrementar todos os elementos de um vetor de 1, a função *kernel* deverá implementar um único incremento fazendo $vetor[id] = vetor[id] + 1;$, onde *id* será o identificador de cada *thread*. Quando o *kernel* for disparado para execução por um número N de *threads* (onde N corresponde ao número de elementos do vetor), cada *thread* incrementará um elemento do vetor, correspondente ao seu identificador.

CUDA agrupa as *threads* em **blocos** e os blocos em uma **grade**. Na Fi-

gura 4.5 podemos visualizar a relação entre *threads*, blocos e grade. Uma grade possui vários blocos e um bloco possui várias *threads*. Sempre que um *kernel* é disparado, o programador define quantas *threads* irão executar esse *kernel* e como elas serão organizadas em blocos (quantas *threads* por bloco). Os blocos podem ser unidimensionais, bidimensionais ou tridimensionais, significando que as *threads* terão identificadores com 1, 2 ou 3 índices, respectivamente. Por sua vez, uma grade também pode ter até três dimensões, significando que os blocos poderão ter 1, 2 ou 3 índices de identificação. A Figura 4.5 ilustra o caso de uma grade bidimensional com dois blocos também bidimensionais, cada um com duas *threads*.

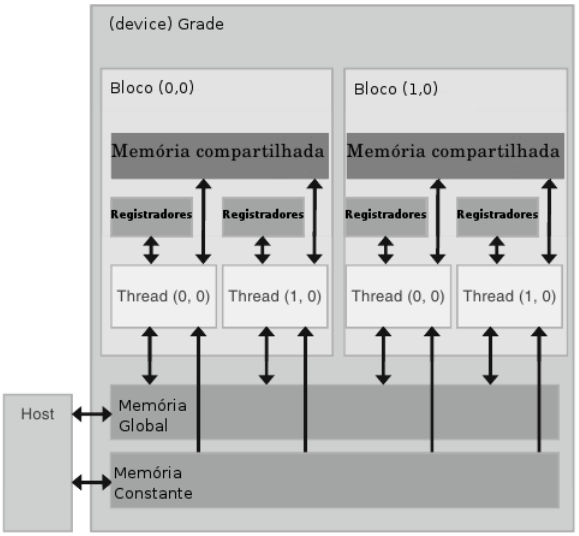

Figura 4.5: Organização das threads e da memória da GPU. Fonte: KIRK; HWU (2010)

A estrutura de memória da GPU abrange vários tipos de memória, com diferentes tamanhos e velocidades de acesso, e é um fator relevante no projeto de um código. Os **registradores** são a memória de acesso mais rápido e são alocados com exclusividade para cada *thread*. Dentro de um bloco, as *threads* compartilham o espaço de **memória compartilhada** (*shared memory*), exclusivo de cada bloco, que pode ser usada como uma memória de rascunho. A **memória global** pode

ser acessada por todas as *threads* de uma grade e também pela CPU. Trata-se da memória mais lenta e de maior capacidade de armazenamento. Por fim, as GPUs também possuem uma área de memória com acesso apenas para leitura, chamada **memória constante** (*constant memory*). A memória constante pode ser escrita e lida a partir da CPU, e apenas lida a partir das *threads* de uma grade. A Figura 4.5 mostra, também, os diferentes tipos de memória da GPU. Na tabela da Figura 4.6 temos as diferentes formas de declaração das variáveis em *CUDA*, com o tipo de memória alocada, escopo da variável e seu tempo de vida.

CUDA Variable Type Qualifiers

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__, __shared__, int SharedVar;`	Shared	Block	Kernel
`__device__, int GlobalVar;`	Global	Grid	Application
`__device__, __constant__, int ConstVar;`	Constant	Grid	Application

Figura 4.6: Tipos de variáveis em CUDA. Fonte: KIRK; HWU (2010)

No modelo de execução de CUDA, as *threads* de um bloco são escalonadas para execução em grupos chamados *warps* (um *warp* contém 32 *threads*). Nesse modelo, uma instrução é carregada de cada vez e todas as *threads* do *warp* executam a mesma instrução, aplicando-a sobre a parte dos dados que compete a cada *thread*. Uma vez escalonadas, as *threads* de um *warp* executam até terminarem o *kernel*, ou requisitarem um dado que está na memória global. Nesse caso, enquanto o dado requerido é lido da memória, outro *warp* pode ser escalonado para execução. Para o caso de aplicações onde é necessário garantir que todas as *threads* de um bloco cheguem juntas a um determinado ponto da execução, CUDA oferece um mecanismo de **sincronização por barreira** por meio da função `__syncthreads()`. Essa função

faz a *thread* esperar que todas as *threads* do mesmo bloco cheguem a esse ponto para então continuar o seu processamento.

O fluxo de execução de um programa CUDA segue, tipicamente, os seguintes passos: lê os dados de entrada e faz as inicializações necessárias; reserva espaço de memória na GPU; transfere os dados de entrada da memória da CPU para a memória global da GPU; executa um (ou mais) *kernels*; transfere os dados processados da memória da GPU para a memória da CPU; exibe os resultados e executa as finalizações necessárias.

4.3.1 Código em CUDA

Para a implementação do método FDTD em GPU, usamos como base o algoritmo proposto em (DE DONNO et al., 2010, 2013). O fluxo principal de execução do algoritmo é apresentado na Figura 4.7, onde a alternativa de salvar os resultados em disco é opcional. Assim como nas versões sequenciais e *multithreading*, implementamos duas versões do algoritmo paralelo, uma usando condições de contorno de Dirichlet e outra usando PML. Em ambos os casos, o controle das iterações temporal é feito na CPU que dispara três kernels a cada passo de tempo.

Na versão com condições de contorno de Dirichlet, temos 3 *kernels* por iteração no tempo. O primeiro *kernel* calcula as condições de contorno, o segundo *kernel* calcula os valores de H_z e o terceiro *kernel* calcula os valores de Ex e Ey. Para a versão com PML, os dois primeiros *kernels* atualizam os valores dos campos H_{zx} e H_{zy}, e o terceiro *kernel* calcula as atualizações de E_x e E_y. A opção de salvar os resultados intermediários, quando selecionada, é feita de forma cíclica e vem acompanhada da transferência de dados da *GPU* para *CPU*. Como a transferência de dados da memória da GPU para a memória da CPU tem um custo alto, só a

realizamos quando a opção por salvar os resultados intermediários é ativada ou no final do processamento.

Em ambas as versões, a função de sincronização `cudaThreadSynchronize()` é invocada entre as chamadas dos *kernels* para garantir que a CPU aguarde a conclusão dos *kernels* anteriores. Essa sincronização é necessária para atender aos requisitos lógicos do algoritmo.

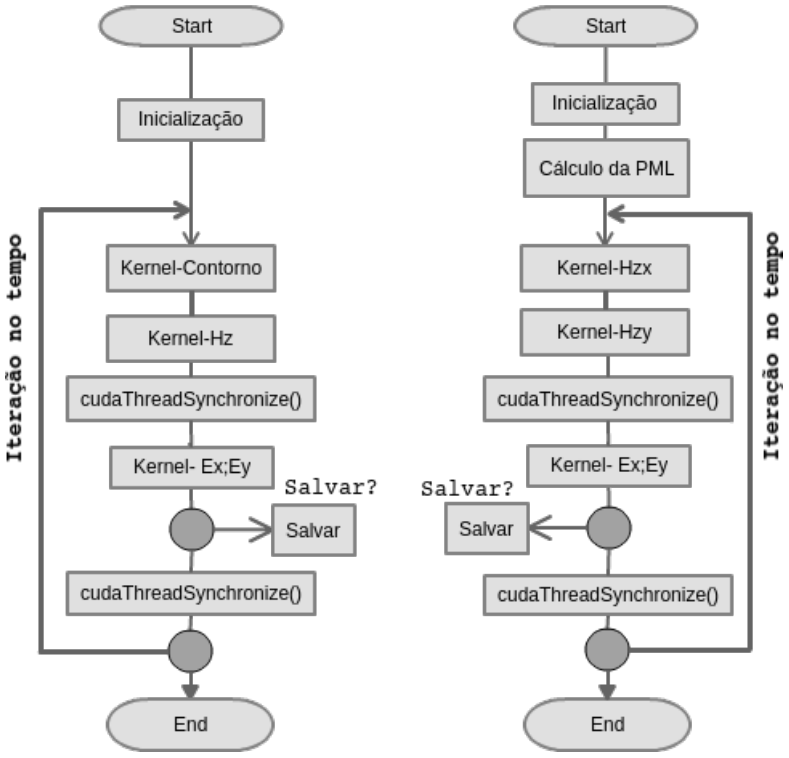

Figura 4.7: Algoritmos GPU-CUDA do método FDTD: (a) Dirichlet (b) PML

Com base nos fluxogramas apresentados na Figura 4.7, foram implementadas duas versões de código. A primeira versão, denominada de **V1**, configura a grade com o número de blocos igual ao número de linhas da malha. Cada bloco tem um número de threads múltiplo de 32, sendo responsável pelo cálculo de diversos pontos

da malha, conforme o código abaixo:

```
for (j=ini_j;j<ny-1;j=j+pulo) {
        c_Hz[I+j]=c_Da_Hz[I+j]*c_Hz[I+j]+c_Db_Hz[I+j]*(dx*(
           c_Ex[I+j] -c_Ex[I+j-1]) - dy*(c_Ey[I+j] -c_Ey[I_1+j
           ]) );
}
```

onde, *pulo* é igual ao número de threads que compõem o bloco.

Na Figura 4.8 apresentamos o *kernel* implementado na versão **V1**. Cada *thread* avança em passos do tamanho do bloco calculando os valores do campo, no exemplo temos blocos de três *threads* executando.

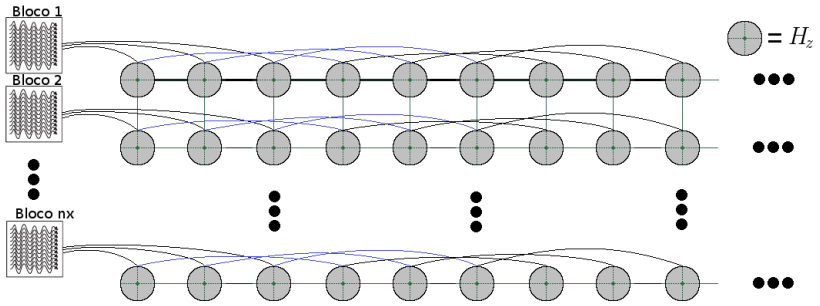

Figura 4.8: Kernel - V1

A segunda versão, denominada de **V2**, segue uma estrutura similar à implementação proposta em (DE DONNO et al., 2013). Nesse caso, a grade é organizada em blocos retangulares com dimensão múltipla de 32 (Figura 4.9). Neste esquema, a implementação dos *kernels* faz com que cada *thread* seja responsável pelo cálculo de uma posição da malha espacial dos campos. A dimensão dos blocos é um parâmentro de entrada.

Seguindo as sugestões de otimização do uso da memória da GPU propostas em outros trabalhos (BALEVIC et al., 2008; DE DONNO et al., 2013; VALCARCE; DE LA ROCHE; ZHANG, 2008), experimentamos o uso da memória compartilhada

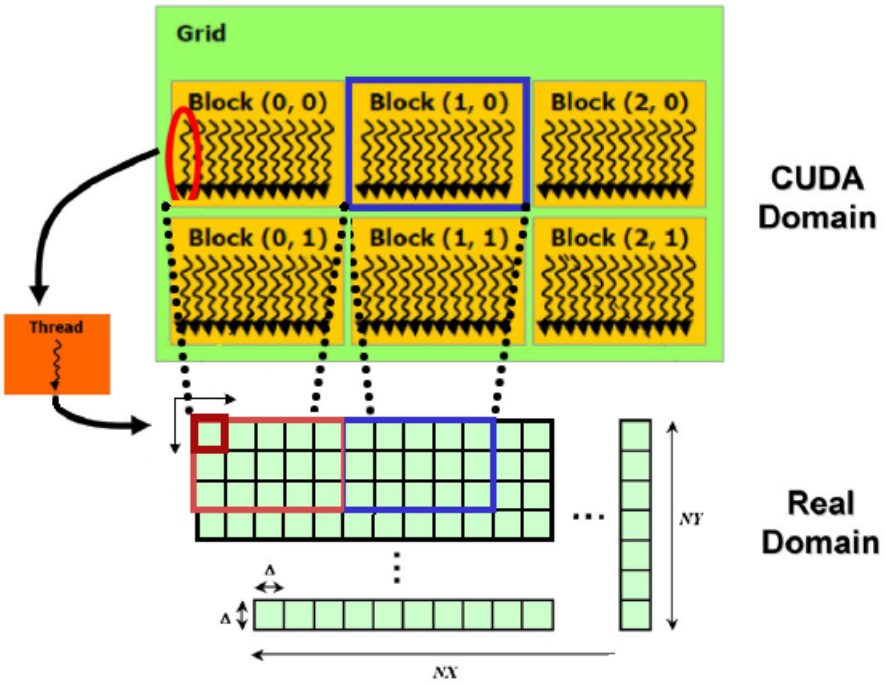

Figura 4.9: Kernel - V2. Fonte: DE DONNO et al. (2013)

entre *threads* do mesmo bloco para armazenar valores que são acessados mais de uma vez em um mesmo *kernel*. Esta abordagem apresentou ganhos apenas para o *kernel* que atualiza os valores de E_x e E_y, visto que ambos utilizam informação da mesma posição de H_z.

Usamos o mesmo critério apresentado na seção 4.1 para validar os algoritmos para GPU. Os resultados são apresentados na Tabela 4.3, onde a 1ª coluna é referente a implementação do exemplo com solução analítica; e a coluna para a implementação PML com tamanho de 10 células em cada direção. As diferenças observadas entre os erros máximos relativos de **V1** foram iguais a **V2**. Como as diferenças relativas foram baixas, consideramos os algoritmos válidos.

Tabela 4.3: Tabela $\|E_{rel}\|_\infty$

Dimensão	C.C. tipo Dirichlet	PML (N=10)
201×201	$3,570\,10^{-7}$	$1,883\,10^{-7}$
401×401	$1,312\,10^{-8}$	$7,458\,10^{-6}$
601×601	$1,840\,10^{-8}$	$3,462\,10^{-3}$
801×801	$1,809\,10^{-8}$	$9,554\,10^{-6}$
1001×1001	$2,045\,10^{-8}$	$1,254\,10^{-4}$

5 AVALIAÇÃO DOS ALGORITMOS PROPOSTOS

Neste capítulo, apresentamos as métricas usadas para avaliar os algoritmos propostos para implementação do método FDTD e os resultados obtidos. Comparamos o desempenho obtido pelas versões sequencial e paralela (em CPU e GPU). A comparação será feita em etapas, ou seja, apresentaremos o desempenho do código sequencial, depois o desempenho do código *multitreading* e por fim o desempenho do código paralelo para GPU. Para medir o ganho de desempenho dos algoritmos paralelos, tomaremos como base o melhor resultado obtido para o algoritmo sequencial.

5.1 Configurações do sistema

Para avaliar os algoritmos propostos, usamos um computador com a seguinte configuração: processador *quad-core* Intel i7-960 3.2 GHz com *hyper-Threading*; 32 Kb de Cache L1, 256 Kb de Cache L2 e 8 Mb de Cache L3; 16 Gb de memória RAM DDR3 1067 MHz. A *GPU* usada foi a NVIDIA Tesla C2070 com 6 GB GDDR5, 448 unidades de processamento, permitindo até 1024 *threads* por blocos; o desempenho dessa *GPU* alcança 515 *Gflops* utilizando precisão dupla e 1.03 *Tflops* com precisão simples. O computador possui sistema operacional Ubuntu 14.04; e os compiladores GCC 4.8.2 e NVCC 5.5.

5.2 Métricas de desempenho

Usaremos três métricas para avaliação do desempenho dos algoritmos propostos:

- **tempo de execução**: tempo em segundos necessário para executar o processamento central dos algoritmos;
- **speedup**: razão entre o tempo do algoritmo sequencial e o tempo do algoritmo paralelo; e
- **speed[MCells/s]**: quantidade de células da malha processadas por unidade de tempo.

A métrica *speed[MCells/s]* é usada no trabalho de Donno et al (DE DONNO et al., 2013) e é definida na Equação (5.1) como milhões de células processadas por segundo:

$$speed[Mcells/s] = \frac{N_x\, N_y\, N_t}{10^6\, T} \tag{5.1}$$

onde N_x e N_y são as dimensões da malha de nós utilizada para a discretização, N_t é o número de passos no tempo e T é o tempo de execução da implementação medido em segundos.

Tomando como referência outros trabalhos da literatura, os experimentos usarão grades com dimensões descritas na Tabela 5.1. Em todos os experimentos consideraremos 1000 iterações no tempo.

A Tabela 5.1 apresenta em sua primeira coluna a dimensão da discretização em quantidade de células. Variamos o tamanho da malha da discretização espacial de 300×300 até 6900×6900 (tamanho máximo limitado pelo espaço de memória

Tabela 5.1: Dimensão X alocação de memória

Dimensão da malha	Memória (Mb) por campo	Memória (Mb) caso homogêneo	Memória (Mb) caso heterogêneo
300^2	0,69	2,06	6,18
1500^2	17,17	51,50	154,50
2100^2	33,65	100,94	302,81
2700^2	55,62	166,85	500,56
3300^2	83,08	249,25	747,76
3900^2	116,04	348,13	1044,39
4500^2	154,50	463,49	1390,46
5100^2	198,44	595,32	1785,96
5700^2	247,88	743,64	2230,91
6300^2	302,81	908,43	2725,30
6900^2	363,24	1089,71	3269,12

global da GPU usada, considerando precisão dupla e uma estrutura de algoritmo preparada para espaços heterogêneos). A segunda coluna nos traz a informação da quantidade de memória necessária para a alocação de um dos três campos calculados. A terceira coluna mostra a quantidade de memória (aproximada) necessária para implementação do método FDTD em C considerando um meio de propagação homogêneo. A quarta coluna contém a quantidade de memória (aproximada) necessária para implementar o esquema o método FDTD para um meio heterogêneo. Observe que a quantidade de memória para o caso heterogêneo é três vezes superior ao caso homogêneo pois temos a necessidade de declarar duas matrizes a mais por campo para descrever o meio de propagação.

5.3 Avaliação do algoritmo sequencial

Os algoritmos sequenciais expostos na seção 4.1 foram compilados com cada uma das opções de otimização de código do compilador *GCC*: "$-O0$"(sem otimiza-

ção), "$-O1$", "$-O2$"e "$-O3$". Na Tabela 5.2 temos os resultados obtidos com os algoritmos sequenciais.

	C.C. tipo Dirichlet				PML			
	O0	O1	O2	O3	O0	O1	O2	O3
300^2	2.53	0.74	0.65	0.60	6.17	2,92	2.73	2.27
1500^2	63.45	25.85	25.71	25.31	94.14	40.47	39.65	38.38
2100^2	124.62	51.23	50.89	50.43	178.71	74.77	72.51	72.00
2700^2	205.50	81.73	80.21	79.46	292.79	122.91	121.96	121.06
3300^2	307.25	122.49	120.46	120.20	429.85	174.33	172.33	170.18
3900^2	428.98	173.68	171.69	170.24	597.89	250.90	249.52	247.83
4500^2	570.81	232.64	229.93	228.73	791.64	334.84	327.62	325.17
5100^2	732.46	298.28	294.58	292.24	1014.02	431.95	427.09	421.52
5700^2	920.01	389.77	383.73	383.54	1263.45	525.73	515.99	511.94
6300^2	1117.02	445.81	438.78	438.28	1545.79	633.14	629.19	620.51
6900^2	1344.99	558.39	552.73	549.94	1874.89	901.37	876.68	882.07

Tabela 5.2: Tempo de processamento medido em segundos do algoritmo sequencial para os dois tipos de condições de contorno implementadas

Podemos notar que o uso das otimizações do compilador *gcc* resultaram em uma redução significativa do tempo de execução do algoritmo sequencial. Observamos também que as otimizações "O2" e "O3" obtiveram resultados muito próximos, com sutil destaque da opção "O3". Para comparação com as implementações paralelas, utilizaremos o menor tempo obtido com o algoritmo sequencial (otimização "O3") como *benchmarck* para o cálculo das demais métricas.

O gráfico apresentado na Figura 5.1 apresenta o desempenho das duas versões do algoritmo sequencial (com as condições de contorno tipo Dirichlet e a *PML*), ambas usando a otimização **"O3"**. Observe que o algoritmo com PML teve um aumento significativo no tempo de processamento. Isso ocorre em razão da divisão do campo Hz em duas componentes e da necessidade de incluir camadas extras de células que neste caso têm tamanho 25 em cada direção. O código com PML demora em média [1] 45% a mais do que a versão com Dirichlet, valor similar ao aumento da

[1] Média da razão entre os tempos da $PML/Dirichlet$, desconsiderando a dimensão 300x300

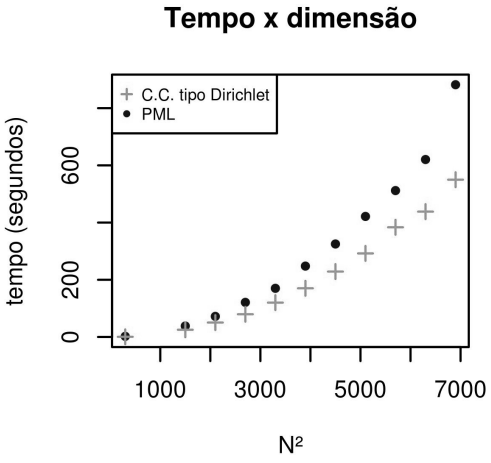

Figura 5.1: Desempenho dos algoritmos sequenciais

memória dado pelo uso da PML.

5.4 Avaliação do algoritmo paralelo para CPU

Os algoritmos da versão paralela com *multithreading* expostos na seção 4.2 possuem como parâmetro de entrada o número de *threads* utilizadas em seu processamento. Realizamos testes variando o número de 2 até 16, de dois em dois, para verificar qual caso obtém o melhor desempenho. Vale ressaltar que o número de *threads* que maximiza o desempenho do algoritmo pode variar dependendo do *hardware*.

A Tabela 5.3 apresenta os resultados das 4 melhores implementações para cada condição de contorno utilizada. No Anexo 1, apresentamos os resultados com todas as quantidades de *threads* avaliadas.

Os tempos de execução mostraram o melhor resultado quando utilizamos

	C.C. tipo Dirichlet				PML			
	2	4	6	8	2	4	6	8
300^2	0.36	0.62	0.61	0.66	1.46	1.59	1.36	1.31
1500^2	24.38	25.41	26.21	27.38	37.72	37.10	38.03	40.06
2100^2	47.45	48.73	50.80	53.60	67.63	71.37	73.27	77.41
2700^2	77.24	81.09	83.57	88.57	112.92	117.23	121.04	127.61
3300^2	115.54	121.34	124.88	132.24	163.84	173.24	178.72	188.66
3900^2	162.57	171.95	175.92	185.08	228.97	241.19	249.52	263.17
4500^2	218.03	228.57	235.03	247.07	302.61	320.57	330.39	347.39
5100^2	277.01	292.77	300.96	316.66	394.39	415.14	426.82	447.46
5700^2	351.69	362.05	372.77	391.75	491.72	515.43	530.11	557.93
6300^2	418.24	445.36	458.28	481.82	595.38	625.15	644.70	677.44
6900^2	516.14	527.46	546.19	573.27	784.71	788.49	796.56	830.09

Tabela 5.3: Tempo de processamento medido em segundos do algoritmo *multithreading* para os dos tipos de condições de contorno implementadas, variando o número de threads

poucas threads, no caso apenas duas. A Figura 5.2 é composta por dois gráficos relativos aos melhores resultados dos códigos com condições de contorno de Dirichlet e PML. O primeiro apresenta os resultados de *speedup* e o segundo o número de milhões de células calculadas por unidade de tempo, variando-se as dimensões da malha de discretização.

Podemos notar que, independente da dimensão do problema, o ganho de desempenho foi muito pequeno. Os valores de *speedup* ficaram muito próximos da média de 1.05 (desconsiderando a malha 300^2).

5.5 Avaliação do algoritmo paralelo para GPU

Para o ambiente de programação paralela em *GPU*, temos duas propostas de algoritmo (V1 e V2). Na versão **V1** temos como parâmetro de entrada o número de threads por bloco (sendo cada bloco responsável pelo cálculo de todas as células

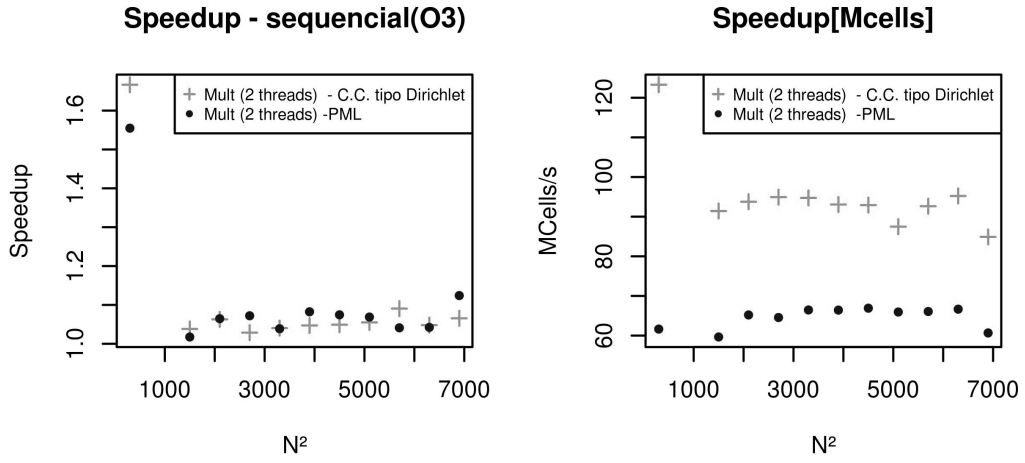

Figura 5.2: Desempenho dos algoritmos *multithreading*: (a) Speedup e (b) Speed[Mcells/s]

de uma linha). Como as threads são escalonadas para processamento em *warps* (grupos de 32 *threads*), testamos sempre tamanhos de blocos múltiplos de 32. A Tabela 5.4 mostra os tempos de execução obtidos para a versão **V1**. De forma geral, a configuração com blocos de 32 threads obteve os melhores resultados.

	C.C. tipo Dirichlet				PML			
	32	64	96	256	32	64	96	256
300^2	1.59	1.59	1.59	1.59	3.46	3.46	3.45	3.45
1500^2	3.90	3.82	4.03	3.90	5.61	5.41	5.65	5.70
2100^2	7.62	7.49	8.10	7.82	10.78	10.54	11.12	11.59
2700^2	12.62	12.56	13.82	13.45	17.85	17.70	19.08	20.12
3300^2	18.84	18.90	21.11	20.82	26.38	26.34	28.70	30.36
3900^2	25.78	26.31	30.02	29.69	36.30	36.86	40.47	43.24
4500^2	34.96	35.45	40.85	40.59	48.71	49.14	54.71	58.54
5100^2	44.93	45.64	53.41	53.21	63.37	64.19	71.63	76.74
5700^2	58.53	60.35	67.91	67.35	78.71	80.11	90.27	96.87
6300^2	72.27	74.65	85.15	83.99	97.86	99.09	112.11	119.87
6900^2	83.76	84.80	102.24	100.77	116.45	118.70	135.12	144.77

Tabela 5.4: Tempo de processamento medido em segundos do algoritmo V1 para os dois tipos de condições de contorno implementadas

Na versão **V2** temos como parâmetro de entrada a dimensão dos blocos, uma

vez que o desempenho do algoritmo é afetado por esse valor. Na Tabela 5.5, mostramos o desempenho das quatro dimensões de blocos que apresentaram os melhores resultados para as duas versões do algoritmo (Dirichlet e PML).

	C.C. tipo Dirichlet				PML			
	1x128	1x256	1x512	1x1024	1x128	1x256	1x512	1x1024
300^2	0,23	0,18	0,20	0,28	0.41	0.27	0.28	0.38
1500^2	4.24	3.70	3.67	3.99	4.88	4.78	4.94	5.13
2100^2	7.16	6.99	7.39	8.21	9.33	8.98	9.38	10.34
2700^2	12.17	11.83	11.64	12.18	16.70	15.32	15.06	15.74
3300^2	18.47	17.44	17.38	18.51	23.64	22.57	22.27	23.71
3900^2	24.70	23.82	23.81	24.68	31.54	30.43	30.68	31.93
4500^2	33.17	31.82	31.75	33.75	43.01	40.99	40.73	43.53
5100^2	42.52	41.45	40.74	41.96	55.20	53.70	53.61	56.53
5700^2	53.29	51.80	51.37	53.09	69.20	66.72	65.59	68.41
6300^2	65.76	63.85	63.39	66.34	83.45	80.80	80.56	85.25
6900^2	78.20	75.79	74.75	77.26	100.18	97.36	95.63	99.66

Tabela 5.5: Tempo de processamento medido em segundos do algoritmo V2 para os dois tipos de condições de contorno implementadas

Observamos que os melhores resultados foram para blocos linha. Em particular, o bloco com dimensão 1×512 foi o que obteve, em geral, o melhor resultado para ambos os algoritmos, mas a diferença foi bastante pequena de uma configuração para outra. Foram realizados testes com outras dimensões, estes resultados se encontram no Anexo 1. Podemos notar que a versão **V2** obteve resultados melhores do que a versão **V1**.

Avaliamos o ganho de desempenho obtido com a versão **V2**, calculando o *speedup* (em relação à versão sequencial) e o número de células calculadas por unidade de tempo. A Figura 5.3 apresenta os resultados obtidos. O primeiro gráfico mostra os resultados de *speedup* em relação ao código sequencial (com otimização **"O3"**), e o segundo gráfico mostra o número de milhões de células calculadas por unidade de tempo, variando-se as dimensões da malha de discretização. Note que a partir da dimensão 1500×1500 os resultados se mostraram estáveis.

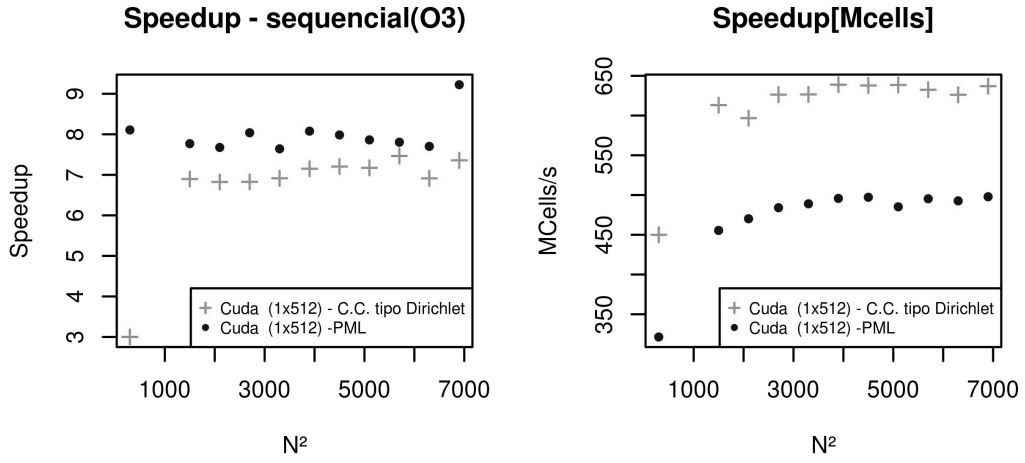

Figura 5.3: Desempenho dos algoritmos paralelos para GPU (V2): (a) Speedup e (b) Speed[Mcells/s]

O gráfico 5.4 apresenta o melhor resultado de cada algoritmo apresentado neste trabalho.

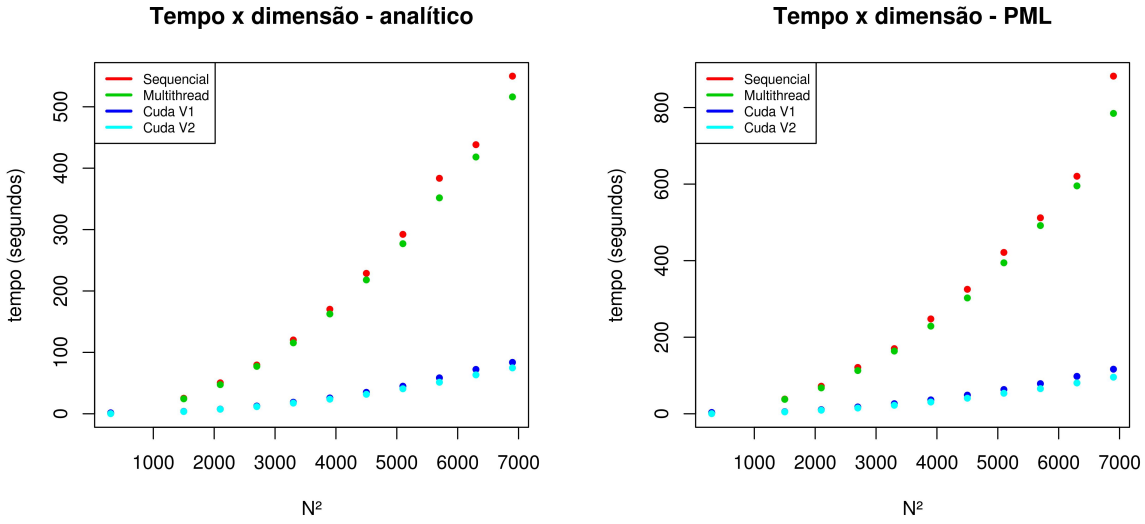

Figura 5.4: Tempo de execução de todos os algoritmos implementados

Podemos verificar que o código paralelo para GPU permitiu uma redução

significativa do tempo de processamento para os dois tipos de condições de contorno. O código paralelo para CPU, por outro lado, não apresentou ganhos significativos em relação à versão sequencial.

A Tabela 5.6 apresenta um resumo das medidas de *speed-up* e *speed[MCells/s]* para todos os algoritmos. Nela incluímos a média [2] dos resultados obtidos. Ressaltamos essa medida pois os resultados mostraram uma certa estabilidade, independente da dimensão da matriz, a partir de dimensões acima de 1000. A medida de dispersão usada foi o desvio padrão dos resultados observados, que é raiz quadrada da média dos desvios em relação a média ao quadrado.

Valores Médios		Speed-up	Speed[MCells/s]
Sequencial (O3)	Dirichlet	-	88,74 (1,94)
	PML	-	61,08 (3,01)
Multithread (2 threads)	Dirichlet	1,05 (0,017)	93,37 (1,04)
	PML	1,06 (0,030)	64,89 (2,58)
Cuda V1 (32 threads)	Dirichlet	6,46 (0,171)	523,22 (11,74)
	PML	6,73 (0,338)	410,38 (5,08)
Cuda V2 (1x512)	Dirichlet	7,08 (0,229)	627,33 (12,24)
	PML	7,98 (0.463)	486,26 (13,7)

Tabela 5.6: *S*peedup e *Speed[MCells/s]* para todos os algoritmos implementados

Ambos os algoritmos em CUDA apresentaram resultados muito promissores. A melhor implementação em GPU obteve um ganho médio de sete a oito vezes quando comparado com a versão sequencial com otimização.

As análises feitas até o momento só levam em consideração o tempo de execução do algoritmo *FDTD*, desconsiderando os tempos de escrita em disco, comum a todos os algoritmos quando desejamos salvar os resultados. Assim como para as aplicações em *GPU* estamos desconsiderando o tempo de alocação de espaço e transferência de dados da CPU para a memória global da GPU e o tempo de transferência

[2] Média dos *speed-up* e speed[MCells/s], desconsiderando a dimensão 300x300

dos dados processados da memória da GPU para a memória da CPU.

Dado que salvar os resultados tem a sua importância, como por exemplo sua utilização em outro software vamos apresentar os tempos de transferência de dados entre a GPU e CPU. Assim como o tempo de escrita em disco, considerando que todos os algoritmos podem gravar os resultados. Estes serão apresentados de forma geral, pois os tempos de escrita e transferência de dados depende apenas do hardware utilizado e não do algoritmo.

Na Tabela 5.7 temos os tempos médios de transferência de dados. A primeira coluna apresenta o tempo de alocação e transferência de dados para a *GPU*, a segunda apresenta a média de 50 medidas de tempo para transferir os dados da GPU para CPU. A coluna "GPU-CPU ($\times$ 50)"tem o tempo total das 50 medidas, a coluna "escrita"apresenta o tempo médio (50 observações) de escrita dos dados em disco; e a coluna "escrita ($\times$ 50)"apresenta o tempo total das 50 medidas de escrita em disco. No anexo 1 temos as medidas de desvio padrão associada aos resultados da Tabela 5.7.

	CPU-GPU	GPU-CPU	GPU-CPU ($\times$ 50)	Escrita	Escrita ($\times$ 50)
300x300	0.09	0.00	0.04	0.00	0.07
1500x1500	0.08	0.01	0.74	0.17	8.68
2100x2100	0.13	0.03	1.44	0.52	26.02
2700x2700	0.16	0.05	2.37	1.00	50.23
3300x3300	0.22	0.07	3.47	1.61	80.29
3900x3900	0.29	0.10	4.87	2.34	117.22
4500x4500	0.37	0.13	6.48	3.18	159.07
5100x5100	0.45	0.17	8.30	4.16	208.09
5700x5700	0.56	0.21	10.43	5.23	261.29
6300x6300	0.67	0.25	12.73	6.43	321.56
6900x6900	0.79	0.30	15.21	7.76	388.01

Tabela 5.7: Tempo de transferência em segundos dos dados dos campos eletromagnéticos

Como pode ser observado, o tempo de transferência dos resultados é conside-

rável. No entanto, o tempo de transferência de apenas um campo entre a *GPU-CPU* não é muito elevado. Mas se este for feito em intervalos de tempo muito curto teremos uma perda considerável no desempenho adquirido pelo uso da *GPU*. Por isso devemos nos restringir a devolver o mínimo de dados possível a *CPU* para mantermos um bom *speed-up*. Outro ponto que devemos chamar atenção é para o tempo de escrita em disco que é muito elevado, devendo ser evitado o quanto possível. Note que o tempo de armazenamento em disco é tão elevado que se torna mais viável o recálculo dos campos que seu armazenamento.

5.6 Considerações finais

Os resultados obtidos foram promissores dado que o ganho no tempo de processamento devido ao uso de GPU foi bem significativo. No entanto, o algoritmo **V2** baseado no trabalho de D. Donno (DE DONNO et al., 2010, 2013) apresentou resultados inferiores aos apresentados no artigo. Em principio, acreditamos que os resultados diferem do artigo por não utilizar memória de textura e possivelmente pela diferença nas estratégias de alocação das constantes que descrevem o meio.

O método FDTD tem como característica relevante ser considerado *memory bound*, pois a quantidade de memória necessária para seu processamento é muito elevada(ANDERSSON, 2002). Em D. Donno, as grades utilizadas chegam até a dimensão de 5100×5100 células, que corresponde ao valor limite da *GPU* utilizada, 1 Gbyte, sugerindo que os seus resultados foram gerados para o caso de um meio homogêneo. Nessas condições, as constantes associadas às propriedades físicas do meio não precisam ser alocadas em forma matricial, ao contrário do nosso caso em que é necessário 1/3 da memória para sua alocação.

Como trabalhos futuros, podem ser realizadas algumas alterações na estru-

tura do algoritmo para atingir melhores ganhos de desempenho. Uma possibilidade é mudar a forma de alocação das propriedades físicas para meios heterogêneos. Em muitas aplicações temos a interação de uma onda eletromagnética com um objeto. Nestes casos é possível dividir o domínio em retângulos, onde cada retângulo tem suas próprias características. Assim, a memória necessária para caracterizar o meio pode ser reduzida, aumentando o desempenho do algoritmo.

Além disso, também podemos tentar o uso da memória de textura como foi sugerido em (DE DONNO et al., 2010). Por fim, a implementação do método FDTD para as equações de Maxwell no caso tridimensional em *GPU*. Para o caso tridimensional esperamos que o uso de memória compartilhada, de forma similar ao código **V2** em *CUDA*, deve apresentar um ganho de desempenho relativo superior ao caso bidimensional devido à maior quantidade de reúso da informação durante o processo de atualização do campo eletromagnético realizado em cada passo de tempo.

6 CONCLUSÃO

A computação científica possui uma demanda muito elevada de processamento de dados e esta aumenta constantemente. Em especial, na área da matemática aplicada existem inúmeros problemas que necessitam de um poder computacional muito elevado para viabilizar a execução das simulações em tempo hábil. Um exemplo é a implementação numérica das equações de Maxwell, onde seu custo é naturalmente muito alto e pode aumentar expressivamente de acordo com a implementação.

Neste trabalho foram apresentados algoritmos para resolução das equações de Maxwell através do esquema de diferenças finitas de Yee para três ambientes de programação: *CPU com código sequencial*, *CPU com código paralelo* e *GPU*. Todos os códigos foram validados por meio de exemplos analíticos e numéricos. Por fim, realizamos uma série de testes com diversas dimensões de malha para avaliar o desempenho dos códigos em todos os ambientes.

A implementação paralela aplicada a processadores multinúcleos não apresentou um ganho de desempenho significativo. A implementação paralela para GPU (usando *CUDA*), por outro lado, obteve ganhos bastante significativos de desempenho em relação a aplicação sequencial otimizada. Estes ganhos foram em média 8 vezes superior a melhor implementação sequencial e chegando a 9 vezes considerando o maior tamanho de malha testado.

Logo, neste trabalho podemos verificar e quantificar o quanto a tecnologia de programação paralela aplicada a processadores gráficos pode acrescentar em desempenho no estudo de problemas eletromagnéticos através do *FDTD*.

REFERÊNCIAS

ADAMS, S.; PAYNE, J.; BOPPANA, R. Finite difference time domain (FDTD) simulations using graphics processors. In: DOD HIGH PERFORMANCE COMPUTING MODERNIZATION PROGRAM USERS GROUP CONFERENCE, 2007, Pittsburgh. **Proceedings...** IEEE, 2007. p. 334–338.

ANDERSSON, U. **Yee bench- A PDC benchmark code**. Stockholm: Department of Numerical Analysis and Computer Science, 2002 (Relatório técnico, TRITA-PDC-2002:1).

BALEVIC, A. et al. Accelerating simulations of light scattering based on finite-difference time-domain method with general purpose GPUs. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTATIONAL SCIENCE AND ENGINEERING, 11., 2008, São Paulo. **Proceedings...** New York: IEEE, 2008. p. 327–334.

BERENGER, J. P. A perfectly matched layer for the absorption of electromagnetic waves. **Journal of Computational Physics**, Orlando, v. 114, n. 2, p. 185–200, 1994.

BRYANT, R. E.; O'HALLARON, D. R. **Computer systems**: a programmer's perspective. 2. ed. Boston: Prentice Hall, 2011.

CARVER, R. H.; TAI, K.-C. **Modern multithreading**: implementing, testing, and debugging multithreaded java and c++/pthreads/win32 programs. Hoboken: Wiley, 2005.

CHELLAPPA, S.; FRANCHETTI, F.; PÜSCHEL, M. How to write fast numerical code: a small introduction. In: LÄMMEL, R.; VISSER, J.; SARAIVA, S. (Ed.). **Generative and Transformational Techniques in Software Engineering II**:

International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers. Berlin: Springer, 2008. p. 196–259. (Lecture Notes in Computer Science, 5235).

DE DONNO, D. et al. Introduction to GPU computing and CUDA programming: a case study on FDTD. **IEEE Antennas and Propagation Magazine**, New York, v. 52, n. 3, p. 116–122, 2010.

DE DONNO, D. et al. GPU-based acceleration of computational electromagnetics codes. **International Journal of Numerical Modelling: Electronic Networks, Devices and Fields**, Chichester, v. 26, n. 4, p. 309–323, 2013.

DEMIR, V.; ELSHERBENI, A. Z. Compute unified device architecture (CUDA) based finite-difference time-domain (FDTD) implementation. **ACES Journal**, v. 25, n. 4, p. 303–314, 2010.

EL ZEIN, G.; KHALEGHI, A. Emerging Wireless Communication Technologies. In: LABIOD, H.; BADRA, M. (Ed.). **New Technologies, Mobility and Security**. Berlin: Springer, 2007. p. 271–279.

HAND, J. W. Modelling the interaction of electromagnetic fields (10 MHz–10 GHz) with the human body: methods and applications. **Physics in medicine and biology**, Bristol, v. 53, n. 16, p. R243–R286, 2008.

INMAN, M. J.; ELSHERBENI, A. Z.; SMITH, C. E. FDTD calculations using graphical processing units. In: IEEE/ACES INTERNATIONAL CONFERENCE ON WIRELESS COMMUNICATIONS AND APPLIED COMPUTATIONAL ELECTROMAGNETICS, 2005, Honolulu. **Proceedings...** New York: IEEE, 2005. p. 728–731.

KASHDAN, E.; GALANTI, B. A new parallelization strategy for solving time-dependent 3D Maxwell equations using a high-order accurate compact implicit scheme. **INTERNATIONAL JOURNAL OF NUMERICAL MODEL-**

LING: ELECTRONIC NETWORKS, DEVICES AND FIELDS, Chichester, v. 19, n. 5, p. 391–408, 2006.

KIRK, D. B.; HWU, W.-M. **Programming massively parallel processors hands-on with cuda**. Burlington: Morgan Kaufmann, 2010.

KRAKIWSKY, S. E.; TURNER, L. E.; OKONIEWSKI, M. M. Graphics processor unit (GPU) acceleration of finite-difference time-domain (FDTD) algorithm. In: INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, 2004, Vancouver. **Proceedings...** New York: IEEE, 2004. v. 5, p. V–265–V–265.

LIMA, C. B. **Análise de dispositivos eletromagnéticos para hipertermia usando o método FDTD**. 2006. 120 f. Tese (Doutorado em Engenharia Elétrica) — Universidade Federal de Santa Catarina, Florianópolis, 2006.

MUR, G. Absorbing boundary conditions for the finite-difference approximation of the time-domain electromagnetic-field equations. **IEEE Transactions on Electromagnetic Compatibility**, New York, v. EMC-23, n. 4, p. 377–382, 1981.

O que é CUDA? Disponível em: <`http://www.nvidia.com.br/object/cuda_home_new_br.html`>, Acesso em: 30 ago. 2015.

PATTERSON, D. A.; HENNESSY, J. L. **Computer organization and design**: the hardware/software interface. Burlington: Morgan Kaufmann, 2013.

PEREIRA, N. R. **Comparação do desempenho do FDTD com implementação em CPU e em GPU**. 2012. 70 f. Dissertação (Mestrado em Engenharia de Computadores e Telematica) — Universidade de Aveiro, Aveiro, 2012.

STALLMAN, R. M. **Using the GNU compiler collection**: for gcc version 5.2.0. Boston: GNU Press, 2010.

STRINGHINI, D.; GONÇALVES, R. A.; GOLDMAN, A. Introdução à Computação Heterogênea. In: JORNADAS EM ATUALIZAÇÃO EM INFORMÁTICA, 31.,

2012, Curitiba and CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 32., 2012, Curitiba. **Proceedings...** Porto Alegre: Sociedade Brasileira de Computação, 2012.

SULLIVAN, D. M. **Eletromagnetic Simulation Using The FDTD Method**. New York: Wiley, 2000.

TAFLOVE, A.; HAGNESS, S. C. **Computational Electrodynamics**: the finite-difference time-domain method. 2. ed. Boston: Artech House, 2000.

VALCARCE, A.; DE LA ROCHE, G.; ZHANG, J. A GPU approach to FDTD for radio coverage prediction. In: IEEE SINGAPORE INTERNATIONAL CONFERENCE ON COMMUNICATION SYSTEMS, 11., 2008, Guangzhou. **Proceedings...** New York: IEEE, 2008. p. 1585–1590.

YEE, K. S. Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. **IEEE Antennas and Propagation Magazine**, v. 14, n. 3, p. 302–307, 1966.

ANEXO A - TABELAS COM OS TEMPOS DAS SIMULAÇÕES

	2	4	6	8	10	12	14	16
300^2	1.46	1.59	1.36	1.31	1.74	1.63	1.71	1.77
1500^2	37.72	37.10	38.03	40.06	39.53	39.27	39.51	39.70
2100^2	67.63	71.37	73.27	77.41	76.09	76.01	76.07	76.78
2700^2	112.92	117.23	121.04	127.61	125.62	125.88	126.00	126.93
3300^2	163.84	173.24	178.72	188.66	185.85	186.85	187.08	188.06
3900^2	228.97	241.19	249.52	263.17	259.32	260.82	261.25	262.42
4500^2	302.61	320.57	330.39	347.39	343.49	345.20	345.49	346.89
5100^2	394.39	415.14	426.82	447.46	443.00	444.77	445.62	446.75
5700^2	491.72	515.43	530.11	557.93	551.57	554.36	555.51	557.40
6300^2	595.38	625.15	644.70	677.44	672.65	675.39	676.35	678.03
6900^2	784.71	788.49	796.56	830.09	823.65	825.75	826.52	828.36

Tabela 1: Tempo de processamento do algoritmo *multithreading* para PML

	1x32	1x64	1x128	1x256	1x512	1x1024	2x32
300^2	0.35	0.30	0.41	0.27	0.28	0.38	0.33
1500^2	6.34	5.24	4.88	4.78	4.94	5.13	6.00
2100^2	12.17	10.18	9.33	8.98	9.38	10.34	11.41
2700^2	22.80	17.82	16.70	15.32	15.06	15.74	22.71
3300^2	30.34	25.33	23.64	22.57	22.27	23.71	28.64
3900^2	41.14	33.52	31.54	30.43	30.68	31.93	38.53
4500^2	56.04	45.94	43.01	40.99	40.73	43.53	53.26
5100^2	70.45	58.40	55.20	53.70	53.61	56.53	66.97
5700^2	88.72	74.37	69.20	66.72	65.59	68.41	84.13
6300^2	108.74	88.39	83.45	80.80	80.56	85.25	102.42
6900^2	129.47	105.31	100.18	97.36	95.63	99.66	120.51

Tabela 2: Tempo de processamento do algoritmo CUDA V2 para PML

	2x64	2x128	2x256	2x512	4x32	4x64	4x128	4x256
300^2	0.31	0.31	0.32	0.34	0.38	0.37	0.36	0.40
1500^2	5.45	5.30	5.54	5.94	6.82	6.59	6.56	7.07
2100^2	10.25	9.90	10.19	10.98	12.96	12.35	12.24	12.94
2700^2	18.50	17.17	16.81	17.79	24.29	20.86	20.07	20.83
3300^2	26.06	24.56	24.80	25.84	32.12	30.62	30.15	31.28
3900^2	34.78	33.57	34.55	36.02	42.78	41.15	40.59	43.00
4500^2	47.45	44.86	45.27	47.10	60.71	56.41	55.45	57.74
5100^2	60.46	57.90	59.29	62.53	74.34	71.66	70.96	74.00
5700^2	76.51	72.03	72.64	76.02	93.44	89.27	87.87	91.75
6300^2	91.83	87.77	89.41	93.99	114.46	109.47	106.75	110.81
6900^2	108.41	104.48	106.22	110.45	135.19	129.39	128.33	133.42

Tabela 3: Tempo de processamento do algoritmo CUDA V2 para PML

	C.C. tipo Dirichlet	PML
300^2	0.38	0.66
1500^2	6.49	6.84
2100^2	6.62	6.68
2700^2	6.30	6.78
3300^2	6.38	6.45
3900^2	6.60	6.83
4500^2	6.54	6.68
5100^2	6.50	6.65
5700^2	6.55	6.50
6300^2	6.06	6.34
6900^2	6.57	7.57

Tabela 4: Speed-up- multithread (2 threads) vs sequencial (O3)

	C.C. tipo Dirichlet	PML
300^2	2.61	6.49
1500^2	3.63	6.05
2100^2	5.17	5.92
2700^2	4.90	5.31
3300^2	4.64	5.61
3900^2	5.19	6.02
4500^2	5.06	5.80
5100^2	5.19	5.98
5700^2	5.25	5.77
6300^2	4.91	5.71
6900^2	5.24	6.81

Tabela 5: Speed-up- CUDA V1 (32 threads) vs sequencial (O3)

	C.C. tipo Dirichlet	PML
300^2	3.00	8.11
1500^2	6.90	7.77
2100^2	6.82	7.68
2700^2	6.83	8.04
3300^2	6.92	7.64
3900^2	7.15	8.08
4500^2	7.20	7.98
5100^2	7.17	7.86
5700^2	7.47	7.81
6300^2	6.91	7.70
6900^2	7.36	9.22

Tabela 6: Speed-up- CUDA V2 (1x512) vs sequencial (O3)

	C.C. tipo Dirichlet	PML
300^2	150.00	39.65
1500^2	88.90	58.62
2100^2	87.45	61.25
2700^2	91.74	60.22
3300^2	90.60	63.99
3900^2	89.34	61.37
4500^2	88.53	62.28
5100^2	89.00	61.71
5700^2	84.71	63.46
6300^2	90.56	63.96
6900^2	86.57	53.98

Tabela 7: Speed[Cells/s]- Sequencial (O3)

	C.C. tipo Dirichlet	PML
300^2	250.00	61.64
1500^2	92.29	59.65
2100^2	92.94	65.21
2700^2	94.38	64.56
3300^2	94.25	66.47
3900^2	93.56	66.43
4500^2	92.88	66.92
5100^2	93.90	65.95
5700^2	92.38	66.07
6300^2	94.90	66.66
6900^2	92.24	60.67

Tabela 8: Speed[Cells/s]- Multithread (2 threads)

	C.C. tipo Dirichlet	PML
300^2	450.00	321.43
1500^2	613.08	455.47
2100^2	596.75	470.15
2700^2	626.29	484.06
3300^2	626.58	489.00
3900^2	638.81	495.76
4500^2	637.80	497.18
5100^2	638.44	485.17
5700^2	632.47	495.35
6300^2	626.12	492.68
6900^2	636.92	497.86

Tabela 9: Speed[Cells/s]- CUDA V1 (32 threads)

	C.C. tipo Dirichlet	PML
300^2	391.30	257.14
1500^2	322.81	354.89
2100^2	451.84	362.37
2700^2	449.17	319.74
3300^2	420.30	358.93
3900^2	463.72	369.71
4500^2	448.11	361.35
5100^2	461.66	369.20
5700^2	444.76	366.21
6300^2	444.81	365.00
6900^2	453.26	367.73

Tabela 10: Speed[Cells/s]- CUDA V2 (1x512)

	CPU-GPU	GPU-CPU	GPU-CPU (× 50)	escrita	escrita (× 50)
300x300	0.08	0.00	0.00	0.00	0.00
1500x1500	0.00	0.00	0.17	0.01	8.56
2100x2100	0.01	0.00	0.30	0.02	14.85
2700x2700	0.00	0.00	0.42	0.03	20.87
3300x3300	0.00	0.00	0.50	0.03	25.04
3900x3900	0.00	0.00	0.63	0.06	31.45
4500x4500	0.01	0.00	0.67	0.05	33.69
5100x5100	0.01	0.00	0.73	0.08	36.49
5700x5700	0.01	0.00	0.85	0.18	42.62
6300x6300	0.03	0.00	0.90	0.11	44.97
6900x6900	0.01	0.00	0.92	0.10	45.85

Tabela 11: Desvio padrão amostral dos tempos de transferência.